THE UNIVERSITY OF NEWCASTLE UPON TYNE
COMPUTING LABORATORY


# The Exploitation of Parallelism on Shared Memory Multiprocessors

by

Michael A. Stoker

PhD Thesis

September 1990

# *Abstract*

With the arrival of many general purpose shared memory multiple processor (multiprocessor) computers into the commercial arena during the mid-1980's, a rift has opened between the raw processing power offered by the emerging hardware and the relative inability of its operating software to effectively deliver this power to potential users. This rift stems from the fact that, currently, no computational model with the capability to elegantly express parallel activity is mature enough to be universally accepted, and used as the basis for programming languages to exploit the parallelism that multiprocessors offer. To add to this, there is a lack of software tools to assist programmers in the processes of designing and debugging parallel programs.

Although much research has been done in the field of programming languages, no undisputed candidate for the most appropriate language for programming shared memory multiprocessors has yet been found. This thesis examines why this state of affairs has arisen and proposes programming language constructs, together with a programming methodology and environment, to close the ever widening hardware to software gap.

The novel programming constructs described in this thesis are intended for use in imperative languages even though they make use of the synchronisation inherent in the dataflow model by using the semantics of single assignment when operating on shared data, so giving rise to the term *shared values*. As there are several distinct parallel programming paradigms, matching flavours of shared value are developed to permit the concise expression of these paradigms.

# *Acknowledgements*

Although the research behind this thesis and its subsequent writing are attributed to one individual, I feel that it is fitting to mention the other people who helped make it possible. First and foremost, I would like to thank my supervisor, Professor Peter Lee, for his comments and guidance over the course of my research. Moreover, I am grateful for his time and encouragement given during the latter phases of writing this thesis.

Secondly, I would like to thank Dan McCue for his comments and suggestions regarding the implementation of my work. In addition, I would like to thank him for reading and commenting on early drafts of this thesis.

Finally, I acknowledge the financial support from the Science and Engineering Research Council during the production this thesis.

# *Table of Contents*

# *List of Figures*

# *Chapter 1*

# Introduction

Ever since the first general purpose digital computers were constructed around the early 1950's, the users of computers have sought greater processing power than that which was currently available. Despite the tremendous leaps made in processing speeds, due mainly to rapid advances in hardware technology, users' expectations for performance have always outstripped the supply. Early requirements for processing speeds in the range of thousands of operations per second have now been replaced by requirements for processing speeds approaching thousands of millions of operations per second. Experience has shown that no matter how powerful a computer may be, there will always be some large application, or for that matter, a combination of many small applications, that will either consume or even exhaust its entire processing potential.

Over the last forty years, hardware technology (involving processors, memory and peripherals) has undergone rapid transformations. Huge increases in performance have been obtained over the first generation of valve-based systems, with this increase not only helping to offset, but at the same time fuel, the demand for more performance. The advent of the transistor in the 1960's, and LSI (Large Scale Integration) in the 1970's (subsequently VLSI in the 1980's) have had a twofold effect on the computer generations they heralded. Each new generation of components was faster, cheaper, smaller and more reliable than its predecessors. Therefore, initially, the main attraction was an improved price/performance characteristic, but later this gave way to the idea of constructing new machines more powerful than ever before. This was in part made possible by the higher reliability of the new components as it was not unreasonable for some early computers to crash daily (if not more frequently) resulting in high operating costs and maintenance bills. But improvements in both hardware and software technologies have meant that failures now happen far less frequently. Undoubtedly the two seductive factors of good price/performance and high reliability have contributed to the wide scale acceptance of computers into society and thus have ultimately secured their future.

Computers have been applied to a diverse range of tasks and the number of applications in which computers are being used is rapidly increasing. As more powerful machines become available, new and more challenging software

applications are being devised to take advantage of the high processing performances now on offer. Moreover, existing applications are being scaled up to match the new availability of processing power. For example, in weather forecasting more accurate predictions than ever before are being obtained. This is happening because the process of forecasting [Kaln79] is centred around solving partial differential equations represented on a three dimensional mesh. A reduction in the mesh size, which while tremendously increasing the amount of processing required for a solution, also gives rise to more accurate results.

For a significant proportion of applications the real bottleneck in a computer system is the processor, as it actually does the work of executing programs. An obvious path to greater processing power is to simply employ a faster processor, but this is not quite so simple in practice. If a significantly faster processor is to be used, then the other components of the computer system (memory and disks) may also have to be upgraded. Adding a faster processor to a system can change its balance from being *compute-bound* to *memory* or *input-output bound*, thus reducing the overall benefit gained from the introduction of the new processor. Furthermore, faster processors based on the leading edge of technology are often very expensive and if further components have to be upgraded, then this merely compounds the cost. While the economic cost of performance is important there is another more important long term factor to be considered. The electronic-based technologies that produced the *computer revolution* are rapidly being extended to their physical limits and will unable to be a continuing source of performance increase. Hence, new sources of performance improvement must be found to meet ever increasing performance requirements.

Naturally one could turn to technologies other than electronics in the pursuit of faster machines, as in the case of optical computers. However, a large amount of research regarding the construction and manufacture of optical computers remains to be completed before such far-ranging steps can be taken. A more technologically conservative approach to the problem, however, is not to build a faster traditional computer from superior technology, but instead, to improve upon the original computer model by re-examining the fundamental way in which it works. Originally, computers were envisaged as functioning in a serial fashion executing a single operation at a time. As the study of computer systems developed, intra-processor parallelism (e.g. instruction pipelining) and system level parallelism (e.g. multiprogramming by overlapping input and output requests) were introduced. From the user programming point of view, these kinds of parallelism were buried in the hardware and operating systems so no changes

in programming style were needed. Furthermore, in large machines special purpose processors were added to off-load input/output activity from the central processor (e.g. channels on IBM 360 series). These uses of parallelism in single processor systems have been applied very successfully over the years and continue to be very important, even so, greater performance is needed still. Thus, the next logical step is to introduce multiple central processors into processing systems to form true multiprocessors. Although this is quite straightforward in theory, there are many engineering and other pragmatic problems that have to be solved to produce viable multiprocessors and therefore to put *parallel processing* into practice. Parallel processing can be described as the process of partitioning an application into a number of components, and distributing the components over a group of processors so that each component can execute concurrently, with a view to executing the application as efficiently as possible.

## 1.1 Applications and Technology

One of the most influential factors on the design of commercially viable computer systems has been the projects or application domains in which the resulting computers were intended to be used. Parallel processing has tended to be used in applications that deal with massive amounts of possibly distributed data, or in applications that have real-time constraints. Recent areas where parallelism has been exploited include: structural analysis, weather forecasting, petroleum exploration, fusion energy research, medical diagnosis, aerodynamics simulations, artificial intelligence, expert systems, industrial automation, remote sensing, military defence, genetic engineering and socioeconomics.

Projects can constrain the performance, physical size and most importantly the cost of prospective computer systems. For example, the flight control system of an aircraft must be able to operate in real time, be accommodated by the aircraft, and be paid for out of the total cost of the aircraft. The other major influence on computer design, mentioned earlier, is technology. The factors of technology and application domains have caused a broad spectrum of widely differing computers to be produced. Almost all commercially available machines are aimed at general purpose work and are based on the traditional computing model (i.e. the von Neumann model described in chapter two), with variations in price largely accruing from the sizes, speeds and complexities of central processors, main memories and peripherals.

There have been many attempts at producing parallel architectures in the past with varying degrees of success [Fern81, Stok81, Thor81]. But no parallel architecture has been generally accepted as being universally superior to the traditional von Neumann model. One reason for this lack of architectural change, which was especially true in the 1960's, was that the technology for the effective construction of parallel architectures was simply not available. The extra components necessary for a parallel machine were too expensive and with the extra components there were just too many failures for acceptable machine reliability. Later, in the 1970's when technology had advanced, reliability was not the main drawback, but instead, it was primarily the cost of manufacture that prevented the spread of parallel machines. Multiprocessing relies on employing multiple processing units, but at that time processors were expensive and it was often the case that it was cheaper and more efficient to build a bigger, faster processor than to use multiple processors. Evidence to support this claim can be gained from Grosh's Law [Knig66]. Fortunately in the early 1980's with the advent and proliferation of cheap, reliable microprocessors the converse situation arose. Microprocessor (CMOS and NMOS) based multiprocessors have become more economic than fast uniprocessors due to the rapid surge in microprocessor technology [Dong85]. The rate at which this technology is advancing is significantly faster than that which is associated with traditional mainframes (bipolar technology), hence the future of microprocessors seems very bright indeed. Consequently, as long as microprocessors flourish, the multiprocessors that utilise them will at least in theory be able to offer aggregate performances superior to those of the fastest uniprocessors.

However, not all multiprocessors are simply based on standard microprocessors and at the moment a variety of parallel architectures exist, with no single parallel architecture really dominating. The main reason for the diversity found in parallel architectures is that the traditional von Neumann model has to be generalised or abandoned to give a model that incorporates parallelism from the outset. Furthermore, parallel processing is associated with a number of specialist markets, such as supercomputing and fault tolerance, which have to apply parallelism in a manner that most suits their application area's needs. (Further information regarding parallel architectures can be found in chapter two.) Parallel machines currently available include shared memory multiprocessors (e.g. Encore Multimax and Sequent Symmetry), distributed memory multiprocessors (e.g. Intel hypercube and NCube), mini-supercomputers (e.g. Stardent Titan and Alliant FX/80) and fully fledged supercomputers (e.g.

Cray Y-MP and NEC 10). While these machines differ greatly in design and capability they have all evolved from the basic von Neumann model.

There have been many proposals put forward, and several machines built, along completely different lines as alternatives to the von Neumann model . These machines include array processors (e.g. DAP and Connection Machine) and other exotic machines such as the Manchester Dataflow Machine. Although these machines are described as being general purpose computers it is common to find them being used exclusively for special purposes such as image processing or computer science research. For reasons that will be explored next, this thesis focuses on the evolutionary architecture of the shared memory multiprocessor, that extends the von Neumann model through the use of multiple processing units, while retaining the concept of a single global memory.

Although there is a wide variety of multiprocessors now becoming available, a large proportion of every-day computing is carried out by traditional uniprocessor machines. With the exception of the parallel supercomputers, symmetric shared memory multiprocessors (SMMs) have a number of advantages over these established uniprocessor systems which include:

- high performance,
- low cost,
- reconfigurability,
- availability.

The price versus performance measure for SMMs compares very favourably to mainframes. This stems from the fact that multiprocessors are generally constructed from off-the-shelf microprocessors, therefore, allowing them to incorporate the very latest microchips and to ride the technology curve as microprocessor speeds increase. Following on, it is quite common for the imminent need to upgrade a computer system to be recognised during its lifetime. This ideally means substituting an existing processor type for a faster one, given that the architecture can withstand the change. Most multiprocessor systems are designed so that they are constructed in a modular fashion, which enables processor and memory boards to be replaced with relative ease. In large uniprocessor systems, it is usually much harder to replace processing elements because they can be closely tied to other components of the system. Finally, many multiprocessor systems can operate with less than a full complement of processors, allowing processors to be bought only if they are required and in the case of processor failures, the multiprocessor remains functional, albeit in a degraded

state. Thus, SMMs can be seen as strong contenders to be the successors to uniprocessors, at least in the traditional mainframe market. To bolster this claim further, SMMs in common with other multiprocessors based on standard microprocessors have an economic advantage over some of their rival parallel architectures. More specifically, standard processing components are cheap, reliable and widely available due to the proven technology of their construction, as opposed to some of the more exotic parallel machines that rely on customised VLSI chips. In addition, a shared memory multiprocessor can be viewed as a group of tightly coupled uniprocessors executing a variant of a common operating system like the Unix operating system. This simplifies the job of migrating existing software from uniprocessors to a shared memory multiprocessor, which is not always such a straightforward task in the case of other multiprocessor architectures such as private memory machines.

Even though parallel programming has been studied for many years it is still a rapidly expanding subject made all the more so by the current proliferation of multiprocessors. Parallel programming is a very challenging area to research, with many important questions remaining unanswered, and ultimately will have an effect on the shape of future computing systems. Further to this, it can be observed that multiprocessors, are growing in importance and frequency in common use. Good methods of programming and operating these machines have to be researched in order to satisfy the demand generated by the rapidly expanding community of users. At the current time, the growth of parallel programming is being stifled as it is often considered a taxing process with its application restricted to expert users. With the introduction of new programming languages and operating systems, the appeal of parallel programming will broaden and multiprocessors will be able to fulfil their role as high performance computing engines.

## 1.2 The Software Shortfall

At the current time it is quite possible to design and code programs that execute concurrently on multiprocessors. Notwithstanding this, the crux of the matter is that the techniques that are being used in the construction of parallel programs do not always make the best use of the parallelism inherent in an algorithm or the parallelism provided by a machine. What is more, some of the widely used techniques are at odds with some of the established cornerstones of good programming practice. Great advances have been made in areas of software engineering such as high level languages, the information structuring techniques

found in object-oriented programming, and the elegance of declarative programming. Unfortunately, in many cases practical parallel programming has failed to keep pace with these innovations and remains more or less unchanged since its inception. This has happened partly because many of the applications to which parallelism has been applied are based on existing sequential codes almost exclusively written in Fortran. Since these codes are often very old and were written by non-computer specialists, it is generally impractical to rewrite them, and thus parallelisation code must be added in a way which changes the original Fortran source code as little as possible. This has had the repercussions that any poor structure or bad programming in the original program is retained in the parallel version, and the growth of new parallel programming languages is being stifled as often people see no necessity to change over to using them.

In contrast many interesting research concepts for parallel languages and constructs have been entertained and much work has been carried out discovering their strengths and weaknesses. Many of the problems that plagued early parallel programming, such as maintaining mutual exclusion when working with shared data, have been countered with mechanisms such as monitors or more radically by message passing techniques. However, parallel programming still seems to be as difficult as ever it was and for many practical programming jobs the older and more basic techniques are still used. For example, when dealing with shared data variants of locking are the most commonly used synchronisation mechanisms because of their expressive power and simplicity. But by the same token, they are also one of the least structured synchronisation mechanisms and therefore the most open to abuse.

### 1.2.1 The Roots of the Problems of Exploiting Parallelism

There are some links between the largely separate evolutions of computer hardware and programming languages. This is not very surprising since each component needs the other in order for them both to offer an effective computing platform. There is little point, other than for research, in having a programming language that cannot be *effectively* executed, or a machine that cannot be effectively programmed. In general, when an important advance has been made in one field, this had led to a corresponding advance being made in the other. For example, the advent of vector processors led to the development of vectorising compilers for Fortran and the creation of vector programming languages such as

Actus [Perr79], and the invention of LISP led to the construction of symbolic processors for its effective execution, albeit some years later.

The recent emergence of general purpose multiprocessors has been driven by the availability of the hardware necessary for their construction. As mentioned earlier, the VLSI revolution that yielded faster smaller components, spurred supercomputer designers, and also assisted in conventional microprocessor design. Over the last five years, refinements in manufacturing techniques have led to order of magnitude advances being made in microchip performance and to large quantities of relatively cheap, high performance processor and memory chips becoming available. With such excellent building blocks, computer designers have once again turned to multiprocessor architectures to form the next generation of general purpose computers.

The concept of multiprocessing is a straightforward extension to the von Neumann model. Instead of a single processor executing the instructions held in a memory, multiple processors are employed executing multiple instructions, possibly held in different memories. Such a concept for greater system performance is appealing and intuitive but therein lies the problem. Although it is possible to visualise how such a processing systems works, it is relatively difficult to write programs to take full advantage of the parallelism offered by such systems. For instance, on a parallel machine with N processors each with a sequential performance of K MIPS (Millions of Instructions Per Second), the theoretical peak performance of a single job is NK MIPS. With a small number of processors (1 to 3) this peak can often be achieved but for larger numbers (10 to 20) it may take more intricate programming, and for massive parallelism (100 to 200+) only a few specialised algorithms can achieve peak performance. Thus, while it may be straightforward and economical to assemble vast collections of processors and memories, it is no easy task to take an arbitrary application and implement a parallel program which runs on such a multiprocessor, and achieves even a decent fraction of the machine's theoretical peak performance. Broadly speaking the roots of the problems involved with parallel computation are connected with the distribution of a program and its data over a number of processors and coordination of the resulting separate threads of control.

### 1.2.2 Bridging the Gap

The task of exploiting the parallelism offered by multiprocessors can be described by three major characteristics: (i) the distribution of a computation over

a number of processors (and memories); (ii) the synchronisation of the communications between the separate parts of the computation; (iii) and the form that the communications should take. Thus, the way to exploit parallelism is to derive a series of programming abstractions that are elegant, efficient and easy to use, to govern the factors that underlie the three characteristics.

The main ideas behind parallel programming have been developed over thirty years from stimuli originating from both hardware and software sources. Many classic problems in concurrency, such as the dining philosophers problem [Holt83], originated from the solutions to real problems found in timesharing operating systems. A major factor in shaping the style of operation of a parallel programming language has been the hardware and software environment in which it was to be used. Essentially, in common with other pragmatic themes, the designers of parallel languages have had to make tradeoffs between sometimes conflicting concerns. For example, the most common tradeoff is between speed of operation and the degree of program abstraction. Simple but primitive mechanisms, such as semaphores, have the attribute of fast operation. However, because they have little inherent structure associated with them, their misuse can lead to unstructured and error-bound programs. Conversely, well formed abstraction mechanisms, such as monitors, can lead to clear and concise programs, though sometimes at an undesirable penalty in performance. Thus, the challenge set to parallel language designers is to devise programming mechanisms that are convenient for the programmer to work with and to some degree portable, but at the same time, are efficiently matched to their underlying target parallel architectures.

In a similar vein, in order to retain efficiency, programmers should employ algorithms that are well matched to their implementation language. If certain operations were considered to be efficient in a particular language and others less so, then it would seem sensible in the pursuit of high performance to maximise the use of the faster operations, even if changes had to be made to the original algorithm. Unfortunately, the real world is not always so accommodating and it may happen that no algorithmic changes can be made. Thus, this would tend to imply that in order to accommodate different emphases there must be a variety of different parallel programming constructs. Hence, what is required to redress the disparity in capability between the current parallel hardware and programming languages, is a range of well defined parallel programming abstractions that

allow programmers to use parallelism constructs that have the appropriate structure for their applications.

## 1.3 Areas for Research

The hardware basis for this thesis is the shared memory multiprocessor and the problems addressed are primarily software concerns, yet having said this, there is still a wide choice of the particular area for research. There have been many attempts at producing parallelism-exploiting programming languages, but for several reasons (discussed fully in chapters two and three) no single language or family of languages have emerged into general use. One of the reasons for this is that the job of effectively exploiting parallelism cannot solely be addressed by a programming language alone, but instead, must be considered in terms of a complete programming environment. Given a bare multiprocessor, five basic areas must be considered: program design, programming languages, operating system software, program monitoring tools, and architectural extensions.

For this thesis the most important of the five areas is the parallel programming language. A programming language assumes the role of a boundary between users and the system; it defines the base abstractions out of which programs are constructed by guaranteeing the properties of its programming constructs, thereby setting requirements for the operating system and hardware to fulfil. For instance, if a portable parallel language allows the existence of shared global variables, each valid implementation of the language must preserve the shared variable semantics irrespective of whether physically shared memory is available on a particular machine. In the past, most parallelism constructs have been designed bottom-up by starting with features that the hardware provides directly and evolving constructs that are easily implementable into programming languages. This style of development has led to efficient but non-portable programs. Therefore, it is the responsibility of the language designer to provide constructs geared more towards the needs of the programmer than to the system, while at the same time not making unreasonable implementation requests of the hardware and operating system. So, once a hardware platform and a style of programming language have been fixed is it feasible to design a run time system and set of tools to comfortably support parallel programming. Thus, while this thesis concentrates mainly on parallel programming language issues, a brief outline of the alternative research areas of the environment follows.

Traditionally, program design has been a paper and pencil exercise, but increasingly through the advent of software design tools programmers are turning towards more automated design methods. It is now possible to assemble elaborate parallel programs via the use of graphically based tools that permit the visualisation of the structure of an emerging parallel program [Dong86]. Another prominent issue in program design is that parallel programming is not simply founded on an expressive language, but in addition, programmers should design parallel programs to work to standard models or paradigms. Using a well understood model as the basis for a program's design often means that the debugging and optimisation of the program can be done very quickly as the programmer is already familiar with the kinds of problems that can arise. Furthermore, certain algorithm design styles can be evolved so that they implicitly take advantage of the underlying hardware. An example of this can be seen in the design of the LAPACK numerical library [Demm87]. This library provides parallelised linear algebra routines which have been purposely designed to work at the highest level of parallelism possible, so minimising the time overheads associated with moving massive amounts of data between processors and memories. Parallel program design can benefit enormously from tools and standard techniques because they help to manage the increased complexity that can result from parallel programs, but they are only an aid to programming and cannot solve some of the fundamental problems associated with parallel programming.

In a timesharing environment (e.g. the Unix operating system), the operating system must play the role of resource manager and share finite hardware resources amongst a number of users and provide programming interfaces (system calls) to enable programmers to access these resources. In a timesharing parallel programming environment the job of the resource manager is made harder as special care must be taken when scheduling groups of cooperating processes. This is because there can be hidden user-defined synchronisation dependencies between concurrently executing processes that could lead to poorer than expected performance from the processes if the operating system blindly rescheduled them. For instance, if a process that held a crucial lock was suspended by the operating system, all the processes that wanted to acquire the lock could waste their timeslices by waiting for the lock to be relinquished. Clearly, what is needed is an operating system which offers attractive (quick and efficient) parallel system calls to programmers, such as Mach [Acce86], allowing some central control to be applied to ensure that user-level and system-level synchronisation policies do not compete against one another. In addition, other facilities such as secure memory

management, workload partitioning, error recovery, and remote distribution of processes could be provided to make the implementation of parallel languages easier.

Even though a program may have been designed to a standard pattern, coded in an elegant language and executed by an advanced run time system, programming errors can of course still occur. An error in a parallel program can either give rise to an incorrect result, or it can cause the program to execute more slowly than anticipated (a performance error). The primary tools of a programmer in tracing and fixing these errors have been debuggers and program profilers. If parallel languages and operating systems are developed, then programming tools must also be developed commensurably or the job of examining the behaviour of parallel programs will become intolerable. A programmer should be able to monitor a program in the abstract forms found in its design, taking into account the sharing and partitioning of data, as this kind of approach is the natural continuation of debugging techniques that have evolved from *absolute* to *source level* debugging.

Finally, during the course of the development of computer architectures, it has been noted that in some cases certain key operating system functions, such as memory management, are best delegated to the hardware for efficiency reasons. This idea can be extended so that certain critical constructs of programming languages also have direct hardware support. For example, consider the atomic *test-and-set* operation, and the research currently being undertaken to produce VLSI chips to support the Linda parallel programming language [Ahuj86]. Naturally, due care must be taken to identify the exact functions to be locked into the hardware. Experience has shown that this approach can have its pitfalls, as in the case of the ill-fated Ada based Intel 432 [Orga82]. Further to this, the RISC school of microprocessor design aims towards producing processors that are operated by a minimum of instructions by shifting as much complexity as possible into the compiler. Clearly, research into architectural extensions must be founded on a wealth of practical knowledge about the operational behaviour of parallel programs on shared memory multiprocessors which implies, in turn, that research of this kind is an ongoing and long term process. Thus, since the demands of parallel programming are not thought to be well enough understood at this time, hardware extensions are paid scant attention in the bulk of this thesis.

## 1.4 A Portent of the Chapters

This thesis is organised into seven chapters and explores the implementation and exploitation of parallelism. It focuses latterly on shared memory multiprocessors but continues with the second chapter which is a broad survey of parallelism issues, intended to illuminate the reader regarding the diversity and depth of research into parallel processing. The chapter is divided into three sections dealing in turn with parallelism in computer architectures, computer operating systems and system software, and parallel programming languages. As a substantial amount of research has been undertaken to investigate parallel processing, it is unrealistic to compile a definitive survey of parallel processing here. Nevertheless, key issues can be identified and hence the material that is presented sets out the main hardware and software options faced by someone who wishes to exploit parallel processing. The third chapter puts the material presented in the survey into perspective by identifying the important common characteristics found in parallel activities. As parallel architectures and their indigenous styles of programming appear at first sight to differ widely, it is somewhat rewarding to find a large degree of commonality between the competing approaches. In addition, the methods of designing parallel algorithms are discussed with a view to showing how such parallel algorithms can best be supported by programming language constructs and parallel architectures. In the final part of the chapter, a basis (or framework) is developed that can be used in the design of new parallel programming concepts.

The fourth chapter is a presentation of new work relating to the exploitation of parallelism. A parallel programming model is developed that uses a form of single assignment variable, called a *shared value*, to specify and synchronise information exchange between parallel threads of control. The aim of designing the model is to abstract away the essence of parallel programming from a single language or architecture so that it can be applied across a range of languages. Drawing on previous work and paying attention to the framework from chapter three, shared values are discussed and their capabilities and limitations outlined. During this discussion, two similar parallel programming languages that employ shared values are used to illustrate short example programs. As this research is aimed at solving a real programming problem, chapter five is a short overview of an implementation of a programming language that supports the use of shared values. Information is presented on the process of compiling source programs and how shared values can be efficiently implemented on shared memory multiprocessors. The testing and evaluation of several parallel programs written

using shared value constructs is discussed in chapter six. The evaluation is primarily achieved by comparing shared value-based programs with their counterparts written in current parallel programming languages. Some comments are made on the implementation of shared values and how these affect the results observed. Finally chapter seven concludes the thesis by summarising its goals and attainments. The latest work on shared values is presented, which reflects the current thinking and new directions that could be explored. In addition, some brief comments are made on the overall success of this work and how the shared value approach to parallelism compares to other work on parallelism that has emerged over the last four years.

# *Chapter 2*

# The Nature of Parallelism

The task of compiling a complete survey of parallelism research is only marginally less difficult than trying to net all the fish in the North Sea with a single cast. The notion of parallel activity is present throughout many aspects of computing science, ranging from software examples such as a parallelised quicksort algorithm, to hardware examples such as a central processor operating concurrently with its peripheral devices. Over the last five years parallel processing has grown into a mature research area encompassing a wide range of topics so this survey chapter has been divided into three sections:

- parallelism creation by hardware,
- parallelism management by operating systems and tools,
- parallelism exploitation by programming models and constructs.

The hardware section of this chapter describes some of the notable parallel architectures, yielding interesting material to allow the successes of the different approaches to be compared. Moreover, information is presented about the kinds of applications that classes of parallel architectures are aimed at supporting and the corresponding operational tradeoffs that are made. The section on operating systems describes the importance of the operating system to the exploitation of parallelism, resulting from its job as a process and resource manager; and the role of software tools in development of programs. Software tools have also been included in this section because they can be viewed as part of the system software. One reason for this choice is that the architectural abstraction portrayed by an operating system should be fully integrated into the software tools it supports, so that the tools can relay pertinent system information to users in a convenient format. The section on parallel languages concludes this chapter by listing existing programming constructs that allow the exploitation of parallelism. This is the most important section of the chapter, but even so, only the main concepts of parallel programming are presented as there are too many parallel languages to discuss each one individually. Issues arising from these three sections is discussed further in the third chapter and is used to form the requirements for parallel processing mechanisms.

## 2.1 Parallel Architectures

The central hardware theme of this thesis is shared memory multiprocessors, but inspiration can be drawn from the examination of alternative multiprocessor architectures. There have been many proposals for the design of computers based on parallel architectures. Some architectures have included parallelism for reliability by replicating important components; such as is typically found in fault tolerant systems (e.g. Tandem NonStop™ [Katz78]). However, the primary reason for the use of parallelism in a computer design is to obtain an increased performance over a sequential design for a given hardware technology.

As the reliability and complexity of the constituent components of computers have increased and their size decreased, it is now possible to put complete functional units on a single VLSI chip. Hence, it is now possible to construct parallel architectures that contain hundreds or even thousands of processors. Parallel computer architectures based on arrays of dedicated processors are direct offshoots from VLSI technology and have limited general use [Mead80], but others such as the dataflow architectures highlight important concepts of general applicability. As computer architectures have developed, many classification systems have been devised to try and capture the most important characteristics of each machine [Feng72, Shor73, Hünd77, Hock81, Skil88]. In this chapter only the general issues are presented in a simple categorisation to give a flavour of parallel architecture research, with there being no attempt at a formal analysis.

### 2.1.1 Flynn's Categorisation

The basic classification scheme introduced by Flynn for computer architectures [Flyn66] groups together architectures with similar modes of operation. The scheme does not go into specific details regarding the hardware breakdown of individual architectures, but nevertheless, it has proven useful as a method of identifying architectures with related operating characteristics. Other more complex categorisations and extensions to the basic scheme [Kuck78, Dunc90] have been proposed to capture more quantative information, though these are not discussed here.

Flynn proposes that all computer architectures can be classified as one of four types according to the way in which their processing elements handle instructions and data.

- The single-instruction stream single-data stream (SISD) representing conventional scalar processing systems.
- The single-instruction stream multiple-data stream (SIMD) including most array processors such as Solomon and Illiac IV.
- Multiple-instruction stream single-data stream organisations (MISD),
- Multiple-instruction stream multiple-data stream (MIMD) including multiprocessor organisations.

The simplicity of the categorisation is its main advantage, but it is flawed in so much that no widely recognised architecture can be fitted into the MISD category.

### 2.1.2 SISD - Conventional Processors

One of the earliest views of a digital computer architecture came from John von Neumann [vonN58]. He proposed an effective computing system model that is still in use today. His model consists of a *store* and a *processing element* that perpetually perform a *fetch-execute* cycle, see Figure 2.1. The processing element selects and fetches an instruction from a sequence of instructions held in the store. The element then decodes the instruction and finally executes it. With the store being used as an updatable memory to hold the instructions and the partial results of a computation, it can be said to hold the *state* of the computation. This follows from the *side-effect* nature of the model which means that the action of an assignment has the side-effect of modifying the data object bound to a global variable. It is this notion of state and the ordering of events necessary to achieve a particular state that is the cause of many difficulties when considering parallel programming. For example, the idea of a global variable that can be modified by many parts of a program is an efficient mechanism for intra-program communication, but the unconstrained use of global variables can produce unstructured programs that are very hard to understand.

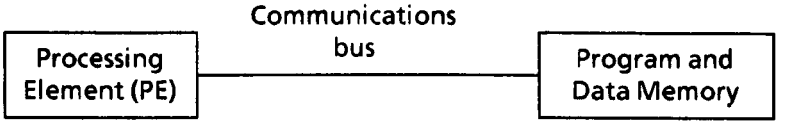



Figure 2.1 - von Neumann model.

As has been mentioned in the first chapter, parallelism has been introduced into von Neumann systems in many ways most of which are invisible to an

applications programmer. Dedicated processors have been attached to the bus to handle the execution of instructions to interface to external media such as disks and networks, and to execute instructions involving floating point numbers. Memory interleaving allows several memory requests to be active simultaneously permitting the overlapping (pipelining) of several fetch-execute cycles. A pipelined cycle can consist of: instruction fetch, decode, operand fetch, arithmetic execution and result storage. Arithmetic and logical operations inside the central processor have also been parallelised by using techniques found in devices such as carry-lookahead and carry-save adders.

Nearly all scalar processors are descended from the von Neumann model and this hardware trend has to some extent been mirrored by traditional programming languages. A large proportion of programming languages are based on control flow models that utilise active agents to execute a stream of instructions which act upon passive memory locations. For multiprocessors the picture is the same; many multiprocessors extend the von Neumann model simply by adding extra processors and memories, and parallel programming languages likewise by supporting multiple threads of control with additional scoping rules for multiple memories. Thus, it can be seen that the von Neumann model has had a profound effect on both sequential and parallel computer systems. Even so, it is not the only model that can be used as the basis for parallel programming nor is it the best, as shall be seen later.

### 2.1.3 SIMD

There are three basic types of SIMD processor, that is, processing systems characterised by the synchronous application of a master instruction over a number of related operands. The different types are: array processors, vector and pipelined processors, and associative processors.

#### Array Processors

An array processor consists of one control unit and a number of directly connected processing elements. Each processing element is independent, having its own registers and storage, but only operates on command from the control unit. Three examples of actual array processors are Solomon [Greg63], Illiac IV [Barn68] and the DAP [Hock81].

## Vector and Pipelined Processors

Although the two types of machine in this category are completely different it is very common to find machines that combine both of these parallel processing techniques for maximum efficiency.

By using vector processing a machine operation can take vectors as operands and can return a result that is also a vector, as opposed to scalar processing in which only scalar operands can be manipulated. Vector operations cut out the need for the separate operand fetches and instruction fetches/decodes that would be needed to operate on each element of a vector if similar operations were to be performed by a scalar processor. Vector processors can be grouped into two classes according to the way in which vectors can be referenced. In the first group, *memory-to-memory*, source, intermediate and result vectors can be directly retrieved from memory (e.g. TI-ASC). In the second group, *register-to-register*, operands have to be accessed indirectly through special vector registers (e.g. Cray-1, Fujitsu VP-200). As vector operations can take many machine cycles to set up and many more to execute, several vector operations can be in progress simultaneously by the use of pipelining.

Although pipelining techniques do not fall under the strict definition of SIMD they are presented here because they are often matched with vector multiprocessors. Pipelining has been incorporated into three areas of computer operation and can be classified as one of the following: arithmetic, instruction or processor. Arithmetic pipelining is found in the computational units of machines such as the Star-100, Cray-1 and Cyber-205 and is concerned with speeding up the execution of arithmetic and logical operations. Instruction pipelining, as mentioned before, is very common and is found on most machines including uniprocessors. Processor pipelining, as it name suggests, operates at the much larger scale of inter-processor parallelism.

A pipelined arithmetic unit is a time-multiplexed version of the array processor with a number of functional units which can either be tailored to a particular operation (e.g. Cray-1) or can perform different operations, in some cases simultaneously (e.g. TI-ASC). These units are arranged in a production line fashion, staged to accept a pair of operands every $\Delta t$ time units. The memory in a pipelined system is arranged to facilitate high-speed data transfer to produce the source operands which are entered a pair every $\Delta t$ time units into the designated pipeline. The resulting stream of operands can then be returned to memory or fed into another pipeline, a procedure called *chaining*. If sufficiently large vectors are

entered into a pipeline, the speed up over a scalar processor is approximately equal to the number of pipeline stages. The total execution time is longer for a pipelined processor than that for a completely parallel processor, but fewer hardware components are needed. However, as the length of a pipeline grows, its set up and flushing times increase proportionally, which means high overheads are incurred when operating with small vectors.

Vector pipelined machines have proved to be very effective at floating point computation, but only rarely do these machine achieve a high percentage of their theoretical peak performances. Two of the reasons for this are that often the quantities involved in real applications do not map well into the fixed vector sizes of a machine (e.g. a problem of size 65 is a bad match for a vector pipeline of size 64), and the task of moving data between registers, memories and secondary storage becomes so complex and congested that valuable processing time is wasted waiting for data to be shunted around.

### Associative Processors

The characteristic of associative processors is that the processing elements are not directly addressed by location or sequence, but rather by content. Processing elements are activated when a generalised match relation is satisfied between an input register and the characteristic data contained in each of the processing elements. Only the elements that match perform the control unit's operation during which time the other elements remain idle. Associative processors have been constructed since the 1950's but technological limitations meant that early machines were not cost effective [Slad56, Meil81], though newer machines have been more successful (e.g. STARAN, MAP, and MPP [Batc80]).

### Analysis of the SIMD approach to Parallelism

An obvious comment that can be made about SIMD machines is that they sacrifice some generality to other parallel processing models, and by implication some performance, when executing certain problems because of the master-slave configuration of the processing elements. Nevertheless, SIMD machines such as the Cray processors, GF-11 and Illiac IV have been used very successfully for intensive floating point applications and machines such as the Connection Machine [Hill85] likewise for symbolic applications. From the hardware perspective, SIMD machines only require a single complex (expensive) master processor making do with simpler processors as the slaves, while MIMD machines generally replicate all processors evenly. Operationally, Beetem [Beet87] argues

that SIMD machines are more effective at multiprocessing than MIMD machines because of the synchrony of the processors which obviates the need for explicit additional synchronisation. In a similar vein, machine and program designs are simpler as the rules for sharing data are very rigid: either all processors get the data or none of the processors get the data. These sharing rules are qualified though, as many SIMD machines have masking instructions to allow a processor to ignore data that is not intended for it. As a consequence of using regular sharing rules, however, resulting programs have structures that can be easily debugged because of the uniformity of processing found in their algorithms. In addition, each processor needs only a relatively small private memory to hold data as program instructions are held in a single program memory. This compares favourably against conventional distributed memory architectures who must replicate program instructions for each processor, but is about the same for conventional shared memory machines who can similarly share program code.

A negative point, however, is that the optimal allocation of work when mapping a large problem onto a fixed size array is known to be NP-complete [Nico88] so even potentially highly parallel algorithms may not perform well on SIMD processors. To summarise, SIMD machines offer a viable parallel processing architecture both economically and operationally, but the choice of a particular architecture must be made with respect to its intended role, as some applications will work very well while others will achieve little better than scalar performance.

### 2.1.4 MIMD - Conventional Multiprocessors

Two very different types of computer system fall under the MIMD classification. The first is a multicomputer system defined by Hwang and Briggs [Hwan85] as:

> a *multiple computer system* consists of several autonomous computers which may or may not communicate with each other;

while the second type, a multiprocessor system, is defined as:

> a *multiprocessor system* is controlled by one operating system, providing interaction between processors at the process, data set and data element levels.

Parallel processing can be attempted on multicomputer systems but multiprocessors generally accumulate much smaller overheads when handling

the parallelism, boasting superior software and run time support permitting the construction and execution of highly efficient parallel programs.

A MIMD multiprocessor architecture is composed out of a group of processors, some memory and a connection layout to allow the processors and the memory to communicate. The processor configuration differs from the SIMD model in that all of the processors are nominally of equal status and capability, not master-slave or add-on processors, enabling each processor to operate asynchronously and independently. Furthermore, if a MIMD multiprocessor is composed of identical processors, then processing can be performed *symmetrically*, in that computations are not tied to an individual processor and can migrate between processors as desired. In contrast though, the types of processors in a MIMD multiprocessor can be varied, mixing general purpose processors with those that have specialised capabilities.

There are two schools of multiprocessor design: one view is that the processors share a single memory and use it for mutual communication, whereas the other view is that there need be no shared memory and communication takes place by means of message passing. Machines that use shared memory are often described as being *tightly coupled* with there being many interactions between its processors, while conversely, distributed memory machines are *loosely coupled* with there being fewer interactions because of the higher communication cost. In addition, Arvind and Ianucci [Arvi83] have pointed out that architectures that provide a shared memory abstraction must: (i) make provision to tolerate long latencies for memory requests; and (ii) achieve unconstrained, yet synchronised, access to shared data.

To address the former problem many solutions have been tried out including: arranging for small uniform memory access times, using special task switching hardware to suspend tasks until their data arrives, or more fundamentally, by using a data driven model (i.e. dataflow). To address the latter problem there have been several solutions including, once again, special hardware at both the instruction and memory location levels (e.g. in the HEP), and sharing and synchronisation techniques based on coherent caches. Thus, maintaining a shared memory abstraction is a severe test for an architecture, though, if shared memory is given up in favour of a distributed memory model, new problems arise at the programming level regarding the partitioning and transference of data between memories. As each approach to memory addressing has some merits, but also some

drawbacks, some machines have been designed combining both shared and distributed memory.

### Shared Memory Multiprocessors

A shared memory multiprocessor architecture is characterised by having a single, physical global memory which is directly accessible to all its processors, i.e. a single shared address space. In addition, the key idea behind true shared memory systems is that the access time to a piece of data is independent of the processor making the request, that is, there is a uniform access time for fetching and storing memory elements. With this being the case, the actual interconnection scheme between memory and processors can be completely hidden from application software. However, because of the complete sharing of all memory, these systems suffer most from effects such as memory bank and bus contention. Hence, the limiting factor on the maximum number of processors the system can usefully accommodate is the aggregate memory bandwidth. In order to minimise memory contention, processors often have large cache memories to reduce the number of memory requests. If suitable caching strategies are employed then the shared memory architecture works very well in practice allowing the current generation of multiprocessors to utilise up to thirty processors. Interestingly enough, however, the use of cache memories makes shared memory machines seem more like distributed memory machines, as cache accesses are much faster than memory references, but no explicit programming of the cache can be done. Further information on caching techniques is presented in chapter five.

As each processor must be able to reference any memory location only connection layouts that allow full interconnection of processors and memories can be used. Conceptually the simplest way of achieving a full system interconnection is to use a single shared bus. This has the advantages of being relatively cheap and easy to configure but set against these is the drawback that the bandwidth of the bus restricts the system to a small number of processors when compared to distributed memory machines; with the bus being a potential single point of failure. A modest extension to the single bus scheme is to employ two buses, one for reading the other for writing. This may seem attractive but in practice not much is gained as often memory requests need to use both buses, with multiport memory being required to support the multiple buses. Using multiport memory means that multiple control, switching and arbitration logic is needed for each memory unit to synchronise the accesses from separate buses. This makes

multiport memory a most expensive and limited commodity (due to physical space requirements) but it has been implemented on several older machines including the Univac 1100/90 and the IBM System 370/168. A more radical method of increasing the bandwidth, however, is to make a separate bus available from each processor to each memory unit, an arrangement called a *crossbar* as found in the Alliant FX/8. This method gives the best bandwidth and reliability but can still suffer from contention if multiple requests are received at a memory unit simultaneously. Other disadvantages with the crossbar switch are the high complexity of the interconnection components and the associated high cost. Sample interconnection schemes are shown in Figure 2.2. Shared memory has been used in the construction of many multiprocessors ranging from supercomputers such as the Cray X-MP [Lars84] and the IBM 3090 [IBM85] to more economical systems like the Encore Multimax [Enco87] and the Alliant FX/8 [Alli86, Perr86].

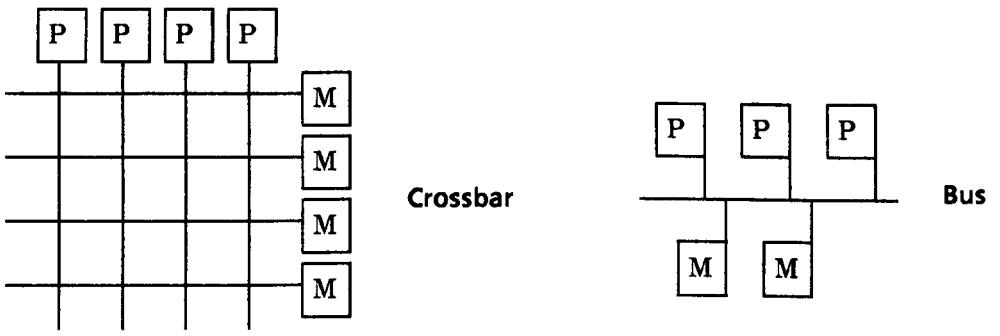



Figure 2.2 - Shared memory architectures.

## Distributed Memory Multiprocessors

A distributed memory system is configured so that each processor has its own local memory, forming a node, with there being no globally accessible memory. Thus, the only way in which data can be shared between a group of nodes is to move it explicitly between them, a technique which is often called message passing. The time taken for the movement of data is dependent on the node connection layout, from which a measure of distance between nodes can be derived. As the connection layout may be visible to application programmers this distance factor is of importance in algorithm design when considering the usage of data relative to its position. A large number of connection schemes have been used in distributed memory multiprocessors [Feng81], ranging in complexity from a linear array with each node communicating with adjacent nodes, to a hypercube,

in which there are $n$ connections to each of a node's $n$ nearest neighbours [Ratt85, Seit85]; three connection schemes are shown in Figure 2.3.

Distributed memory machines have the advantages over shared memory machines that they are easier to built, requiring only simple point-to-point communication buses, and they are more extensible as there need be no complete interconnection of nodes and therefore no shared bottlenecks. However, with the restriction of private memories, programming can become more difficult as consideration must be given to the efficient distribution of program and data to minimise inter-node communication costs. Two well known message passing multiprocessors are the Intel iPSC [Gehr88] and the transputer-based Meiko Computing Surface.

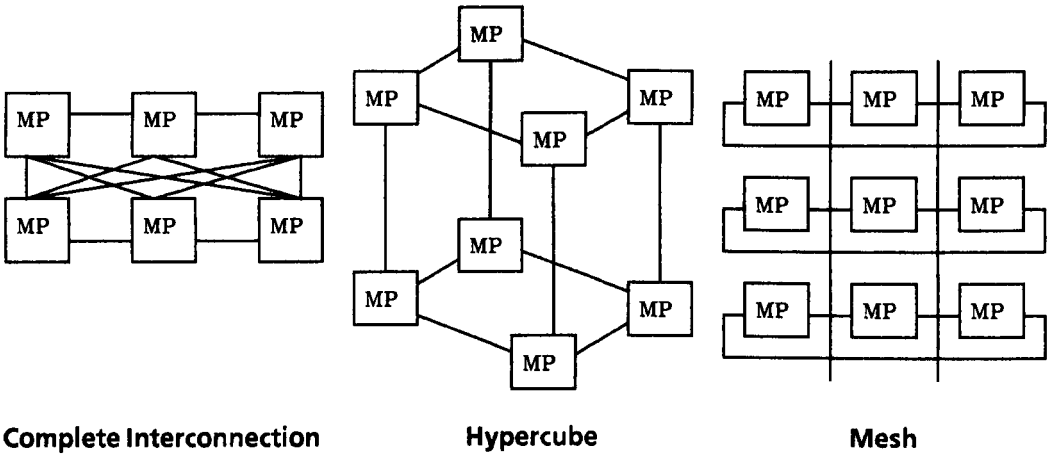



Figure 2.3 - Distributed memory architectures.

## Hybrid Multiprocessors

Hybrid systems possess attributes from both shared memory and message passing systems. Such a system may be organised so that all the memory is divided into local sections for each processor, with the operating system or special routing hardware simulating a single shared global memory. Memories and processors are interconnected by a series of multistage switches that must allow, through some sequence of stages, every processor access to every memory. Switch structures need not be of regular design, see Figure 2.4, as long as a switch structure can claim complete processor to memory interconnection. From the programmer's viewpoint, programs for hybrid systems are coded like shared memory systems though the actual performance considerations resemble those of message passing systems (i.e. a shared memory abstraction with non-uniform

access times); but in most cases the penalties for poor data distribution are considerably less. This deception is permissible for most applications, however, the programmer must be aware of the details of the hardware layout in order to produce efficient code. As the illusion of shared memory has to be maintained, the limiting factor on the number of processors in a system is the aggregate communication speed. This limitation, however, is not as extreme as with shared memory systems because, in the case of the BBN Butterfly, for instance, adding extra processors incorporates the addition of extra memory bandwidth [Crow85]. Other examples of machines of this type are the IBM RP3 [Pfis85], NYU Ultracomputer [Gott83], C.mmp [Wulf72], Cedar [Gajs84] and the Denelcor Hep [Smit81] whose special support for parallel programming is mentioned in the third section of this chapter.

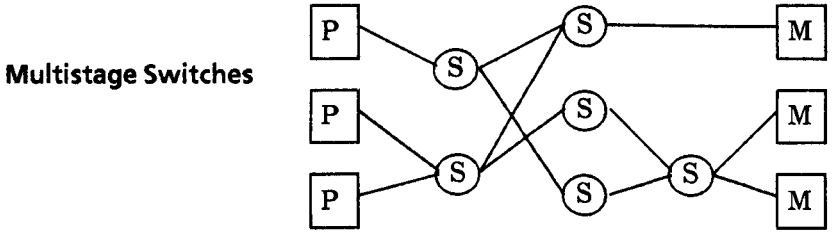



Figure 2.4 - Switch-interconnected shared memory architecture.

### Analysis of the MIMD approach to Parallelism

In general, it is easier to build yet harder to program private memory machines as opposed to shared memory machines of similar processing potential. This is because in shared memory machines, engineers do the hard work of designing and building memory arbitration and coherency protocols so that programmers can work with a single shared memory. In turn, this means that programmers do not have to be concerned with the precise location of programs and their data, in order to produce efficient or even working programs. Conversely, private memory machines can have greater total memory, CPU to memory bandwidth, and number of processors as they are freed from the task of maintaining a true shared memory abstraction. Similarly, private memory machines built on hierarchical structures such as trees are scalable so that arbitrarily large machines can be constructed. Hybrid multiprocessors offer a good compromise between the two extremes, but currently they have been developed largely as research machines, with only a few machines making inroads into the commercial marketplace.

In comparison to SIMD machines, MIMD multiprocessors are used for mainstream computing tasks as well as more specialised parallel applications. This results from the MIMD model being perceived as being a closer match to the SISD model than SIMD. Applications have been easily ported to MIMD machines from SISD giving MIMD machines a good general appeal as opposed to SIMD machines which are largely viewed as special purpose devices. To redress this balance somewhat, certain machines combine both SIMD and MIMD approaches (e.g. Cray Y-MP), retaining generality, and offering great computational power.

### 2.1.5 Exotic Architectures

While it may be possible to place every architecture into a category in Flynn's classification scheme it is more productive to split off a number of architectures and look at them separately. Although there are many architectures that could be mentioned at this point only four models are discussed here: VLIW, systolic arrays, the dataflow model, and graph reduction, all of which are radical departures from the von Neumann approach.

#### VLIW Architectures

Machines of this type are highly parallel architectures that offer an alternative to more conventional multi and vector processors. Whilst resembling ordinary multiprocessors VLIW's (Very Long Instruction Word) utilise a tightly coupled, single flow of control mechanism that can be thought of as being similar to the operation of horizontal microcode. The pioneering VLIW architecture, the ELI-512, consisted of 16 32-bit RISC microprocessors with access to a combination of local and global memory. Programs specified very fine grained control over virtually every resource of the machine with each long word instruction containing operation fields to control all of the individual processors. Sequencing was performed by a single flow of control as found in SIMD architectures. However, each processor could perform a different operation from its neighbours, unlike vector instructions, with all processor communication being controlled by the instructions. As a practical point there was no central program store, rather each processor fetched the relevant portion of the instruction word from its own program store. Figure 2.5 shows the outline of a VLIW architecture.

The intended method of programming a VLIW is by using a *trace scheduling* compiler. Trace scheduling is a process for predicting the control flow in programs so that very long instructions can be generated from sequential source code. If this process can be done successfully then effective VLIW machine code can be

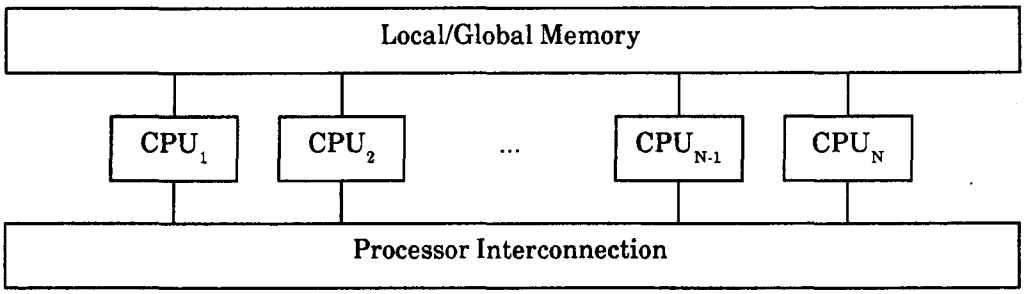



Figure 2.5 - VLIW architecture.

constructed, however, if not (due to complex control flow in the code) then some potential parallelism may be lost. Only one company (Multiflow) has exploited VLIW hardware technology, but the novel approach of trace scheduling for instruction execution has had some influence on the design of optimising compilers.

### Systolic Arrays

Due mainly to the advances made in VLSI technology it is now cost effective to build arrays of dedicated processors to perform specialist functions. One such type of processing array is the systolic array described by Kung and Leiserson [Kung78]. A systolic array is a regular set of interconnected cells each capable of performing some simple fixed operation upon data which flows through the array at regular beats. This kind of processing is similar to arithmetic pipelining but far more complex functions can be computed by systolic arrays than by simple arithmetic pipelines. Array configurations that have been suggested include the one dimensional linear array, the two dimensional square array, the triangular and hexagonal arrays, and binary trees. Applications for these arrays have included matrix addition, matrix multiplication, equation solving, and searching.

Other designs that have been proposed for processing arrays include some for reconfigurable arrays (e.g. CHiP [Snyd82]). Such an array consists of three components: processors, a switch lattice and an array controller. The ability to reconfigure an array greatly expands the range of functions it can compute. Arrays of processors can often compute functions in the minimum time-complexity (e.g. matrix multiplication in $O(n^2)$ rather than $O(n^3)$) but in general they are perceived as being difficult to design as no widely recognised language is used to specify their layout. In addition, arrays of processors suffer from mapping

difficulties similar to SIMD machines when mapping large problem sizes onto smaller fixed array sizes.

### Dataflow Architectures

Dataflow was first proposed by Dennis [Denn79] with a view to providing a technology-independent computing model which could offer better programmability than the von Neumann model in application areas. The principle idea behind dataflow systems is that only the availability of operands should influence the order in which program statements are executed. Conventional machines operate by taking instructions one at a time from a memory and executing them, a process known as control flow. With dataflow, instructions can be executed as soon as their operands become available. This is accomplished by compiling a dataflow program into a dataflow graph where nodes represent instructions and arcs are the program structure (data dependencies). During program execution data tokens flow along the arcs of the graph between nodes. Tokens are held at a node until all the tokens necessary for that statement to execute have arrived. When this happens the node is said to *fire*, consuming the first token from each of the input arcs and emitting data tokens along all of its output arcs. As there is no explicit ordering of the firing of statements, other than the data dependencies, statements can be executed as soon as their data tokens arrive, possibly in parallel.

There are two working designs for dataflow architectures. The first is *static dataflow* due to Dennis where only one token can exist on an arc at a time, there being no method for distinguishing between tokens. Control tokens are used to send acknowledgements to guarantee proper timing when transferring data tokens from node to node. A drawback with static dataflow is that separate iterations of an iteration construct cannot be unfolded if the total number of iterations is not known at compile time, as only one iteration can be in progress at a time due to the restriction that only one data token can exist on an arc at a time. In *dynamic dataflow* [Gurd85, Arvi80] data tokens are tagged enabling multiple tokens to exist on an arc. This allows iterations to be unfolded making this a maximally parallel model. Figure 2.6 shows architectures for both static and dynamic models in which data tokens circulate along the arrows.

Several dataflow machines have been built, including the Manchester Dataflow Machine [Gurd85], but they have run into interesting difficulties. As parallelism is exploited at the smallest level, that is individual instructions, it is possible for many computations to proceed in parallel given a suitably parallel

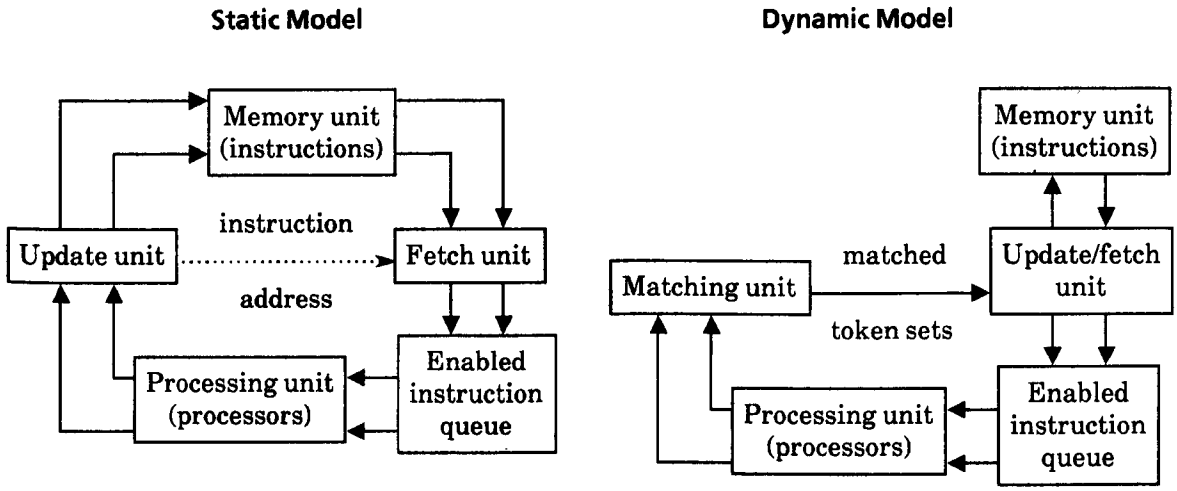



Figure 2.6 - Dataflow architectures.

application. This can be bad if the machine's memory becomes filled with computations waiting for data tokens and no new tokens (to fire the waiting ones) can be accommodated. Hence, the problem is one of *throttling* the computation to control the degree of parallelism and this is now a major research area [Sarg87].

## Graph Reduction Architectures

Graph reduction [Wads71] can be used as an efficient method for implementing functional languages. Executing a program written in a functional language consists of evaluating a series of expressions that have a natural representation as a graph. Evaluation proceeds by means of a sequence of simple steps called reductions that are local transformations of the graph. Due to the non-interfering nature of the reductions, described in the Church-Rosser Theorem [Chur41], reductions may safely take place in a variety of orders. The evaluation terminates when there are no further reducible expressions. Figure 2.7 shows the evaluation of $f\,7$ in the functional language Miranda where $f\,x = (x+2)*(x-5)$.

Graph reduction is an inherently parallel activity as at any moment a graph may contain a number of redexes (reducible expressions) and it is very natural to reduce them simultaneously. On a shared memory multiprocessor the process of graph reduction poses fewer synchronisation problems over a conventional approach. This follows from there being only one shared graph in the reduction model as opposed to shared program text and data structures in the conventional model. Graph reduction is also inherently distributed as a reduction is a local transformation of the graph and no shared bottlenecks, such as an environment,

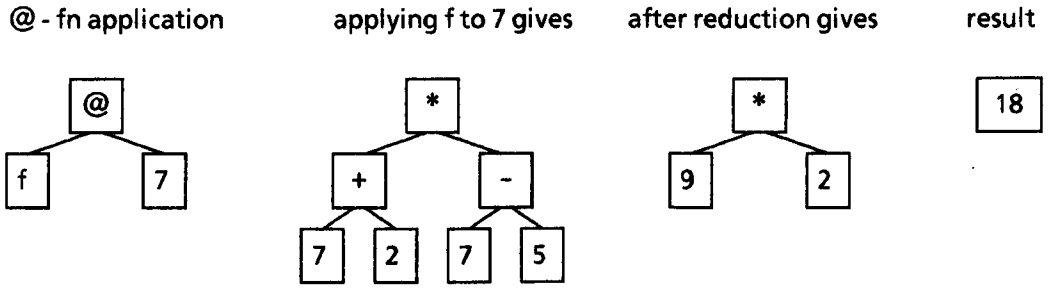



Figure 2.7 - Graph reduction.

need be consulted to perform a reduction. As a usable model, graph reduction introduces no new parallelism constructs and so may be easier to understand than the complex interdependencies found in current parallel programs. Moreover, at all stages of a reduction the computation is well defined, which cannot always be said of imperative style parallelism. (Additionally, the graph is in a form which is more suitable for the analysis of the data dependencies than a conventional program.) However, graph reduction does have some drawbacks, similar to dataflow, in that parallelism is exploited at the level of single instructions which can have the serious side-effect of high memory usage. Some graph reduction architectures are being built such as the Rediflow machine [Kell84] and the Grip Machine [Peyt90], though these are actually implemented on von Neumann machines and are not available commercially.

### 2.1.6 Architecture Summary

Of the many types of parallel architecture the MIMD systems derived from the von Neumann model proliferate, except for the parasitic incursion of vector processors. Many SIMD processors have been constructed but almost by definition are regarded as specialised and deemed to be separate from mainstream computing. The argument of SIMD versus MIMD can be thought of as one which sets many small simple processors against fewer larger more complex processors. In the main, current programming languages have tended to favour the MIMD approach as it has been easier to divide a problem into tens of pieces that synchronise after many hundreds of instructions for MIMD machines rather than the hundreds or thousands of pieces necessary for SIMD machines that synchronise after every instruction.

Shared memory multiprocessors are vying with distributed memory machines to satisfy the demand for parallel processors and it seems that technology will probably have the final say over which will dominate. Plans have been made for

large shared memory machines (around 100 processors), and hybrid machines already exist that contain several hundred processors to compete with the large distributed memory machines. In contrast, advanced operating system techniques and increases in inter-node communication speeds are helping distributed memory machines become more like hybrid multiprocessors. This allows a hitherto unknown degree of portability between distributed memory machines as the interconnection layouts are submerged beneath the operating system. Moreover, with fast communication speeds future distributed memory machines will be able to offer very efficient support for shared address spaces.

Thus, the precise future of computer architectures is unclear but two points of note have emerged:

(i) Future high performance architectures will be multiprocessors so that they can take full advantage of parallel processing. Such machines may consist of a group of asymmetrical processors (perhaps to allow fast floating point or vector operations), or may in fact be symmetrical processors given that fast floating microprocessors can now be manufactured (e.g. Intel i860).

(ii) Many future high performance architectures will support a shared address space either directly, or indirectly via the operating system or special routing hardware (i.e. non-uniform shared memory access time - NUMA).

Multiprocessors are a very attractive (cost effective) and natural method of obtaining high processing performance, and with peripheral benefits such as fault tolerance, seem too important to be ignored. In addition, physically shared memory can be used to integrate microprocessors and tightly coupled SIMD processors, such as vector processors, permitting a powerful combination of SIMD and MIMD parallel processing techniques. Furthermore, the shared memory abstraction is emerging as a more usable programming model than private memory because it avoids many of the problems in distributing applications over hundreds of memories, and some parallel programming languages require it (more information in chapters two and three). Of course, distributed memory machines could be used to simulate shared memory machines in order to support shared memory languages. However, this job is difficult to carry out efficiently for many real applications without some direct programmer intervention.

## 2.2 Operating System Issues and Tools

This section covers two areas of operating system influence on parallel programming: (i) resource management and parallel processing primitives, and (ii) tool support. The first area describes the role of the operating system as a resource manager in a parallel processing environment and outlines the facilities it should offer to application programmers and compiler writers. Software tools have been shown to be useful in the development of programs, all the more so when considering parallel software, and there is a wide variety of tools currently in use. Tools range from specification and design aids, to much needed monitoring and debugging facilities. Auto-parallelising compilers and program restructurers also fall into this category.

Before commencing this discussion on operating systems this is a reasonable juncture to introduce some terminology. Due to the wide range of research into parallelism many terms are used interchangeably and it can sometimes be confusing if terms are used loosely. The fundamental definition of interest here is that which is used to represent control flow.

Definition: A sequential program specifies the sequential execution of a list of statements, the execution of which gives rise to a *thread of control* that is sometimes called a *process*.

There are two ways to implement process concurrency, one uses only one processor, the other many.

Definition: In *multiprogramming*, many processes share the same processor and each receives a share of the time available [Dijk68] - this is termed pseudo concurrency.

Definition: In *multiprocessing (multitasking)*, many processes are distributed over a number of processors with each process executing on its own processor [Jone80] - this is termed real concurrency.

Multiprogramming is used in uniprocessor systems to simulate concurrency by interleaving the execution of processes on a single processor. It is also used by some multiprocessor operating systems to perform load balancing when the number of processes able to be run exceeds the number of available processors, though this too is commonly called multiprocessing. From the definition

multiprocessing also takes place when the number of processes able to be run is less than or equal to the number of available processors.

When a parallel program is analysed in terms of its component processes no assumptions can be made about the rate at which each process executes, except that they notionally make progress toward completion. This is known as the *finite process assumption* and using this assumption the correctness of a program is independent from the number of processors, but not the number of processes, a program in executed on.

One further definition added at this point makes a distinction between parallelism in a single job and parallelism between a group of jobs. Multiprocessing would seem to cover both types of parallelism, but they are fundamentally different at the programming level and at the level of the operating system.

Definition: In *parallel processing*, the constituent processes of an application are distributed over a number of processors with each process actively executing on its own processor.

One way in which parallel processing differs from simple multiprocessing is that in parallel processing, one can talk about a linear speedup with respect to the number of processes being applied to a problem. Clearly this can only happen if all the processes of a job execute on their own processors, and do not have to share processors with other processes of the same or other jobs.

In general, a process is assumed to be an entity similar in functionality to a process found in operating systems that support virtual memory, such as the Unix operating system. This assumption carries with it some implementation details about such processes which makes a process something more than just a simple thread of control. A thread of control can be represented, minimally, by some code, a heap, a stack and some register values, whereas the representation of a process is a superset of such a thread of control with additional operating system state such as page, process and file table entries. When a process context switch takes place, the data structures maintained by an operating system have to be updated, as well as the change over to the new thread of control. This has given rise to processes of this kind being called *heavyweight*, due to the relatively long time it takes to perform operations on them such as context switches, creation and deallocation.

### 2.2.1 Resource Management

Operating systems have played a dual role in the development of parallel processing. Historically, operating systems have provided the stimulus for parallel programming research, as often their job has been thought of in terms of a collection of concurrent activities. (Parallel languages designed for the construction of operating systems include: Concurrent Pascal [Brin75], Edison [Brin81], and Modula-2 [Wirt80].) More recently, however, operating systems are being written with application-level parallel processing in mind. Hence, system call facilities and tools are being provided to support efficient parallel processing and multiprocessing rather than the multiprogramming of old.

The range of jobs for which operating systems are expected to take responsibility has grown as computer architectures have developed. Responsibilities for virtual and real memory management, file structures, input/output peripherals, reconfiguration (in the case of partial failure or expansion), exception handling and not least process management have all fallen to the operating system. Two underlying principles, identified by Finkel [Fink86], have been followed during the evolution of operating systems.

*The resource principle*: an operating system is a set of algorithms that allocate resources to processes.

*The beautification principle*: an operating system is a set of algorithms that hide details of the hardware and provide a more abstract environment which is uniform across architectures and implementations.

Operating systems responsible for implementing parallel processing on multiprocessor machines have a more difficult job than those merely implementing multiprocessing. Conventional operating systems must follow the principles set down by Finkel making sure that dynamic problems such as *deadlock* and *starvation* do not arise (these terms are explained further in chapter three). Operating systems committed to parallel processing must also perform these duties, but at the same time try to ensure tangible benefits can be obtained from the use of intra-job parallelism. Clearly, conflicts may arise between the interests of a parallel job and those of the remainder of the system. However, there is no algorithmic set of rules to follow to obtain the optimum system throughput with optimum parallel program performance. Moreover, it seems unlikely any such rules will be formulated, except in special cases, due to the high complexity and the influence of unpredictable factors such as the execution times of programs.

Nevertheless, heuristic methods are used to resolve the situation by, in some cases, simply reducing the priority of parallel jobs when the system is busy (e.g. Encore Umax [Enco87]), or more forcefully by partitioning up the available resources so that parallel jobs execute in their own mini-environments (e.g. BBN Butterfly Chrysalis [Thom88]). Neither of these solutions seems entirely satisfactory as each one is biased too much in favour of one extreme. This may be precisely what is wanted when considering timesharing or real-time systems, but in general a less extreme solution would be better.

### Multiprocessor Operating Systems

There are three classes of multiprocessor operating systems, with each having a completely different level of support for parallel processing: (i) separate supervisors, (ii) master-slave, (iii) symmetric.

Separate supervisors is the simplest scheme to implement as it is a natural outgrowth from sequential operating systems. Each processor in the system has its own private memory, so it runs its own copy of the operating system, maintains its own files and accesses its own input/output devices. In effect, this is a multicomputer organisation as the operating systems are loosely coupled with there being no direct support for parallel processing.

A master-slave operating arrangement is an attempt to off-load processing from a master processor to several subordinate processors. Operationally, the master processor assumes responsibility for running the operating system itself, performing tasks such as servicing interrupts and communicating with peripherals, while the slave processors run user programs. This scheme has the advantage that it supports some parallel programming, but it does have the limitation that all the operating system calls have to go to the master processor. This can have the effect of choking the parallelism as slave processors wait idly for the master processor to come round to servicing their requests. Investigations have been carried out to find out the extent to which the master processor can bottleneck the system [Ensl77]. Their conclusions were, that the performance advantages gained from adding slave processors diminished very rapidly, so discrediting this method of multiprocessing.

In a symmetric operating system, similar to a symmetric processing system, each processor has equal status and can run either the operating system or user programs. There is only one shared copy of the operating system and similarly all peripheral resources are pooled allowing equal access. There are many variants of

symmetric operating systems with the simplest being the *floating master*. In this subclass, only one process is allowed to enter the operating system kernel at a time, forcing all others to wait. This can be viewed as a master-slave arrangement with temporary mastery passing between processes. Other more complex variants allow concurrent accesses to operating system data structures, with special synchronisation code used to ensure consistency (e.g. Encore Umax and Sequent Dynix [Oste86]). These operating systems are run on tightly coupled shared memory multiprocessors.

In addition to the classes of operating system mentioned previously, distributed operating systems have also been used with multiprocessors. A distributed operating system is one that runs on many machines simultaneously but gives users the illusion of a single machine. Such systems are similar to pure multiprocessor operating systems but must use a form of message passing to exchange data between processes, as it is possible for a group of cooperating processes to be located on several heterogeneous and physically distributed machines. Mainly, effective distributed processing is considered not to be communication intensive (i.e. it is loosely coupled), as opposed to parallel processing which sometimes is. This stems from the underlying process models that are used to communicate between remote machines (e.g. client-server models). When the distribution of an application is necessitated because key resources are only available on selected machines, often, processes on a machine will have to wait while operations are performed remotely - meaning that there is no real parallelism as only one thread of control is active at a time. Furthermore, the distribution of applications can have significant communication and synchronisation overheads making it an unsuitable approach for many problems. Distributed operating systems have been an active research area in many universities yielding many rival systems including: Agora [Bisi88], Argus [Lisk82], Ameoba [Mull86], Eden [Alme85], Isis [Birm85], Mach [Acce86] and V [Cher85].

### Parallelism Support

Multiprocessor operating systems support parallel programming by offering the hardware resources of a machine to programmers in a palatable form and by handling some of the more mundane parts of parallelism management. In a symmetric operating system running on a shared memory multiprocessor, one of the most useful abstractions to give to a programmer is of course shared memory. Thus, memory protection must be maintained between users but relaxed between processes cooperating on a shared job. Two examples of models of sharing are the

Umax *share()* call that allows a process to share memory with its children, and System V shared segments that allow a block of shared memory to be linked into the address space of an arbitrary process. Other interprocess communication mechanisms include *sockets* and *pipes*.

Fundamental process handling facilities are provided by operating systems for the creation and deletion of processes. In symmetric operating systems, facilities are also provided for multiprogramming and multiprocessing processes. These facilities are often optimised so that parallel programs can be load-balanced over the available processors. Figure 2.8 gives an indication of the kind of scheduling that is performed by an operating system by illustrating a typical process state diagram. The systems calls that are necessary to support such an environment include: create, destroy, suspend, resume, change priority, block, wakeup, and dispatch.

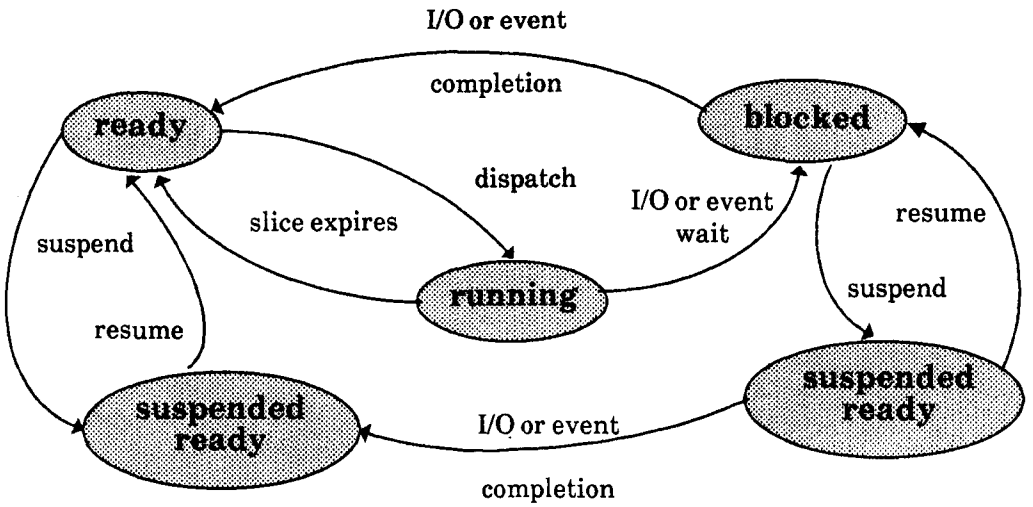



Figure 2.8 - Process state diagram.

To synchronise a group of cooperating processes synchronisation mechanisms such as System V semaphores have been developed. Additional mechanisms for the provision of exception and signal handling are also common.

Another of the responsibilities of an operating system is its role in debugging and the gathering of statistics. Both of these activities can be simplified if an operating system provides interfaces whereby relevant system information can be easily obtained. If this is not the case, the onus of collecting information about a parallel program is left entirely on a user, which can lead to further complications in already complex programs. In addition, the ability to checkpoint a parallel program can give usability and fault tolerance benefits. This is because

checkpointed programs can be restarted from the last checkpoint, obviating the need to reexecute any costly computations needed to reach that point. (Further information on debugging parallel programs can be found at the end of this section.)

### Alternatives to Processes

Although processes have been used as the standard units of control flow they are unsuitable for many parallel programming applications. As far back as the early 1970's some operating systems contained *lightweight* threads of control, commonly called *threads*, to implement asynchronous input/output operations. A thread can be viewed as an activation of a procedure and differs from a process in two major ways: (i) many threads can execute in the same address space, and (ii) threads have much less associated state information. Moreover, lightweight threads are generally implemented on top of the operating system giving an additional level of software support to applications. (If this is the case then processes are still created and scheduled by the operating system with the lightweight threads executing in the address spaces of the processes.)

Using the facilities of a user-level threads package for parallel processing has some advantages over using similar facilities provided by an operating system, including:

- functionality,
- portability,
- efficiency,
- abstraction.

One of the most important reasons for using a user-level threads library is that different functionality can be provided by constructing superior interfaces to standard system calls. Furthermore, with parallelism expressed in higher level constructs a certain degree of freedom can be obtained from the underlying operating system, allowing portability across systems. As user-level constructs use only a minimum of system calls, and have less operating state overhead, programs are more efficient as less time is spent in the management of parallelism. For example, with the Brown Threads Package on an Encore Multimax typical times for user-level (thread) operations are often measured in microseconds, while the times for similar (process) operations performed via the operating system, in milliseconds. If all parallel programming were done at the kernel level, every system call would require a context switch to system mode as

well as the time for the operation itself. Thus, user-level threads provide a greater degree of abstraction than the base operating system allowing the construction of more understandable programs. However, this benefit is not without its drawbacks.

One of the foremost penalties of forsaking the operating system is the according loss of security through memory protection. As threads execute in the same address space it is possible to get unintentional memory interference between them. This can take the form of threads accidentally corrupting user data structures or even other threads' data structures, such as their stacks. In addition, it is likely that no information about the internal structure of a threads program can be communicated to the operating system. Thus, a thread responsible for coordinating a group of worker threads would be suspended if the operating system suspended the process that was executing it - thereby forcing all the workers to wait until that process (and the thread) were restarted. Furthermore, if a thread running in a process together with several other threads page-faults, that process and all of the threads it is supporting will be suspended until the page is brought into memory. Solutions to some of these problems have been implemented in operating systems such as Umax, which allows gang scheduling so that either all the processes in a gang run or none of them do, and Mach, which allows user-level handling of page-faults. These are steps in the right direction but more comprehensive support for threads is needed.

There have been many instances of lightweight threads with properties accorded to them by their intended roles. Threads have been used inside operating system kernels to implement non-blocking operations, and at the user-level to provide alternatives to process-level concurrency. Threads execute in a shared address space, therefore requiring shared memory (generally physically shared), and seem a natural method for exploiting parallelism on shared memory multiprocessors. Operating systems that support lightweight threads include: Amoeba, Mach, Topaz [McJo87] and V. Portable lightweight threads libraries include: Brown Threads [Doep87], C Threads For Unix (CMU), ConcurrentC (Perdue) and Presto [Bers88].

### 2.2.2 Tool Support

The idea of using tools to assist programmers in the construction of programs has been around for many years. Tools have been provided as aids to programmers to help them manage the complexity that is associated with developing correct and

efficient software. Tools come in many different forms, with each having a specific job - just like the tools in a conventional tool bag. Every tool has its part to play in contributing to the process of developing high quality software, hopefully, in the shortest possible time. Of the many different types of tools, ranging from editors to source code management systems, only a few are especially relevant to parallel programming, and these can be broadly classified as:

- auto-parallelising compilers and restructurers,
- design tools,
- monitoring and debugging tools.

Tools that allow programs to be distributed across several machines (e.g. Avalon [Detl88] and Arjuna [Shri89]) are not mentioned here as they have more to do with issues of physical distribution and reliability than parallel processing. Also, tools that facilitate parallel programming by providing programming constructs are not covered here, as they are mentioned in the first section on operating systems and are revisited later in the software section of this chapter.

## Auto-parallelising Compilers and Restructurers

The aim of the tools discussed in this section is to produce code, that runs efficiently in parallel at run time but starting with ordinary sequential code. Many different techniques fall under this heading, with vectorisation being the most well known. Almost all the work carried out on tools of this kind has been undertaken with conventional procedural languages such as C and Pascal, with by far the largest proportion on variants of the Fortran language. Fortran has been used for the majority of numerical software, and therefore almost by default, is currently the most frequently used language for parallel processing resulting from the widespread use of auto-parallelising tools.

To initiate this discussion on compilation techniques a short overview of the process of compilation is presented from Almasi and Gottlieb [Alma89]. This overview is intended to explain some of the terminology and is referenced in chapter five in connection with compiler optimisations. After the overview several parallelisation methods are discussed starting with vectorisation.

Modern compilation can be envisaged as a two stage process, a *front end* that analyses programs and a *back end* that generates machine instructions so that

programs can be executed on a particular machine. More specifically, Aho and Ullman [Aho86] list the stages of compilation as:

1. Lexical analysis.
2. Syntax analysis.
3. Intermediate code generation.
4. Code optimisation.
5. Code generation.
6. Table management.
7. Error handling.

The first five stages proceed more or less in sequence, while the last two go on in parallel with them. The accepted border between the two parts of a compiler is the fourth stage, which can involve both parties.

In the first stage, a *lexical analyser* takes a program written in a high level language and breaks it down into *tokens* which represent atomic symbols such as constants, operators, separators and identifiers. These tokens are then passed to a *syntax analyser* which builds a *parse tree* to check the grammatical structure of the program, signalling errors if necessary. The parse tree consists of *basic blocks*, which are straight-line sections of code, with no jumps out except at the end and no jumps in except at the beginning. The flow of control among the basic blocks within a procedure is represented by a *flow graph*, with the calling relationships between procedures in a program shown via a *call graph*. In addition, special tables are constructed to hold information corresponding to the names used by the program, together with type and other related data. The process of generating all these tables and graphs is called *semantic analysis*, and when this is complete, intermediate code can be generated.

Intermediate code preserves the semantics of the (syntactically correct) original program but expresses them in a format that can be more easily analysed and translated into machine code for a variety of machines. Hence, it contains no references to hardware resources such as registers or specific memory locations.

Code optimisations are geared towards making programs run faster, or to use less memory, occasionally combining the two. Effective optimisation can be accomplished by using the knowledge acquired in the front end of the compiler to suggest possible valid code transformations, and the built-in knowledge of the back end of the compiler to assess which transformations are profitable. Knowledge from the front end comes in the form of an extensive set of

interprocedural (global) and intra-basic block dataflow and dependence analyses. Some of this analysis is straightforward and is described in the following steps.

1) Divide the program into basic blocks.
2) Construct a control flow graph where nodes represent basic blocks and arcs represent possible paths of control.
3) Divide the control flow graph into structured subparts called *intervals* made up of one or more basic blocks. Such an interval might correspond to a loop and expressing it in this form makes it more amenable to dependency analysis.
4) Within each basic block, categorise the occurrence of each variable as one of:
   (a) Variables whose use is confined to the basic block (e.g. temporary variables).
   (b) Those USED in the basic block but defined elsewhere.
   (c) Those DEFINED in the basic block and potentially used elsewhere.
5) For every USED variable in each block, find all possible definitions in other blocks. This is the process of finding the *reach* of each variable, sometimes called *u-d (use-definition) chaining*, and forms the core of global data dependency analysis.

With this information the back end of the compiler can perform further analyses to implement specific parallelisations, the form of which are dependent on the target hardware. Other optimisations, common to sequential machines [Hwan85], are also performed such as: (i) common expression elimination, (ii) invariant expression movement, (iii) strength reduction, (iv) register optimisation, and (v) constant folding.

Finally, or perhaps during the optimisation stage, code is generated for the target architecture by the back end of the compiler. Code generation is a straightforward process given that all the hard work has been done in the semantic analysis and code optimisation stages.

### Vectorisation

Vectorisation is a code optimisation process performed by a compiler to generate vector instructions for vector processors. The idea was introduced by Muraoka [Mura71] and was subsequently implemented by Kuck [Kuck72,Kuck74].

Essentially, the job of the front end of a vector compiler is to look for potential parallelism in a sequential program, while the back end, must subsequently try to turn the promises of parallelism into reality by selecting and scheduling vector instructions. Effective vectorisation of complex programs can be a very difficult job, as not only must algorithmic transformations be applied to produce potentially vectorisable code, but also these transformations must yield code that precisely suits the target architecture, with respect to vector lengths and the timing of vector operations. Optimisations are usually performed in two phases [Hwan85].

Extended Optimisation

(i) Intrinsic function integration (e.g. SQRT, SINE).
(ii) Subprogram inlining.
(iii) Reductions of iteration numbers in nested DO loops.
(iv) Reordering of the execution sequence to reduce pipeline overhead.
(v) Temporal storage management.

Vector Extended Optimisation

(i) Full vectorisation.
(ii) Pipeline chaining.
(iii) Pipeline antichaining.
(iv) Vector register optimisation.
(v) Parallelisation (e.g. using multiple pipes simultaneously).

To vectorise a piece of code, the dependency graph generated by u-d chaining for that piece of code is first checked for cycles. In practice, u-d chaining can lead to conservative estimates of dependency which can prohibit potential parallelism. Other tests which are faster and more accurate have been described by Kuck [Kuck84]. Four types of dependency may exist between statements $S_v$, $S_w$ ($w > v$):

(a) Flow dependence: the value of a variable computed by $S_v$ is read in $S_w$.
(b) Control dependence: $S_v$ and $S_w$ are in separate branches of a conditional $S_u$, with $u < v,w$ (e.g. if $S_u$ then $S_v$ else $S_w$).
(c) Antidependence: a value of a variable read by $S_v$ is recomputed in $S_w$.
(d) Output dependence: both $S_v$ and $S_w$ compute values for the same variable.

Direction vectors can be computed, holding the sense of the dependency between statements, to be used for cycle checking. If no cycles are found, the statements of a vectorisable loop are then inspected to see if they can be mapped into vector instructions of the target architecture. If cycles exist in the code,

however, further analysis can be done to see if they can be broken down or classified as one of a known type of reduction or recurrence operation, that can be executed with a vector instruction or replaced by a subroutine call. For example, dependencies in the forms (c) and (d) can be removed by recoding.

In order for a Fortran compiler to automatically vectorise an inner DO loop, the code contained by the loop must meet certain criteria [Myer86]:

- it must reference at least one array,
- it must use either arithmetic, relational, or logical statements,
- it must not contain any GO TO instructions, IF tests, subroutine calls, or input/output statements,
- it must not contain a multiply dimensioned array; only one index may vary at a time within the loop,
- it must not contain any irreducible data dependencies within the loop.

If the code meets all these conditions then it can be vectorised, but in many cases the code will not meet one or two criteria. This situation can be improved by giving the programmer some written guidelines on how to write code that avoids unnecessary dependencies and can be vectorised more efficiently. Further to this, almost all hardware vendors have their own parallelising compilers that accept directives from a programmer to indicate what should be vectorised. Some of the more advanced compilers, such as Alliant's [Alli86], have a feedback mechanism whereby a programmer is told where there are data or control dependencies prohibiting vectorisation - the idea being that these can be removed later by recoding. In addition, compilers often support an ad-hoc extended language with extra constructs for explicit parallel programming. This basically transfers all the responsibility for parallel programming to the user, making the compiler's job much simpler. In contrast, the most recent version of the Fortran language, the 8X standard [Paul82], contains extensions so that operations can be applied directly to arrays and vectors. This is a kind of halfway-house between auto-parallelisation and explicit parallel programming, making vectorisation more explicit and hopefully a more efficient process, but no full implementations of a Fortran 8X compiler have yet been released. (Other vector programming languages have been proposed and are discussed in the final section of this chapter.)

As an example of vectorisation only the first of the following two code fragments can be easily vectorised.

```
      DO 10 I = 1, N
10        A(I) = B(I) + C(I)
```

After vectorisation the loop above is A(1:N) = B(1:N) + C(1:N), while next fragment has a flow dependence attached to the vector A,

```
      A(1) = 1
      DO 10 I = 2, N
10        A(I) = B(I) + A(I-1)
```

## Auto-parallelisation and Restructuring

Much of the analysis that is performed by vectorising compilers can be applied to generate code for MIMD machines. Often compilers offer both vectorisation and *concurrentisation* facilities, as some machines such as the Cray Y-MP and the Alliant FX/80 combine both MIMD and vector processing capabilities. Similar restrictions to those described for vectorisation apply on the format of code for it to be concurrentisible. However, there is a major difference between the two methods of parallel execution, which is that vector operations function at instruction level parallelism, whereas MIMD machines must process much larger chunks of work in order to overcome the set up times for their threads of control. Once again, the major source of work in programs is loops, which have to be partitioned so that some loop iterations are executed by one processor, some more by the next, and so on. This process is called *loop spreading* and is returned to in the fourth chapter.

Another major source of parallelism that can be exploited by MIMD machines is interprocedural parallelism. Not surprisingly, the task of dependency analysis is more complex than before as procedures can modify non-local data. Moreover, the parameter interface is not always well defined as difficulties arise when dealing with pointers. Nevertheless, some research has been done in this area and working tools such as the Parafrase-2 restructurer have been devised [Poly89].

Parafrase-2 is a source to source program restructurer, that takes sequential programs written in high level languages (currently Fortran, C and Pascal) and transforms them, over a series of stages, into high level code which can be parallelised easily by an auto-parallelising compiler. The difference between this kind of approach and simple auto-parallelisation is that it is more portable, in the sense that it supports multiple languages and theoretically, can be used in conjunction with almost any MIMD/SIMD machine's auto-parallelising compiler.

The restructuring works by first translating the input code into an intermediate representation, then optimising the code over a number of stages towards a given target architecture, and finally translating the massaged code back into the input language. Data dependency analysis is performed by variants of Banerjee's *gcd* and *bounds* tests [Bane88] with adaptations to handle symbolic terms. Another major analysis section of the tool is the Static Program Analyser (SPA). The role of the SPA is to provide estimates of a program's execution time that can be (i) communicated to the user; (ii) used to make a prediction of the maximum theoretical speedup; and most importantly (iii) used to assist in subsequent program transformations. Several methods are available for making timing estimates.

In the simplest model, it is assumed that there are unlimited processors available for use and there are no delays for synchronisation and memory accesses. This gives the theoretical shortest time for execution and can be used as the basis for comparison with other models. A more realistic model assumes that memory accesses are the overriding factor in the execution of a program. In this model, the types and the number of memory accesses are collated, leading to fair approximations of actual execution times [Gall88]. Alternatively, another model assumes that all memory references take unit time and it is the operations that can take variable amounts of time. This approach has limited use, but if used in conjunction with actual timings, can give some indication of the effects of memory contention.

To provide actual estimates of a program's run time, the longest path (in the case of multiple) through a program is analysed; a variant on this method uses an average based on many different routes through the program. Timings can then be used together with the code paralleliser to check if a parallelisation leads to a quicker execution of the program, or if the overheads in the parallelisation outweigh the gains.

Interprocedural analysis is the other major area of dependency analysis in Parafrase-2. The aim here is to parallelise loops which contain calls to subroutines which would otherwise be passed over in parallelisation. Procedures, by definition, try to hide information and are a source of difficulties for dependency analysis. The three main objectives of this analysis are:

- Reference Information: how and when objects are referenced, including the specific part of the object for structured object like arrays.

- Aliasing Information: when two or more apparently different references to objects actually refer to the same object. This can happen directly if two names refer to the same object or more subtly, when two names refer to objects that overlap in storage. Aliasing in languages like C, that allows pointers to objects, is worse than in Fortran though some work has been done in this area [Weih80].

- Constant Propagation: propagating constants across procedure boundaries to lead to more accurate dependency analysis within procedures.

There are two ways of performing the dependency analysis for procedures. The first is to insert an entire procedure at the point of call, expanding it so that it is completely inline. This gives very accurate dependency information but has several serious flaws. For instance, it is time and space inefficient - both potentially increasing exponentially [Sche77], and fails for recursive calls. Another, more effective method, is to analyse each point of call by trapping all of the information flow in and out of the procedure.

In summary, Parafrase-2 is a source restructurer that can automatically perform transformations enabling vectorisation and more general auto-parallelisation. Most research has been done with the Fortran language because of its dominant position as a numerical programming language and because its semantics do not contain troublesome entities such as pointers. Similar work is being carried out with language like C, which makes considerable use of pointers, as many of the underlying analysis techniques employed are still applicable. Related research on program restructuring has been carried out by the PSC [Alle84], KAP [Davi86], VAST [Brod81] and PTRAN [Alle88] projects.

### Other Research

In this section three other approaches, to add to the two more prominent techniques of automatic parallelisation, are briefly discussed. These are: (i) Tree Height Reduction, (ii) Trace Scheduling, (ii) APL-parallelism.

Tree height reduction works by making use of the associative, commutative and distributive properties in algebraic expressions. The process involves splitting up complex expressions into functionally independent parts for parallel evaluation by multiple processors. A comprehensive algorithm has been developed by Baer and Bovet [Baer68] which uses only associativity and commutativity -

though tree height reduction is not a widely used technique due to the small amounts of parallelism it yields in comparison to other methods.

Ideally, it would be convenient if there was sufficient parallelism inside each basic block of a program to make it profitable to execute such instructions in parallel. However, block sizes tend to be small, implying that the speedups that can be obtained are also small, around three at maximum for Fortran programs. The Bulldog compiler [Elli85] tries to increase this amount of parallelism by discerning the control paths through a program (traces) so that several basic blocks can be executed in parallel, a technique called trace scheduling. This has the advantage over vectorisation in that only standard inter-basic block dependencies have to be investigated as opposed to complex loop dependencies. In general, the compiler only guesses with high probability which way a conditional branch will follow, so it has to generate a trace for each eventuality. This can lead to many traces being generated for a group of basic blocks but this turns out to be an acceptable method of trading memory for performance. Further parallelism is obtained by unrolling inner loop bodies to give a form of vector parallelism. Trace scheduling has been around for many years and is the intended method for compiling code for VLIW machines mentioned earlier. However, it does not seem to have caught on and is not as popular as vectorisation or auto-parallelisation.

The high level language APL has also been the subject of a research effort into automatic parallelisation. One of its inherent advantages over other languages is its rich semantics which allow operations to be performed on entire vectors and arrays. Furthermore, it contains no loop constructs or conditional branches. This means that large blocks of parallelism are implied in APL programs and can easily be exploited to give parallel execution without intricate dependency analysis. Unfortunately, almost all APL systems are based on interpreters, as opposed to using compilers which are more efficient. The reason for this is APL's flexibility, which allows objects to change type and size during a program's execution, and APL also supports some functions which are very difficult to compile. Nevertheless, an experimental compiler called the E-compiler [Chin86] has been produced which generates vector instructions from APL source for the IBM 3090 VF. A recent study has been done to compare parallel execution times for vectorised Fortran against times for vectorised APL [Chin88]. The somewhat pleasing conclusion of the study was that APL derived vector code did execute as fast or faster than the Fortran derived vector code. APL is not a widely used language, nor is it ever likely to be, but evidence now points to the conclusion that

working with high-level operations and compound objects such as arrays, provides a good starting point for the exploitation of parallelism.

## Design Tools

To write a good parallel program one must start with a good design. If it has been decided that auto-parallelisation is not suitable for an application, possibly because no such tool is available for the implementation language, a parallel program must be hand crafted to do the job. This task can be made much easier by the provision of software tools to assist in the specification of the parallelism. Design tools give programmers the flexibility to come up with clear designs, by abstracting away some of the fine detail regarding the implementation of the parallelism in favour of more high level approaches. This is also true of newer high level parallel programming languages. However, using a tool allows a programmer to continue using current languages by retrofitting additional functionality. Design tools are also useful when coming to grips with programming multiprocessors that use a revolutionary programming model, such as in the case of the Poker Environment [Snyd84] for the CHiP systolic array.

One might argue that interacting with an auto-parallelising compiler is in some sense designing a parallel program - but this process is simple compliance to the compiler's model of parallelism, which in all probability is nothing more than vectorisation. For example, Parafrase-2 can display a dependency graph side by side with its corresponding piece of code. This facility allows a programmer to adjust the code until extraneous dependencies are removed from it, possibly enabling vectorisation. However, this interaction would not be necessary at all, if the compiler could perform the transformations on the code by itself.

To specify more elaborate parallel systems with arbitrary structure, more flexible methods of interaction are needed for programmers. An example of such a tool is the Schedule/Trace package [Dong86]. The package allows a programmer to describe a parallel program graphically and it will generate the control code necessary to implement the parallelism. Naturally, there are some restrictions on the format of the parallel programs as not all types of interprocess interactions are supported.

The other area of influence covered by design tools is program simulation. Paraphrase-2 has a facility for predicting the time taken to execute a program, but this estimate can only be given after the code has been written - albeit that not all of the program has to be written, only the segment of interest. A more specialised

tool for designing programs is Transim, which is used for specifying Occam programs for execution on transputers. This tool uses a fixed language and hardware but does accept abstract descriptions of programs and generates fairly accurate performance predictions. Probably the major drawback with this tool is, though, that users must supply appropriate data to govern some of the timings. Naturally, if these data values are not representative of actual data, nonsensical timing predictions will be returned.

## Monitoring and Debugging Tools

The tools useful for monitoring programs can be classified into two groups: *debuggers* and *profilers*. Debugging is often described as the process of locating, analysing and correcting faults. More specifically, debuggers assist in the process of examining *what* a program does, making it helpful in locating *algorithmic* faults that cause programs to produce incorrect results. Along side a debugger, the role of a profiler is to help understand *how* programs execute, so that their *operational* behaviour can be optimised to give maximal efficiency. In actuality, this is not a disjoint classification as often algorithmic and operational information can be gained from the same tool.

### Parallel Profilers and Visualisers

As one of the primary uses of multiprocessors is for the execution of large computationally intensive numerical applications, it seems essential to carry out performance tuning through the use of software tools. A large number of monitoring tools and techniques exist and a comprehensive survey has been compiled by McDowell & Helmbold [McDo89].

One of the oldest program monitoring tools is the profiler. Its function is to indicate to a programmer where a program has spent most of its execution time, so enabling subsequent optimisation. In the case of the Unix profilers *prof* and *gprof*, data is collected during a program's execution and is displayed in the form of a list of procedure names, together with the percentage and absolute execution times spent in each one. This information can be augmented by adding the procedure call graph and calling frequency. Procedure data is collected by procedure call counting, and timing information is collected by sampling the program counter periodically and applying some statistical properties of the program to produce figures for the entire run. There are a few problems with this kind of profiling, as sometimes due to statistical errors and certain program behaviour, such as recursion, the profiler will return wildly inaccurate times. Sequential profiling of

this kind can be extended to parallel programs so that each process collects its own information.

However, in parallel programs, other factors that do not arise in sequential programs, such as interprocess communication, are of vital importance and have to be measured. Events such as interprocess communications are indicative of the data flow within a program, which is something that is not covered at all by conventional control flow profilers. Some of the most important information about a program that uses message passing is the data regarding the sending and receiving of messages, where such messages are used to control the execution of the program (the details of this are discussed in the last section of this chapter). As an additional general problem, profiling user level threads is difficult if profiling information is only collected at the process level.

Problems Encountered in Monitoring Parallel Programs

The most obvious difficulty in monitoring a parallel program is the potential for it to generate huge amounts of monitoring data. This can happen because parallel programs, by their nature, tend to have more complex data and control flow than sequential programs, meaning that more information has to be collected to capture the precise state of a computation. Moreover, the adage that correct results do not imply a correct program is certainly true, as the total number of program behaviours can grow exponentially with the number of processes a parallel program uses. In addition, there are several other problems more specific to parallel processing:

Non-determinism : this happens when the outcome of a particular operation cannot be predicted in advance, because it is dependent upon time factors prevalent at the instant of its execution. More often that not, faulty behaviour of this kind may be difficult to repeat or in fact may be non-repeatable.

Probe effect : exact information cannot be gathered about a program without disturbing its underlying computations. This usually comes to light in time critical code, which can have its behaviour altered due to the overhead of executing the monitoring code. For instance, tracing a parallel program can sometimes remove the occurrence of errors by forcing the serial execution of program code. Alternatively, hopefully to reduce this influence, an external process can be used to snoop on a program. This mechanism can work quite well on shared memory systems [Aral88] but is less useful with distributed memory

systems. Time critical code can be found in real-time systems, or in more commonplace locations such as run time scheduling.

Clock synchronisation : in the absence of a single global clock, it is difficult to reason about the state of a program at any given instant. Spatially distributed events happen at their own pace, making it difficult to take an accurate snapshot of how a program is progressing. This means that is it not possible to single-step a program in the traditional sense so that one event occurs at a time.

Visualisation Tools

To combat the problems of trying to unravel parallel program profiling data, potentially approaching megabytes in size, graphical interfaces to profilers have been developed. These allow events in a program to be filtered and displayed in a palatable fashion so that crucial events can be identified. Visualisation can either be performed during the execution of a program or at some later date. Naturally, if real-time visualisation is used, some interference (probe effect) will perturb the working of a program. This is especially true of distributed memory systems, as profiling data has to be exchanged between nodes in competition with real data, before it becomes out of date. Of course, even postprocessing of the data has some impact on a monitored program's performance, as the data still has to be collected.

Bernstein *et al* describe two visualisation tools [Bern89] that have been designed to work with the Preface Fortran preprocessors [Bern88]. These tools are intended for use on shared memory multiprocessors and cover the three aspects of program visualisation.

(1) Collection of data at time of call, possibly by interacting with the operating system or a hardware monitor.
(2) Preprocessor support so that tracking routines can be inserted for every instance of a parallelism construct.
(3) Viewing of data, in real-time or after program execution has finished.

Timing data generated from a profiled run can be attributed to one of the following sources:

(1) Application work - user program, system time and idle time.
(2) Waiting (e.g. at barriers).
(3) Parallelisation/synchronisation routines.

To display this information succinctly, timings can be converted into a two dimensional bar for each process of a program, with each bar subdivided vertically into periods of parallel and serial activity. Bar sections are horizontally divided into time components for: (i) parallel activity, (ii) serial activity, (iii) parallelisation overhead, (iv) system time, (v) idle time. An example of the type of output generated by the Preface-2 display tool for a parallel program consisting of three processes is shown in Figure 2.9.

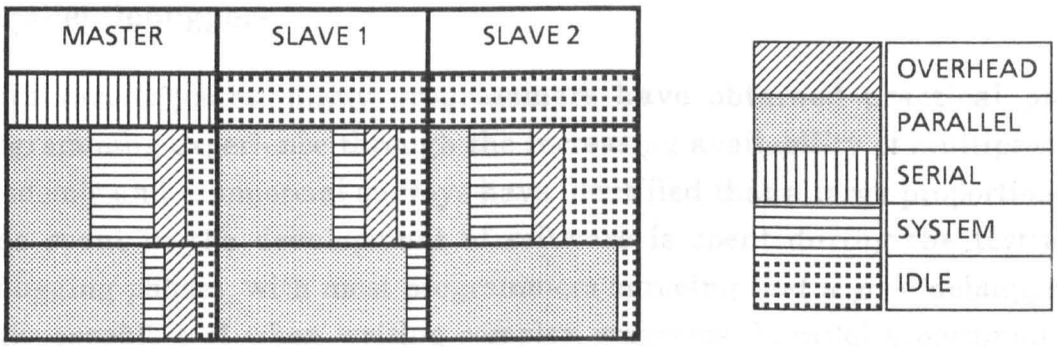



Figure 2.9 - Visualisation of a parallel program.

Other visualisation work has been carried out by Dongarra *et al* [Brew88] in the Memory Access Package (MAP). In this model, references to array elements are displayed instead of control flow information. For instance, every time an array element is read or written, a corresponding change is made to a screen display. This tool is meant to be used as a post processor, necessarily, because of its high cost of collecting the information. Even so, information from the tool can be used to check the memory accessing behaviour of an algorithm so that it can be compared to access patterns of rival algorithms.

Summary

Program monitoring tools come in many different forms, with one of the first [Russ69] being an interactive system which presented a graphical display of potential parallelism in Fortran programs together with any detected bottlenecks. The analysis of early Fortran programs led to the discovery that most programs spend most of their time in a few loops [Knut71], and work was also done by Kuck [Kuck72, Kuck74] which showed the speed up of programs and the utilisation of processors for a number of standard problems.

As the motivation behind parallel programming is greater program performance, the areas of performance evaluation and improvement are being

integrated into debugging. A parallel program that does not run as quickly as expected can sometimes have a performance fault. The methods for examining the execution of a program, and rectifying the problem, are in essence the same as those used in tracking algorithmic errors. Information such as memory access patterns and thread waiting times are common to both disciplines, though performance monitoring also entails collecting some more specialised data such as: message passing statistics, message send/receive delay times, excessive synchronisation waiting, memory channel utilisation and system snapshots.

## Parallel Debuggers

In recent years, many programmers have obtained practical parallel programming experience through the increasing availability of multiprocessors. Academic and commercial surveys have identified that a large proportion of the time spent in the development of software is spent during the testing and debugging phases, with most programmers agreeing that a good debugger is an indispensable tool when writing complex programs. Parallel programming has traditionally been thought of as being a more demanding discipline than that of sequential programming because of its very nature, i.e. many operations happening at the same time. However, the main source of errors in parallel programs comes not so much from the operation of the threads of control themselves, but from the communication framework between them.

### Problems and Errors Found in Parallel Programming

For shared memory machines, the most common fault involves errors in accessing shared variables. Either a value is shared when it should be private, or private when it should be shared. This situation can arise through improper (unintentional) data sharing either, (i) temporally or (ii) locationally. In a temporal error, because of faulty synchronisation, data is shared incorrectly between threads of control when its access should be controlled. For example, if several threads that update a shared sum are permitted to access the sum at the same time, it is likely that the value of the sum will become inconsistent, with updates being lost. In a location error, data is placed in a region that is not suitable for its use, in which it is made unintentionally shared or private. This can result from an oversight of the compiler or from programming errors due to a lack of understanding of the shared memory mechanism. For example, if an operating system allowed memory to be shared at the page level, errors would occur if

intended private variables were allocated on the same memory page with data that was shared, or if subsequently, the page was to be shared.

Distributed memory machines do not have the same type of sharing faults found in shared memory machines, but instead errors can arise in the flow of messages between nodes. One of the most common of these flow-errors is mismatched messages, where a thread is expecting one type of message but receives another. This can cause the node to get out of step with its partners, leading to a partial or complete failure of the program.

Some parallel programs contain unwanted non-determinism in the form of race conditions. Such a condition can arise when the correctness of executing a statement is dependent upon the prevailing timing conditions. These situations, generally, occur in shared memory systems, because of their need to specify synchronisation to control access to shared data. Nevertheless, message passing systems can also contain race conditions resulting from poorly designed inter-node communication protocols.

Apart from the faults which parallel programming can introduce, there are also some difficulties in the mechanics of debugging. One such difficulty is that an error can occur in one part of a program and reveal its symptoms in a completely different section of the code. This problem can occur in serial programs, but in parallel programs it can be much worse, as different code sections can be executed by separate processors. The programmer is then faced with the problem of tracking backwards through the execution of a program, which may have behaviour that is hard to reproduce or is non-deterministic. Moreover, this cannot always be done, as the original error can be quickly obscured by the actions of other threads before it can be investigated by a debugger.

In problems that have faulty timing or synchronisation, the number of processors that a parallel program is executed on can be a factor in its correct functioning. Most parallel programs start life running sequentially or with limited parallelism. Once some confidence has been achieved, large amounts of parallelism are used, where possible, but this can lead to the uncovering of latent faults. This problem originates from not being able to thoroughly test all routes and combinations in a program and can come as a nasty surprise to a supposedly correct program.

As might of been gathered, not all today's parallel software started life as parallel code, as many existing *dusty deck* serial programs have been parallelised.

Moreover, almost all the numerical algorithms that are used in parallel programs are simple modifications of existing (serial) algorithms and are not customised parallel algorithms as such. Therefore, although it may be quicker, it has been observed that it is generally harder to hand parallelise and debug parallel programs that originated from serial programs, than to write new parallel code from scratch. The leading reason for this is that serial programs were largely designed with no thought to parallelism, with the result that tangled control flow and the use of certain data structures often seem to conspire to force unnecessary serialisation. Furthermore, when serial code is parallelised, programmers often have to contend with bad programming practice, such as poor modularity of data, coupled with a lack of understanding about what the code actually does. Fortunately, these problems reflect the transition in writing software, from serial to parallel code, and may become less severe as newer code gradually replaces the old.

Methods of Debugging

There are a number of techniques that have been used for debugging parallel programs. These methods include:

Traditional debuggers: sometimes called *breakpoint debuggers*, these tools operate by permitting a user to stop a program during its execution, examine the state, then continue, or reexecute from the beginning in order to stop at an earlier point in the execution. This method of working is called cyclic debugging as it involves examining repeatable states and state transitions. Tools of this type have been extended to cover multiple processes simultaneously (e.g. cdb and pdbx) but fall foul of the fact that parallel programs often have non-repeatable behaviour.

Event-based debuggers: these tools view the execution of a program as a sequence of events and are generally concerned with analysing *event histories* (in a similar way as event-based monitoring). Deterministic replay of non-deterministic programs can be obtained if events can be recorded simultaneously. Event data can be used in several different ways: (i) browsing, (ii) replay, (iii) simulation. To browse the information, filters are needed to select events of note. Sometimes all of the events can be examined by producing what amounts to a *movie* of the execution of a program. When a program is *replayed*, events are used to control the reexecution of the program, allowing the use of traditional debuggers without changing the program's behaviour. Following on from this, events can be used to

simulate the environment of a program, enabling further debugging to take place, without reexecution of the program.

As an alternative approach to using events, the occurrence of events can be used to control the execution of a program. This can be achieved by defining sequences of events with corresponding actions, in a similar style to path expressions [Bate88]. Thus, when an event occurs some action can be taken, such as signalling the event, or perhaps if the event was signalled as a result of an oversight in its specification, the specification could be altered and the execution of the program continued.

Static Analysis: tools of this kind perform similar checks to those performed in the dependency analysis phases of auto-parallelising compilers. Checks are made for structural faults in programs, arising from synchronisation and data-usage errors. Examples of such errors are deadlock and the reading of uninitialised variables. Note that, such analysis does not prove that a program works - as it does not examine a program's function. However, analysis of this kind can be useful in locating non obviously faulty code, and in circumstances where other debuggers cannot be used because of the probe effect. Other research is being carried out into formal methods, whereby parallel programs are translated into a formal model of concurrency, such as Petri-nets, to permit more comprehensive program analysis to take place [Hall90].

Operating System Support for Parallel Debuggers

It may be all very well to reason about the functionality of a debugger, but many of its capabilities are derived from the facilities offered by its host operating system. A sequential debugger is expected to provide the following operations, either implemented in hardware or by the operating system [McDo89]:

- to read or write a register or memory location,
- to set and trap breakpoints,
- to trap program exceptions,
- to single step a program,
- to trap memory accesses.

A parallel debugger must supply all of these facilities for each process that it supports and furthermore provide:

- the ability to trap on any interprocess communication,
- the ability to modify/insert/delete such messages,
- the ability to control environment factors such as timers.

Hence, there are lessons to be learnt for operating system designers, to provide mechanisms to enable the collection and dissemination of both program and system wide statistics. Failure to do so creates an unnecessary barrier between the execution of a program and the ability of a programmer to monitor it. This may be deemed acceptable when working with high levels of program abstraction, but it is in effect a false reality, as no low level information is available to effectively support the monitoring and debugging of such abstractions.

### Summary of Monitoring Tools

As tools become more elaborate and offer more functionality, there is a tendency to combine the role of several tools under the banner of a single notional tool. By starting at the design stage, tool designers can simplify their job if there is some common framework for tools to be integrated to exchange information about the characteristics and implementation details of programs. Moreover, the jobs of monitoring and debugging a program are becoming intrinsically linked by sharing common data and techniques, with graphical front-ends permitting high levels of user interaction. Some tools allow programs to be monitored in terms of complex objects and operations [Shen90, LeBl90] rather than the low level machine approach common to old-style debugging. Admittedly, the performance tuning of a program can involve the examination of some rather esoteric data, but the ways in which this data can be collected and displayed is merely an outgrowth of more common monitoring techniques.

Probably the most useful attributes of a parallelism monitoring tool are a method to unobtrusively observe a parallel program, and a way to meaningfully sequence the events so observed. These attributes are especially important when considering scalability in parallel programs. Currently, monitoring tools generally deal with only a few threads of control. In the machines of the future, methods will have to be developed so that hundreds or possibly thousands of threads of control can be represented if necessary, without an appreciable loss of usability. Thus, one of the capabilities of such a tool would be to allow the debugging of a single or a group of threads, while the other threads continue as normal. This may not be too difficult for some programs, but for others which have time-dependent constructs such as timeouts, things will not be quite so simple.

## 2.3 Parallel Programming Mechanisms

Many surveys, almost too numerous to mention, have been made of parallel programming constructs and techniques [Alma89, Andl77, Andr82b, Bal89, Bald87a, Bloo79, Karp87, Perr87]. But, given the rapid pace of research no survey could ever claim to be complete, except perhaps at the time of its writing. With the evolution of the understanding of parallel systems, and software engineering practices such as data abstraction, parallel programming styles have changed so that these interests might be reconciled. However, even as the earliest constructs of parallel programming have become so well known that they have nearly passed into folklore, the fickle nature of parallel programming means that it is common to find them appearing thinly disguised even in the newest of languages. Furthermore, due simply to the huge number of parallel languages, there is not the time to make a complete survey of them here. Thus, this section concentrates on the central techniques of parallel programming while omitting some of their derivatives.

When designing a parallel programming notation, Ghezzi [Ghez85] has identified that there are three main issues to be addressed. These can be expressed in a slightly modified form to give: (i) how threads of control are specified, (ii) how these threads communicate, and (iii) how they synchronise their operation. Over the next five sections it will become apparent that many different parallel programming techniques can be applied to solve the same problem. Of course, each method has its own merits and tradeoffs, but as such, there are no definitive guidelines to producing the most effective parallel software.

### 2.3.1 Specifying Parallel Execution

There have been many proposals for syntactic notations for specifying parallel execution in a program. In some of the early attempts, no differentiation was made between the definition of a thread of control and its synchronisation constraints. This lead to a succession of mechanisms which imposed specific synchronisation policies that attempted to trade expressive power for ease of use.

#### Coroutines

Coroutines were devised as a means of obtaining concurrency in programs by exploiting the pseudo concurrency offered by a single processor. Coroutines allow the transfer of control among a collection of routines (procedures) that do not exhibit a hierarchical relationship with respect to the transfer of control [Conw63a].

The transfer of control between coroutines is achieved by the *resume* command whose syntax is "**resume** *routine*". The execution of this command transfers control to the named routine after saving enough state information so that control can return to the instruction following the resume. This enables control to be transferred back to the original routine by the named routine executing another resume. The coroutine model allows many threads of control to exist but allows only one thread to be active at a time. Coroutines have been implemented in a number of older languages including SIMULA [Nyga78], BLISS [Wulf71] and Modula-2 [Wirt80].

### Fork and Join

The syntax of the *fork* statement [Denn66, Conw63b] is "**fork** *routine*", the action of which creates a new thread of control that runs concurrently with the *forking* thread. The *join* statement, is a control flow synchronisation construct whose invocation "**join**", causes the suspension of the calling thread until all of its children (*forked* threads) have terminated. There is a more primitive variant of the join construct "**join** *m,g*", which has a more complex interpretation. The execution of this form of the command indivisibly subtracts one from *m* (normally a shared variable) and either goes to the statement labelled *g* if *m* is zero, or otherwise executes the statement that follows the join, usually a *quit* that terminates the program.

The fork and join constructs are the most powerful mechanisms for creating and terminating threads of control as there are no encumbering rules or syntax for their use. Consequently, programs that scatter such statements throughout their code, executing them conditionally and iteratively, become hard to understand because there are no inherent relationships between any pair of fork or join statements. Software that makes use of these constructs includes the Unix operating system [Ritc74], and the languages PL/1 and Mesa [Mitc79].

### Cobegin

The *cobegin* statement is a parallelism construct similar in operation to the fork statement, allowing the denotation of a block of statements that are to be executed concurrently. Declarations take the form of "**cobegin** $S_1 \parallel S_2 \parallel .. \parallel S_n$ **coend**", which means that each of the statements $S_1 .. S_n$ are to be executed in parallel. Each of the $S_i$ may be of any statement type including another cobegin. The cobegin-coend constructs are more structured and less powerful than the fork-join combination, as each cobegin must be matched with a corresponding coend.

Variations of this statement, first introduced as *parbegin* by Dijkstra [Dijk65a], have been included in the languages Edison [Brin81] and Argus [Lisk82].

### Doall

This construct can be thought of as a specialisation of the cobegin statement, in which statements marked for parallel execution must appear inside a loop construct that allows the programmer to use an index mechanism to identify eligible statements, rather than writing each one out explicitly. Often as a result of this, the number of parallel statements in a *doall* is generally determined at run time as it is expressed as a parameter, as opposed to cobegin whose number of statements is fixed at compile time.

There are many variations on the basic doall that can be used to specify a variety of loop behaviours [Wolf89]. For example, the sequential Fortran **DO** statement will execute loop (a) deterministically, while the results of loop (b) are non-deterministic as loop iterations may be executed in parallel in an unspecified order with no synchronisation enforced between them.

(a)
```
DO (I = 2, N-1)
    A(I) = A(I+1) + A(I-1)
ENDDO
```

(b)
```
DOALL (I = 2, N-1)
    S1 : A(I) = A(I+1) + A(I-1)
ENDDO
```

(c)
```
FORALL (I = 2, N-1)
    S1: A(I) = B(I) + C(I)
    S2: ..
ENDFORALL
```

(d)
```
DOACROSS (I = 2, N-1)
    A(I) = B(I) + C(I)
ENDDOACROSS
```

In loop (c), all the values of *B(I)* and *C(I)* are fetched, all the additions are done, then all the values of *A(I)* are stored. Furthermore, the semantics of *forall* force $S_1$ to be executed for all values of $i$ before $S_2$ is executed. In loop (d), iterations are executed concurrently with the provision that indexes are allocated to threads in strict ascending order.

### Process declarations

In this scheme, some of the procedures that comprise a program have a special denotation marking them as procedure bodies for threads of control that will execute in parallel. Pratt [Prat84] gives a discussion of how progressive relaxation of the constraints on the sequential calling and execution of procedures eventually leads to parallel threads of control. Process declarations provide a more modular approach for declaring parallelism than say the cobegin statement because of their ability to support data abstraction, therefore aiding understandability and modifiability. Some examples of this kind of construct are to be found in the

languages Concurrent Pascal [Brin75], Modula [Wirt77], CSP [Hoar78], PLITS [Feld79], Ada [DoD80], and SR (Synchronising Resources) [Andr82a].

### 2.3.2 Thread Communication and Synchronisation

There are essentially two ways in which threads of control can communicate, either they make use of shared memory and share variables directly, or shared memory is not used (it may not be available) and they communicate by making procedure calls to explicitly exchange messages. Threads of control may need to synchronise their operation for two reasons. Firstly, to moderate accesses to shared data, and secondly to moderate their control flow. *Control* synchronisation affects only the control flow of the participating threads, while *data* synchronisation represents a synchronisation point and an information exchange. As a general observation, on one hand programs that make use of shared memory have to be concerned about synchronising accesses to shared data by selectively delaying the execution of threads. On the other hand, those that make use of message passing need to be concerned with the distribution of data and the form of message exchange between threads of control, and not with synchronisation which happens automatically.

When shared variables are used for interthread communication, synchronisation must be enforced between threads to ensure that they access the shared data in a consistent manner. It is useful to identify two types of synchronisation: *mutual exclusion* and *conditional synchronisation*. Mutual exclusion ensures that the execution of a sequence of statements can be treated as a single indivisible operation when considering thread interleavings. Such a sequence of statements that must be executed indivisibly in order to preserve program consistency is called a *critical region*. Mutual exclusion is achieved by only letting one thread operate inside a critical region, with any others being blocked from entering the region until that thread has left. The term mutual exclusion, naturally enough, refers to the subsequent mutually exclusive executions of the critical region. For example, consider a group of threads that sum a vector. Each thread sums a portion of the vector, then updates a counter representing the number of elements summed and the sum itself. When a thread updates the counter and the sum, this should be treated as an indivisible (or atomic) action so that the relationship - the counter holds the number of elements summed and the sum holds the total, always holds true for other threads. Mutual exclusion is a method for making the effect of simultaneous accesses be the same

as if they had been made in some unspecified but legitimate sequential order. This requirement is known as the *serialisation principle.*

Conditional synchronisation is used to delay a thread until some specified condition is met. This can be viewed as being a generalisation of mutual exclusion by using an arbitrary condition to block competing threads. A sample application that uses conditional synchronisation is the implementation of a shared buffer. The semantics of reading from the buffer are: if the buffer is not empty remove a value, or else wait until a value arrives. The semantics of writing are: if the buffer is not full then insert a value, or else wait until there is space in the buffer. To guarantee consistency when operating on the buffer, both reading and writing operations must of course be executed atomically.

Four constraints are imposed on the operation of primitives and algorithms to solve critical region problems. Firstly, machine instructions are executed indivisibly and memory references are serialised to create predictably reliable machine operations. Secondly, the finite process assumption holds, so that a critical region is executed in a finite time. Thirdly, threads operating outside of their critical regions cannot prevent other threads from entering their own critical regions. Finally, threads must not be indefinitely postponed from entering their critical regions. A synchronisation mechanism must be able to demonstrate that it does not violate any of these constraints when attempting to synchronise a given problem.

Algorithmic solutions to critical region problems have abounded for many years but many have contained several minor flaws. Dekker's algorithm was one of the earliest to demonstrate a good (but not complete) solution [Rayn86], with improvements made subsequently by Dijkstra [Dijk65b] and later Lamport [Lamp74]. One of the difficulties that was encountered in writing good mutual exclusion algorithms was that solutions were written using only conventional sequential instructions, such as conditionals ('if' tests) and assignments. As research into synchronisation progressed, special purpose synchronisation constructs were developed, sometimes implemented by hardware, to make synchronisation code much easier to write. Moreover, as the use of parallel programming became more widespread, new languages supporting ever more elaborate synchronisation constructs have become available.

The synchronisation constructs described in the following sections take an *operational* view of synchronisation behaviour. That is, the execution of a parallel program is viewed as a sequence of indivisible steps formed by merging the

sequences generated by the program's threads of control. Not all interleavings of the steps will produce correct results, hence a synchronisation mechanism is needed to enforce constraints on the possible interleavings to ensure correctness. This technique of viewing synchronisation as such an ordering of events is useful when explaining how synchronisation mechanisms work. However, as the number of threads increases, the number of possible interleavings grows very rapidly making exhaustive analysis very difficult. Formal methods for describing synchronisation in terms of axioms and inference rules have been developed to counter this situation [Floy67, Hoar69], though these are not discussed here.

## Hardware Synchronisation Primitives

To ease the task of writing purely software based synchronisation protocols, computer architectures have sometimes offered support in the form of special hardware instructions which are executed atomically, even though they may notionally consist of several individual machine instructions.

### Busy-waiting

Busy-waiting is the most basic synchronisation mechanism of all. When a thread wishes to enter a critical region it makes a procedure call that operates on a variable known as a *lock* (or spinlock). Hardware support in the form of an atomic *test-and-set* instruction is provided to implement operations on locks as they must be performed atomically. If a lock variable is *unset* when the spinlock procedure is invoked, the calling thread *sets* it, and then returns to continue with its execution. If the lock is *set* when spinlock is invoked, however, the thread waits (spins) by repeatedly testing and trying to set the lock until it is successful. When the thread leaves a critical region it calls another procedure (spinunlock) to unset the lock.

Busy-waiting is best suited to applications where critical sections of code consist only of a few short instructions [Hoog83] or when alternative thread scheduling operations take a relatively long time. It is inefficient for threads to busy-wait for long periods of time, as they consume useful processor cycles that could be otherwise employed by other threads. For simple synchronisation protocols, busy-waiting is an acceptable synchronisation mechanism; however, for more complex problems, such protocols can be very difficult to design, understand and therefore prove correct. This follows from the basic nature of the mechanism which places great responsibility on a programmer to decide what synchronisation is required and how to provide it. Moreover, programming mistakes can be easily made by placing calls to spinlock operations in the wrong position or by omitting

them altogether. Machines that provide test-and-set operations include the IBM 370 series and the Encore Multimax. A related operation, *compare-and-swap*, can also be used to represent locks with this being found in IBM 370/168 machines.

### Fetch-and-Add

If many threads of control compete for access to a shared variable, it is possible for these threads to find themselves being blocked for a large proportion of their execution time. A non-blocking primitive, *fetch-and-add(S,P)*, seeks to reduce the time spent in synchronisation by allowing multiple threads to go ahead and execute instead of just one. This mechanism can be viewed as extending the test-and-set mechanism by returning an integer value rather than a boolean. This is done by adding an integer $P$ to a shared variable $S$ and returning the old value of $S$. These semantics are useful for implementing programs where threads need to simultaneously access separate parts of shared data structures such as queues. The fetch-and-add instruction is being implemented on the NYU Ultracomputer and the IBM RP3.

### Denelcor HEP

This computer architecture has a variety of special features and instructions to support efficient synchronisation. Every location in shared data memory and every general-purpose register has an extra bit used to indicate its access state, either *full* or *empty*. Furthermore, a load instruction can be made to wait until its source location is full and then indivisibly set the access state to empty. Similarly, a store instruction can be made to wait until its storage location is empty and then indivisibly set the access state to full. Instructions involving registers can be even more elaborate, as requirements for both the source to be full and the destination empty, can be imposed. To implement the atomicity of these register operations an extra bit is used to indicate that the register is *reserved* and that its value is on the way.

These low-level synchronisation instructions are translated into higher-level operations by HEP Fortran. A variable can take one of three states: full, empty or reserved. For example, given Fortran variables A, B and C the following statements can be executed:

```
A = $C            wait for C to become full, assign to A, set C to empty,
$C = A            wait for C to become empty, assign to C, set C to full,
A = VALUE($C)     read the value of C regardless of access state,
```

| | | |
|---|---|---|
| A = | SETE($C) | *read the state of C regardless, set C to empty,* |
| A = | WAIF($C) | *wait for full, do not empty C,* |
| B = | FULL($C) | *test for full, return result,* |
| B = | EMPTY($C) | *test for empty, return result,* |
| | PURGE($C) | *empty C.* |

Supplemental Note

Recent work carried out by Herlihy *et al* [Herl90] has produced a formal way of describing the properties of synchronisation mechanisms in terms of their *linearizability*. The use of this measure has allowed a hierarchy of the power of synchronisation mechanisms to be drawn up, and has led to the development of the idea of *wait free* synchronisation based on manipulating shared queue data structures.

## Software Synchronisation Primitives

Even if there is no hardware support available, effective synchronisation primitives still can be written. However, the implementation of these primitives is not always straightforward as a number of factors such as blocking, buffering and determinism have to be taken into account.

### Blocking Mechanisms

In busy-waiting, threads that are blocked consume resources while they are waiting and also appear to a thread manager as active threads of control. If a critical region consists of a large number of instructions, synchronisation mechanisms that suspend threads in a queue can be more efficient. In the case where there are more processors than threads of control, busy-waiting is acceptable. However, if the converse situation is true, suspension of threads is generally better. That is, threads that are blocked give up their share of processor time until they are unblocked and can run. This behaviour can lead to the avoidance of certain faults when using busy-waiting. For example, consider a non-preemptive scheduler that has to schedule $t$ threads on $p$ processors ($t > p$). If $p$ threads busy-wait on spinlocks there is no way that any other thread can execute to release them. Hence, almost all software-based synchronisation mechanisms are either non-blocking or instead suspend threads that are not ready to execute in some kind of a queue.

### Buffering Mechanisms

In a message passing system, it is sometimes inconvenient for a thread to wait to communicate with a thread that is not ready. Hence, two styles of message passing synchronisation have been developed. In a traditional *synchronous* message passing system, when a thread wishes to send a message it is blocked (suspended) until after the receiving thread has executed its receive command. Moreover, when a thread wishes to receive a message, it is blocked until after the sending thread has executed its send command. In an *asynchronous* message passing system, threads that send messages do not wait until they are received before continuing their execution. This can happen because messages are implicitly stored in some order (buffered) by the message passing mechanism, allowing senders and receivers to execute at their own rates, provided enough storage is available for undelivered messages.

Asynchronous message passing systems often have rules governing the management of their thread message queues. Facilities that are generally provided are methods for a thread to selectively discard messages from its queue without having to read them, and the implementation of message priority schemes. Simple priority schemes may store the messages in a single ordered queue, with more complicated schemes maintaining multiple queues. Provision can also be made for messages that are sent to threads that do not exist. For example, they may be discarded or perhaps queued until such time as a receiving thread is created or may even cause a receiving thread to be created.

### Non-deterministic Choice

In many parallel programs, a thread of control may have to perform any one of a number of actions depending upon the state of the computation. For example, consider a thread responsible for updating a screen display that receives messages from a group of worker threads. To implement the display thread, a construct is needed to continually check if any of the workers have produced a message without blocking indefinitely while checking on any one worker. A mechanism to do this was suggested by Dijkstra in the form of guarded commands [Dijk75]. There are two guarded command tests, a one-off conditional and an iterator. For the conditional, each of the guards (boolean tests) is evaluated then one statement corresponding to one of the true guards is executed - the choice of statement being made non-deterministically. For the iterative construct, execution occurs while

any of the guards are true, non-deterministically choosing a statement to execute from among the true guards at each iteration.

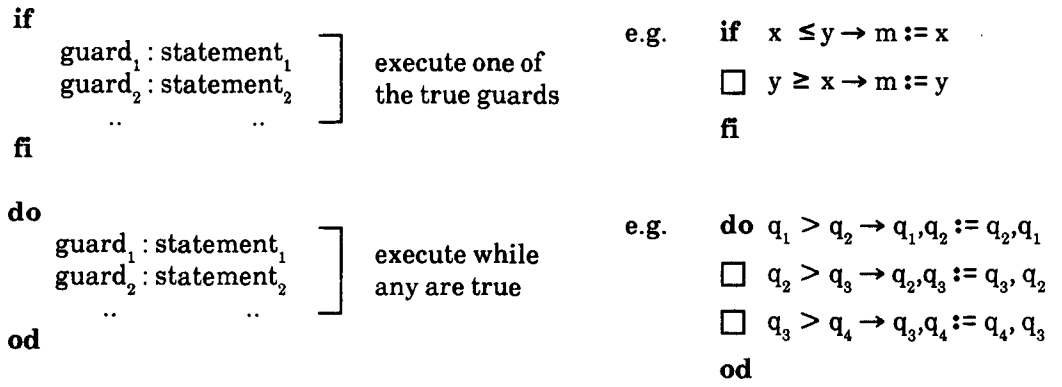



Figure 2.10 - Guarded commands.

Figure 2.10 illustrates the templates of the two forms of guarded commands, with accompanying examples from the original paper. Parallel programming languages that exploit variants of guarded commands include Occam (**ALT** statement) and Ada (**select** statement).

### 2.3.3 Communication and Synchronisation via Shared Variables

Although the idea of communication between threads of control via shared variables is conceptually trivial, ensuring that meaningful and correct communication takes place, however, is not. A hierarchy of software-based synchronisation mechanisms have evolved, progressively offering more abstraction and comfort to programmers. The cost of this process of abstraction, though, has been borne by the implementations of these mechanisms, often resulting in large operating overheads at run time. This tradeoff is quite acceptable when the intended use of these mechanism is considered, which is the programming of resource allocation problems (e.g. dining philosophers). In such problems the emphasis is on elegantly and reliably expressing a correct solution, rather than a need for absolute efficiency. However, there are many other types of problems to which parallel programming can be applied that have quite different emphases on their requirements.

## Semaphores

Semaphores provide a more abstract view of synchronisation than simple busy-waiting and do not need any special hardware instructions for their implementation [Dijk65a, Dijk68]. A semaphore is an initially non-negative, integer-valued variable on which two operations are defined: **P** (wait) and **V** (signal). Given a semaphore *s*, a thread calling P(*s*) is delayed until *s>0* and then executes *s:=s-1*; the test and decrement being executed as an indivisible operation. A call to V(*s*) performs, as an indivisible operation, *s:=s+1* and wakes up a thread suspended on *s* if there is one. Hence, when a thread issues a wait operation on a semaphore and encounters either zero or a negative value it will be suspended until another thread has signalled on that semaphore and given it a positive value.

There are many variations on the basic semaphore scheme, such as binary semaphores and counting semaphores, but all implementations of semaphores are assumed to exhibit *fairness*. This guarantees that no thread delayed while executing a P operation will be delayed forever if V operations are performed infinitely often. This fairness condition has to be imposed because many threads can wait on the value of a semaphore, and if events were left to chance, it would be possible for a thread never to be released if other threads were allowed to proceed in precedence. In practice, fairness is easy to obtain, as threads waiting on a semaphore can be stored in a first-in-first-out queue.

While semaphores can be used to express almost any kind of synchronisation behaviour, they remain unstructured primitives with a high potential for misuse. Similar to busy-waiting, the execution of each critical region must begin with a wait operation and end with a signal operation. If this is not done the mechanism breaks down, in all likelihood causing program failure. Furthermore, there are no inherent differences in the way in which mutual exclusion and conditional exclusion are expressed, making the interpretation of a piece of synchronisation code more difficult.

## Read and Write Locks

This mechanism is more flexible than busy-waiting, differing by using two distinct lock types. If a variable is not locked, a thread can apply either a *read* lock or a *write* lock. A thread wishing to read a variable may do so if it is not locked or if there is only a read lock. If there is a write lock, however, the thread must wait until the write lock is relinquished. In the case of a thread wishing to write, if

there is any kind of lock, the thread must wait until it is given up and then apply a write lock of its own.

This kind of locking can provide a more convenient method for programming some problems, such as readers-and-writers [Holt83, Deit84], than other less structured mechanisms like semaphores. Programs are more understandable because some idea of the intention of a thread towards shared data can be noted from the types of locks it employs. Locking of this kind has been greatly extended by introducing many new lock types and rules, examples of which can be readily found in database systems [Gray78].

### Barriers and the Force

The barrier construct was developed to synchronise programs that operate by partitioning large data structures over a number of processors (sometimes called the single program multiple data SPMD paradigm). A barrier is used to synchronise threads of control by forcing them to wait at a common point in their shared code, the barrier, until a specified number of them are waiting. A barrier is initialised with an integer value that represents the number of threads that it will stop. When a thread arrives at a barrier, it checks the number of waiting threads. If this value is one less than the initial setting of the barrier, it continues execution after waking up all of the waiting threads, otherwise it suspends itself.

The force [Jord84,Jord85] is a generalisation of the barrier. Here, all threads wait at the barrier until the final one arrives. When this occurs, this thread executes the code between the **barrier** and its **end barrier** statement, while the others remain waiting. After the active thread has finished executing its block of statements, all threads continue executing at the statement following the **end barrier** statement. Notice, if there is no code between the **barrier** and its **end barrier** statement, the construct behaves like a normal barrier that traps all the threads. Figure 2.11 illustrates how calls to barrier routines, with a barrier *X*, are used in source program code.

### Conditional Critical Regions

Conditional critical regions (CCRs) [Hoar72, Brin72, Brin73b] were designed to overcome some of the structuring problems associated explicit locking-type synchronisation mechanisms. Shared variables are declared in groups called *resources*, to allow a compiler to check that shared variables are only referenced from CCRs associated with the corresponding resource. Code generated by the

(a)
*statements*
...
**barrier**(X)
*statements*
...

(b)
*statements*
...
**barrier**(X)
*statements executed by only one thread*
**end barrier**
*statements*
...

Figure 2.11 - (a) Barrier construct and (b) Force construct.

compiler guarantees mutually exclusive execution of the CCRs associated with a given resource. A resource is declared as follows, "**resource** *r* : $v_1, v_2, \ldots, v_n$", with variables $v_1, v_2, \ldots, v_n$ being accessed by means of CCR statements. These take the form of "**region** *r* **when** *B* **do** *S*", where *B* is a boolean expression that references only variables that are in *r* and local to its own thread of control, and *S* is a list of statements. Furthermore, a variable can be in at most one resource and variables in resource *r* can be accessed only in CCR statements that name *r*.

Conditional critical regions are good at providing synchronisation but they have two operational drawbacks. Firstly, they have been shown to be costly to implement, because of the large amount of run time checking required to enforce the synchronisation. In addition, program listings can be quite hard to follow as statements performing operations on resource variables are dispersed throughout the code, meaning that the entire program has to be studied to discover how a resource is used.

## Monitors

The monitor concept was developed independently by Hoare [Hoar74] and Brinch Hansen [Brin73a] as an extension of the secretary concept [Dijk65a]. A monitor is an example of a modular design, encapsulating the definition of a resource and the operations that manipulate it. This overcomes one of the drawbacks of conditional critical regions by decoupling the details of a monitor's implementation from the ways in which it can be used. Hence, monitors and their descendants feature in parallel programming languages that are used to program applications which contain numerous shared resources, such as operating systems.

A monitor consists of a collection of *permanent* variables that are used to store the state of the resource, and some procedures to implement operations on the resource. The values of the permanent variables are retained between activations

of monitor procedures and can only be accessed from inside the monitor. There is an initialisation section inside a monitor that must be invoked before any of the monitor procedures can be called. This section of code, for example, may be responsible for initialising internal synchronisation variables, or may in fact be empty. Monitor procedures are written as conventional procedures, possibly using parameters and local variables, but they have the additional property that execution of monitor procedures is guaranteed to be mutually exclusive. This property means that if multiple threads try to make calls to monitor procedures, only one will succeed, and the others will have to wait, making it impossible to access resource variables concurrently. The syntax of the declaration of an example monitor *mname* and procedures $mp_1, \ldots, mp_N$ is presented in Figure 2.12.

```
mname : monitor
    var Declarations of permanent variables

    procedure mp1 (parameters)
    var Declarations of the variables local to mp1
    begin
        Code to implement mp1
    end

        ...

    procedure mpN (parameters)
    var Declarations of the variables local to mpN
    begin
        Code to implement mpN
    end

    begin
        Code to initialise the permanent variables of mname
    end
```

Figure 2.12 - Monitor declaration.

Calls to monitor procedures are made by prefixing the name of the monitor to the name of the procedure. For example, *mname.mp(arguments)* will activate procedure *mp* of monitor *mname* upon invocation.

There have been a number of proposals for realising the internal conditional synchronisation required for monitors. The earliest proposal is based on *condition variables* on which two operations are defined, **wait** and **signal** [Hoar74]. The execution of *condition*.**wait** causes the invoking thread to be blocked and to relinquish its control of the monitor. The execution of *condition*.**signal** causes a blocked thread to wake up if there is one, with the invoker proceeding as normal. (This is similar to *events* that have been developed for use outside of the monitor

environment.) More control over thread scheduling can be obtained by using the *conditional wait* statement, which takes the form **wait**(*B*), where *B* is a boolean on which the invoking thread waits. Further extensions and refinements to these synchronisation operations can be found in Concurrent Euclid [Holt81, Holt83], Concurrent Pascal, Mesa [Lamp80], Modula, Modula-2 and Pascal-Plus [Wels79].

Unfortunately, the synchronisation properties of monitors are not so attractive as might seem at first glance. Admittedly, monitors provide a tidy abstraction of a resource manager with a well defined interface - but therein lies the problem. Consider two threads that have to access two shared resources controlled by monitors. As a result of their programming, the first thread enters the first monitor and at the same time the second thread enters the second monitor. Now, assuming that both threads are successful in obtaining their respective monitor ownerships, then if they have to access the other's monitor before relinquishing their own, both threads will be suspended indefinitely as neither can give way. This deadlock scenario is not limited to pairs of threads and monitors, and can be generalised to many threads and many monitors. Hence, the implications of nested monitor calls have been investigated [List77, Hadd77] and it turns out that this kind of synchronisation fault can occur with almost any non-trivial synchronisation construct.

### Serializers

A serializer [Atki77] completely encapsulates the resource it manages, to form a *protected resource*, but with the resource itself remaining a distinct object within the serializer. Serializers are descended from monitors in that they inherit many of the latter's attributes. Serializers consist of a set of descriptions of internal objects, with operations to manipulate the objects, and the guarantee that the execution of serializer operations is mutually exclusive. However, serializers also incorporate a mechanism for increasing concurrency when working with resource data by enabling the invocation of resource operations from outside the control of the synchronisation mechanism. This is achieved by enabling threads to temporarily leave the serializer and enter the resource to perform the operations.

Serializers have two native features, *queues* and *crowds*. The *enqueue* operation specifies the queue a thread should wait in and the corresponding condition to leave that queue. A serializer automatically restarts the thread at the head of a queue if its condition is met. Crowds are unordered collections of threads used to handle synchronisation resource state information, i.e. they keep track of what threads are in the resource and what operations are currently being

executed. The normal sequence of events for a thread requesting access to a shared resource is:

| | |
|---|---|
| enter | *gain possession of the serializer,* |
| enqueue | *release possession of the serializer and join a queue,* |
| dequeue | *leave the queue and regain possession of the serializer,* |
| join_crowd | *release possession of the serializer and enter resource,* |
| leave_crowd | *leave the resource and reenter the serializer,* |
| exit | *release the serializer.* |

Figure 2.13 is an example of a serializer, written in CLU, to realise the readers and writers problem with a first-come-first-serve priority scheme [Bloo79].

```
first_come_first_serve = serializer is read, write, create;
    rep    =  record[   waiting_q      :  queue,
                        readers_crowd :  crowd,
                        writers_crowd :  crowd,
                        db            :  data_base ];
    create =  Initialisation code for the serializer to create the wait queue and,
              read and write crowds.
    read   =  proc<s:cvt, k:key> returns<data>;
                  queue$enqueue<s.waiting_q> until
                      <crowd$empty<s.writers_crowd>>;
                  d : data
                  crowd$join<s.readers_crowd> then
                      d:= data_base$read<s.db,k>;
                  end;
                  return <d>
              end read;
    write  =  Code to handle write requests.
end first_come_first_serve;
```

Figure 2.13 - Serializer for FCFS server.

Serializers present a more structured approach to writing resource management software than monitors, by way of maintaining a more supportive client-server abstraction. Serializers have established methods for handling request type information (e.g. request times) by the use of queues, and use the crowd construct to handle resource management. These improvements lead to a mechanism that is easier to use and more reliable than the basic monitor mechanism [Bloo79]. Other monitor-like mechanisms (e.g. Pascal-Plus envelopes) can give similar improvements in usability, however, the overhead involved with the operation of these mechanisms can make them unsuitable when resource

operations consume only small amounts of time, i.e. less than the time spent in synchronisation. Hence, monitors and their derivatives can be thought of as being good for coding heavyweight resource management problems - but inappropriate for the exploitation of lightweight parallelism.

### Path Expressions

Path expressions are a synchronisation mechanism that was first defined by Campbell and Habermann [Camp74], though many extensions and variations to the original concept have followed [Habe75, Laue75, Camp76, Flon76, Laue78, Andl79]. The particular variation discussed here is taken from the Path Pascal programming language [Camp79].

Path expressions are a centralised declaration of all the synchronisation constraints for a resource. These definitions are clearly divorced from their implementation and are enforced by code generated by a compiler. A path *controller* keeps track of the operations executed on each instance of a resource, and ensures that the operations executed on that resource conform to some legal ordering. When path expressions are used, a module that implements a resource has a structure like that of a monitor. The structure contains permanent variables which store the state of the resource, and procedures which realise the operations on the resource. One or more path expressions in the header of each resource define constraints on the order in which operations are executed, as no synchronisation is programmed in the procedures. The syntax of a path expression is "**path** *path-list* **end**", where a *path-list* is a mixture of operation names and path operators. Path operators include "." for concurrency, ";" for sequencing, "*n:(path-list)*" to specify up to *n* concurrent executions of *path-list*, and "*[path-list]*" to specify an unbounded number of concurrent executions of *path-list*. Figure 2.14 illustrates the use of path expressions to provide the synchronisation for a bounded producer-consumer buffer.

The path expression indicates that: (i) executions of *put()* are mutually exclusive; (ii) executions of *get()* are mutually exclusive; (ii) a get() operation is always preceded by a put() operation; and (iv) at most $N$ put()'s have not been followed by a get() operation.

Path expressions are an elegant notation for expressing synchronisation constraints expressed operationally (i.e. in terms of procedure executions), but they are poorly suited for specifying conditional synchronisation. Many problems

```
module    buffer
    path    N:(1:(put); 1:(get)) end
    var     Local state variables for buffer
    procedure get()
    procedure put()
    begin
        Initialisation code for buffer
    end
```

Figure 2.14 - Path expressions for a bounded buffer.

that require conditional synchronisation, such as scheduling problems, can only be coded with path expressions by the introduction of additional mechanisms.

### 2.3.4 Communication and Synchronisation via Message Passing

A message passing system can be viewed as a derivative of the semaphore mechanism that has been extended so that when a message passing procedure is invoked, data is transferred as well as synchronisation provided. Typical forms for these calls are "**send** *expression-list* **to** *destination-designator*" and "**receive** *variable-list* **from** *source-designator*". In a message passing system, all communication and synchronisation can be performed by exchanging messages, so there is no need for any shared variables, making message passing systems ideal for use in distributed memory environments. The two main issues in the design of message passing systems are how the source and destination designators are specified, and how communication is synchronised.

#### Specifying Channels of Communication

When viewed together the source and destination designators of a pair of send and receive commands form a communications channel. The simplest kind of channels are those that permit one-to-one communication between threads, sometimes called *point-to-point* communication. More advanced forms of communication include *broadcasts*, in which a thread sends a message to all threads (one-to-many) and *multicasts*, in which a thread send a message to a group of threads. Various schemes have been proposed for naming channels, with the simplest method being *direct naming*. In this scheme, thread names are simply used as the source and destination designators. However, direct naming can be

inhibiting as implementation information about precise names and numbers of threads have to be hardwired into program code. For example, a resource server would not be expected to know in advance the identities of all its clients.

A more flexible approach to channel naming is *mailboxes*, where a list of global names exists. Each name can appear as a destination address in any send and as the source address in any receive command. This scheme allows some degree of anonymity between threads and easily permits many-to-many thread communication. Unfortunately, for distributed memory systems it has been reported that there is a high communication overhead associated with this mechanism [Gele82]. There is a special case of mailboxes, though, in which a name may appear as the source designator in receive commands of only one thread. Such mailboxes are known as *ports* [Balz71] and allow many-to-one communication and selective many-to-many communication if ports can be shared.

The source and destination designators can be fixed at compile time, which is *static channel naming*, or run time, which is *dynamic channel naming*. Dynamic naming is more powerful and can lead to more elegant and flexible programs than static naming. Unfortunately, this power has the respective penalty of making some programs much harder to understand and to reason about. In addition, in common with dynamic behaviour of any sort some run time overheads are incurred in the management of the dynamic naming.

Not all message passing systems operate by using pairs of explicit send and receive commands. In an *implicit receipt* system, when a message arrives for a thread, a new thread of control is created to automatically invoke the receive procedure. Such a mechanism is useful in the construction server processes that contain multiple threads of control.

Many parallel programming languages have chosen message passing as their interthread communication mechanism. For example, one of the early notations, CSP [Hoar78], uses synchronous message passing via direct statically named channels. Thus, an output command takes the form of *destination*!*expression*, where *destination* is a thread name and *expression* is a simple or structured value. An input command takes the form of *source*?*target*, where *source* is a thread name and *target* is a simple or structured variable local to the thread containing the input command. Two threads communicate if they execute a matching pair of input/output statements (target and expression match if they have the same type).

The result of the communication is that the value of the expression is assigned to the target variable, with both threads then proceeding asynchronously.

Due to the large volume of research into distributed systems, many distributed programming languages have been proposed that exploit message passing for interprocess communication. The idea here is to hide the distribution of threads by restricting their interactions to message-based communication. (Other research has investigated the provision of virtual shared memory across a number of machines (e.g. Agora), though difficulties can arise if programmers are unaware of the non-uniform access times that occur.) Message-based languages include: Argus [Lisk82], Distributed Processes (DP) [Brin78], Gypsy [Good79], PLITS [Feld79], StarMod [Cook80], and SR [Andr82a]. Further information on distributed programming languages can be found in a recent survey [Bal89].

## Message Passing Abstractions

There are some higher level mechanisms that are commonly associated with message passing systems. These represent convenient abstractions that make certain distributed programming tasks easier for the programmer. All of the abstractions described can be implemented with variations of the basic *send* and *receive* primitives.

### Remote Procedure Call

The remote procedure call is a convenient mechanism for the implementation of *client-server* systems. The idea behind the client-server model is that resources are completely encapsulated and administered by threads of control known as servers. Other threads that wish to access a resource, the clients, send messages to the server thread instructing it to perform the desired operations on the resource, possibly returning results. Remote procedure calls are often used in distributed systems to hide the fact that clients and servers can be located on separate machines.

Remote procedure calls can take the form of "**call** *service(value-args, result-args)*", where *service* is the name of a channel, *value-args* is the data sent to the appropriate server, with *result-args* being the values returned. On the other side of the service, there are two basic approaches to designing server interfaces. In one approach the remote procedure is declared like a normal procedure with annotations to say that it is a server, and to indicate which parameters are used to supply inputs and which should return results. At run time the remote procedure

is executed as a server thread which waits for the receipt of a message. A thread that wishes to invoke a remote procedure sends a message to the server and then blocks until the results of the remote procedure call are returned. When the server receives a message, it examines its input parameters, executes its body, and then returns a reply message via its result parameters.

A different approach for implementing servers is taken in the Ada programming language [DoD80]. Threads of control in Ada are called *tasks*, and remote procedures are called *entries*; which are ports into a server thread specified by means of an **accept** statement. Ada also offers another mechanism for interthread communication, which is that tasks declared in the same procedure can share variables. An example of an Ada task, named *ResourceController*, designed to control access to a given resource is presented in Figure 2.15.

```
task ResourceController is
    entry GetControl;
    entry RelinquishControl;
    entry GetStatus(Y : inout INTEGER);
end ResourceController;

task body ResourceController is
begin
    loop
        select
            when Condition1 => accept GetControl
                -- entry GetControl's statements;
        or  when Condition2 => accept RelinquishControl
                -- entry RelinquishControl's statements;
        or  when Condition3 => accept GetStatus(Y : inout INTEGER)
                -- entry GetStatus' statements;
        else
                -- default statements;
    end loop;
end ResourceController;

        ...

ResourceController.GetControl;
    -- use the resource
ResourceController.RelinquishControl;

        ...
```

Figure 2.15 - Ada resource controller task.

The resource controller loops indefinitely accepting calls to the entries *GetControl*, *RelinquishControl* and *GetStatus*. When the *select* statement is executed, each of the boolean guards is evaluated. If a guard is true, an accept sequence is said to be *open*. (If several sequences are open, Ada does not specify

which entry will be accepted.) Here, the guards are used to ensure that if and only if, a call to GetControl has been accepted only calls to RelinquishControl will be accepted - enforcing mutual exclusion on the resource and implying that GetStatus can only be called when no task is using the resource. If several tasks call GetControl simultaneously, only one call will be accepted and other calls will wait in a first-come-first-serve queue. This remote procedure call mechanism is called a *rendezvous* and an example of a call to GetStatus is shown in Figure 2.16.

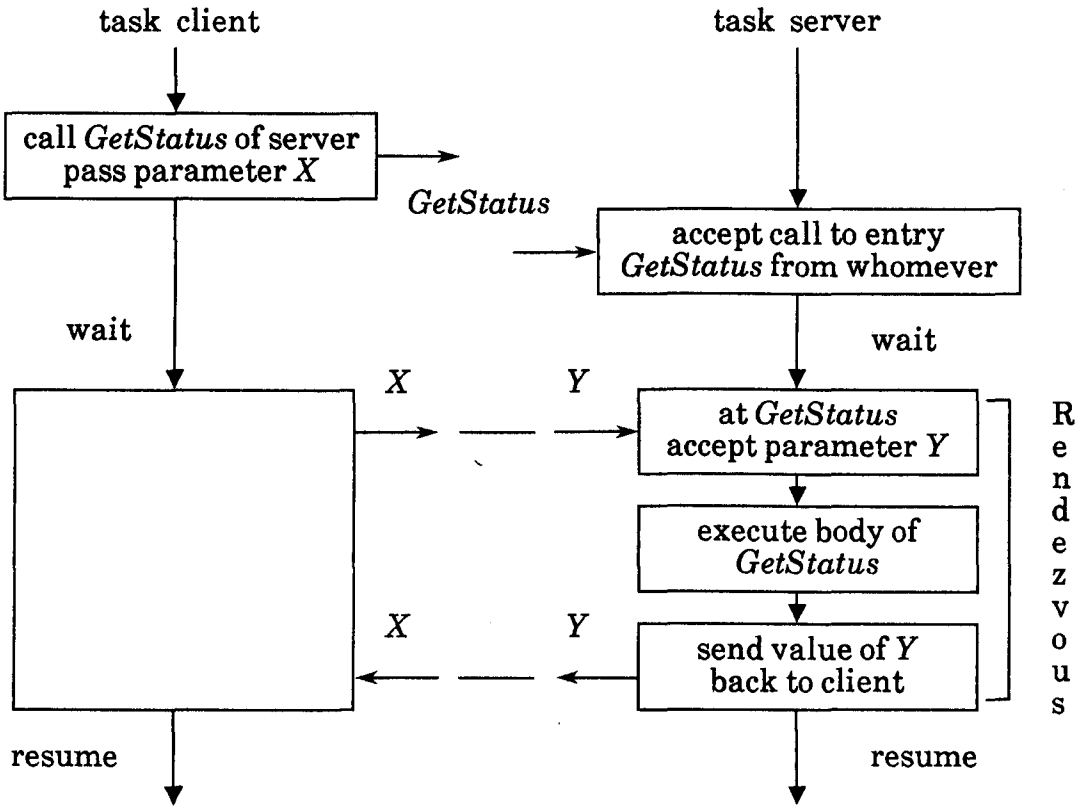



Figure 2.16 - Ada Rendezvous.

In a rendezvous, if the first task does not wish to wait for the second task, it has the option of doing something else and trying to establish the rendezvous again later. This is achieved either by polling the other task or by using timeouts to check if the other task is ready to rendezvous.

### Atomic Actions

Just as individual statements can be regarded as being executed atomically, arbitrarily large operations can be executed atomically if they are packaged in *atomic actions*. A major use of atomic actions is the implementation of transaction

processing on distributed systems, particularly in connection with fault-tolerance [Kohl81]. An atomic transaction [Reed79, Lamp81], is an all-or-nothing computation that either succeeds, installing a complete collection of changes to some variables or aborts, installing no changes. (For purists, atomic actions also have a number of secondary requirements related to partial failures, though these are not discussed here.)

The motivation behind atomic transactions is that in a concurrent environment many independent objects may have to be updated at the same time so that the information they hold is consistent. Atomic actions provide a method of wrapping up a group of transactions so that they are seen to execute together as an indivisible operation. Implementations of atomic actions often use variants of the two-phase commit protocol to synchronise the changes to objects. Even so, atomic actions are not a foolproof synchronisation mechanism as deadlocks similar to those found in nested monitor calls can occur in nested atomic actions. Additional mechanisms can be employed, however, to circumvent these problems by defining the relationships between atomic actions [Shri90].

### 2.3.5 High Level Models of Parallelism

The parallelism mechanisms that have been discussed up until now have been viewed more or less as extensions to existing languages providing functionality for parallel programming. The models presented in this section, however, take more radical views of parallelism, sometimes using no explicit parallelism commands at all. The models described briefly here are: Linda, vector languages, object-oriented languages, functional languages, logic languages, and dataflow languages. These languages represent a large proportion of the higher level models used for parallel programming, but do not represent all of them as special purpose mathematical languages such as DINO [Rosi89] and DEQSOL [Kono89] are not covered.

#### Linda Primitives

The Linda primitives [Gele82, Carr85] provide a method of communication between threads that is both flexible and portable. The primitives form a parallel processing language that can be inserted into almost any programming language, with implementations having been successfully performed on both shared memory and distributed memory multiprocessors. The model assumes the existence of a *tuple space*, which is conceptually, a globally accessible associative memory. The

tuple space can be manipulated by the four primitives *in()*, *out()*, *rd()* and *eval()*, defined as follows:

| | |
|---|---|
| **in( .. )** | *returns a matching tuple and withdraws it from the tuple space,* |
| **rd( .. )** | *returns a matching tuple but leaves it in the tuple space,* |
| **out( .. )** | *adds a tuple to the tuple space,* |
| **eval( .. )** | *forks a new thread of control to add a tuple to the tuple space.* |

The first three primitives execute atomically so that race conditions do not occur when adding or removing tuples from the tuple space. The new thread that is created by eval() is not executed atomically, however, as it may perform many operations during the course of its execution, including any of the Linda primitives.

Tuples can be thought of as ordered collections of fields, where fields can be of any primitive type. For example,

| | |
|---|---|
| 5 | *a tuple of one integer field,* |
| "hello" | *a tuple of one field of characters, i.e. "hello",* |
| "A", 1, 2, 3.1 | *a tuple of four fields of types character, integer, integer, real.* |

Tuples are entered into the tuple space by means of the appropriate Linda primitives and remain there until they are withdrawn or the program ends. It is possible to have multiple tuples with the same template and even with exactly the same values in their fields. When this happens, identical tuples are treated as being indistinguishable from each other. Tuples are withdrawn from the tuple space by matching them against a supplied template. For a match to occur all of the following conditions must hold:

- a tuple and a template must have the same number of fields,
- corresponding fields of the tuple and template must be type consonant,
- corresponding data items must be equal,
- there must be no corresponding formal fields, i.e. all fields have values.

If a rd() or an in() command is executed and there is no matching tuple in the tuple space, the calling thread is suspended. However, if a matching tuple is found, each field of the tuple is bound to its corresponding field of the template. In the case of there being many possible matches for a template, a tuple is selected at random. Non-blocking versions of rd() and in() are also supported which return the value one and a matching tuple, or the value zero otherwise.

## Vector Programming Languages

As the dependency analysis involved in vectorisation can, in some circumstances, be very complicated, vector-level parallelism can be exploited explicitly by vector programming languages. Many such languages have been proposed (e.g. VECTRAN [Paul75]), though most are machine dependent (e.g. Illiac IV CFD Fortran), but Actus [Perr79] contains a suite of constructs that try to abstract way from the underlying hardware while still retaining efficiency for parallel execution. A brief description of Actus is presented which outlines some of its parallel processing features.This is followed by a short overview of Fortran 8X which contains many Actus-like features for parallelism but extends them to multidimensional arrays.

Actus allows the declaration of vectors, arrays and related parallel constants such as sequences. For example,

```
scalar    :  array[1..m, 1..n] of integer;   { two dimensional integer array }
parallel  :  array[1:m, 1..n] of integer;    { one dimensional array (1..n) }
```

Furthermore, as Actus is based on Pascal, vectors can be of compound types (i.e. records) and can be indexed by any ordinal type. Assignment and arithmetic operators in Actus are overloaded so that they can be applied to vector operands. Moreover, existing Pascal constructs, such as **if, case, while** and **for,** are overloaded to operate on either vectors or scalars. For example,

```
A[1:N] = B[1:N] + C[1:N];                { vector addition/assignment }
if A[1:N] > 0 then                       { test each element of A[1:N] }
    A[#] = A[#] - 1;                     { for those elements > 0 }
else
    A[#] = A[#] + 1;                     { for those elements <= 0 }
```

A *sequence* is a parallel constant, which can be used in vector assignment, with its template for declaration being "**parconst** *identifier* = *start*:(*increment*) *finish*;". For example,

```
parconst biglist = 1:100;                { values 1..100 }
parconst gaplist = 10:[10]100;           { values 10, 20, 30, .. , 100 }
```

One of the most important features of the language are *index sets*, which are used to identify individual data elements from vectors. Index sets can be manipulated like conventional sets by arithmetic operators (e.g. +, -, *), and link neatly with the specialised Actus control constructs **any, all** and **within.** For example,

```
within 1:100 do                          { in the range of values 1..100 }
    if A[#] < 0 then A[#] = A[#] + 1;    { test each each, assign if true }
if any (B[1:N] > A[1:N]) then            { succeed if any are true }
    B[1:N] = 0.0;
if all (C[1:N] > A[1:N]) then            { succeed if all are true }
    C[1:N] = 0.0;
```

As mentioned earlier, the Fortran 8X language also contains many facilities for supporting parallel execution. The idea behind Fortran 8X is to make parallelism in programs easier to declare, detect and therefore exploit. It does this by allowing operations to be applied to entire arrays and by providing array manipulating functions (similar to APL) which can be internally parallelised.

For example, "**REAL, ARRAY**(-10:5) :: VECTOR1", declares an array called vector1, size 16, with subscripts ranging from -10 to 5. The shape of an array is defined by its rank (number of dimensions) and its extents (sizes of those dimensions). Fortran 8X supports the reshaping of arrays and the specification of subarrays from larger ones. In addition, elements can be selectively masked.

Array constants are supported by a variety of notations, for example, [0,0,0,1,0,0,0,1] can be written as [2[3[0],1] or [2[0,0,0,1]]. Furthermore, array ranges have shorthand notations, for example, [1,2,3,4,5] = [1:5] and [2,4,6,8] = [2:8:2] where the final digit is the step factor.

Assignment and arithmetic operators are overloaded to work with arrays, and many built-in array handling functions are provided. For example,

| | |
|---|---|
| ALL | *if all elements are .TRUE. the result is .TRUE.,* |
| ANY | *if any elements are .TRUE. the result is .TRUE.,* |
| COUNT | *number of .TRUE. elements,* |
| MAXVAL | *maximum value in an array,* |
| MINVAL | *minimum value in an array,* |
| PRODUCT | *multiplies all elements together,* |
| SUM | *sums up all elements.* |

Further operations are included to check on the sizes of the extents of arrays and to perform array storage management operations (e.g. PACK and UNPACK).

Other research is also looking at specifying parallelism through the use of array valued operations, culminating in a language called Booster [Paal89]. This language is a sub-language centred around data structures and their methods of access, and has been designed to be used with existing imperative languages like C and Fortran. In Booster an array data structure is represented by a *shape* and

can be accessed by means of several user-defined *views*. Further syntax is provided for accessing the individual elements of a shape. Thus, the intention is to provide a method for specifying the operations to be performed on a shape in a understandable yet abstract setting, so that an optimising complier for an arbitrary parallel architecture can extract the maximum amount of parallelism from a program.

Finally, a generalisation of the idea of vector and matrix programming can be found in the data-parallel programming model [Breu88, Rose87]. Data-parallel computation is aimed at exploiting the parallelism offered by SIMD array processors, such as the Connection Machine, and therefore uses a notion of tightly coupled, fine grain computation. Basically programs are written in an imperative language, such as C, which has been extended to allow the explicit declaration of parallel data objects and incorporates a suite of operators (some new, some overloaded) to manipulate such parallel objects. Parallel data objects can be structured types like vectors or matrices, or can be made up from groups of individual records. A program begins execution on the master processor of a SIMD machine operating sequentially, but any parallel objects used by the program are distributed over the SIMD processors, being held in local memories, so that operations involving the parallel objects can be executed concurrently. Thus, the data-parallel model operates at a higher level of abstraction than simple vector operations by taking advantage of the greater functionality and the local memory of each processing element.

### Object-Oriented Languages

Object-oriented languages, such as Smalltalk, are traditional languages in which the set of variables defining a program's state is partitioned into small subsets called *objects*. An object-oriented programming language enables programmers to define new classes of object, where each object is an *instance* of one class. The internal state of an object is represented by a collection of *instance variables* as defined by the class. Each class defines a set of named *operations* that can be performed on instances of that class. Objects interact with each other by sending messages. A message causes one of the recipient object's procedures to be activated, possibly making it change state, and may result in further messages being sent to other objects.

The construction of software using object-oriented techniques is distinguished by its ability to reuse program code by: (i) supporting data abstraction (via encapsulation), (ii) supporting generic operations, and (iii) using inheritance to

derive new objects. For example, if a class *C* directly inherits from a class *B*, then class *B* is the parent and *C* is a child class. Object-oriented languages therefore possess a collection of powerful mechanisms for program construction that may well cause the von Neumann languages to be usurped as the standard programming model.

Object-oriented programming is one of the major research areas in parallel programming as it combines parallelism with leading edge programming language research. Many models of parallelism (e.g. Actors [Agha86]) are based around the idea of collections of objects that execute concurrently. Although it is not possible to cover all the models here, a short list of some of the major techniques is presented. To obtain parallelism in an object-oriented model one can: (i) allow objects to be active without receiving a message; (ii) allow a receiving object to continue after it produces its result; (iii) send messages to several objects at once; and (iv) allow the sender of a message to proceed in parallel with the receiver. (This happens until the sender actually uses any result from the receiver, at which time the sender blocks until the result is produced). In addition, some models of parallelism allow an object to be multithreaded and so internally exploit parallelism.

However, although object-oriented languages are a good starting point for parallel programming, this pathway to the production of effective parallel programs is not always an easy road to follow. For instance, partitioning a program into concurrently executing objects can be a difficult task, suffering from the same problems encountered in the construction of message passing programs. Likewise, if an object utilises multiple threads of control, this can sometimes be undesirably exposed by derived objects thus breaking the rules on object encapsulation [Snyd87]. This is especially true for parallel object oriented languages that are mere extensions of existing sequential languages, though, purposely designed parallel languages do fare better. Object-oriented languages have often been seen as a good way of expressing distributed programming applications and this has given rise to the development of many such languages. Some examples of parallel object-oriented languages include: Beta [Kris87], Concurrentsmalltalk [Yoko87], Emerald [Hutc87], Orca [Bal88], and POOL-T [Amer87].

## Logic Languages

Logic programming languages are declarative languages in which programs are sets of relationships between terms, i.e. they are Horn clauses involving

objects such constants, variables and structures. Constants are represented by strings of alphanumeric characters that start with lower case letters, while variables start with upper case letters. Relationships are either stated as *facts* (predicates) or *rules*. A fact states that a relationship holds between two terms (e.g. father(algol, pascal).). A rule takes the form $A \leftarrow B_1, B_2, .. , B_n.$, which means that *A* holds if $B_1$ holds and $B_2$ holds and so on (e.g. grandfather(X,Y) ← father(X,Z), father(Z,Y).). The *head* of a clause is the term which is implied to be true, i.e. in a rule it is the term to the left of the arrow and in a fact it is the term itself.

The execution of a logic program is a query to see if a specific relationship holds between certain terms, which is known as the *goal*. Clauses are activated by a process called *unification* [Nils80] which finds the most general substitutions of variables that make two expressions identical. A clause is called if its head matches the current goal. When this happens the predicates in its body are called sequentially. If they all succeed, the clause is exited as in a conventional procedure call, otherwise, if a predicate fails, the system *backtracks* to the most recent decision point and follows an alternative path from that point. If no paths lead to success then the clause has failed. The total path traced out by the system is called the *execution graph*.

The most well known logic programming language is Prolog [Ster86], but what is somewhat surprising is that there are three distinct generations of the language, Prolog I-III. In Prolog II, extensions are made to allow the expression of the idea of *difference* (as opposed to equality) and to allow the delayed execution of goals. In Prolog III, rules can be annotated with constraints to speed up their execution by rejecting incorrect solutions as soon as possible. The solution of such constraints can itself be used as the basis for a programming model and is considered to be a highly parallel activity. Some research has been carried out into the parallel execution of constraint languages [Bald87b] though this is not discussed further here.

At least three types of parallelism can be exploited implicitly by the parallel implementation of logic programming languages:

- and-parallelism,
- or-parallelism,
- unification parallelism.

And-parallelism exploits the fact that for a rule to be true, its subgoals must evaluate to true. From the previous example, "grandfather(X,Y)" is only true if "father(X,Z)" is true and "father(Z,Y)" is true. Hence, both queries can be explored in parallel, returning false if either one or both fail, or returning true if they both succeed. To implement and-parallelism the variables of a clause must be able to be typed as input or output variables, so that the information flow between threads of control can be determined. This allows the construction of thread pipelines of the form "produce( ..., X), consume(X, ...)".

Or-parallelism exploits the fact that for a rule to be true, at least one of its subgoals must evaluate to true. Hence, separate branches of a query can be explored in parallel returning true if one or both succeed, or false otherwise. As all the alternatives for a goal are examined simultaneously, there is no need for backtracking to be used. However, the implementation of or-parallelism is quite tricky because separate branches of a query can make independent bindings to variables that are shared between the branches. Thus, some form of scoping must be enforced so that queries following separate branches do not interfere with one another. In addition, if all the separate branches of a program were to be explored in parallel, large demands for memory can be created (as each thread of control must maintain its own copies of shared data). Thus, as a solution a fixed number of threads of control can be used but a load balancing mechanism should exist to enable threads with lots of available work to release some of it to otherwise idle threads of control.

In unification parallelism, the mechanics of the pattern matching that underlies the execution of logic programs is parallelised. If a program consists of many clauses then it is natural to examine them in parallel to speed up the unification process. However, this is not always a straightforward task to perform as in conventional Prolog the ordering of clauses is significant and there is a control flow command called the *cut*. Moreover, due to the optimised nature of the unification process very little parallelism can be effectively exploited, because of the small grain size of the work, and arguments have also been put forward in favour of sequential unification [Mitz86].

As noted above exploiting parallelism in logic languages is not an easy task, as many semantic and implementation difficulties can arise. Some research has focussed on automatically parallelising conventional Prolog, while other research has taken an easier option of defining a subset of Prolog and parallelising that (e.g. Datalog). By way of contrast many researchers have proposed extensions and

new logic languages to enable explicit parallel programming. For example, CS (Communicating Sequential) Prolog uses a notion of time and can suspend the execution of goals until certain conditions are met. A Concurrent Prolog program extends sequential Prolog by introducing read-only annotation of variables (shared logical variables) and a commit operator "|". Resulting programs are a finite set of guard clauses which operate in a fashion that closely resembles guarded commands, with the resulting synchronisation mechanism being that threads are suspended on undetermined read-only variables. For example, if separate threads are used to evaluate each of the goals "goal1(X,Y), goal2(X,Z), goal(X)", if any thread binds *X*, its value is immediately made available to the other two threads. However, problems arise in the implementation of the guarded commands. The semantics of the guarded commands dictate that there should be no effects on a program's environment until after a guard has committed. But, in languages that allow guards to bind shared variables, conflicts can arise due to multiple threads trying to bind the same variable. This can be resolved by excluding variables from guards as in Flat Concurrent Prolog; or by making temporary bindings and then making them permanent on commitment as in Parlog; or as in Concurrent Prolog by making any changes visible outside the clause only if the guard commits.

Multiple messages can be passed between threads by using a form of shared logical variable known as a *stream*. A stream consists of two parts, a *head*, which is a message, and a *tail* which is a stream. Hence, because of the recursive definition of streams they can be used to send an unlimited number of messages. Further information on the semantics and the development of concurrent logic programming languages can be found in a survey paper by Shapiro [Shap89].

### Functional Languages

At present there are many avenues of research into parallel programming based on functional languages. Functional languages support a model of computation that is based upon pure mathematical functions. There is no notion of assignment, variables, or control flow mechanisms such as conditional or iterative constructs. Programs are entirely composed of expressions and functions which can be evaluated by a process called graph reduction (mentioned earlier). In graph reduction, no parallelism constructs are used to direct program execution, as parallelism comes naturally from the process of reducing a program graph. Hence, programmers need not be concerned with performing detailed program tuning, nor is there any need to originate new parallel programming languages. However,

an arbitrary functional program is not guaranteed to have many parallel reductions, even if the application being programming could be phrased as a parallel problem.

Other work in the automatic parallel execution of functional programs has focussed on parallelising Lisp dialects such as Scheme. In the approach followed by ParaTran [Tink88] a preprocessor introduces parallelism into sequential code at points determined by static analysis. Normally, analysis of this kind is quite hard to carry out effectively in languages like Lisp due to their use of pointers. However, ParaTran optimistically schedules code for parallel execution without regard for possible side-effects resulting from accesses to shared data. At run time, though, the ParaTran system detects and corrects data dependency violations using an automatic history and rollback mechanism. Nevertheless, the parallelism offered by auto-parallelisation and graph reduction is only *algorithmic* parallelism (i.e. that which is inherent in the algorithm). Hence, other approaches in parallel functional programming follow more aggressive courses of action in the exploitation of parallelism allowing the coding of explicitly parallel algorithms.

Parafunctional Languages

The next stage in complexity after implicit parallelism is an approach using what are termed *parafunctional* languages. These languages accept functional programs *annotated* with commands to control the location and sequencing of operations. In the ParAlfl language [Huda87], annotations take the form of *mapped expressions* and *synchronising expressions*. Mapped expressions are used for indicating the points in a program where parallelism can be exploited. For example, the sequential expression f(x) + g(y) can written as the parallel form (f(x) on 0) + (g(y) on 1), where *0* and *1* are processor identifiers. This is a static declaration of parallelism and has the drawback that it binds the functions to specific processors, so reducing portability. A more flexible form of the expression can be written as (f(x) on left(self)) + (g(y) on right(self)), where *self* is a processor identifier and *left* and *right* are functions that return the identifier of the processor in the specified direction (assuming a mesh layout of processors).

ParAlfl uses lazy evaluation in common with other functional languages which means that expressions are not evaluated until they are needed. In order to increase efficiency, however, eager evaluation can be used to force evaluation of expressions before they are needed. Synchronising expressions provide a mechanism for imposing an ordering on a sequence of events. For example, the

expression "**synch** (ab) **in** a : f(x) + b : f(y)", indicates that the operation labelled *a* should proceed the operation labelled *b*. In the following example the synchronisation expression permits zero or more evaluations of the expression labelled *a* followed by the expression labelled *b*.

```
synch (ab) * in f(l) + g(l)
where f(lst) = if null(lst) then nil
                    else ... a : f(tail(lst)) ...
       g(lst)= if null(lst) then nil
               else ... b : g(tail(lst)) ...
```

The kind of lock-step behaviour generated by *synch* expressions is similar to that specified by path expressions and regular expressions.

Parafunctional languages separate the functional aspects of a program, that are concerned with the computation, from the operational aspects (i.e. parallelism). In common with implicit parallelism, parafunctional programs contain no side-effects, meaning that timing issues are not a problem. Furthermore, program annotations can be considered to be natural and concise, do not affect the correctness of a program, and are portable to a number of functional languages.

Extended Functional Models

Conversely, many functional programming languages have been modified, or extended, to include explicit parallel programming commands. Blaze [Meth85] is a Pascal-based language for parallel scientific programming that supports algorithmic parallelism as well as explicit parallelism through the use of a parallel loop construct. Modified languages include MultiLisp [Hals87], and Qlisp [Gold88] which contain several mechanisms for supporting parallel execution. For example, Qlisp provides the **qlet** and **qlamba** constructs to fork and join threads of control, with locks and events being used for additional synchronisation. In addition, there is the **catch** and **throw** control flow construct which is used to implement dynamic exits, and the *futures* control construct (first described in MultiLisp). Futures allow a thread of control to manipulate unevaluated objects as along as the objects are referenced indirectly (through pointers). A thread will block if it directly references an unevaluated object until the object have been evaluated. Unfortunately, the use of these explicit parallel programming mechanisms can be just as problematic as similar mechanisms in imperative languages, resulting in common difficulties such as deadlock and non-serialisability.

## Dataflow Languages

Dataflow languages are quite similar to functional language with the exception that one assignment can be made to each value. Statements can be executed as soon as their operands become available, which means that once again algorithmic parallelism can be exploited. Some examples of early dataflow languages include LUCID [Wadg85] and VAL [McGr82].

The single assignment rule means that side-effects cannot cause problems, and in common with functional languages, *locality of effect* can be exploited when executing programs. That is, once a value is fixed it cannot be changed anywhere else in a program therefore limiting the interactions between program segments. Moreover, dataflow languages do not have the aliasing problems encountered with procedural languages. Unfortunately, early dataflow languages encountered problems in efficiently supporting large data structures such as arrays, because a completely new copy of such a data structure needed to be produced after every write operation on the data structure. In response to this problem newer dataflow languages such as ID [Arvi88] make use of data structures called I-structures, which allow an array to absorb one write per array element - thus improving storage management efficiency.

Other prominent work has been carried out in the parallelisation of SISAL programs [McGr83] taking the form of translating a SISAL program into an intermediate code for subsequent compilation and optimisation [Feo90]. Optimisations are geared towards efficient storage allocation, the elimination of extraneous copy instructions, and of course parallelisation. After optimisation, the intermediate code is translated into C code for final compilation into an executable program. Such executable programs have been demonstrated to have performance comparable to similar sequential Fortran, with promising results also having been obtained from parallel runs [Feo90]. One important feature of the SISAL compilation process is that with the output consisting of C code, SISAL programs can be effectively targeted to a wide range of parallel architectures and take advantage of the optimising compilers available for those machines. Other research into dataflow languages includes Strand, which uses a parallel model of statement execution.

### Strand

Although the syntax of Strand [Fost90] makes it appear like a logic language, and indeed it can trace its origins back to PARLOG, it is in fact a functional

language. The computational model it uses dispenses with backtracking and states that all statements that are activated in a program execute concurrently. In practice, activated statements are suspended until their value operands become available - as with conventional dataflow. However, due to performance limitations in the Strand virtual machine, statements are only executed in parallel if they are annotated in a fashion similar to that employed in parafunctional languages. Hence, the onus of identifying useful parallelism is transferred to the programmer, with the system dumbly following those instructions.

There are three methods whereby parallelism can be exploited with Strand. Fine grain parallelism can be exploited by executing individual statements in parallel. Medium grain parallelism can be exploited by executing procedures in parallel that communicate via streams. Finally, coarse grain parallelism can be exploited by using the foreign language interface. This allows a Strand program to act as a parallel harness to call and schedule procedures written in other languages.

The notation used by Strand is similar to that used in Prolog, which is somewhat confusing to readers as program interpretation is quite different. Strings starting with upper case letters are variables and those starting in lower case are constants. Synchronisation in Strand is achieved by the use of single assignment variables, which suspend threads that read them until the values of the variables are determined. Figure 2.17 shows a full implementation of a bounded buffer written in Strand. Communication is realised between the producer and consumer by the use of a shared stream called *Buffer*. Initially, Buffer has three free slots which will accommodate three messages before the producer thread has to wait for the consumer thread to read those messages and extend the buffer.

The interpretation of the program is as follows. When the producer thread is first called, the parameters in its invocation are matched against the first definition of the producer body. As the parameters match those in the formal template, the guard test of the first definition of the producer body is evaluated. As this succeeds the first time round (N=100), the corresponding body is executed. This code generates a new message then calls itself to generate the remaining messages, remembering that in Strand recursion does not involve backtracking. Meanwhile, when the consumer thread is first called, its parameters are matched against those in the first definition of the consumer body. If *[msg|Tail]* is found to

```
% invocation code
Buffer := [M1, M2, M3 | Tail]          % create a buffer with 3 slots and a tail
producer(100, Buffer)                  % start producer generating 100 messages
consumer(Buffer, Tail)                 % start consumer

% code for the producer
producer(N, [Message | Tail]) :-       % wait for buffer space
    N > 0 |                            % check generation count
        NewN is N - 1,                 % decrement count
        Message := msg,                % generate a message
        producer(NewN, Tail).          % recurse to generate remainder

producer(0, [Message | Tail]):-        % no more messages to send
        Message := done.               % close stream

% code for the consumer
consumer([msg|Tail], NextSlot):-       % wait for message
        NextSlot := [X|OtherSlots],    % extend buffer
        consumer(Tail, OtherSlots).    % consume rest of messages

consumer([done|OtherSlots], _ ).       % terminate
```

Figure 2.17 - Bounded buffer in Strand.

be unbound when the match is made the consumer thread waits until it receives a value (i.e. from the producer) before continuing. When the consumer thread executes its body, it turns out that, in this case, the message is ignored and only the buffer is extended.

Eventually, when the producer thread recurses and the guard test of the first definition fails (i.e. when N=0), the second definition of the producer body is matched and executed. This results in a special data value being sent to the consumer signalling the end of data. When the consumer thread comes to read this message, its secondary body definition is matched and executed (which in this case is empty).

# *Chapter 3*

# Parallelism Issues

The aim of this chapter is twofold: firstly, to explore what can be expected from the application of parallel processing to the execution of programs, and secondly, to discover how parallelism can best be exploited on shared memory multiprocessors. The construction of effective parallel programs relies on three things:

- a sufficient set of programming constructs must be provided to allow the coding of a desired algorithm,
- a collection of paradigms for parallel programming should exist, to enable the explicit design of parallel programs,
- there should be some guidelines about matching paradigms to constructs, with references to the efficiency of one paradigm over another for solving a given type of problem.

The first section of this chapter describes a measure for characterising parallel computations in terms of the ratio of useful work performed by a thread of control to its synchronisation overhead. This measure is very helpful when matching parallel algorithms to parallel programming language constructs and eventually to parallel architectures. Subsequent sections describe the different approaches to parallel programming and relate the types of parallelism that they exploit to the architectures to which those approaches are most suited. After this, the faults that can arise in parallel processing are discussed followed by a description of a set of requirements that the designers of parallel programming mechanisms may wish to follow. These requirements are intended as a set of guidelines rather than golden rules, because some of them are in some sense in conflict with each other, as in the case of efficiency versus safety. Thus, the final choice over which requirements to adhere to is partly specific to the problem being solved and also somewhat subjective. The final section in the chapter summarises the approaches that have been used to enable parallel execution, such as auto-parallelisation and explicit parallel languages, discussing what it means to adopt one approach in preference to another.

In a paper exploring the limits to the efficiencies of parallel computations Cvetanovic lists eight major factors that can influence the performance of a parallel program executing on a given multiprocessor architecture [Cvet87].

(1) The amount of parallelism inherent in the application.
(2) The method for decomposing a problem into smaller subproblems.
(3) The method applied to allocate these subproblems to processors.
(4) The grain size of a subproblem executed on each processor.
(5) The possibility of overlapping processing with communication.
(6) The data-access mode where data items are accessed either directly from global memory, or first copied to local memories then accessed from there.
(7) The interconnection structure (hardware).
(8) The speed of processors, memories, and an interconnection network.

The first five factors in the list are concerned with the properties of the problem being solved and the way in which the problem is programmed. From the point of view of this thesis these are the most important factors, with the remaining three being fixed by the choice of multiprocessor architecture. Methods for decomposing problems into subproblems for subsequent execution in parallel and matters pertaining to how these subproblems are scheduled are discussed later in this chapter. However, besides the parallelism in the application itself, the grain size of the useful work that is performed in parallel can have the overriding say on the effectiveness of a parallel program. For in order to be efficient, a parallel programming mechanism must schedule the work that it defines as *suitable for parallel execution* into threads of control to be executed in parallel with as little overhead as possible.

## 3.1 Parallelism Granularity

One of the most frequently used measures applied to parallel computation is the *grain size* or *granularity* of the parallelism that is exploited. This is a measure of the number of instructions in a parallel computation between synchronisation points. Figure 3.1 is a modified table of grain sizes from an original table by Bell [Lee89].

For shared memory multiprocessors the lowest level of granularity that can be effectively exploited is medium grain, resulting from the not insignificant overhead costs of setting up and synchronising multiple threads of control. Of course, greater efficiency can be obtained by executing parallelism with larger grain sizes due to an increase in the ratio of useful work to parallelism overhead.

| Grain Size | Synchronisation Interval (instructions) | Source of Parallelism | Typical Architecture |
|---|---|---|---|
| fine | < 10s | parallelism inherent in a single instruction or data stream | vector processors |
| medium | 100s | parallel processing within a single application program | shared memory multiprocessors |
| coarse | 1000s | multiprocessing concurrent processes | distributed memory multiprocessors |
| very coarse | 100000s | distributed processing across local area networks | groups of workstations (multicomputers) |
| infinite | ∞ | independent computations | all multiprocessors |

Figure 3.1 - Grain size classification for parallel architectures.

Shared memory multiprocessors optimistically need O(100) instructions to set up a lightweight thread of control, with O(10) instructions to perform synchronisation. Parallel programming mechanisms that can spawn sufficient threads of control to permit the parallelism available in an application to be exploited by the hardware, but are large enough to amortise the overheads, will be considered to be effective for shared memory multiprocessors. This idea is touched on again in the fourth and fifth chapters and is demonstrated experimentally in chapter six.

At the vector level, fine grained parallelism necessitates synchronisation after every vector operation, while for processes, synchronisation points in coarse grain parallelism do not occur for many thousands of operations. Figure 3.2 is a map that shows the relationship between the degree of parallelism available at a given granularity [Alma89], in which shaded regions indicate hardware exploitation, with clear regions representing software parallelism. (The scale on the map is broadly representative of real figures but should not be taken literally in all cases.)

One of the main ideas contained within this map is that problems which are based on manipulating objects like arrays, traditionally operated on by iteration constructs, are good areas for looking to apply many processors in parallel. A natural way to achieve this is by spreading the object over the processors, though sometimes the amount of work done by a thread of control can be quite small. Alternatively, in problems which can be decomposed into functionally separate

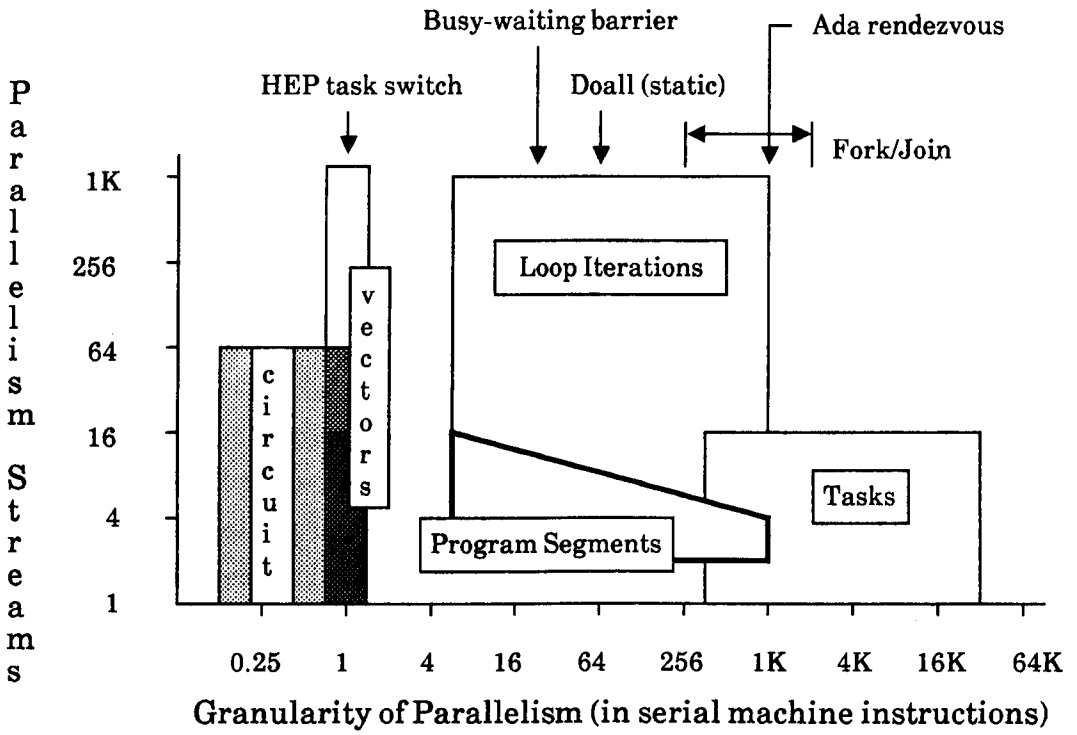



Figure 3.2 - Parallelism/granularity map.

tasks, by way of contrast these are relatively few in number but individually tend to involve more instructions.

A variety of synchronisation mechanisms have been added to the map to indicate the relative cost of each mechanism. Those mechanisms that have hardware support take anything from one instruction to around twenty. Those that are more elaborate take tens to hundreds of instructions, while those that make operating system calls can take several thousand instructions.

If the grain size of the potentially parallel parts of a problem suggests that the problem is not suitable for parallel execution on a given architecture, operations can sometimes be *blocked* together to reduce overheads. For example, scheduling the individual parts of a vector instruction on separate processors of a shared memory multiprocessor does not sound very appealing, even if there are sufficient processors to enable a completely parallel (one-to-one) execution. However, blocking together many parts of the vector instruction so that each processor executes many such parts without stopping for synchronisation can lead to useful performance gains being obtained given sufficiently long vectors. This technique can be applied to many problems where the granularity of the parallelism is less than the optimum size for an architecture, if there are no inhibiting dependencies

between computations and there are sufficient operations to make blocking worthwhile. Several experiments with the grain size and blocking properties of sample problems are investigated in chapter six.

## 3.2 Programming Styles and Constructs

Before discussing the methods by which parallel programs can be produced a short overview of the styles of programming languages that can be used to code such programs is presented. Currently, now more than ever before, many models for programming are being investigated and used practically by researchers. For a given programming model, there are often several languages in use to cover its spectrum of thought, the exact number depending on that model's perceived importance. Parallelism is an issue in programming that cuts across established language boundaries, as it is deemed a feature of a programming language rather than an end in itself. Thus, as many different language styles purport to supporting parallel execution, the task of choosing the best one, or even a good one, is not easy.

In his book, Wadge in a slightly tongue-in-check way characterises five types of programmer by the languages that they use [Wadg85]:

- Cowboys who exploit the quick and dirty features of von Neumann languages and their underlying hardware.
- Wizards who use formal methods for program specification and design, together with logic and functionally based programming languages.
- Preachers who put their faith in adherence to structured programming to tame the excesses of von Neumann languages.
- Boffins who don't care about using a particular style of programming language but rely on software tools to produce correct programs.
- Mr. Fixit who believes the von Neumann languages are not fundamentally flawed and can be made more presentable by cleaning up some of their unpleasant side-effects.

These classes correspond quite closely to genuine approaches to programming philosophy, and illustrate the quite deep divisions that exist between the separate camps. Basically, there are three views on the shape of the near term future of programming. Firstly, there are the fundamentalists who believe in staying with traditional methods of programming because of the large investment that has been made in existing software and the training of personnel. Secondly, there are the evolutionaries who believe that in order to accommodate an ever expanding

and broadening range of demands, programming must change with the times. This does not entail the wholesale abandonment of old techniques, rather a change of emphasis and the inclusion of some new ideas. Thirdly, and finally, there are the revolutionaries who believe that traditional methods were a product of their time and the future for programming lies only in a complete break with the past. From one viewpoint this revolutionary message seems very attractive, because new models give a fresh start to include new ideas, and liberation from some of the problems that plague traditional programming. From another viewpoint, however, some of the promise held by these languages has not been realised because of difficulties in producing efficient software and hardware to execute these languages. Hence, there is often a high degree of inertia in switching to revolutionary models, as some people are loath to change from what they know (and think they understand) to more abstract programming models.

To reinforce the difference between evolutionary and revolutionary models, it is useful to classify programming languages as either of the *imperative* or *declarative* style. Broadly speaking imperative languages represent the heart of traditional programming, with declarative languages representing the new blood (although the lambda calculus which is the foundation for functional programming has been around since the 1930's [Chur41]). Figure 3.3 shows the breakdown of programming languages as either imperative or declarative in nature [Bald87a].

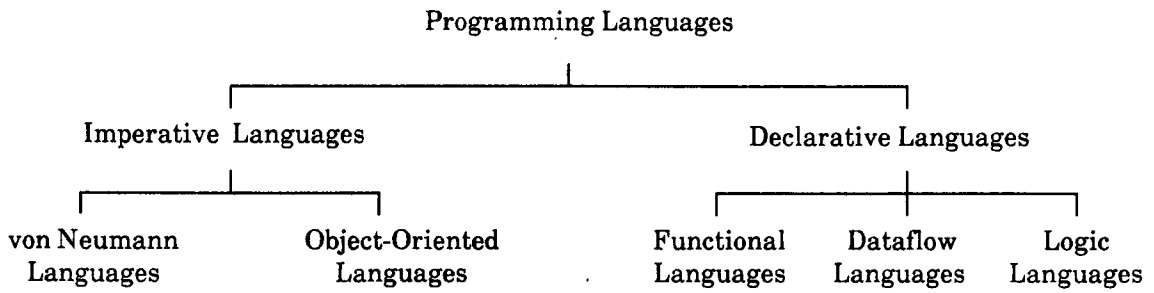



Figure 3.3 - Breakdown of programming languages.

Imperative (procedural) languages are those that are most widely used and understood. Languages such as Fortran, Algol and their descendants are imperative because their programs take the form of a series of steps (a solution path), which if executed correctly and in a valid ordering, will compute the solution to some problem. One criticism that has been levelled at imperative languages is that this need to specify an exact solution path by giving primitive

machine directives and having to consider the order of every program statement is an extra burden on programmers who could be more productively engaged in problem solving. Moreover, some solution paths are so complicated, and in some sense over specified and serialised, that potential parallelism in an algorithm is obscured and not exploitable by auto-parallelisation.

On a more positive side, imperative languages have attracted a reputation for being an effective programming model because they often allow programming down to the level of a bare machine in order to promote storage or execution efficiency. This can happen because imperative languages are based on direct abstractions from the von Neumann model and can let hardware features such as the existence of multiple processors and shared memory percolate up to the programming level. The control flow model of imperative languages is a flexible platform for exploiting parallelism, with a wide range of parallelism constructs having been developed (see chapter two). Moreover, often new control flow constructs can be added (with care) to imperative languages while maintaining the semantics of existing programming language constructs. Thus, in general, parallel algorithms can be expressed in whatever format thought most appropriate, enabling the grain size of an algorithm to be precisely and explicitly tuned to match that of the underlying hardware.

Declarative languages by way of contrast take a much higher level view of problem solving as they hide away the details of the underlying hardware. In such languages (e.g. Miranda and Prolog), only a goal and a sufficient set of relationships between program objects are required. These allow the abstract machines (interpreters) employed by declarative languages to generate solution paths by themselves. This bodes well for auto-parallelisation and for naturally parallel languages because there is not the over-specification found in imperative languages to restrict program analysis. However, the efficient realisation of parallelism can still be difficult because of problems in efficiently matching the potential parallelism in a program to its underlying hardware (in most cases von Neumann machines). Here, the abstract model used by a language may be a poor model for execution on a von Neumann machine, with the mismatch being exacerbated by the importance of matching the grain size as well. Furthermore, explicit parallelism constructs that can improve the matching between software and hardware generally have to follow a different approach to normal (e.g. parafunctional languages in chapter two), as declarative languages abandon the traditional notion of control flow.

If good (efficient) implementations of declarative languages are available, the sequential solution of problems that are basically procedural or algorithmic in nature can be executed as well as corresponding solutions written in sequential imperative languages. For example, searching and list manipulation problems are on the whole handled very well by declarative languages, with the advantage that the resulting programs are easier to read and write than their imperative counterparts. But, whether this is true for the implementation of parallel programs is an open question. A more clear point, though, is that for applications that interact with their environment (e.g. hardware devices) or applications that are controlled in real time, good solutions come from the flexibility of imperative languages because of their immediacy to their hardware environment. By the same token declarative languages often offer more convenient semantic notation to express complex behaviour to abstractly model the physical world, but nonetheless seem to lack the capability to link such a model to its environment without resorting to imperative style constructs. Thus, while declarative languages may offer the promise of easy parallelism, whether this is best exploited by shared memory multiprocessors or specialist hardware is not a cut and dried issue.

Merely saying that a programming language is imperative or declarative can imply some information about the nature of a language, but further insight can be gained from other classification schemes. Figure 3.4 shows Treleaven's classification, which groups programming languages according to their control mechanism (how instructions are sequenced) and data mechanism (how information is structured) [Trel82].

| Control Mechanism | Data Mechanism | |
|---|---|---|
| | Shared Memory | Private Memory |
| control driven | von Neumann | communicating processes |
| pattern driven | logic | actors |
| demand driven | graph reduction | string reduction |
| data driven | dataflow (I-structures) | dataflow (tokens) |

Figure 3.4 - Treleaven's language classification.

Basically Figure 3.4 can be thought of as a high level summary of the types of programming language available to a parallel language designer. For control

driven languages, the main method of exploiting parallelism is by using explicit control constructs. Nevertheless, auto-parallelising compilers are rapidly becoming more widely available and will prolong the lifetime of existing languages by providing almost direct routes to portable parallel software. For the other types of control mechanism, parallelism has most often been exploited implicitly by an interpreter or run time system. This has happened because the model of evaluation, be it pattern, demand or data driven, only specifies the orderings of the events crucial for a program to work correctly, leaving the others to be determined by the run time system. Thus, these models at first seem to provide much better starting points for auto-parallelisation than control driven languages, however, sometimes the reality falls somewhat short of the dream as difficulties have been encountered in the implementation of such parallel run time systems (see chapter two). Hence, hybrid languages have been developed which combine multiple control mechanisms. Parallelism in these languages can be expressed explicitly by special control constructs such as in Qlisp, or implicitly such as in Strand which is pattern driven with data driven synchronisation.

Historically, the majority of work carried out in parallel programming has been focussed on control driven shared memory designs, although private memory designs now seem in vogue. Research into the other models of programming has been steady, though there is some reluctance to adopt one of these models when progress is still being made in improving the familiar control driven languages.

## 3.3 Algorithms and Models for Parallel Programming

When designing an effective parallel program it is productive to use a naturally parallel algorithm. This is true whether the parallelism is exploited directly by explicit parallelism constructs, or implicitly by a compiler or an interpreter. As luck would have it, the choice of the best algorithm is not always easy as in some applications, such as those involving numerical analysis, a variety of similar algorithms may be available to do the job. When an algorithm is executed sequentially, it is generally straightforward to give bounds for its time and space complexities. For a parallel algorithm, however, it is sometimes hard to reason about its performance because such an algorithm may have a poor sequential performance (as compared to rival algorithms), but may out perform its rivals when executed in parallel [Wrig90]. Nevertheless, one of the most important lessons to be followed when writing a parallel program is to use an appropriate

algorithm and data structure [Bent86]. Doing so can greatly improve the quality and efficiency of a solution.

Parallel algorithms can be constructed by following one of the recognised methods, of which there are many classifications, though most are variations on a few basic themes. This state of affairs has arisen because almost all these classifications are based upon intuitive reasoning originating from practical experience, rather than some set of absolute rules rooted in formalism. Thus, classification schemes tend to reflect programming models loosely based on their authors' favourite parallel programming languages, as opposed to being true abstractions of parallel processing. Nevertheless, some formal work has been carried out into parallel algorithm design.

Analytical Models

An analytical model of a computer is one in which abstractions of its hardware capabilities such as the instruction set, number of processors, and physical layout of memory and processors are used for algorithm design. The most basic machine model is that of the Random Access Machine [Aho74] which models a serial computer. This can be extended in a number of fashions to form a Parallel-RAM machine (PRAM), which is essentially a collection of RAMs all accessing the same memory [Boro82, Fort78, Snir82]. Basically, PRAM models differ according to the level of concurrency that is permitted when performing operations and accessing memory locations. Models such as Message Passing-RAM also exist representing machines with private memories.

Analytical models are used to specify algorithms in terms of individual low level instructions. Thus, typical algorithms that can be specified in this way are those pertaining to numerical analysis, those for searching and sorting, and those used in special hardware devices such as systolic arrays. Hence detailed analysis of these algorithms at the level of instruction counting is possible, to determine their efficiencies and limitations. However, because of the low-level approach to algorithm design, analytical models are not suitable for expressing abstract problems where information is structured in terms of groups of complex objects. Moreover, the analytical models can also be very architecture dependent as they are constructed out of primitive machine operations. Thus, algorithm designers should have other methods to fall back on to represent problems that are too complicated to reduce to the basic machine level.

## Models for Parallel Algorithms

The following classification scheme for parallel algorithms tries to draw out the strands of truth from several classifications by not specialising or focussing on a particular processing model. This it does reasonably successfully, although as a classification scheme it is neither rigorous nor complete.

Expressed most simply, parallel activity can be *competitive*, *cooperative* or *independent*. In competitive parallelism, threads of control executing different programs compete for time on shared hardware resources. Here, there is no cooperation or synchronisation between threads of control. In cooperative parallelism, separate programs or more commonly separate parts of the same program execute together, cooperating and synchronising in the joint execution of some shared job. In real life applications, however, it is possible for an algorithm to be both competitive and cooperative, through the use of more active threads of control than processors. Finally, threads of control executing by themselves on separate processors, sharing no resources, exhibit independent parallelism.

In most cases cooperative parallelism is the most tenable method for gaining significant performance benefits for an application, with there being two obvious strategies for exploiting this parallelism. One strategy is to design a parallel algorithm in terms of a network of concurrently executing, specialist threads of control. This approach is known as *functional partitioning*, as the overall function of an algorithm is partitioned into subfunctions for execution by separate threads of control. Here, it is normal for each thread to have a single role and for data to flow between threads as required. Models such as pipelining, and large-scale or macro dataflow are good examples of this paradigm.

The other strategy for implementing cooperative parallelism is to replicate program code by having it executed by multiple concurrent threads of control, and then to distribute the data across them. This is known as *data partitioning*, as program data structures or parts of a large data structure are partitioned into substructures for manipulation by replicated workers. In some sense, the data structures can be viewed as being static and control is passed over them by the worker threads, making this paradigm the dual of functional partitioning. Vector processing and its multiprocessor equivalent of loop spreading are the two most representative examples of this paradigm.

Of course, many real life parallel programs are a combination of both parallelism models. For example, consider a pipeline that consists of several

stages, with each stage being executed by a separate thread which transforms elements from an input stream to form an output stream. A program may have several such pipelines to efficiently compute a wide range of functions. If one such function was found to be required more frequently than the others, perhaps multiple copies of the appropriate pipeline could be employed to enable multiple data streams to be processed in parallel. As a dual example, consider a group of replicated threads that service infrequent but identical transactions. If greater throughput is required, increasing the number of replicated workers is not a solution, as it is the complexity and not the number of transactions that is the bottleneck. Hence, a solution may be found by pipelining the processing of a transaction so that the work previously done by a single thread is partitioned into several threads.

A more specialised, though more informative, scheme comes from Carriero and Gelernter [Carr89] which classifies parallel *concepts* as: (i) result parallelism, (ii) agenda parallelism, and (iii) structure parallelism. These roughly equate to data partitioning, a combination of data and functional partitioning, and functional partitioning in that order.

A program that uses result parallelism is, unsurprisingly, designed around the production of some final result, usually a complex or large data structure such as an array. Threads of control are created with the purpose of calculating individual data items that will be deposited in their respective positions in the final result data structure. Thus, a thread may perform many types of operation before producing its final result. In many cases one thread of control is created to produce a single element, therefore fixing the number of threads for a given problem size and making resulting programs of this type highly parallel for large problem sizes. For example, in ray tracing computer graphics the displayable image can be represented by a two dimensional array of pixels, which can be computed in parallel by assigning an individual thread of control to compute each pixel.

Agenda parallelism involves a transformation or a series of transformations which are to be applied to all elements of some set in parallel. Algorithms of this type consist of a number of discrete stages. At each stage, all threads cooperate to complete the available work on a particular data structure before moving onto the next stage, and possibly another data structure. This means the number of threads that can be engaged to perform processing is flexible and the work can be partitioned at run time for good load balancing.

An algorithm that uses structure parallelism (also called specialist parallelism) is broken down into a logical network of cooperating threads with each thread having a single specialised role. Under the strictest interpretation, the group of threads employed by such an algorithm is fixed in both size and function, however, it seems reasonable to include algorithms that can be decomposed into specialist units, but can dynamically alter their processing configuration by creating and terminating threads of control on demand.

Speculative versus Conservative Parallelism

The models for parallelism just described cover how parallelism is exploited by an algorithm, however, one must also consider what parallelism is to be exploited. On one hand, the conservative view could be taken to only execute the work that is absolutely necessary to compute a program's results. That is, the work that lies on the solution path of the program. On the other hand, one could speculate about the form of a solution path and perform operations (in parallel) that may lie on the solution path. As an example of speculative parallelism consider a conditional statement that has two branches. Both branches of the conditional could be evaluated in parallel ahead of the conditional being evaluated, data dependencies permitting, even though only one branch would ever be needed. Highly significant speculative performance gains, relating to a *combinatorial implosion*, have been reported in artificial intelligence applications based on searching algorithms [Korn82]. Here, a choice of searching techniques exists (e.g. top-down, breadth-first) and it is not possible to tell in advance which algorithm will perform best. Hence, the best solution is to execute a combination of both search techniques taking the answers from whatever search completes first. In fact, this approach gives improved performance even if the searches are multiprogrammed, and when it is executed in parallel, gives rise to superlinear performance gains over both of the original purely sequential searches.

Speculative parallelism is the basis of trace scheduling and OR-parallelism, and can be naturally exploited by other parallel computation techniques such as graph reduction [Peyt86]. However, there are some complications in the implementation of such systems. For instance, identifying suitable instructions for speculative execution can be quite difficult because the work must be of a suitable grain size. Moreover, even when speculative parallelism can be identified, controlling its evaluation can be quite involved. For instance, at any given time during the execution of a program there will be tasks that must be done and those that may be done. If sufficient resources are available, speculation can

occur, but this must happen in such a way that any changes to the state of a program made by a speculating thread are not observed by other threads until that work is actually needed. In addition, in cases when it has been determined that a branch of a computation is not needed, the thread of control following the branch may have to be terminated and its results discarded. If there is only one level of speculation this does not cause too many problems, but in the case of many levels of speculation (e.g. nested conditionals) several threads of control may have to be stopped which could involve a considerable amount of time overhead from housekeeping processing.

### Techniques of Parallel Programming

Once a model for a parallel algorithm has been decided upon, it must be coded using an appropriate parallel programming technique. In many cases this technique will be a natural extension of the parallel model that has been used, but in other cases a variety of techniques will be applicable.

The most influential factor in the choice of parallel programming technique concerns whether the algorithm that is to be coded requires shared or distributed memory. Lauer and Needham [Laue79] argue that the expressive power of techniques based on shared variables is equivalent to that based on message passing. However, not as much can be said about the efficiency of choosing shared memory or message passing without direct recourse to the problem. At a higher level of abstraction, Carriero and Gelernter [Carr89] classify parallel programming techniques as: (i) message passing, (ii) live data structures, and (iii) distributed data structures. They make the point that these three techniques closely correspond to their classification of parallel concepts.

According to Carriero and Gelernter if message passing is used to structure a parallel program, all data values are encapsulated within separate threads of control that communicate only by exchanging messages. A thread may contain many data items and can only reference others by making explicit send and receive calls to their respective managing threads.

A live data structure can be viewed as a special case of message passing in which there is one thread of control per data value. Moreover, such threads only send one message representing their final data value upon termination. These message are usually left in some global area so that they can be subsequently collected by some other thread of control.

Distributed data structures are manipulated by threads of control that communicate by referencing shared data objects rather than by coupled pairs of send and receive operations. Such a data structure is said to be distributed in the sense that many threads of control can operate on it simultaneously, with the data structure residing in shared memory. For example, the process table in a time sharing operating system, and blackboard systems [Fost90] can be thought of as distributed data structures.

Carriero and Gelernter argue that the process of writing a parallel program starts with the choice of a natural parallel concept for its design. Once this has been decided, a parallel program can be coded and tested so that its performance can be assessed. Finally, if the program has proven ineffective, it may be transformed into an alternative parallel program that uses a more efficient but less obvious concept. Figure 3.5 illustrates the perceived relationships between the different methods of implementing parallel algorithms.

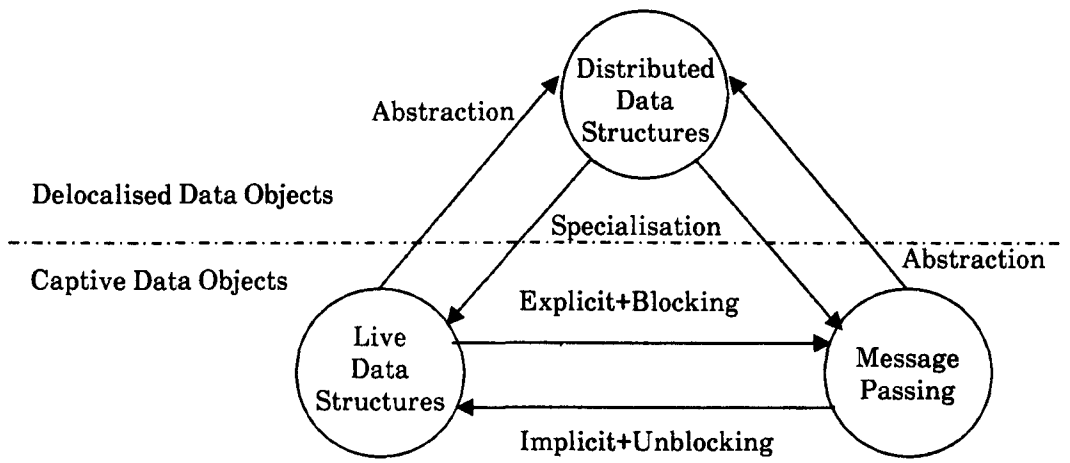



Figure 3.5 - Transforming parallel algorithms.

Practically, the changes that are made to ineffective parallel programs relate to matching the grain size of the parallelism executed by threads of control more closely to that preferred by the target architecture. On one hand, in shared memory machines, distributed data structure programs that lack parallelism can be recoded as live data structure programs because of the fast inter-thread communication mechanism shared memory provides. On the other hand, if the same problem were to be recoded for a distributed memory machine, message

passing may provide a better model because expensive interthread communications can be managed more carefully.

## Issues in Parallel Algorithms

Although classifying models for parallel algorithms can lead to clear ways of designing parallel programs, often important implementation details are glossed over. For example, whether a parallel algorithm is deterministic and if not, what this implies about the behaviour of the algorithm. However, several important characteristics of parallel algorithms can be identified and used to classify them. These properties are:

- the nature of the thread control structure (hierarchical or peer),
- the nature of the work allocation (static or dynamic),
- the nature of thread synchronisation (regular or irregular),
- the nature of the work (uniform or non-uniform),
- the extent of the parallelism available.

Note that the nature of the communication between threads (shared memory or message passing) is not present in the above list because it is an implementation technique covered in the previous section. In addition, there is no mention of processor utilisation in terms of useful work executed, because this is a system measurement and is not specific to an individual problem.

### Thread Control Structure

There are two methods for structuring threads of control in a parallel program. Firstly, if the threads are arranged in a master-worker configuration, this is a hierarchical structure with the master being responsible for creating, terminating and possibly synchronising its workers. Hence, the master thread usually has a more complex implementation than its lightweight children. Secondly, if there is no hierarchical relationship between threads, then they are peer threads. In practice, an algorithm that uses this form of control distributes responsibility for controlling a program over its threads. Programs of this kind often have good load balancing and fault tolerance properties because important worker threads can be easily replicated.

Not surprisingly many real life applications are best solved by combinations of hierarchical and peer thread structures. Examples of combined control structures can be found in recursive thread structures that are produced by unwinding recursion, and replacing it by a combination of recursion and thread creation. For

example, quicksort [Hoar62] can be parallelised in this way. However, if recursion is unwound then it is possible to create too many threads unless some throttling is applied, as the amount of useful work done by a thread can be less than the overhead needed to set it running.

Work Allocation

If a static work allocation algorithm is used as the basis for a parallel program, the number and function of its threads of control are fixed at compile time, resulting in a fixed thread graph. Programs of this form are easier to write and reason about than similar programs written using dynamic behaviour, as the interactions between threads can be predicted more easily. One implementation point is that, as almost all of the threads of control in a static model are created at the same juncture, less overheads can be incurred than if they are created in a piecemeal fashion. Furthermore, if the work can be evenly distributed over the workers and there is no external load on the system, then static work allocation is more efficient than dynamic allocation as there are less run time overheads.

In a dynamic work allocation algorithm, the number and possibly the function of its threads are determined at run time or can change during the execution of a program, resulting in a mutable thread graph. Algorithms of this type are more flexible and adaptable than static algorithms, but pay the price in terms of greater program complexity and potentially higher overhead costs. However, in cases where the amount of work cannot be determined at compile time or otherwise cannot be equally distributed among the threads of control, it may be such that the penalties of poor load balancing far outweigh the overheads of dynamic work allocation. This is also true for situations where a parallel program has to share resources with other programs. When this happens, static thread arrangements are limited by the speed of the slowest worker, while dynamic arrangements can compensate for external loading, provided they have a sufficiently elaborate work allocation mechanism. Examples where dynamic work allocation can be effectively employed are adaptive equation solving algorithms. These algorithms operate iteratively by homing in on a solution by focusing on an ever decreasing part of the original solution space. If only one division of the solution space were to be made then it is likely that towards the end of the algorithm only a few of the original threads of control would be actively employed. However, if the solution space is repartitioned at every iteration then the maximum number of threads can be employed for the entire duration of the algorithm.

Synchronisation and Work Distribution

Although these are separate issues it is constructive to combine them into a table and examine the types of algorithm model that are generated. Figure 3.6 is such a table and contains typical example algorithm models for each class. Basically non-uniform work distribution corresponds to functional partitioning (i.e. different threads doing different jobs) whereas uniform work distribution corresponds to data partitioning (i.e. different threads doing the same job).

| Work Distribution | Synchronisation Interval | |
|---|---|---|
| | Regular | Irregular |
| Non-Uniform | pipelines & networks | logical networks & transactions |
| Uniform | static data partitioning & SIMD | dynamic data partitioning |

Figure 3.6 - Classification of models for parallel algorithms.

In practice, the easiest algorithms to debug and code are those that operate synchronously with uniform work distribution, however, efficiently matching such algorithms to real applications can be quite difficult. On the other side of the coin, real applications tend to have natural analogues in asynchronous systems that combine threads of control with varied roles. In general, while it may be easy to design such processing groups, it is not easy to code and debug such systems because of the complexity which arises from executing multiple operations in parallel, and the resulting interactions between such operations.

Parallelism Extent

This is a dual measure of an algorithm combining its properties which reflect the number and complexity of the processors that could be usefully employed in its execution. One of the reasons why these factors have been joined is so that they mirror one aspect of the classification of multiprocessors as SIMD versus MIMD, i.e. very many simple processors versus fewer more complex processors. Likewise, even among MIMD machines similar comparisons can be made between microprocessor-based supercomputers and their rivals like the Cray Y-MP.

Parallelism extent is partitioned into five categories that are related to the computer architectures that are most appropriate for exploiting each extent of parallelism, without excessive multiprogramming. The categories are defined intuitively rather than by more common methods for measuring computational

power such as MIPS, MFLOPS and KLIPS. This is so because these measures are notoriously bad indicators of computer performance as there is no standard way for comparing the figures that are quoted. Moreover, figures can vary significantly between the normal rate (sustained) and that which only can be obtained under theoretical conditions (peak rate). Formal methods for quantifying the processing capacity of computing systems are emerging that permit architectural designs to be analysed for their operational characteristics [Lipo87]. However, such methods are not needed here because of the intuitive nature of the categories, but they may be of use if the classification is subsequently refined.

The main reason why different architectures appear in separate parallelism extent categories is that, in general, architectures cannot be effectively scaled up or down by arbitrary amounts. Hence, architectures only feature in the categories for which they were purposely designed. It is simply not possible to multiply up the number of processors and the amount memory in an architecture to get a more powerful one because physical limitations cannot be scaled so conveniently. For instance, bus-based shared memory machines are limited by the transmission rate of the bus (ultimately the speed of light), and distributed memory machines, such as hypercubes, are limited by the number of physical connections that can be made to a given node. Research is going on to build scalable multiprocessors out of replicated units with structures like those of trees to remedy these problems (e.g. NON-VON [Shaw84] and TRAC [Lipo87]), but as might be expected, writing parallel programs for these machines will not be easy.

In the limited parallelism category programs are mainly thought of as being sequential because they are based on sequential algorithms that provide little scope for explicit parallelisation. However, even in these programs some parallelism can be exploited even though a program is executed by a single processor. Uniprocessor architectures described in chapter two often include support for parallelism in the form of instruction pipelining and multiple functional units. For superscalar processors, optimising compilers can reorder instruction streams so that several arithmetic operations can be in progress simultaneously. This means that programs that were thought to be sequential or possess only a small number of independent instructions can be effectively executed by exploiting what limited parallelism is available.

The moderately parallel category covers the parallelism that can be exploited by up to say thirty non-trivial processors. Respectable performance gains over sequential programs can be obtained from parallel programming at this level and

many types of multiprocessor architecture are capable of providing these kinds of performance gains, ranging from groups of workstations to tightly coupled shared memory multiprocessors.

The parallelism in the highly parallel category is expressed in terms of tens of processors. Here, only hybrid and distributed memory machines are viable architectures, though there are plans for shared memory machines of this size. Specialist array and vector processors also operate at this level of parallelism, though at a much smaller level of granularity.

The very highly parallel category covers parallelism in the range of hundreds of processors. The machines that are capable of supporting this level of parallelism are distributed memory multiprocessors that offer supercomputer performance. Hence, these machines tend to be used for specialist numerical work or run more general work by being partitioned into smaller multiprocessors.

The machines that operate in the massively parallel category are generally radical departures from the von Neumann model, giving parallelism in terms of (tens of) thousands of processors. The processors found in such machines are very simple, with small instruction sets, and are capable of operating on very small pieces of data at a time (e.g. individual bits giving very fine grained parallelism). Such an architecture of this type is the DAP mentioned in chapter two.

## 3.4 Problems in the Parallel Execution of Programs

When designing parallel programming constructs it is important to be aware of the kinds of parallel processing errors that can arise. Two types of fault are linked particularly with parallel processing, which can be termed *performance faults* and *operational faults*. A performance fault causes a parallel program to execute more slowly than its algorithmic implementation would suggest. Sometimes, in very extreme cases, performance faults can affect the correctness of a program by highlighting limitations in time dependent synchronisation. For example, consider a program that makes use of message passing implemented with timeouts. Certain runs of the program could fail if random delays caused timeout periods to be unexpectedly exceeded. More significantly though, an operational fault such as deadlock can cause the complete failure of a program by stopping some or all of its threads of control from proceeding with their execution. Operational faults are really a subset of algorithmic faults, which are common to

both sequential and parallel programs, and correspond to algorithmic flaws in a program that prevent it from producing a correct result.

A large proportion of the time spent developing complex parallel software can be spent correcting performance faults. When a parallel program is designed, implicit assumptions are made about the lengths of time sections of code will take to execute, so that program threads can be efficiently distributed over system processors. If this estimation is done badly, then a parallel program can execute slower in parallel than it does sequentially. Even if a good allocation of threads to processors can be made, for many very large problems the speedups obtained by parallel processing begin to decline when tens or hundreds of processors are used. Many possible reasons for this exist, some of which (such as contention) are architecture dependent, but others come from the serialisation and overhead attributable to the operating system. In moderately parallel applications it is possible that this effect is not noticeable, but for highly parallel systems serious problems can arise, effectively putting a premature upper bound on the parallelism that can be exploited. Other performance related issues include creating the optimum number of threads for a parallel program, and choosing the most appropriate form of synchronisation for coordinating these threads of control.

If threads of control have unsynchronised read and write access to a shared resource, then a race condition can occur between such threads of control. Such a condition corresponds to an error in a program's logic where synchronisation should have been used to enforce consistency. Race conditions can lead to operational faults of which the most well known is deadlock. When this phenomena occurs, the threads that are deadlocked cannot make any further progress towards completion without external intervention. Deadlock has been researched for many years and the conditions for it arising are well documented [Coff71]:

- mutual exclusion,
- non preemption of threads in critical sections,
- waiting for other resources while in critical sections,
- circular waiting.

The two basic approaches that have been developed to handle deadlock are the techniques of *prevention* and *avoidance*. In deadlock prevention, the prevailing operating conditions of a concurrent system are such that some or all of the necessary conditions for deadlock are inhibited. Hence, it is possible to prove that any program operating in such an environment can never deadlock. When it is

undesirable or not possible to remove the necessary conditions for deadlock from a system, a strategy of avoidance can sometimes be used instead. In such a system, concurrent programs are executed under the aegis of some application specific rules. These rules ensure that a program can complete its job, but in so doing must avoid situations where deadlock could occur. For example, in problems of resource allocation (e.g. devices or memory) Dijkstra's Banker's Algorithm [Dijk65b] models the state of a resource by labelling it as either being in a *safe* or an *unsafe* state. If the resource is in a safe state, threads using it are guaranteed that they will not deadlock and will complete in a finite time (assuming that critical section rules apply). However, if the resource is in an unsafe safe, there is a chance that threads using the resource will deadlock if events occur in a particular sequence. Therefore, the Banker's Algorithm only grants those requests that result in the resource remaining in a safe state. Unfortunately, the algorithm only works under a strict set of rules, some of which are unreasonable for many applications. For instance, the Banker's Algorithm is unsuitable for programs that cannot, in advance, accurately predict their resource requirements, or that must adhere to strict time deadlines. Thus, if for some reason neither of the two basic techniques of overcoming deadlock can be used, *detection* and *recovery* mechanisms also exist to locate and break deadlocks, hopefully returning at least some threads of control to operating states.

Although deadlock is fairly well understood there is a similar problem called *livelock,* which is more insidious and occurs mainly in transaction systems. Consider a thread of control whose execution is progressing toward its goal, but discovers that it cannot complete its task because another thread holds some vital resource. In an attempt to resolve this conflict, the first thread then tries to recover by recommencing its execution. In some situations this strategy will overcome the problem, however, if the second thread turns out to need a resource held by the first thread, a livelock can occur. Here both threads need a resource held by the other, and both threads never terminate as they cycle endlessly trying to avoid a deadlocked state. As the threads of control do not stop executing livelock can be difficult to detect unless there is some mechanism for monitoring the progress of a program.

Even though a parallel algorithm may withstand careful scrutiny it can still fail due to problems of *thrashing.* Once again, threads do not necessarily cease execution but become so bogged down executing synchronisation or communication code that they cannot make progress towards their goals. For example, consider a producer thread that sends a stream of messages to a

consumer once the consumer signals that it is ready, and stops once the consumer signals that it is busy. If the producer sends messages too quickly, the consumer may not have time to signal to producer to stop sending messages, as it is too busy fielding the incoming messages. Clearly, thrashing problems are influenced by the timing and storage properties of the hardware environment in which programs are executed, bringing in an element of non-determinism into the execution of such programs. Furthermore, thrashing (and livelock) can occur through improper scaling of problems, in which programs may appear to work correctly by producing correct results in one configuration, but may fail in a larger or different configuration.

A more subtle operational fault is *indefinite postponement.* This relates to certain threads never getting an opportunity to execute due to some bias in a scheduling or work partitioning algorithm. Other terms that have been applied to this phenomena are starvation and fairness. In most cases, this problem is confined to system software, as thread scheduling is generally handled automatically and is not the responsibility of a user. However, unfairness can occur in dynamic algorithms, such as work allocation strategies that allocate work on a first-come-first-serve basis.

## 3.5 Requirements For Explicit Parallelism Constructs

Some years ago Andrews [Andr76] suggested that the following areas must be addressed when designing new parallel programming constructs:

- expressiveness,
- reliability,
- security,
- verifiability.

The expressiveness of a language must be sufficiently rich to enable a wide variety of operational policies to be naturally expressed. Thus, programmers should have the freedom to design programs using an appropriate paradigm, and not be constrained to using a potentially obtuse model for solving the problem in hand. In practice, this means being evenhanded in the provision of language constructs by not trying to build in features that preclude a more efficient implementation in alternative styles or omit them altogether. Furthermore, constructs should exhibit uniformity, whereby similar concepts have compatible representations and similar operations have the same general interpretation.

The reliability goal is directed towards the compile time detection of time dependent faults arising from parallel activity. This can be realised through the use of high level (structured) constructs which are suitably constrained in their capabilities, to exclude or minimise the effects of parallelism features such as dynamic process structures and non-determinism. The use of such features has to be restricted because static compile time checking is, on the whole, unable to accurately predict the errors that dynamic behaviour can lead to. Only the possibility of faulty behaviour can be detected, which if used as an indicator of faults would lead to the classification of many working programs as being incorrect. Moreover, in any case, no indication is given of the likelihood of an erroneous state being reached. In general, the laudable aim of compile time reliability is difficult to achieve for real-world programs. Even so, it can have an important structuring effect on programs, hopefully leading programmers to a better understanding of their code.

Early reliability checkers were used to detect multiple references to shared variables in the hope of highlighting faulty accesses. Two examples of compilers that enforce more elaborate reliability checks are Concurrent Pascal and Linda. In Concurrent Pascal, references to shared variables must only occur in special parts of a program (conditional critical regions), which enables the compiler to generate special code to enforce mutual exclusion. In the Linda compiler, simple matching is performed on the tuple communication primitives. This is to check whether tuples that are written by an output primitive can subsequently be read, and conversely, to see if tuples that are read, have at some stage been written. Tuple analysis of this kind is not rigorous, in so much that it simply looks for the existence of the primitives and does not examine actual program control flow to see if the primitives are executed in some correct ordering. However, quick checks of this kind are easy to implement and do pick up some of the more obvious programming errors.

The motivation behind the security (integrity) goal, is to enforce access restrictions and to offer a *guaranteed service*. Access restrictions are enforced to offer a *secure service* which prohibits unintentional accesses to shared data and excludes external tampering. Consider a server in a concurrent system that handles a device or manages some data. This server must offer a clearly defined interface to its resource, with there being no other ways of obtaining access. The idea here is to stop malicious or naive users from corrupting the shared data or simply viewing it when it is in an inconsistent state. Often the complexity of a concurrent system is such that there is no global understanding of the interactions

between its atomic parts. Thus, to counter such systems, in a guaranteed service, valid resource operations are expected to execute as specified, taking whatever actions are available to avoid errors and inconsistency resulting from possibly unforeseen circumstances.

Some of the checking that is needed to enforce access restrictions can be performed at compile time, for efficiency, and is related to the analysis employed to ensure reliability. However, other checking must be done at run time to handle dynamically occurring problems. Moreover, additional run time checking must be used to ensure a guaranteed service. This may take the form of arbitrating execution problems such as deadlock and livelock, handling exceptions, or enforcing policy decisions such as fairness on scheduling mechanisms.

The verification of a program is the process of proving its correctness by showing that it always meets its specification. As parallel programming becomes more commonplace, it is useful to be able to reason about the correctness of a program in a formal way. Certain language constructs make this task easier, while others make it more difficult. Constructs that encapsulate data, so reducing interactions and side-effects, are a boon to verifiability, while capabilities that allow the run time bindings of variables, such as passing procedures as parameters, are a hindrance. For example in the general case, programs written in CSP can be formally analysed, while those written in C cannot.

It is widely accepted that the ability to prove a program correct can be useful, giving a programmer confidence that a program will actually do what it is supposed to do. But, there is no general agreement about the relative importance of this program property with regard to other program properties such as efficiency and usability. When designing a parallel programming style or model it is advantageous not to preclude the ability to perform a formal analysis, but in all probability the force of the other considerations will win out.

While it would be nice to believe that it is possible to evenhandedly satisfy all of the requirements set out for parallel programming mechanisms, experience has shown that language designers must trade off one requirement against another. Thus, one conclusion that can be drawn is that parallel programming may be best served by a collection of complementary mechanisms, with each mechanism focussing on doing its job as well as possible. Following this policy, it should be possible to develop simple and intuitive programming abstractions that can be

used for a large proportion of parallel programming (easing the current situation), but nevertheless are not all things to all men.

## 3.6 Methods of Exploiting Parallelism

From the material that has been presented in this and the previous chapter, it should be apparent that there are many ways in which parallelism can be exploited. Even when considering only one parallel architecture, i.e. shared memory multiprocessors, the choice is not made much simpler because many of the software methods for exploiting parallelism are independent from the underlying hardware. To summarise, to address the problem of exploiting parallelism described in this thesis one could consider the following:

Auto-parallelising Solutions

- writing tools to transform sequential programs into parallel programs,
- devising abstract machines (interpreters) to execute programs in parallel.

Programming Language Solutions

- writing new programming languages (explicitly parallel, implicitly parallel or sequential),
- devising specialist parallel programming languages for use with existing languages.

Hybrid Solutions

- using graphically based tools for program construction, that automatically generate the parallelism code required to implement a given design.

Software Libraries

- the utilisation prewritten parallel codes.

Auto-parallelising Solutions

The auto-parallelisation of imperative programs at first seems a painless way of exploiting parallelism in a convenient and portable manner. The fundamental technique of auto-parallelisation is vectorisation, whose success relies on three factors: (i) the code is vectorisable, (ii) the compiler can spot this, and (iii) the target hardware has appropriate instructions to exploit the parallelism. Thus, vectorisation (and its associated techniques) cannot guarantee effective parallel

programs, because such techniques are essentially a series of stepwise transformations that produce concurrent programs by starting with sequential programs.

Auto-parallelising interpreters (for declarative languages) similarly rely on three factors: (i) there is parallelism implicit in a program (algorithmic parallelism), (ii) the available parallelism can be exploited by the interpreter's model of parallelism, and (iii) the grain size of the parallelism exploited by the interpreter can be efficiently matched to its target architecture. Hence, the limitations of these techniques are similar to those encountered in vectorisation.

One of the characteristics of auto-parallelisation is that it hopefully leads to the exploitation of parallelism via an existing sequential programming model. On one hand this makes the job of the programmer easy and the job of the parallelising tool writer quite hard, as the parallelising tool has to do all of the work of detecting and exploiting the parallelism. On the other hand, however, a programmer may have chosen an algorithm that has very limited parallelism, giving an auto-parallelising no scope to work, where in fact a parallel algorithm could have been used instead. Thus, to promote effective parallel software programmers have to be steered towards using parallel algorithms and one of the most successful ways of doing this is by providing suitable programming constructs.

Programming Language Solutions

As the scope of programming language design encompasses far more than issues of parallelism, the task of defining the definitive parallel programming language cannot be undertaken in isolation from other programming language research. Over the last twenty years many programming languages have been proposed that either use parallelism as a feature, or were purposely designed to exploit parallelism. Unfortunately, as people have continued to use more traditional languages or moved to other languages (e.g. object oriented languages) much of the impact of these parallel languages has been lost, although some of their ideas for parallelism mechanisms have been perpetuated in later languages. Thus, for this thesis, while it would be desirable to propose a complete programming language, it is somewhat impractical due to the large volume of work that would be involved.

At the current time a large proportion of parallel programming is undertaken using sequential languages which include parallelism extensions in the form of

library calls and compiler directives. Extensions have been made to both declarative and imperative languages. For example, in the case of Fortran extensions have been made in an disorderly fashion which has led to many different but similar language dialects. While diversity of this kind encourages competitive research, in many situations effort is wasted in producing equivalent but incompatible constructs. A more productive approach to extending existing languages is to define a specialist parallel programming language (e.g. Linda), and then apply its use equally when it is used in conjunction with its host languages. Hence, this route to parallelism has the attribute of uniformity across a range of languages, and also means the programmers can continue to use familiar languages (which may be the most suitable for the job in hand). Thus, the parallel programming constructs that are developed in this thesis are intended to be viewed as a parallel programming sub-language, that can be used with existing imperative languages. Nevertheless, at some later time it may be expedient to design a complete programming language, purpose built, to harness the new parallelism constructs.

Imperative languages are used as the host languages for this research because the work is centred around producing shared data structures that can be manipulated in parallel. This kind of shared memory programming model is the one which is most suitable for efficient execution on shared memory multiprocessors (and is also quite efficient for hybrid architectures as well). In addition, the work carried out into data structuring techniques (in object-oriented systems) seems to be an important programming area and using imperative languages allows the freedom to explore various parallelism constructs. Having said this though, declarative languages and styles of programming offer high levels of abstraction that are liberated from the details of the underlying hardware. Moreover, declarative programming can naturally lead to programs that exhibit high locality of reference and low coupling between procedures, which means that such programs can be executed very efficiently taking maximum advantage of hardware facilities such as caching. Thus, while imperative languages are used as the staple programming languages for the research in this thesis, the influence of declarative languages on the form of the parallelism constructs is evident.

### Hybrid Solutions

Software tools are an important aspect of a programming environment, but nonetheless at the current time programming languages seem to be a more

important area for parallelism research. At some time in the future, programming metaphors (bases for programs) may be able to be expressed visually, without recourse to any code, and at that time parallel programming will be liberated from textual programming languages. However, in the short term programming language development seems to be the best way to evolve parallel programming styles for the future.

Software Libraries

One method of creating a parallel program is to make use of prewritten library routines, devised by an expert, that execute in parallel. A good example of an application of this technique is the Lapack numerical library [Demm87]. Many key numerical analysis functions are encoded within the library, and so for many scientific problems some parallelism will be automatically exploited when library functions are invoked. Using software libraries is a very attractive approach to parallelism because it offers a quick route to often very effective parallel programs. However, libraries are not without their faults as tradeoffs must be made in their implementations because of the general lack of portability between different types of parallel architecture. Moreover, the programming effort to construct a library may be considerable as libraries may have to be modified for each new machine that is supported (even if they have similar architectures). In addition, the use of parallelised library routines written in imperative languages can lead to nasty side-effects when the routines interact with the rest of a program. Errors such as these can be very unpredictable and difficult to pin down, especially if a programmer does not have a good working knowledge of how the parallelised library works.

At the current time, large standard software libraries are used for specialised computing concerned with numerical analysis or computer graphics. With advances in software engineering it may be possible in the future to maintain libraries of parallelism routines with general purpose capabilities to widen the accessibility of this route to parallelism. In addition, the notion of software reuse itself is one that some people believe is a ready answer to some of the problems in engineering large pieces of software - saving both time and money. Nevertheless, the problem remains that libraries must be written in some programming language, so it is the implementation language of a library that must overcome most of the problems associated with parallel processing.

# *Chapter 4*

# Shared Values

Over the course of the previous two chapters aspects of parallelism ranging from multiprocessor hardware to parallelism constructs and parallel algorithms have been discussed. Much of this information is generally applicable to the study of parallelism, though some of it is more specifically aimed toward specialised multiprocessor architectures and parallel language approaches. The focus of this chapter is to present original material on the concept of using *shared values* as a method of explicitly exploiting parallelism in programs. The ideas behind the semantics of shared values are drawn from a diverse range of sources and as a result, the use of shared values is thought to be widely applicable to parallel programming rather than being just an approach specialised for certain types of applications. Likewise, from a hardware perspective, although shared values are most suitable for use with shared memory multiprocessors, there are no fundamental reasons why they cannot be used successfully with other multiprocessor architectures.

Thus, the main body of this chapter is devoted to an explanation of the operation of shared values. This explanation is opened by a short overview of the ideas behind the use of shared values and considers the types of problem that shared values are designed to solve. Following on, a description of the semantics and syntax of shared values is presented, interleaved with examples explaining how shared values can be used for parallel programming. This description is the basis of the *Tyger* parallel programming model, which is the model of parallelism specifically developed to support shared values. Finally, the closing section of this chapter departs a little from the shared value theme. It tackles some of the problems in the production of effective parallel software, not addressed directly by shared values, taking the form of a general outline of the remainder of the system software envisaged for a complete shared value environment.

## 4.1 Philosophy of Shared Values

One way in which parallelism can be exploited by multiprocessors is through the expression of application programs in terms of groups of cooperating threads of control. As noted in earlier chapters, two strategies for creating parallel programs are possible. Firstly, explicit parallelism constructs can be written directly into

programs to force parallel execution. Secondly, software tools can be used either to analyse a program and automatically generate the parallelism code (e.g. vectorisation), or be used to execute a program in a way that is naturally parallel, with only algorithmic dependencies controlling the exploitable parallelism (e.g. dataflow). Nevertheless, no matter which method is used to exploit parallelism, effective parallel programs cannot be produced by arbitrarily partitioning programs into multiple threads of control.

In point of fact, any program can theoretically be partitioned such that there is one thread of control per instruction. If such a program is executed correctly in parallel on a shared memory multiprocessor (as opposed to pseudo concurrent execution), in all but trivial cases coordination between threads is necessary to reflect the dependencies between statements, to ensure correct instruction sequencing, and to manage the execution of the threads themselves. When these overheads for communicating data and synchronising multiple threads of control are taken with the overheads for thread creation and management, such overheads will more than outweigh the useful work performed by such a program. From the viewpoint of efficiency, this makes parallel execution of this kind very unattractive. A more likely scenario for parallel execution, therefore, is that the number of actual threads of control is bounded by the number of processors on a multiprocessor and not only by the problem size. Thus, although conceptually there may be many threads of control in a program these will be partitioned for execution by a smaller number of actual threads of control. For example, this technique is essentially the one followed by loop spreading when simulating vector operations, and while it is somewhat better than the totally parallel approach it can still be inefficient if an allocation of work to an actual thread of control is smaller than the time taken to make the allocation. Thus, if a program is analysed, certain activities may be identified as being suitable for parallel execution according to some efficiency criteria, and it is only these activities that are partitioned and executed in parallel, the rest being executed sequentially.

If lightweight threads of control are employed to execute a parallel program, thread management overheads can be kept to a minimum. More importantly though, the amount of useful work computed by a thread should exceed the amount of work taken to create and manage that thread, so that some performance benefit will be obtained if sufficient numbers of threads can be utilised. Ideally, if there are no interactions between parallel threads of control the maximum efficiency benefit can be derived from their parallel execution. That is, no time is wasted on the overheads of communication and synchronisation, and

as a result, the ratio of useful work performed by a program to its parallelism overheads is as high as possible. In many programs, however, assuming there are no thread interactions is impractical as few applications can be decomposed into completely independent parts. Many parallel programs are composed from threads that interact with each other by exchanging information and synchronising their execution, producing complex overall patterns of control and data flow. Interactions between threads of control can occur for many reasons, though these can be linked to the low level types of dependency between statements identified by Kuck (see chapter two). However, at a conceptual level, threads can interact by transferring program data, by propagating thread control information (synchronising), or by a combination of both (e.g. as in message passing). Thus, to produce effective parallel programs, the aim of a parallelisation strategy is to yield sufficient threads of control to enable allow a reasonable number of processors to be applied in the execution of a program, but with the proviso that the threads of control should interact as infrequently as possible. Moreover, as an aid to designing and debugging parallel programs it is advantageous if the patterns of communication and dependency between threads of control are regular and one directional.

To recap, parallelism can be obtained from a program by partitioning its data structures such that worker threads, perhaps independently, compute their own parts of some result. Techniques which have taken up the data parallelism approach include SIMD processing (e.g. vectorisation) and at a higher level of granularity MIMD processing (e.g. loop spreading). Data parallelism can be viewed as a natural extension to von Neumann programming that redresses some of the unnecessary sequentiality that is propagated by conventional programming languages when operating on regular data structures. As an alternative paradigm, parallelism can also be obtained by distributing the function of a program across a group of threads, with each thread performing some unique transformation on its data. More often than not, however, threads have to communicate with each in order to produce a result. A good example of functional parallelism exploited in this way is pipelining. No clear cut rules exist for clarifying which decomposition is preferable for an algorithm, as either decomposition can sometimes be used successfully, or occasionally they can be combined and used together. However, in some cases, the use of either approach can lead to parallel solutions that contain a large degree of parallelism overhead, in the form of communication and synchronisation, which may prove unacceptable for certain applications.

One factor that is important in the design of parallel algorithms is the regularity of the work performed by the threads. Analysis of a program can be made easier if its threads of control perform similar functions. Moreover, specifying that hundreds of threads perform one function is somewhat easier than expressing that hundreds of threads should perform different functions. In the same vein, parallel algorithms based on data partitioning have been shown to exhibit scalable amounts of parallelism. In chapter three, Figure 3.1 showed a map of granularity versus number of threads of control. One of the main points contained in this map is that, in general, data parallelism provides more scope for effective parallelism than functional parallelism. This stems from the observation that as problems are scaled in size, so increasing the size of their data structures, more data parallelism can be applied, data dependencies permitting. However, in the case of a program that exploits functional parallelism, increasing the size of a data structure, does not necessarily increase the scope for parallelism, as it can only be increased if the program can be decomposed into smaller functional units. Irrespective of whether such a decomposition can be performed, undoubtedly the repeated functional decomposition of a program will lead to threads of control which have complex interdependencies and correspondingly complex interactions.

For a significant number of applications, exploiting parallelism in a program by data partitioning is a technique that can provide a sufficient number of realisable threads of control to enable worthwhile performance benefits to be obtained. Moreover, these threads of control may well have few interactions, leading to small parallelism overheads. Hence, the objective for shared values in the *Tyger* programming model is the provision of an elegant notation for expressing parallel activity, that pays particular attention to the efficient exploitation of data parallelism.

## 4.2 Tyger Parallel Programming Model

A Tyger program starts off as a single thread of control executing in its own address space called a *shared value space* (SVS). Over the course of its execution, a Tyger program may consist of many concurrently executing threads of control, spawned from the original thread, that can communicate with each other by reading and writing shared values in their single communal SVS. There are two fundamental types of shared values called *static* and *dynamic* shared values. Static shared values can be thought of, and declared in the same manner, as conventional variables, but with the proviso that they are automatically shared amongst a group of threads of control that share a SVS. Moreover, static shared

values have single assignment semantics which allow them to act as an interthread communication and synchronisation mechanism. Conversely, dynamic shared values are not rooted and lexically scoped to any section of program code like conventional variables, but instead are passive free agents created at run time. After their creation, dynamic shared values reside in the SVS during the execution of their parent parallel program, and can be referenced associatively and manipulated by Tyger constructs in a manner similar to Linda tuples (see chapter two).

Shared values and their associated constructs form the programming part of the Tyger model, but the model extends to cover part of the thread implementation and operating system's view of a parallel program. Information on the structure and external appearance of a group of threads that forms a Tyger program is presented in the *global environment* section towards the end of this chapter.

One important axiom of the Tyger model is that the process of identifying and coding parallel execution should be an explicit process, deriving benefits from a programmer's experience and knowledge of his intentions regarding how a program should operate. One consequence of this axiom is that a programmer could have in mind the effect parallel execution will have on the performance of a program, to enable performance tuning of the program if desired. However, this explicit approach to parallel programming is qualified by the expectation that software tools, primarily compilers, will off-load many of the details regarding the implementation of parallelism, leaving programmers to concentrate on designing and expressing parallel algorithms. Of course, this is easier said than done, but as shared value constructs are geared towards exploiting the regularity offered by data parallelism, the implementation of parallel threads is made much easier than if shared values constructs claimed to efficiently support all types of parallel programming. Thus, the Tyger model can offer a high level view of parallelism to programmers by concentrating on supporting one important type of parallelism very well, and in so doing does not have to make unreasonable requirements for any underlying implementation.

The Tyger model is intended as a vehicle, or pool of ideas, for exploring ways in which parallelism can be exploited in imperative programs through the use of shared values. The model is evolutionary in nature, changing with the introduction of new ideas, rather than being a fixed concrete model. The decision to make the Tyger model mutable was taken to reflect the fact that the provision of programming constructs is always a compromise between design

considerations, therefore making it difficult to arguably obtain the best suite of constructs at the first attempt. The Tyger programming language has been developed to, test out practically, the idea of using shared values for parallelism put forward in the Tyger model.

The Tyger language is a compact parallel programming language that deals exclusively with parallel programming, and hence, has to be embedded in a host language for general use. The execution model followed by the Tyger language is based on procedural control flow, so suitable host languages are imperative languages such as Fortran, Algol and their derivatives. The objective of the Tyger language is to describe parallel control flow and information sharing relationships, and in consequence, the Tyger language cannot be packaged as a black box that can be retrofitted blindly to any imperative language. Tyger mechanisms have to be carefully integrated into a language, to ensure uniformity of operation across Tyger constructs and thereby ease of use, and cannot be thought of as merely an adjunct. In practice, this may mean having to compromise some of the Tyger constructs or restrict some of the host language features. As an example, consider the implementation of Tyger constructs in an object-oriented language such as C++. Here, one reason for using the Tyger constructs would be to allow objects implemented with shared values to be operated on in parallel. Object-oriented languages are based around the manipulation of objects created to a pattern defined by their class. C++ allows the declaration of static data which is shared between all objects of a class by maintaining only a single copy of this static data and making all objects of the class refer to it when necessary. If a C++ program that exploits this feature is executed concurrently, then problems can arise with sharing and synchronising accesses to this static data that do not arise in a sequential execution. This is because static data is implicitly shared and unless explicit steps are taken to share and synchronise accesses to it, programs that use static data cannot work correctly. However, due to the nature of information structuring and hiding used in object-oriented languages, explicit changes cannot always be made to references to static data as the implementation of a class is hidden from its users. Hence, the only safe solution is to disqualify the static data feature from the language. Thus, although it would be convenient to think of the Tyger language as being self contained, restrictions may have to be placed on the host language, and details of the Tyger language itself may vary, according to which language is acting as the host.

Two languages have been chosen as host languages for Tyger constructs to date, C and C++. These languages were chosen as examples of imperative

languages that are in common use but have received little attention from the point of view of automatic parallelisation, therefore requiring any exploitation of parallelism to be done explicitly. This situation has arisen because programs written in C and C++ can suffer from the aliasing of variables and can make considerable use of pointers, which makes the dependency analysis for automatic parallelisation very difficult. This fact has been recognised by many other researchers who have produced explicit parallel programming tool kits in the form of thread libraries for these languages (see chapter two). This work has been useful for carrying out research into parallel programming, but in the main has been too low level when compared with other more integrated approaches followed by other parallel object-oriented languages (e.g. Argus) and parallel declarative languages (e.g. Strand). Thus, Tyger constructs are a method for building on the earlier work done with threads libraries, in order to provide much higher levels of parallel programming abstraction for C and C++ that can rival other high level approaches to parallel programming.

### 4.2.1 Programming Language Constructs

When designing a parallel programming language, three types of programming mechanism have to be addressed: (i) how multiple threads of control are created; (ii) how they communicate; and (iii) how they are synchronised. The Tyger parallel programming language supports the creation of multiple threads of control via parallel iteration constructs to enable data partitioning, and by a general parallelism operator to support functional partitioning. Communication between multiple threads of control is realised by shared values, which are also the method of synchronisation. Shared values come in two varieties, static and dynamic, with static shared values being the most influential part of the Tyger model.

### 4.2.2 Static Shared Value Semantics

A single assignment variable has the property that it can only be bound to a value (assigned) once in its lifetime during the execution of a program. This property is exploited by dataflow systems as a method for controlling program execution, with operations being executed as soon as the values of their operands are made available. The idea of controlling program execution by data availability, however, is not confined to dataflow languages (e.g. it occurs in Strand) and also can be found in languages that make use of synchronous message passing such as CSP (see chapter two). In CSP, if a thread wishes to send a

message to a companion, it must wait until the companion is ready to receive it. Similarly, if a thread wishes to receive a message it must wait until one has been produced. These synchronisation semantics for the flow of data are more specialised than those for one directional dataflow as not only must a consumer thread wait until its data is available, but also a producer thread must wait until its consumer is ready. Thus, dataflow semantics can be used to provide a dual mechanism serving for both interthread communication and synchronisation.

One of the goals of the Tyger model is to produce an interthread communication mechanism that is both elegant and efficient. Shared memory provides an excellent starting point for efficient communication. However, detailed synchronisation code is sometimes needed to ensure that operations occur in a correct sequence. Consider an elementary example of two threads of control using shared memory to communicate, operating under the classic writer to reader relationship. In Scenario 0 no explicit synchronisation construct is used to enforce correct program execution, though is it assumed that operations on shared memory locations are executed atomically. The shared data in this scenario are the shared variables *X* and *XControl*.

```
// Scenario 0
int     X, XControl = 1;
Writer()                          Reader()
{                                 {
    X = 100;                          do {} while (XControl != 0);
    XControl = 0;                     /* use the value of 'X' directly */
}                                 }
```

When the *Writer()* and *Reader()* procedures are executed by separate threads of control, the thread executing the Reader() procedure will be held in a loop until the value of *XControl* is changed by the writing thread. Further on in the execution of the reader the value of X will be used directly in some computation. One of the drawbacks in using the above synchronisation scheme is that two shared variables have to be created, where conceptually there should only be one. A more serious flaw, however, is that the semantic link between a synchronising variable and a data variable is one made entirely by the programmer, therefore, opening an avenue for programming errors. More specifically, one of the common errors in using this form of synchronisation is that program statements can be ordered incorrectly causing synchronisation errors, and possibly leading to program failure due to deadlock.

In Scenario 1, locking is used to control access to shared data in much the same way as before, though this time resulting in a simplified Reader() procedure. The shared data in this scenario consists of *X* and a lock *XLock*.

```
// Scenario 1
lock    XLock = locked;
int     X;

Writer()                        Reader()
{                               {
    X = 100;                        lock(XLock);
    unlock(XLock);                  /* use the value of 'X' directly */
}                               }
```

The interpretation of the execution of the new program is exactly as before, with the benefit of using locking being that more of the sense of the communication between the threads is captured. However, locking is a very powerful, yet unstructured, synchronisation mechanism so it can be difficult to use in practice, and faces problems similar to those outlined in Scenario 0. An aesthetic improvement over Scenario 1 is to use an event based mechanism for synchronisation. The use of an event mechanism provides a more tailored expression of the synchronisation between the threads of control than does locking, in that the reader waits for the event that corresponds to the arrival of a value for X. The syntax for an implementation using an event is illustrated in Scenario 2.

```
// Scenario 2
event   XSignal = cleared;
int     X;

Writer()                        Reader()
{                               {
    X = 100;                        wait(XSignal);
    signal(XSignal);                /* use the value of 'X' directly */
}                               }
```

Once again the same interpretation of the program applies, though normally a thread is suspended and does not consume processor cycles while it waits on an event. This process of program refinement could be continued using mechanisms such as barriers and monitors, however, two fundamental problems will remain. While events (and their ilk) provide an intuitive method of synchronisation for this problem, it is still possible to make programming mistakes when ordering signal() and wait() operations. Moreover, each of the approaches described need to have a synchronisation variable explicitly initialised to a correct state before execution of the procedures is commenced. If a message passing mechanism is used to communicate information between writer and reader, however, the former

problem is alleviated as communication and synchronisation are enmeshed into a single pair of commands. Furthermore, no explicit initialisation may be needed, although a programmer must ensure that there are no messages already present on the communication channel. The syntax of such a message passing arrangement is illustrated in Scenario 3.

```
// Scenario 3
channel XChannel;

Writer()                              Reader()
{   int X;                            {   int Y;
    send(XChannel, X = 100);              receive(XChannel, Y);
}                                         /* use the value of 'X' indirectly by referencing Y */
                                      }
```

When the *Writer()* procedure is executed by a thread of control, it puts a copy of the value of *X* on the shared communication channel (*XChannel*) and then continues execution (with the channel buffering the value in a FIFO queue). When the *Reader()* procedure is executed by a thread of control, that thread will be suspended until a value can be obtained from XChannel. Once a value has been made available it is removed from the channel and bound to the local variable Y, with the reader then continuing and using the value of Y in some subsequent computation.

Message passing neatly combines communication and synchronisation by replacing direct sharing semantics with copy semantics. However, although one source of programming errors is removed another source of errors is created, as there is the need for a programmer to deliberately partition the set of program variables between threads of control. Moreover, consideration must be given to how data is to be explicitly moved between threads of control when it is required. The tasks of partitioning and designing protocols for moving data around in message passing systems can be an integral part of problems that require a model of physical distribution (e.g. fault-tolerant program replication across multiple machines), but for problems that use a model of conceptually shared data, such tasks are just extra work for programmers.

One of the aims of using static shared values is to combine the immediacy of access found in shared memory systems with the compact synchronisation mechanism found in message passing systems. Hence, static shared values combine aspects of dataflow synchronisation behaviour with shared memory accessibility. In the single assignment model utilised by dataflow systems, only one assignment can logically be made to each variable. Further assignments to the same variable are treated as invalid and are disallowed; this rule is also

followed for shared values. Thus, it may seem that static shared values are nothing more than single assignment variables, though while this is partly true, shared values exhibit some semantic differences that will be described in coming sections. Scenario 4 illustrates a static shared value solution to the writer to reader problem.

```
// Scenario 4
SV int X;
Writer()                          Reader()
{                                 {
    X = 100;                          /* use the value of X directly */
}                                 }
```

A static shared value *X* is declared as shared to the *Writer()* and the *Reader()* procedures. When the Reader() procedure is executed by a thread of control, that thread will execute until it needs to use the value of X, at which time, it will be suspended until the value of X is available. As soon as the value of X is determined, it will be used immediately by the reader in some calculation. When the Writer() procedure is executed by a thread of control, the thread assigns a value to X, thereby making this value available to any thread that needs it, and so releases any suspended threads. Thus, the use of a static shared value permits a once-and-for-all communication between threads of control, as the value held by a shared value cannot be changed once is has been set, though it can be read as often as required.

## Shared Value Description

A simplified view of the Tyger programming model permits the existence of two elementary types of variables. Firstly, there are those that are treated as conventional variables, being local to the thread that uses them and not shared between multiple threads of control (i.e. traditional variables). Therefore, there are no synchronisation constraints on operating with these variables, and no race conditions or thread interleavings to be considered. Thus, for efficiency reasons, it is safe to reassign these variables when necessary as this cannot affect other threads of control. Secondly, there are shared values, which are shared between threads of control, acting as an interthread communication and synchronisation mechanism, that can only be bound to a single value. An attempt to bind another value to such a variable will raise an exception, while references to an unbound shared value result in the suspension of the referencing threads until a value is made available (i.e. the shared value is bound to a value). Thus, the name "shared value" represents that fact that only write-once values can be shared between

threads not rewriteable variables. One important point to note regarding the notion of sharing of data between threads of control is that it is selective and modular. That is, shared values do not have to be declared as global variables and can appear as local variables that a thread can share with its children, via a parameter passing mechanism.

Using single assignment variables as a communication and synchronisation mechanism seems an attractive way of extending imperative languages to encompass parallelism. However, there are two drawbacks in using the basic single assignment scheme. Firstly, there is the matter of storage management. As many potentially parallel programs are based on the manipulation of large data structures, making new versions of such data structures because of repeated alterations to certain parts of the structures could be prohibitive in terms of the memory required unless adequate techniques for the reclamation of unwanted storage are available (garbage collection). Secondly, as many imperative programs are based on manipulating data structures by using iterative rather than recursive methods, the naming of variables can be problematical in the case of multiple assignment to temporary or intermediate variables. In addition, the reuse of storage is one of the strong efficiency points of the von Neumann architecture, from which shared memory multiprocessors are derived, and should be given due consideration. Furthermore, from the point of view of time efficiency, maintaining multiple copies of data structures may lead to excessive copying overhead, which can have a disproportionately negative effect on parallel program performance if, as a result, the processor-to-memory communication links become saturated.

To ease the transition to the single assignment property of variables in imperative languages, the Tyger language offers a compromise between using single or multiple assignment values to support thread interactions, combatting the storage and naming problems mentioned before. The key observation which is exploited in the Tyger model is that much of the parallelism in a program can be exploited in a series of discrete bursts of parallel activity. For example, a program starts executing with a single thread of control, performs some work involving multiple threads of control, but returns to a single thread of control to perform some final work. This simple example can be generalised to encompass programs that contain many bursts of parallelism, returning to sequential activity after each one.

A Tyger program consists of a sequence of intervals, called *stripes*, that correspond to alternate blocks of program statements in which a program is either operating in a sequential or concurrent manner. Stripes are delimited by parallelism constructs which change the state of a program from being sequential to parallel and vice versa, though otherwise need no explicit denotation. The Tyger model defines that it is not possible to have consecutive sequential stripes, as these are coalesced into a single sequential stripe, nor is it possible to have consecutive parallelism stripes. The former observation is quite natural but the latter requires a little explanation. At the very start and very end of a parallelism stripe there is only a single thread of control, any others having terminated. Thus, the end of one parallelism stripe and the start of the next, delimits a sequential stripe that may, or may not, contain any user program instructions.

The reason why a Tyger program is cast into this seemingly strange mould of alternate sequential and parallel intervals is to allow the reuse of the communication and synchronisation mechanism provided by static shared values. During a sequential stripe, there is no need for the communication and synchronisation semantics of a shared value to be in effect, so it can behave precisely like a conventional variable, with the capability to be assigned multiple times. However, when a parallelism stripe commences, all shared values take on an uninitialised state so that their dataflow semantics can come into force, allowing them to act as an interthread communication mechanism. When a parallelism stripe terminates, that is all but one of the threads has terminated, a shared value reverts back to being a normal variable, ready for immediate reuse or for reuse in a subsequent parallelism stripe.

The stripe model of parallelism employed by Tyger programs envisages that conceptually there is one master thread of control that persists throughout the execution of the entire program. This thread of control represents the activation of a procedure which corresponds to the main body of the program, and is called the *spine* (similar to the term used in graph reduction). There are Tyger constructs that allow the activation of procedures as new threads of control, which execute in parallel with the parent main thread of control. However, the parent cannot terminate the execution of a thread creation construct until all of the newly created worker threads have themselves terminated. Figure 4.1 shows the relationship between a thread diagram on the far right, and its corresponding sequence of Tyger stripes, for a Tyger program consisting of five stripes.

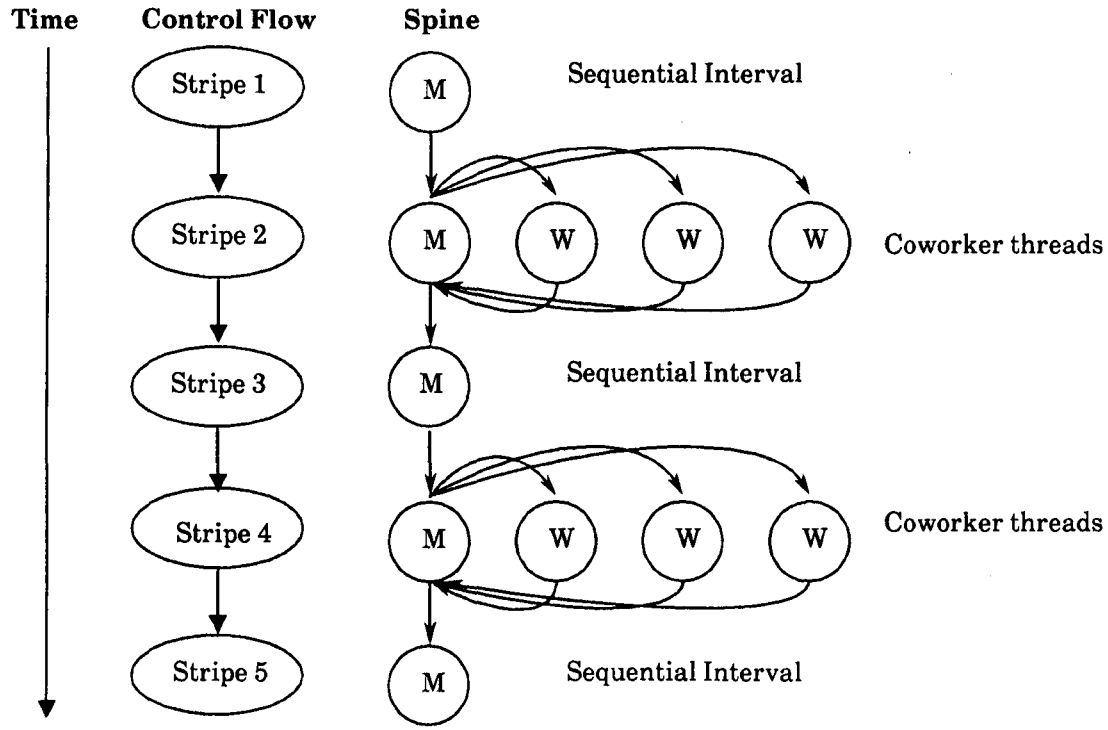



Figure 4.1 - Tyger stripes.

As an example main program, the following code fragment describes a Tyger program that consists of three stripes. An opening sequential stripe in which an initialisation routine (*doMasterStartUp()*) is executed, a parallelism stripe where *doMasterWork()* runs in parallel with *doSlaveWork()*, and a closing sequential stripe in which a concluding procedure (*doMasterCloseDown()*) is executed:

```
void    main()
{
    doMasterStartUp();                  /* sequential stripe */
    doMasterWork() |&| doSlaveWork();   /* parallelism stripe */
    doMasterCloseDown();                /* sequential stripe */
}
```

The " |&|" symbol is called the *parallel-and* operator and is one of the Tyger thread creation operators. In the above program it will cause the creation of a new thread of control to execute the doSlaveWork() procedure in parallel with the doMasterWork() procedure, which is executed by the master thread of control. Furthermore, in common with all Tyger parallelism thread creation operators, the parallel-and operator requires the master thread of control to wait, if necessary, until the worker thread of control has terminated. Further information regarding this operator is presented later in this chapter.

Parallelism stripes can be viewed as a series of fork and join statements in which all of the workers are forked at the beginning and joined at the end of each parallelism stripe. In the current Tyger model, a program can contain only one spine, which means that worker threads cannot themselves create other threads to form substripes. (The reasons for this restriction are presented later in this chapter.) The rules for the creation of threads of control imposed by the Tyger model constrain the way in which parallel programs can be structured, by making them express parallelism in a number of discrete parallelism stripes. Moreover, the threads of control in a parallelism stripe take the form of an asymmetric master-worker configuration. However, as subsequent example code fragments show, these restrictions do not unduly detract from the expressive power of the Tyger constructs, as many parallel applications can be expressed in a single parallelism stripe. Furthermore, the master thread can perform useful work during a parallelism stripe and does not have to remain idle waiting for its workers to finish.

### Static Shared Value Histories

As the Tyger model of parallelism allows static shared values to be reused over the course of a number of stripes, it may be useful to examine a past value held by a shared value. Such contingencies arise in iterative numerical applications when the next state of a variable is computed as some function of its previous states. This need to access past state information has been recognised in other languages based on single assignment such as SISAL [McGr83, Feo90], where there is an operator to retrieve the previous state of an iterator variable. In the Tyger model, a past value taken by a shared value, *sv*, in a previous stripe can be accessed by encapsulating references to sv in the function *old()*. This function takes a shared value as a parameter and returns a shared value as a result. The interpretation of what the old() function returns is dependent on whether it is invoked from a sequential stripe or a parallelism stripe. Consider a thread $T_0$ operating in a sequential stripe with a shared value sv which is acting as a normal variable. The following statements are always true:

| | |
|---|---|
| $T_0$ reads sv | ⇒ $T_0$ obtains the current value of sv. |
| $T_0$ reads old(sv) | ⇒ $T_0$ obtains the previous value of sv, otherwise an exception is raised. |
| $T_0$ binds sv to a value | ⇒ sv is bound to a new current value. |
| $T_0$ binds old(sv) to a value | ⇒ an exception is raised. |

As with normal variables the current value of a static shared value can be inspected, even in cases where a shared value may be undefined when it has yet to be assigned. Thus, intuitively, the old() function operates by returning the previous value held by a shared value, or by raising an exception when this value does not exist or when an attempt is made to alter it.

In a parallelism stripe the old() function operates in the following manner. Consider a group of $n$ threads $T_0 .. T_n$ executing in a parallelism stripe. The role of $T_0$ is to produce a shared value, *sv*, that the $T_i$'s ($0 < i \leq n$) will use in their own computations, with the $T_i$'s possibly binding other shared values as well. The following statements are always true:

| | |
|---|---|
| $T_0$ binds sv before $T_i$ reads sv | $\Rightarrow$ $T_i$ obtains the latest value of sv. |
| $T_i$ reads sv before $T_0$ binds sv | $\Rightarrow$ $T_i$ blocks until sv is bound to a value. |
| $T_0$ binds sv before $T_i$ reads old(sv) | $\Rightarrow$ $T_i$ obtains the previous value of sv. |
| $T_i$ reads old(sv) before $T_0$ binds sv | $\Rightarrow$ $T_i$ obtains the current value of sv. |
| $T_0$ binds sv after binding sv | $\Rightarrow$ an exception is raised. |
| $T_0$ binds old(sv) | $\Rightarrow$ an exception is raised. |

Whereas in a sequential stripe old(sv) always returns the previous value of sv, in a parallelism stripe it will either return the previous value of sv or the current value of sv, depending on whether sv has been bound to a value in that stripe. The way to look at this behaviour is to assume that sv gets a new current state immediately upon entry to the parallelism stripe. This state is undefined and cannot be examined until it has been bound to a value by a thread, but this means that old(sv) is always defined. Unfortunately, there is a minor shortcoming in the usage of the old() function which occurs if old(sv) is referenced during a parallelism stripe in which sv is not written. In this case, a call to old(sv) from the following sequential stripe will return the previous value of sv and not the current value of sv that was returned by the call to old(sv) in the parallelism stripe.

The old() function can be generalised to "old(i, sv)" where $i$ is an integer (i>0), *sv* is a shared value, and $i$ is used to select the $i^{th}$ oldest value of sv. For example, assuming sv has taken the values {2, 4, 6, 8} ranging from the oldest on the left to the current value on the right. The value of sv is 8, the value of old(1,sv) is 6, the value of old(2,sv) is 4, and the value of old(3,sv) is 2. An equivalent but more unwieldy notation for multiple histories can be obtained from nesting calls to the old() function. For example, old(3, sv) is equivalent to old(old(old(sv))). As before, an attempt to read an old shared value that does not exist or an attempt to write to an old shared value will cause an exception to be raised. Figure 4.2 illustrates a

snapshot of a Tyger program's execution in which it is executing in stripe *i*, reading and writing shared values, but it can also read previously written shared values from earlier stripes.

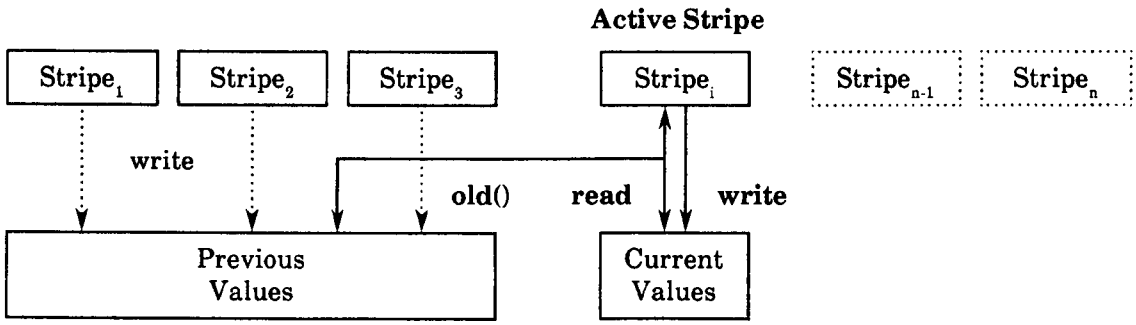



Figure 4.2 - A snapshot of a Tyger program's execution.

Using the history facility provided by the old() function means that static shared values can be initialised at compile time, as compile time can be regarded as part of the initial sequential stripe. However, if there is a special need to obtain the initial state of a shared value rather than some intermediate state, a call can be made to the old() function with a special read only variable. Calls take the form of "old(**InitialValue**, sv)". Thus, the *InitialValue* read only variable is not a constant, but instead is a count which represents the number of assignments made to a static shared value. This mechanism can be used to retrieve the initial state held by a static shared value irrespective of whether it was made at compile time or not.

Figure 4.3 illustrates a sample Tyger-C program with two parallelism stripes. The first two lines of the main procedure declare and initialise two shared values *sumA* and *sumT*. Next, three worker procedures *doA()*, *doB()* and *doC()* are listed. The worker procedures are denoted by the keyword "**PAR**" which means that they can be executed in parallel, but do not have to be. The main reason why the keyword is necessary is to take the place of any return type, as no provision is made in the current Tyger model for the return of a value from a thread that returns a non-void result. When the main procedure is executed it creates a worker thread of control and executes doA() and doB() in parallel. The thread executing doA() binds the shared value referenced by its parameter called *outResult* (i.e. *sumA*). The thread executing doB() reads the value of its parameter called *in* (also sumA) and uses this value in a part of its own computation. In the second parallelism stripe, a thread executing doA() is executed in parallel with a thread executing doC(). As before, the shared value sumA is used to communicate

data between the threads, only this time the previous value of sumA is also needed.

```
void    main()
{   SV float sumA = 0.0;
       float sumT = 0.0;

    doA(&sumA, 1) |&| doB(sumA, &sumT);          /* parallelism stripe */
    printf(" The first sum is %f\n", sumT);
    doA(&sumA, 2) |&| doC(sumA, &sumT);          /* parallelism stripe */
    printf(" The total sum is %f\n", sumT);
}

PAR   doA(outResult, data)
SV float *outResult;
    int  data;
{   float LocalSum ;
    // perform some local work with 'data' storing temporary result in LocalSum
    *outResult = LocalSum;
}

PAR   doB(in, outResult)
SV float in;
    float *outResult;
{   float LocalSum ;
    // perform some local work referencing 'in' and storing temporary results in LocalSum
    *outResult = LocalSum;
}

PAR   doC(in, outResult)
SV float in;
    float *outResult;
{   float LocalSum ;
    // perform some local work referencing 'in' and storing temporary results in LocalSum
    *outResult = LocalSum + old(in);
}
```

Figure 4.3 - Sample Tyger-C program.

One important observation to make about the previous states held by a static shared value that are returned by the old() function is that they do not have a direct correspondence to the number of stripes in a program. That is, "old(i, sv)" returns the $i^{th}$ oldest state of sv, if it exists, which may not have been written in the $i^{th}$ previous stripe. This is so because a shared value may take many values during a sequential stripe or may not be referenced at all during a stripe. Hence, to retrieve a particular shared value history, a programmer should count the number of assignments made to the shared value, rather than the number of stripes in the program between the past value of interest and its current value. The main reason why the retrieval of old shared values is linked to the actual number of distinct old values in a shared value history and not to the number of stripes in a program is to make program code more portable. For instance, the

reference "old(sv)" assumes that an assignment, executed in some past stripe, has made at least one update to *sv*. However, under some alternative stripe semantics using a fictitious operator called, say, "previous(sv)", it is assumed that there has been at least one assignment to sv and that it was made in the last stripe. Of course, there may be counter examples where a programmer may wish to know the value taken by a shared value in some previous stripe. So, prudently, both history mechanisms can be provided for extra flexibility, as this can be carried out for very little implementation cost.

## Programming Language Considerations

Up until this point the discussion of the Tyger model has been couched in quite abstract terms, shying away from the realities encountered in the implementation of Tyger languages. Unfortunately, adequate descriptions of the remaining Tyger programming constructs cannot be supplied without some recourse to some specific implementation language details.

The first implementation of the Tyger model used the C language as a host and was called Tyger-C. As C is a fairly small language, not supporting a high degree of data abstraction, new syntax was developed for the Tyger constructs. To produce executable Tyger-C programs it was envisaged that source programs would be passed through a preprocessor generating C programs as output, that contained explicit lightweight thread constructs to implement the parallelism. Some of the simple Tyger constructs were implemented in this way, but unfortunately, due to complexities in the Tyger model, it was discovered that a complete preprocessor would take on a complexity approaching that of an optimising compiler, hence this route was abandoned.

The object-oriented language C++ was chosen as the next Tyger language. In an object-oriented language, the syntax and semantics of its operations are much richer than those of a simple imperative language, which means in this case, that almost all of the Tyger model can be implemented on top of C++ without introducing any additional syntax. Thus, the use of an object-oriented model provides an excellent short term method for building Tyger constructs for evaluation, though a true Tyger compiler is needed to produce the most efficient code.

As a consequence of using C++ as a language to implement Tyger constructs, shared value objects are created by instancing the *shared value class* with a parameter denoting the number of shared values to be held inside the object

(accessed as a linear array). In addition, all shared value operations are described as being *member functions* of a shared value object, and can be thought of as being tied to a shared value object and not free-floating as before.

Shared Value Notation

Individual static shared values can be used to express fine grained parallel activity, though in many cases to obtain real benefits from parallel execution on shared memory multiprocessors this parallel activity must be blocked into sizes yielding medium grain parallelism or better. One of the easiest ways to accomplish this is to parallelise algorithms that process regular data structures such as arrays. Hence, the array assumes the position of the primary data structure in the static shared value model, and forms the basis for the discussion in the remaining part of this chapter.

A notation to declare and manipulate arrays is not really part of the Tyger model, as it has more to do with the data structures of a Tyger host language. Nevertheless, it is included here as the parallel manipulation of arrays can be made much easier by the provision of suitable syntax. Two areas of array notation need to be addressed. Firstly, arrays of shared values must be declared; and secondly, sections of arrays of shared values need to be specified as parameters to Tyger constructs. In the latter case, a high level mechanism is needed to identify shared value array elements so that information can be exchanged between threads without actually having to examine the value of a shared value. (This idea is quite similar to that of Halstead's Futures in MultiLisp [Hals87], where references to variables can be manipulated on the promise that values for those variables will eventually become available.) For example, consider a master thread that allocates shared value array elements to worker threads. The master thread needs to be able to reference groups of shared values so that it can allocate them as input parameters to the worker threads that compute the actual values (this idea is returned to later in the description of the Tyger iterators).

The notation that has been chosen to manipulate arrays is, unremarkably, modelled on that found in vector and array processing languages. Array elements or sequences of array elements can be identified by using *index sets*. Such sets provide a tidy and compact method of element identification, taking the form of an integer triple *lower-bound* : *upper-bound* : *step* or simply a pair of bounds with a default step equal to one.

Tyger-C notation: here the **SV** qualifier in an array or variable declaration appropriately takes the place of the storage class of the variable. This not only makes parsing such declarations easier, but also means that the shared value qualifier can be easily thought of as a property of a variable, which it is, rather than a specific variable type. Sample declarations of shared values in Tyger-C are:

```
SV int signal                /* integer shared value */
SV int work[1:100]           /* array 1-100 of integer shared values */
SV float work[1:20][2:10]    /* 2-d array of single precision real shared values,
                                indexed 1-20 and 2-10 */
```

Tyger-C++ notation: here shared value objects are created by instancing the shared value class and supplying the number of shared values that the object contains. For example, an array of shared values *work[0:n-1]* can be created by invoking the shared value constructor via the statement "**SV** work(n)". Notice that the use of index sets in Tyger-C++ shared value declarations has degenerated to the point where only an upper bound need be specified, with the lower bound and step defaulting to 0 and 1 respectively. Naturally, complete index sets could be used in shared value declarations, though, in many cases their presence would not add appreciably to the readability or functionality of a program.

One drawback of the shared value class is that it can only be used to represent shared values of a single base type at a time (e.g. integer or float), unless elaborate and none too reliable steps are followed in its implementation. Alternatively, other similar shared value classes could be used to represent different base types such as **SVF** vector(n) for floating point shared values. However, subsequent releases of C++ promise parameterisable (generic) types in which "**SV** vector(float, n)" would create an array called *vector* of floating point shared values, and "**SV** list(int, n)" would create an array called *list* of integer shared values.

### 4.2.3 Control Flow Partitioning Functions

To express the data partitioning of an array of shared values a supplementary mechanism has been provided to coordinate the creation and termination of multiple threads of control; because shared values by themselves are only a mechanism for expressing communication and synchronisation. Of course, one could question the need for an additional thread creation notation; as the task of adding thread creation code could be automated and performed by an auto-parallelising compiler using the techniques that have been described in the software tools section of chapter two. A point in favour of this approach is that the

dependency analysis of the auto-parallelisation should be simpler than that normally encountered in programs written using imperative languages because of the restrictions on the use of shared variables, due to the single assignment property. A negative point, however, is that the host languages in which Tyger constructs have been implemented make use of pointers for accessing structures and arrays, which severely complicates the analysis. In addition, the philosophy behind shared values is that an explicit approach to parallelism should be followed that encourages programmers to design programs with parallelism in mind and to this end, explicit programming language support should be provided to express that parallelism.

Essentially, two types of construct are needed for controlling the creation and termination of multiple threads of control. Firstly, and most importantly, a mechanism is needed to specify the points in a program where multiple threads of control are to be employed. Secondly, a mechanism is needed to limit the number of threads of control that will be used at those points. The specification of parallel activity is supported by the *parallel-and* operator (introduced earlier in the sample programs) and the *Tyger iterators*. The semantics of the parallel-and operator are quite simple so they are discussed first, even though the iterators are the primary thread creation mechanism in the Tyger model.

The thread creation mechanism provided by the parallel-and operator "|&|" allows named procedures to be executed in parallel. Basically, the operator takes two arguments and executes them in parallel as separate threads of control. When the execution of both arguments has terminated, the instruction on the next line following the invocation of the parallel-and operator is executed. The use of the parallel-and operator can also be cascaded permitting many procedures to be executed in parallel. Thus, the syntax of its use is

PARALLEL_AND : FN_CALL"|&|" FN_CALL | PARALLEL_AND"|&|" FN_CALL;

where the *FN_CALL* term can be a procedure or Tyger call. The main use of the parallel-and operator is to express functional partitioning, where several distinct procedures are executed in parallel.

Conversely, the Tyger iterators are designed for use in data partitioning applications in which replicated workers execute code that operates on parts of a shared value data structure. The original idea of an iterator comes from the programming language CLU [Lisk77]. In CLU, an iterator is a data abstraction mechanism that is used to select some part of a data structure. For example, a

sparse matrix can be represented by a linked list of values; the role of an iterator is to provide a mechanism for scanning such a list, without revealing to a user the details of the list's implementation. In the Tyger model, the role of an iterator is similarly to scan over a data structure, only this time, procedures operating on the data structure are executed in parallel under the auspices of the iterator. However, a limitation of this model is that all the worker threads execute the same procedure body. Thus, to perform functional partitioning a programmer must write additional selection code for the thread bodies that will determine a thread's function from an examination of its input parameters. This is somewhat irritating from an aesthetic point of view, as writing the extra selection code is an extra burden on programmers which may prove awkward in some circumstances. For instance, the parameter list of a thread must be the union of all initial thread parameter lists for that parallelism stripe. Hence, the parallel-and operator, described previously, was included in the Tyger model as a more general thread creation operator.

The *CONTROL()* procedure allows a programmer to explicitly set the number of concurrent threads of control that will be employed in a parallelism stripe created by an iterator. The syntax of an invocation of the CONTROL() procedure is "CONTROL(THREADS, TRACE)", where the THREADS parameter is a positive integer representing the number of worker threads to be used by an iterator, and the TRACE parameter is a boolean flag (ON/OFF) controlling the printing of debugging information regarding the allocations of work made to each worker thread by an iterator.

Tyger-C Shared Value Iterators

For the implementation of the Tyger iterators in Tyger-C some unusual syntax was devised to reinforce the bonding between arrays of shared values and the iterators that were used to operate upon them. Basically, the model of parallelism was one where a procedure that was to be replicated and executed in parallel took, upon its invocation, an initial parameter consisting of an iterator and an array of shared values. For example, consider using data partitioning to concurrently operate the procedure *worker()*, which uses the read only parameters contained in *ParameterList*, on an array of shared values *X[1:n]*. This can be brought about by a call to an iterator called *DOALL* in the form of

worker(< **DOALL** X[1:n] >, ParameterList)

When the procedure worker() is called, as above, the role of the iterator is to invoke multiple copies of the procedure worker(), with each copy being executed by a separate thread of control. In addition, each invocation of the procedure worker() receives a reference to the array of shared values (in this case X[1:n]), and a unique triple of lower bound, upper bound and step integer indexes to a section of this array; the intention being that a thread computes the values in the section of the array it is allocated. The corresponding declaration of the header of procedure worker() is below, with the parameters *Y* (an array of shared values), *lower* (integer), *upper* (integer) and *step* (integer) being filled in by the iterator

**PAR** worker(Y, lower, upper, step, ParameterList)

Two types of iterator have been provided in Tyger-C: a static iterator called *DOALL*, and a dynamic iterator called *FOREACH*. The DOALL iterator makes a single, approximately equal, allocation of work to each thread it creates. The FOREACH iterator allocates work one index at a time to each thread it creates, and then threads request new work from the iterator until the work to be done is exhausted. This iterative process increases the overhead of work allocation, but at the same time permits a degree of load balancing. A syntax diagram for these iterators follows.

```
ITERATOR_CALL:  PID "( < " ITERATOR SV_ID DIRECTIVE " >" PARAM_LIST ")" ;
ITERATOR      :  FOREACH | DOALL ;
PID           :  // the identifier of any procedure declared as a PAR ;
SV_ID         :  IDENTIFIER "[ " INDEX_SET "] ";
INDEX_SET     :  INTEGER : INTEGER;
PARAM_LIST    :  // any other read only parameters to the function named PID ;
DIRECTIVE     :  " '?' " | " '!' " ;
```

The DIRECTIVE term is used to select the method for determining the number of threads that an iterator will use in a parallelism stripe. The "?" character signifies that an iterator should use a heuristic to arrive at the number of workers, based on the array size supplied to the iterator and on the external system load (number of available processors). Intuitively, these factors have some part to play in the choice of the optimum number of threads, but one factor that is missing is the complexity involved in computing each array element. If only a few operations are needed to compute an array element, it may be appropriate for a thread of control to compute many hundreds of elements. At the other extreme, if it takes many thousands of operations to compute an array element, perhaps one thread of control per element computation is more appropriate. Clearly, finding the

optimum number of threads that balances thread overheads, the available algorithmic parallelism, and the available machine parallelism is very difficult. Therefore, it would be convenient for a programmer not to have to be concerned with the number of threads that a program uses, leaving this task to the compiler and iterator. To support this view, some research has been done in the area of complexity analysis (see Parafrase-2 in chapter two); but for many problems analysis of the kind performed is insufficient and the only sure way to effectively exploit the parallelism of an application is for the programmer to intervene. This is precisely what the presence of the "!" character in an iterator call allows a programmer to do (this is the default). When this character is passed as a parameter to an iterator, the iterator uses a value for the number of workers set by the CONTROL() procedure, using one thread per processor if it cannot find one. This allows a programmer total control over the number of threads that are created, so that a parallel program can be executed repeatedly and the number of workers varied, perhaps to find the optimum number of workers empirically.

For a taste of iterator usage the following two examples are presented. Consider the execution of the procedure *initialise()* which sets every element of the array of shared values *Y[1:N][1:N]* to *initialvalue*. This can be specified by the statement below, with the choice over the number of threads to be used being left to the iterator

```
initialise(< DOALL Y[1:N*N] '?' >, initialvalue);
```

Some time later, the execution of 5 threads of control that use the procedure *worker()* with the same array Y, partitioning Y at run time, can be specified by

```
CONTROL(5);
worker(< FOREACH Y[1:N*N] '!' >);
```

Notice that in the above examples, only one dimensional array partitioning is used to divide the array. The argument for this now follows. In the allocation of shared value references to threads the references themselves are merely integer indexes and not actual pointer references to specific array elements. Moreover, the form of the correspondence between indexes and array elements is one made by a programmer and not by an iterator. For example, if a thread receives a range of indexes, say 1:5:1, it may use these directly, referencing the array elements subscripted from 1 to 5, or may transform them for use to 2:10:2, referencing the array elements subscripted as 2, 4, 6, 8, 10. One consequence of making the programmer devise the mapping between indexes and array elements is that no

automatic error checking can be performed at the thread level to check that a thread assigns the shared values allocated to it. This is true irrespective of whether the checking is done by the compiler or at run time by an iterator, unless the mapping function between indexes and shared values is also known. At first sight this may seem a shortcoming of an iterator, in that the programmer is responsible for the job of mapping indexes to actual values. However, there are a number of reasons why the programmer is best suited to undertake this task.

First of all, using indexes means that iterator functions merely have to partition a set of integers corresponding to the array bounds supplied in the iterator call. This simplifies the implementation of an iterator, cutting run time overhead to a minimum. What is more, dealing with higher dimensional arrays is not an easy thing to visualise or even express in terms of array partitioning. However, flattening the subscripting of such an array into a single stream of integers is the lowest common denominator, and allows a programmer to compute whatever mapping is most appropriate for the application being solved. For example, consider partitioning the two dimensional array *table[0:9][0:9]*, to a procedure called *worker()*. The following iterator invocation examples illustrate the form of the partitioning instances that are allowed under the existing Tyger-C partitioning semantics extended to two dimensions.

```
worker(< DOALL table[0:9][0:9] >)      /* partition every element */
worker(< DOALL table[i][0:9] >)        /* partition the i^th row */
worker(< DOALL table[0:9][i] >)        /* partition the i^th column */
```

While the forms of partitioning shown above may prove useful in some applications, it is unwise to make them the only method of partitioning. In particular, if a programmer insists that a thread should receive a higher dimensional array as a partition or perhaps an irregular shaped partition, then the existing syntax alone is not capable of fulfilling these requirements. Furthermore, the overhead in implementing two dimensional partitioning is higher than that associated with one dimensional partitioning, especially for dynamic partitioning. Thus, implementing the most general partitioning scheme means that every partitioning call pays the higher overhead price, even if such a call does not need the extra sophistication. In addition, there is a further argument in favour of the programmer defining a mapping between a thread's work partition and the actual work it does. Although this discussion has focussed on arrays as the data structures to be partitioned, it may be possible for a programmer to devise a mapping between integers and data elements so that an

arbitrary data structure, such as a tree, can be partitioned. In the light of this, it seems reasonable to take the most flexible line and merely treat work partitioning as operating on a single stream of integers.

Figure 4.4 shows a Tyger-C implementation of Conway's game of life [Gard70]. The program works by setting a two dimensional array of cells to some given initial state. Next, the state of the array of cells is evolved a fixed number of times by computing the next state of each cell as a function of its previous state, and the previous states of its immediate neighbours. After all the computations have been completed the final state is printed out.

The Tyger program illustrated uses a two dimensional array of shared value integers to simulate the array of cells. At the first invocation of the DOALL iterator, the *makeworld()* procedure is executed by two threads of control. Each thread assigns a number of complete rows of the array of shared values, the exact number depending upon the values of *upperBound* and *lowerBound* that are supplied by the iterator. In the second invocation of the DOALL iterator, five threads execute copies of the *worker()* procedure, once again assigning complete rows of shared values. The program as it stands does not make use of the synchronisation properties of shared values, but could do so if the computation and printing were to be overlapped by executing the *printworld()* procedure in parallel with the worker() procedure during the desired generations. However, some peripheral debugging benefit is gained by using shared values in that checks are made to the effect that no cell is ever assigned more than once during a generation.

Tyger-C++ Shared Value Iterators

After the initial research had been carried out with the iterators in Tyger-C it was decided that an enlarged number of iterators should be provided in Tyger-C++. Five iterator functions have been included in Tyger-C++ to facilitate the expression of a variety of work allocation policies. As before, iterators fall into two classes: either they make a single static allocation of work to each thread, or they make a series of dynamic allocations to form a pool of work for a group of threads. The first two iterators, called *block* and *stride*, allocate work to threads using deterministic methods which are explained later. These static iterators are invoked by calling the *DoAllBlock()* and *DoAllStride()* member functions of a shared value object. For example, given a shared value object *X[0:N-1]* and a

```
/* A Tyger-C program to play the game of life in parallel */

#define    MAXGENERATION   10
#define    nprocessors     5
#define    SIZE            100

/* each thread computes a number of rows */
PAR    makeworld(myworld, upperBound, lowerBound)
SV     int myworld[SIZE][];
int    upperBound, lowerBound;
{      int r,c;
       for (r = lowerBound; r <= upperBound; r++)
           for (c = 0; c < SIZE; c++)
               myworld[r][c] = SomeInitialValue;
}

/* each thread computes a number of rows */
PAR    worker(myworld, upperBound, lowerBound)
SV     int myworld[SIZE][];
int    upperBound, lowerBound;
{      int r,c;
       for (r = lowerBound; r <= upperBound; r++)
           for (c = 0; c < SIZE; c++)
               myworld[r][c] = SomeFunction(old(myworld[r][c]), r , c);
}

void   printworld(myworld)
int    myworld[SIZE][];   /* only used from a sequential stripe so no need for an SV */
{      int r,c;
       for (r =0; r < SIZE; r++)
       {
           for (c = 0; c < SIZE; c++)
               printf("%d ", myworld[r][c]);
           printf("\n");
       }
}

void   main()
{  SV int world[0:SIZE-1][0:SIZE-1];
       int generation;

       /* the initialisation time per element is small so while much can be done in parallel, */
       /* it is not efficient to do so, hence the small number of threads */
       CONTROL(2);
       makeworld(< DOALL world[0:SIZE-1] >);          /* parallelism stripe */

       /* more work is involved per element so increase the number of threads accordingly */
       /* to take full advantage of the parallelism */
       CONTROL(nprocessors);                          /* sequential stripe */
       for (generation = 0; generation < MAXGENERATION; generation++)
       {
           worker(< DOALL world[0:SIZE-1] >);         /* parallelism stripe inside a loop */
       }
       printworld(world);
}
```

Figure 4.4 - Tyger-C game of life.

thread body called *worker()*, a call to create multiple worker threads is "X.**DoAllBlock**(0, N-1, worker)".

As C++ is an object-oriented language it has much richer semantics and syntax than that of C, so the Tyger iterator syntax was modified to make it conform with existing C++ syntax. When the shared value class is instanced and a shared value object is created, a parameter must be supplied which represents the extent of the shared value. For example, a call to the constructor "**SV** list(100)", creates one hundred shared values indexed by array notation, i.e. list[0] to list[99]. The iterators are declared as member functions of the shared value class, and so are considered to be part of every instance of a shared value object. Thus, the Tyger-C "worker(<**DOALL** list[0:99] '?'>)" now becomes "list.**DoAll**(0,99, worker, '?')" in Tyger-C++. What is more, the CONTROL() procedure is also bound to shared value object, so the number of threads must be set for each instance. For example, the Tyger-C call "**CONTROL**(nprocs, **OFF**)" becomes "X.**control**(nprocs, **OFF**)".

The remaining three iterators in Tyger-C++, called *hunk*, *point* and *stream*, allocate work dynamically using variations of a non-deterministic first-come-first-served algorithm, also explained later. These dynamic iterators are invoked by calling the *ForEachHunk()*, *ForEachPoint()*, and *ForEachStream()* member functions of shared value object. For example, given the shared value object *X[0:N-1]* and a thread body called *worker()*, a call to create multiple worker threads is "X.**ForEachHunk**(0, N-1, worker)".

With the provision of a wider range of iterators, a programmer can now more accurately match the run time behaviour of an iterator to that of the problem being solved. In chapter six, some comparative data is presented to contrast the effects on performance of parallel programs obtained by using different iterators on the same problem.

Static Allocation Iterators

The block iterator is the realisation of a simple work allocation policy under which the host array of shared values is partitioned into continuous sections of approximately equal size, there being one section per worker thread. When a worker thread is started it is passed two integers, *lower* and *upper*, that represent the limits of its array section. An additional parameter, the *stride*, is also supplied for consistency with the other iterators and always takes the value one in this instance. For all the iterators the interpretation of these three input parameters is the same. It is that processing should commence with the element indexed by the

lower bound as a starting point, and continue with the next element indexed by adding the stride while this quantity is less than or equal to the upper bound. Thus, the important observation is that if a thread uses the three parameters correctly, all the iterators can be used interchangeably - if there are no restrictions on the order in which the resulting shared values are produced.

If the extent of the shared value object does not divide exactly by the number of worker threads that will be used by the block iterator, the remaining work is allocated one element at a time, together with the original work, to threads upon their creation until the remainder is exhausted. The idea behind this strategy is that, hopefully, allocating the extra work to those threads that are started first will have a good load balancing effect, and will negate some of the overhead costs of starting up the worker threads. As an example of iterator usage, given the shared value object *X[0:n-1]*, the operation "X.**DoAllBlock**(0, n-1, Operation)" invokes user code of the form shown below. A reference to the parent shared value object is passed as the first parameter to the worker procedure. The integers *lowerBound*, *upperBound* and *stride* are filled in by the iterator and are passed to the worker procedure in a manner that is implementation dependent. All that it is important to note is that values for the specially named parameters are supplied by the iterator and then can be used by a thread of control as normal parameters.

```
// procedure Operation() is a worker thread body, not necessarily a member function of X[]
// the code computes the values of a part of an array of shared values called X[]
PAR   Operation(SV X, int lowerBound, int upperBound, int stride, .. )
{
        for ( int i = lowerBound; i <= upperBound; i+= stride )
            X[i] = SomeFunction(i);
}
```

In the case of a shared value object where the CONTROL() procedure has set the number of threads equal to five and n = 62, five threads of control labelled 0 to 4 will be used as workers, with thread 0 initially running the iterator code. The number of array elements allocated to each thread is 13, 13, 12, 12 and 12, implemented as thread 1 receiving the triple (0:12:1), thread 2 (13:25:1), thread 3 (26:37:1), thread 4 (38:49:1), and thread 0 (50:61:1).

The stride iterator does not allocate continuous blocks of the host array to a thread, but instead provides an algorithmic way of partitioning array elements. As before a lower bound, upper bound and stride are supplied to each worker thread, however, this time the stride is equal to the number of worker threads. For example, given the shared value object *X[0:n-1]*, the operation"X.**DoAllStride**(0, n-1, Operation)" invokes user code of precisely the same format as with the block

operator. Once again using the scenario of 5 threads and n = 62, the number of array elements allocated to threads is again 13, 13, 12, 12 and 12. However, this time the allocation of indexes to thread 1 is the triple (0:61:5) giving a sequence (0, 5, 10, ... , 45, 50, 55, 60), thread 2 (1:61:5) giving (1, 6, 11, ... , 46, 51, 56, 61), thread 3 (2:61:5) giving (2, 7, 12, ... , 47, 52, 57), thread 4 (3:61:5) giving (3, 8, 13, ... , 48, 53, 58) and thread 0 (4:61:5) giving (4, 9, 14, ... , 49, 54, 59).

Dynamic Allocation Iterators

The hunk iterator continuously allocates hunks of the host array to worker threads during the execution of a program. The partitioning algorithm is based on the Guided Self-Scheduling work of Polychronopoulos and Kuch [Poly87] which is reported to have good load balancing properties. The main idea of the algorithm is to allocate hunks of work in decreasing size to reduce the chance that threads will be idle for long periods after processing their allocations, waiting for the other threads to finish their own computations. The method that has been implemented operates as follows: given a remaining work load $R_i$ ($R_1 = n$) at time $i$ with $t$ threads allocate elements $x_i$ by

$$x_i = \lceil R_i / t \rceil, R_{i+1} \leftarrow R_i - x_i$$

Alternatively $x_i$ can be computed by

$$x_i = \lfloor R_i / t \rfloor \text{ if } \lfloor R_i/t \rfloor \neq 0 \text{ else } 1, R_{i+1} \leftarrow R_i - x_i$$

Once again the information supplied to a thread at run time, every time it requests work, takes the form of three parameters: lower bound, upper bound and stride (which for this iterator is always equal to one). Given a shared value object *X[0:n-1]* the invocation syntax is "X.**ForEachHunk**(0, n-1, Operation)" with the worker procedure written as illustrated before for the static iterators. Although it is not possible to predict which thread will obtain which work allocation it is possible to say what the allocations will be. In the case of 5 threads and n = 62 the partitioning of indexes for threads (using the alternative method for $x_i$) is $x_1$ (0..11), $x_2$ (12..21), $x_3$ (22..29), ..., $x_{16}$ (60..60), $x_{17}$ (61..61).

Like the hunk iterator, the stream iterator also allocates work continuously at run time, only in this case the block size is fixed at one. This method has potentially the best load balancing properties because of its fine granularity of work distribution, but this also means that it has the highest cost in synchronisation overhead. As an example of this iterator's usage consider a shared value object *X[0:n-1]* with invocation syntax "X.**ForEachStream**(0, n-1,

Operation)" and a worker thread as before. In the case of 5 threads and n = 62 the partitioning of indexes for threads is $x_1$ (0..0), ..., $x_i$ (i-1..i-1), ..., , $x_{62}$ (61..61).

The point iterator is an optimisation of the stream iterator that uses work preallocation to reduce the run time synchronisation overhead. At run time a tentative allocation of work to threads is made by using the mechanism employed by the block iterator. Later on, however, when a thread completes its quota of work it can look for additional work by inspecting the quotas of other threads. As a consequence of this, the actual work assignment is handled in a manner similar to the stream iterator, but with the advantages of there being less synchronisation contention and as a result a faster streamlined version of the point iterator work allocation code can be employed. Given a shared value object *X[0:n-1]* the invocation syntax is "X.**ForEachPoint**(0, n-1, Operation)" with the worker thread syntax as before. In the case of 5 threads and n = 62 the partitioning of indexes is similar to that for the stream iterator, except that each thread has a preallocation of indexes made by the block iterator.

### 4.2.4 Summary of the Tyger Model

This section briefly summaries the capabilities of Tyger-C++ by presenting a short programming example, and then goes on to reveal some of the advantages of combining data abstraction with parallel programming. However, a final example warns against the unconstrained introduction of parallelism into procedural languages.

Consider a Tyger-C++ program that performs a transformation on a two dimensional array *X[0:n-1][0:n-1] (n>0)*. The nature of the transformation is to average each array element with its eight neighbouring elements. A shared value object can be used to represent X and the averaging operation can be applied in parallel to the object via a Tyger iterator. As the current iterators support only one dimensional partitioning, X is referenced as a one dimensional array which makes the indexing code look a little untidy, but otherwise does not affect the program. Four user defined ancillary functions (*shiftUp()*, *shiftDown()*, *shiftLeft()* and *shiftRight()*) are used to calculate the indexes of neighbouring elements, hiding the details of how the array wraps along its edges. The following program fragment in Figure 4.5 illustrates how the transformation can be coded by using static shared values.

The program fragment consists of a *main()* procedure, a worker procedure *average()*, and a global variable *n* which is accessible from both procedures. The

```
// declaration of the shared array size, assuming a square array
int     n = 100;

// worker code to compute a block of rows of the shared array
PAR   average(SV Y, int lowerBound, int upperBound, int stride)
{
    ArrayType     tmp;

    for (int row = lowerBound; row <= upperBound; row+= stride)
        for  (int col = 0; col <= n; col++)
        {
            tmp = (ArrayType) 0;
            for (int s = shiftDown(row); s <= shiftUp(row); s++)
                for (int t = shiftLeft(col); t <= shiftRight(col); t++)
                    tmp += old(Y[s*n + t]);
            Y[row*n + col] = tmp / 9;
        }
}

// code to declare the shared array and create worker threads
void    main()
{
    SV      X(n*n);                     // shared array of n^2 elements

    readData(X, n);                     // sequential stripe to input data
    X.DoAll(0, n, average);             // parallel stripe to compute result
}
```

Figure 4.5 - Tyger C++ program excerpt.

main() procedure declares a shared value object *X* of size *n*n* which is used to represent the shared array of elements. Data is acquired for X by invoking the *readData()* procedure. The call to the DoAll iterator partitions X among a group of worker threads executing the average() procedure. Each worker thread is passed three values which the programmer has taken to represent the group of rows of the shared array that the thread has to compute. Therefore, for every element of each row in its range, a thread finds the average of the element and its neighbours by working with a local variable, and then makes a single update to the shared array X. This short example program could have been easily coded using conventional library calls, however, the resulting program would have been nearly twice as long. Moreover, such a program may have contained many implementation details (e.g. the size of shared memory regions) therefore loosing portability, and perhaps increasing its maintenance requirements as changes to the underlying operating system force changes to the program code.

## Interthread Communication

A static shared value is an elegant method for expressing a single one way communication between a reader and writer thread of control because it retains

its value throughout a parallelism stripe. If multiple communications are required between threads, however, one way in which this can be accomplished is by using an array of shared values. The use of an array, though, means that the maximum number of messages has to be known in advance (i.e. fixed array size). In addition, using an array means both the reader and writer have to maintain counters to index the current communication, which can lead to some rather untidy programming little better than communication via shared buffer. Fortunately, by the use of appropriate data abstractions, shared values can be used to represent a shared communication stream, which is an efficient mechanism for interthread communication adopted by many declarative parallel programming languages (e.g. Strand).

Consider a class called SVStream, that a user can define, consisting of a pair of data items, namely a shared value and a reference to a shared value. Hence, a chain of SVStream elements can be constructed to represent a sequence of messages passing between threads of control. With the overloading properties of Tyger-C++, the syntax of SVStream can be arranged such that assignments can be made to an SVStream via the assignment "=" operator and SVStream elements can be dereferenced as if they were instances of a primitive type. In the following example two threads communicate via a shared SVStream.

```
PAR   producer(SVStream pipe)
{
        pipe = 1000;
        for (int i = 0; i < 1000; i++)
            pipe = i;
}
```

```
PAR   consumer(SVStream pipe)
{
        int noMessages = pipe;
        for (int j = 0; j < noMessages; j++)
            cout << pipe << " ";
}
```

The basic operation of the threads is for the producer thread to send the number of messages, followed by the messages themselves, to a consumer that prints them out. Synchronisation is handled by the shared value portion of the SVStream class which confers shared value semantics on SVStream elements. In this implementation of the SVStream, once a value has been read from a stream it is discarded. While this is an intuitive model, it deviates a little from the shared value theme of persistence which assumes that a shared value is always available for use during a parallelism stripe once the value has been determined. An alternative implementation of SVStream (SVQueue), which retains the persistence of stream values, gives rise to the following consumer.

```
PAR   consumer(SVQueue pipe)
{
    SV QueueHandle nextMessage =  pipe->Handle();
    int  noMessages                =  nextMessage;
    for (int j = 0; j < noMessages; j++)
```

```
        cout << nextMessage << " ";
}
```

In this implementation of a stream, *pipe* can be assigned as before but when its value is examined only the first assignment will ever be seen. To obtain the values of subsequent assignments, the consumer maintains a local reference to the next message to read (nextMessage) and uses this to scan the stream. An initial call is made to the Handle() function of *pipe* to obtain a reference to the start of the stream. Subsequent dereferencing of the handle returns the data part of the current stream element and implicitly moves the reference to the next element in the stream. An application for such a stream with nondestructive reading is the implementation of a communication channel that is shared amongst multiple threads of control. A thread can select values from the stream at its own rate, confident in the knowledge that other threads cannot interfere by removing unread values from the stream. (As an optimisation it is possible to modify the SVQueue class, under certain circumstances, so that stream elements are removed after the last thread has read them in order to reclaim the storage.)

Thus, by using the shared value class as a starting point it is possible to develop user defined data structures that not only have the desired operating characteristics (e.g. variable size, repeatable reading, and shared value synchronisation), but at the same time, retain aspects of the familiar original shared value syntax. Hence programmers can continue to think about parallel programs in terms of shared values, and not be concerned about using new notation that may make programs harder to write down and understand.

## Nested Stripes

The Tyger model so far described permits the construction of parallel programs that consist of a series of independent parallelism stripes in which there is a single level of thread creation. If more than one level of thread creation is allowed in a parallelism stripe, that is a worker thread can itself create new threads of control, several choices are possible for the exact interpretation and semantics of shared value operations within such stripes. For example, consider a shared value object *X[0:n-1] (n>2)* and four thread declarations named A to D coded as follows:

```
PAR A(SV X, ... )
{
    X[0] = 1 ;
    C(X, ...) |&| D(X, ...)
}

PAR B(SV X, ... )
{
    X[1] = X[0];
}

PAR C(SV X, ... )
{
    X[0] = 2;
}

PAR D(SV X, ... )
{
    X[n-1] = 1;
}
```

Given that the threads *A()* and *B()* are created by some main procedure and execute in the same parallelism stripe, the action of thread A() will be to bind a value for X[0], create two new threads of control, *C()* and *D()*, and then wait for them to terminate. The action of thread B() will be to obtain a value for X[0] and bind it to X[1]. The action of thread C() will be to bind a value to X[0], while thread D() operates on some completely separate part of X. The flow of control for this arrangement of threads is shown in Figure 4.6.

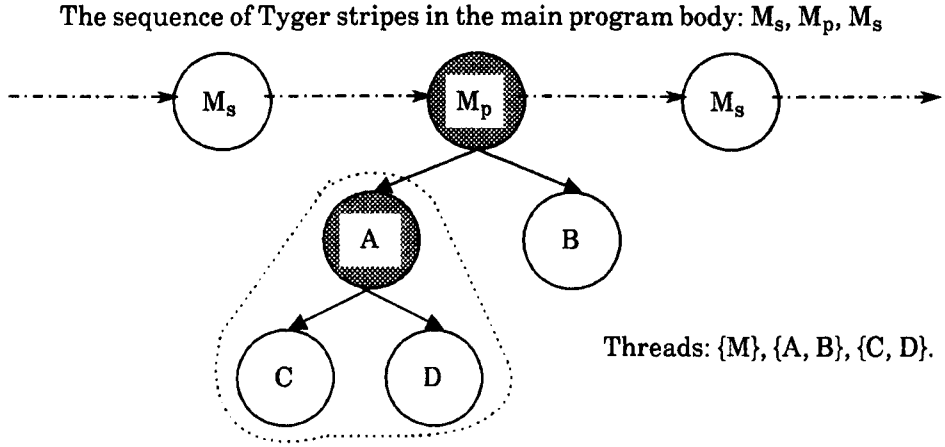



Figure 4.6 - Nested control structure.

Assuming that nested stripes are not allowed, but worker threads can create new threads of control that execute in the current parallelism stripe, leads to the following interpretation of the execution of the thread bodies A() to D(). After being created by the master thread, A() binds X[0] and then creates threads C() and D(). Meanwhile, after being created by the master thread B() looks for a value for X[0] and will find one after A() has executed its first statement. When C() starts executing, in the same stripe as A(), it attempts to rebind X[0] causing an exception to be raised and program failure.

In the alternative scenario, with nested stripes, the execution of the thread bodies is as follows. The threads A() and B() begin execution as before. However, this time A() itself consists of three stripes: a sequential stripe where X[0] is bound to a value; a parallelism stripe where the threads C() and D() are started; and a closing sequential stripe containing no instructions. In addition, the binding made by thread C() to X[0] is a legal operation since no other binding of X[0] is made in C()'s stripe, as A()'s binding to X[0] was made in its previous sequential stripe. Unfortunately, a question arises as to the value of X[0] obtained by B(). If B() executes its read operation on X[0] before C() binds X[0], B() sees the value set by

A(), otherwise it sees the value set by C(). Clearly, this is a race condition and not something that should be permitted by a structured parallel programming mechanism. A possible solution to this problem and other issues raised by the possibility of nested stripes is discussed in the future work section of chapter seven.

### 4.2.5 Dynamic Shared Values

Originally, the Tyger model was envisaged as being solely based on static shared values. This route was followed primarily because the algorithms that were parallelised exploited *strong data partitioning*. In such parallel algorithms, there are no data dependencies between concurrently executing computations so it should be possible to obtain linear speedups proportional to the number of processors applied to a problem, given sufficiently large problems. Furthermore, static shared values also provide support for *weak data partitioning*, in which computations involve read or write dependencies, through their dataflow synchronisation mechanism. However, static shared values provide less support for parallel algorithms that operate on complex shared data structures of the kind classified as distributed data structures by Carriero (see chapter three). Examples of data structures that can be operated on in this way include shared queues, tables and other abstract data structures (e.g. trees).

Using static shared values to represent such structures can be problematic, because if multiple changes have to be made to a field of such a data structure during a parallelism stripe, an entirely new copy of the whole data structure must be produced each time due to the single assignment rule. (This problem is similarly encountered in dataflow languages.) Perhaps what is more of concern though, is that since a conceptually local change to a data structure forces the production of a completely new data structure, only one thread at a time can manipulate the shared data structure. That is, multiple threads cannot make changes in parallel that will result in the production of new data structures, because threads cannot view a consistent overall picture of the data structure. But, strict serialisation of the kind necessary to guarantee correctness when operating in parallel on distributed data structures is often inappropriate because it is often too conservative for many potentially parallel algorithms.

One solution to this problem is to encapsulate distributed data structures inside caretaker threads of control, and to operate threads as clients and servers. However, algorithms of this kind need a more flexible method of sharing

information and synchronising communication than simple static shared values to support efficient spontaneous data transfer. Therefore, in an attempt to follow the expressiveness guideline for parallelism constructs mentioned in chapter three, dynamic shared values were devised to enable the expression of a variety of parallel program behaviours without recourse to unnatural or untidy program code. In particular, dynamic shared values were developed to act as a flexible communication mechanism to give extra support for distributed data structures.

Programming Model

At the time that dynamic shared values were developed, static shared values had only been implemented via Tyger-C. Through the advent of Tyger-C++, with the capability to develop new data types (e.g. streams), the need for dynamic shared values has been somewhat diminished. Hence, only a Tyger-C version of dynamic shared values is presented.

Many of the ideas in the dynamic shared value model came from the Linda programming model described in chapter two. The elements of the Linda model that were thought to be important were the ideas of using associative naming to perform pattern matching on data values, and the necessity of limiting the operations on shared data to a small set of named procedures. These two concepts are important because pattern matching can be a flexible way of referencing parts of shared abstract data structures, and using named operations to reference shared data allows communication and synchronisation to be combined to simplify the task of writing parallel program code. However, one of the aims in developing dynamic shared values was to take these key aspects from the Linda model and specialise them so that extra performance could be obtained through the use of physically shared memory.

The central idea of dynamic shared values, in common with Linda, is the notion of there being a single shared address space within a parallel program. This shared address space, in fact, can be regarded as a container object for shared values and is called a shared value space (SVS). A SVS possesses some intrinsic operations for data manipulation in a similar way that a conventional stack has operations *push()* and *pop()*. A SVS operation returns a status code indicating its success or failure, and where possible, operations are executed in parallel.

A dynamic shared value is formally made up of two parts: an identifier part and an optional data part. Each part of a dynamic shared value consists of a number of typed fields giving dynamic shared values a structure similar to

conventional records and Linda tuples (see chapter two). Identifier fields are used to match dynamic shared values against templates (described later) and can be constituted from strings of characters and all primitive types (e.g. integer, character, float). For example, in Tyger-C an element from a linear array of integers can be represented by the triple *Name*, *Index*, *Value*. If the triple is represented by two identifier fields and a data field, the corresponding value signature is *char**, *int*, (*int*) with the parenthesis indicating that matching is not performed on the last field. One advantage of using a data field in this way is for efficiency, in that signature matching (of the Linda kind) only applies to identifier fields, with no matching operations being carried out for the data field. In addition, it also allows a certain amount of decoupling between threads, as only the key identifier fields need to be known about precisely by a group of cooperating threads, and not the exact format of any optional data.

Primitive Operations

There are four kinds of conceptual operation that one may wish to perform on elements of a data structure that is represented in terms of dynamic shared values. These conceptual operations are: single read, single write, shared read, and shared write. However, the shared write operation is really the compound operation of element replacement, which can be emulated by first locating, deleting, and then creating a new data structure element. The mechanics of the conceptual operations can be divided into three activities: locating a value; obtaining/giving data from/to a value (binding); and controlling the accessibility of a value. It is possible to derive separate functions to perform each of these activities and then to emulate the conceptual operations. For example, a shared read operation is a search followed by a binding. However, splitting the conceptual operations in this way can lead to problems with atomicity. For example, a shared value could be located by a thread, but this value could then be deleted by another thread before the first thread could make its binding. This by itself is not a problem but it makes using such programming mechanisms very difficult because each communication call must be checked to ensure that it is not in a race condition with other competing calls. Hence, the constructs that were developed are a compromise between the appeal of a functional specialisation and the requirements of atomicity.

Every Tyger-C program contains a globally accessible object called *SVS* which has three intrinsic operations defined for it to support dynamic shared values: *find*(), *hold*(), and *release*(). SVS operations are invoked by name and implicitly

operate on a program's SVS object. Alternatively, the optional SVS identifier, *LOCAL* or *GLOBAL*, can be used to qualify each SVS operation invocation to select the SVS to which the operation is to apply. For example, "**LOCAL**:find( .. )" looks for a shared value in the local SVS. The LOCAL SVS (the default) refers to the SVS of the parallel program being executed, while the GLOBAL SVS refers to the SVS of the operating system's environment. Thus, the GLOBAL context can be used to communicate between concurrently executing programs (further elaboration is presented later in the *global environment* section of this chapter).

In addition to the three intrinsic SVS operations there are two helper operations: *sv()* and *expand()*. The **sv**() operation constructs a dynamic shared value from a list of identifier and data fields, returning a dynamic shared value handle to the newly created value. For example, "**sv**("Day", ("Monday"))" creates a shared value with the identifier field "Day" and the data field "Monday". A dynamic shared value handle is a reference to a single dynamic shared value and cannot be changed once it has been set. The **release**() operation (single write) takes a dynamic shared value handle and inserts its associated shared value into the SVS. The operation will succeed if a duplicate shared value does not exist in the SVS, and it will fail returning an error code if one does. For example, "**release**(**sv**("Go"))" writes the dynamic shared value {"Go"} into the local SVS.

The **find**() operation (shared read) takes a shared value signature or template and blocks until it can find a matching value in the SVS. When one has been found, the operation atomically binds any unbound fields in the template to fields in the matching value. The find() operation does not return a handle because the shared value it would refer to could be deleted by another thread, therefore leaving an empty handle. For example, the operation "**release**(**sv**("bin", 1, 100))" creates a new dynamic shared value whose fields can be retrieved by "**find**("bin", ? &X, ? &Y)" where X and Y are integer variables. Note that the "&" character means "take the address of" and the "?" character is used to distinguish between value and variable fields.

The **hold**() operation (single read) searches for and binds a shared value in the same way as the find() operation, but it also returns a handle to the shared value that it matched in the SVS and makes this shared value otherwise inaccessible. One intended use of the hold() operation is to help implement the conceptual shared write operation. Consider a binary tree represented by dynamic shared values as nodes. Nodes are inserted into the tree by searching for their correct position, *modifying* the shared value that will be the parent of the new node, and

then creating the new node itself. The point of insertion of the new node is found by holding a node (starting at the root) and testing whether the node to be inserted should be a left son or a right son. If a son already exists on the appropriate branch, the current node is released back to the SVS and the son located. When the appropriate position has been located, a new version of the father node is created from the existing node and it is released together with its son into the SVS.

One point to note is that during the search for the insertion position only key identifier fields of the node have to be matched, not any of the data fields, therefore, cutting down on the amount of communication overhead. When the insertion point has been found the handle of the matched node and a full node template of empty fields can be passed to the **expand**() operation which will bind the fields of the template to the values of the fields of the node referenced by the handle. This allows the full contents of the existing node to be retrieved and a new parent node to be subsequently constructed. For example, given a shared value handle *Parent* which references a shared value {"node", 6, 10, 20, (1, 5, 3)}, "**expand**(Parent, "node", nodeId, lson, rson, ? &d1, ? &d2, ? &d3)" binds the variables *d1* to *d3* to their corresponding fields. The expand() operation returns a status code, but still carries out the bindings (see matching rules), if there is some mismatch in the number of the fields of the shared value supplied by a handle, and the number of fields specified to retrieve those values.

While the three intrinsic SVS operations give the impression of a message passing environment they actual operate in terms of shared memory. Associative naming is used as a flexible method for building and manipulating data structures, but also it raises the memory abstraction from that of simple shared memory. During the execution of a program, SVS operations construct dynamic shared values in shared memory and arbitrate accesses to them by setting flags controlling their visibility (or availability) in the shared associative address space. The best example of this is the hold() operation which does not actually delete a shared value, but instead makes it unobtainable except via its handle. When a thread terminates and relinquishes any shared value handles it holds, the corresponding shared values are deleted as they can longer be accessed. Thus, shared memory efficiency is exploited at the low level, while message passing semantics at the top level challenge programmers to devise models that overcome the partitioning difficulties encountered in dealing with complex concurrent applications.

## Matching Rules

When a SVS operation is invoked to manipulate dynamic shared values a template must be supplied to identify those values of interest from others in the SVS. The same rules employed by the Linda model are used for matching identifier fields and are not discussed further here. For data fields different matching rules apply. Here, fields either all act as receptors for data or all act as transmitters depending upon the sense of the operation. For reading, fields are matched left to right with values being bound to available fields using a shared value matched on the identifier fields. If there are insufficient data fields in the donor shared value, the operation still succeeds using the bindings it can make, but returns a status code indicating the situation. Similar action is taken if there are too many fields in the supplying value. A positive return value indicates the number of excess donor fields and a negative return value indicates the number of missing donor fields.

For example, the operation "**find**("Row", 1, (? &X))" with integer data variable X can match the value {"Row", 1, (100)} binding X to 100, or the value {"Row", 1, (50, 200)} binding X to 50 and returning an appropriate status code. In addition, the operation "**find**("Row", 1, (? &X, ? &Y))" with integer data variables X and Y can match {"Row", 1, (100)} binding X to 100, but leaving Y alone and returning an appropriate status code.

## Programming Example

One of the quickest ways to sort a list of integers is to use a statistically based partition sort [Noga85]. Such a sorting technique can be made to execute efficiently in parallel by using a combination of static and dynamic shared values. To start with the unsorted list is represented as an array of shared static values, on which some short statistical sampling is undertaken (possibly in parallel) to estimate the range of values in the list. Once this has been carried out, a range of sorting bins is created (represented by dynamic shared values), with each bin covering a unique range of values determined from the sampling so that every list element can be put into just one bin. Next, each element in the unsorted list is inserted, in order, into its bin. Finally, the sorted list, which can again be represented by an array of static shared values, is created from the ordered concatenation of every bin.

One way in which parallelism can be applied to this algorithm is to use a thread of control to manage each bin data structure, and for a separate group of threads to partition the unsorted list by creating dynamic shared values that will

be read by the appropriate bin manager threads. One problem with this implementation is that many threads can be involved, leading to high parallelism overheads. Of course, it is possible to make threads responsible for fielding dynamic shared values for more than one bin but this has two disadvantages. Firstly, the code to do this can be a little untidy, and secondly poor load balancing may result if a few threads end up doing all of the work.

An alternative implementation strategy is once again to have a group of threads partition the unsorted list, but this time each thread inserts values into bins directly (i.e. a distributed data structure approach). Here, the most appropriate number of threads can be used, though poor load balancing can result if threads have to repeatedly compete with one and other when inserting values into a bin. To reduce the time it takes to insert a value into a bin, bins can be stored as dynamic shared values representing binary trees. For example, a dynamic shared value to represent a bin can be made from six fields *"bin", BinId, NodeId, LeftSon, RightSon, Data* (though the first field is not strictly necessary). If -1 is taken to mean *no son*, then given in order the data values 20, 30, 10, 60, 50 for bin *B*, the corresponding bin binary tree is:

{"bin", *B*, 1, -1, 2, 20}, {"bin", *B*, 2, 3, 4, 30}, {"bin", *B*, 3, -1, -1, 10},
{"bin", *B*, 4, 5, -1, 60}, {"bin", *B*, 5, -1, -1, 50}.

Of course executing the operations to generate and retrieve dynamic shared values for a bin can be quite expensive so a simpler data structure such as a linear linked list may be more efficient. However, this is difficult to assess in advance of actually executing a test program because timings are sensitive to factors such as the distribution of the numbers that are sorted and algorithmic parameters such as the number of bins. Nevertheless, using dynamic shared values is a tidy and flexible way to represent the shared bins that allows a programmer to alter the bin data structure without too much recoding.

## Epilogue

Dynamic shared values by themselves are quite a low level mechanism for parallel programming, in common with Linda tuples. They are good for specifying the nuts and bolts parts of operating on shared data, but some skill in using them is needed to code higher level operations such as insert-into-tree() because it is relatively easy to make mistakes in, ordering the communication operations, or in realising potential parallelism. As such, it can be quite difficult to devise effective parallel programs using these mechanisms because often insufficient support is

given to programmers to help them structure their programs. To address this problem, research has been carried out to devise paradigms and models for parallel programs that have natural implementations with tuples or dynamic shared values (see Carriero's classification of parallel concepts and techniques in chapter three). Nevertheless, this is only a partial solution and it is probable that more language support is in fact needed. Only a part of the research carried out on dynamic shared values has been presented here, with ideas on conditional value matching and special control shared values being omitted. While such ideas hold the promise of improving the utility of dynamic shared values, the emphasis of the Tyger model now rests firmly on static shared values as a better (more structured) route to effective parallel software.

## 4.3 System Components

One of the central ideas of this thesis is that effective parallel programming cannot be solely addressed by a programming language but instead must be considered in terms of some larger computing environment. For this thesis, given the shared memory multiprocessor architecture as a hardware platform, the next level up in the computing environment is the operating system. Its role is to make the hardware resources of a machine more accessible to users, by assuming responsibility for managing hardware resources, software resources such as files, and to undertake some of the housekeeping functions to support multiple threads of control. Once the system environment has been established, a compiler or interpreter for a parallel programming language can be written, to marry the cosseted abstractions of the program designer to the hard realities of actual implementation.

As it turns out, the design and implementation of parallel programming abstractions in a language are the most important aspects of the environment in this thesis and so have been covered first, commensurately being paid the most attention. Nevertheless, when the form of the programming language and system abstractions have been determined, software tools can be constructed for program design and monitoring. Such tools must not only operate correctly and meaningfully, but must be capable of exchanging information and being integrated together. Thus, a uniform high level approach should be taken across a set of software tools, founded on collecting low level system information and other implementation details and translating them into the familiar terms found in high level programming abstractions. Hence, three software components, namely

the operating and run time systems, programming language, and software tools, make up a complete computing environment.

### 4.3.1 Ideal Thread Environment

Only a brief outline of the idealised operating system and Tyger run time system, collectively called the *global environment*, are presented here as the main emphasis of this chapter is on shared values. The reason why this material is presented is to sketch the underlying models that are thought to be appropriate for supporting the Tyger model. The intended host architectures for the global environment are shared memory multiprocessors, but it should be possible to support the global environment on distributed memory multiprocessors or across networks of multiprocessors. The global environment is conceptualised as a symmetric multiprocessor operating system that executes jobs by using two tiers of addressing to support parallelism between jobs and parallelism inside jobs. In addition, as the distribution of the global environment is largely hidden, load balancing should be permitted to enable jobs to migrate between host machines. Contemporary operating systems such as Mach [Acce86] and CHORUS [Rozi88] already provide similar levels of functionality, so these seem good starting points for the global environment. Using the Mach terminology, a *task* corresponds to a shared address space where many lightweight *threads* may be executing, with the operating system itself containing many such tasks at any one time. Likewise, the global environment consists of many such tasks that correspond to the activations of programs, with an activation of a Tyger program being given the special name of an *errand.*

Internally, an errand consists of a notional master thread of control, some *worker* threads, and some *helper* threads. All of the threads of control have equal functionality as regards their capability to manipulate the operating system, though the three types of thread have different roles. All threads communicate and synchronise with each other via shared variables, as it is envisaged that threads of control inside an errand are tightly coupled, interacting frequently. Basically, the master thread of control corresponds to the activation of the main procedure of a Tyger program. Its role is to oversee the execution of the errand and to take action to terminate the errand in the case of an unrecoverable error such as a program deadlock. As it turns out, a master may not be a single thread of control as it may create helper threads, delegating roles such as deadlock detection and statistics collection to them. Worker threads correspond to the threads created in a Tyger program by the Tyger iterators or the parallel-and operator. These threads

are used to execute Tyger procedures efficiently in parallel and largely leave administrative duties to the master. Helper threads are invisible to the program level as they are used to simplify the implementation of the run time system. Thus, an errand is an abstraction for a parallel program that hides the details of its implementation, in terms of its number of threads of control, from other programs, and acts as a common repository for operational information for those threads executing inside of it.

Although an errand is a vehicle for executing a Tyger program with internal parallelism, inter-program parallelism can be modelled by a group of concurrently executing errands. Two levels of inter-errand communication need to be addressed. At the bottom level, the operating system must have some fundamental inter-task communication mechanism. Rozier *et al* [Rozi88] argue that message passing is probably the best method for achieving this. Moreover, message passing seems to be a natural solution for overcoming the logical distribution of tasks into separate address spaces and the physical distribution of tasks across separate machines. At the top level, programmers need a communication mechanism that can be used to express complex concurrent behaviour in a familiar and convenient way. Dynamic shared values seem then to be a plausible candidate for this mechanism as they can model parallelism in a way that can be straightforwardly mapped on to a group of errands. At the programming level, it is envisaged that errands are loosely coupled, interacting less frequently and less predictably, so they do not need a direct sharing mechanism for extra efficiency. Instead dynamic shared values provide a conceptually shared associative name space between errands that can be mapped down into the operating system's message passing framework.

To create a dynamic shared value which is shared between several errands two steps are necessary. Firstly, the errands that are to cooperate need to be grouped together so that they share a common *GLOBAL* shared value space (SVS), which was earlier mentioned under dynamic shared values. Secondly, an appropriate command must be issued by a thread in an errand to create a shared value which should be directed to its GLOBAL SVS. The GLOBAL SVS is disjoint from the *LOCAL* SVSs of all the cooperating errands and is also disjoint from those belonging to any other groups. Thus, there are in fact three levels of addressing within the global environment, namely: intra-task, inter-task and intra-group.

### 4.3.2 Supporting Tools

Although the design of the Tyger model has grown out of the familiar control flow models found in conventional programming languages, transforming an algorithm into an effective Tyger parallel program is not always a straightforward undertaking. Fortunately, research with other parallel programming approaches suggests that software design tools can ameliorate this situation. For instance, it is quite common to find interactive software tools that can display information about the control flow and thread structure of a parallel program. For Tyger programs, such a tool would be very useful as it could be used to display the sequence of stripes in a parallel program (also substripes with an extended model). Moreover, if such a tool were to be coupled to an editor, code for a spine thread and its workers could be entered in a window driven fashion rather like the folding editor used with Occam systems. This has the twofold effect of only allowing the expression of valid stripe and thread structures, and allowing programmers to achieve a good overall understanding of the thread control relationships within a program. Another potential capability of such a design tool is the visualisation of the scope of shared information. This could be primarily used as a method of assisting programmers to check the scope of shared and local information to make the job of run time debugging easier.

One of the easiest software tools to implement to support the Tyger model is a run time monitoring tool. Here, low level information can be gathered about the access patterns of a group of shared values merely by examining their synchronisation variables. This means that no extra information gathering code has to be added to a parallel program to be monitored, minimising the perturbation effects on the run time behaviour of the program. In addition, the use of static shared values means that programs will not (intentionally) contain statements that overwrite shared data during a parallelism stripe, making it easier to trace the execution of such a program. However, an external viewing mechanism must be provided to log or display in real time the changes in the synchronisation variables, which may have some small effect on the performance of a test program (due to contention). In a similar vein, if the static shared value history mechanism is exploited by a parallel program, facilities for limited program playback can be easily constructed by providing a viewing mechanism for the shared value histories. Moreover, monitoring and debugging could be arranged conveniently so that a parallel program could be debugged by advancing it a stripe at a time.

Summary of the use of Shared Values

Shared values and their attendant control constructs are useful mechanisms for the construction of parallel programs. Static shared values provide the most support for the data partitioning programming paradigm, which is thought to be the most important method of exploiting significant amounts of parallelism. However, by allowing a programmer to define new classes of object with synchronisation properties based on shared value semantics, two major benefits are gained. Firstly, it is possible to effectively support parallel programs that follow functional decomposition. Secondly, a programmer can structure a parallel program in an appropriate way to carry out its application, rather than some other way to accommodate the basic shared value constructs. Dynamic shared values are also capable of effectively supporting parallel programs that follow data and functional decomposition. But such programs should be constructed by working to standard models for algorithms because dynamic shared values do not provide much help for parallel program structuring.

To enable the effective use of shared values an operating system must offer services similar to those provided by the *global environment*. Key services include the provision of shared memory and the management of multiple lightweight threads of control. Finally, software tools can be used in the design of programs that use shared values to exploit parallelism, and in turn, can be used to inspect the operation of such programs. The use of such tools can lead programmers to a better understanding of their programs which will hopefully dispel some of the mystique surrounding parallel processing.

# *Chapter 5*

# Implementation of Shared Values

After discussing the Tyger model in chapter four the purpose of this chapter is to explain an actual implementation of some of the Tyger constructs. The chapter starts by giving a description of the shared memory multiprocessor architecture that is used to execute shared value based parallel programs, and continues with a description of how static shared values can be implemented mentioning the tradeoffs that can be made. Only static shared values are covered here, as the implementation of dynamic shared values is similar to the Linda work carried out by Carriero [Carr87]. Finally, a short section examines the potential compile time optimisations to static shared values that are possible.

## 5.1 Multimax Multiprocessor

To set the context for the experiments in chapter six, the hardware platforms used to obtain the empirical performance figures are described. These systems were Encore Multimax multiprocessors running the Umax 4.2 operating system (a parallelised version of Unix 4.2). The central component of a Multimax multiprocessor is the system bus. It has a true data transfer rate of 100 Mbytes/s and can accommodate up to twenty function cards. A function card can be a dual processor card, a shared memory card (4 or 16 Mbytes), a mass storage/ethernet card, or the system control card. Various combinations of cards can be used to form a system with the provisos that one system control card is used with up to eight shared memory cards (maximum 128 Mbytes) and eleven processor/mass storage cards (maximum 20 processors). Each dual processor card houses some cache memory (local memory) with logic to ensure cache consistency throughout the entire processing system.

During the course of this research test programs were run on four Multimax configurations derived from two basic machines. Initially, eight National Semiconductor NS32032 processors were configured with 16 Mbytes of memory in one machine, subsequently expanded to eight NS32332 processors with 64 Mbytes of memory (320 system). A second Multimax was acquired with six NS32532 25 MHz processors and 48 Mbytes of memory, later increased to fourteen NS32532 30 MHz processors with 96 Mbytes of memory (520 system). (The problem studies

focus only on the two final systems (320 and 520), although data from experiments carried out on the earlier systems is presented in some of the summary tables.)

The allocation of cache memory associated with each generation of processors was altered by the manufacturer to offset the increased frequency of bus traffic as processor speeds increased. For the 320 system, 64 kbytes of *write through* cache memory were provided per processor card, while in the 520 system this was updated to 256 kbytes per card using *write deferred* cache memory. The availability of cache memory is very influential on the performance of shared values as their implementation has been coded to take advantage of the quicker read and write times. Hence, caching is of paramount importance to successful multiprocessing on shared memory multiprocessors, and of commensurate importance to the success of shared values, so a brief summary of cache operation follows.

### 5.1.1 Caching Strategies and Cache Coherency

When constructing early multiprocessors experimenters found that contention on a shared bus could limit the effectiveness of employing multiple processors, therefore nullifying the benefits gained through the use of parallelism. However, later research has shown that if multiprocessor systems are constructed with sufficiently large cache memories, the sequential performance can be improved because a processor can work out of a cache memory rather than main memory. Moreover, as a direct result of this, in most instances where multiple processors are active there are no serious bus contention problems so enabling effective parallel processing to take place. Nevertheless, it is still possible for pathological behaviour to invalidate the advantages of caching and this phenomenon is explored later in the next chapter.

Cache memories operate by providing fast access to frequently used data items, but in a shared memory multiprocessor system caches are used to allow processors to maintain their own private copies of memory locations so obviating the need to request them from shared memory, with the potential to cause bus and memory bank contention. Problems can arise with multiple cache memories because a consistent system wide picture of the contents of each shared memory location must be maintained at all times. That is, if a processor changes the value of a memory location which is held in its cache and the value of same location also happens to be residing in another cache, the other copy must be marked as invalid as it no longer holds the current state of the location. This problem of maintaining

consistency between caches is known as *cache coherency*. A solution to this problem often adopted in shared bus multiprocessors is to empower the caches to monitor bus transfers and to act on the information accordingly. This kind of design gives rise to the term *snoopy cache*, and several authors have published details on how such devices actually work [Good83, Fiel84, Fran84, Papa84, Rudo84, Katz85, Arch86].

The two protocols used for the snoopy caches in the two Multimax systems are *write-through* caching and *write-deferred* caching. In both methods, when a processor requests the value of a location and the most up to date value is present in that processor's cache it is just read from the cache. On a cache miss, the value is read from the location in shared memory; but in the write-deferred protocol, a special cycle has to be executed in order to force the current state of the location back to shared memory. For cache writes the protocols also differ because of optimisations geared towards reducing the of number bus uses. In the simplest scheme, write-through, each write operation that updates a value in a cache memory forces an update to be made to the corresponding memory location in main memory. All other caches observe this update and if they are holding the *old* value then they mark it as invalid. If subsequently, the value is needed and the value in the cache is invalid, a read operation has to be done to main memory. In an attempt to minimise the bus traffic resulting from write operations several schemes have been proposed [Fran84, Papa84, Rudo84, Katz85] including Goodman's *write-once* scheme [Good83]. This protocol dictates that if a processor writes to a value held in its cache then only the first write generates any bus traffic with subsequent writes executing more quickly as they are deferred and only proceed as far as the cache. If a processor writes to a location owned by a remote cache, a special bus cycle must be executed in which the current value of the memory location is written back to main memory so that the requesting processor can write to it. In practice some 15-20% of memory requests are write operations so the optimisations brought about by write-deferred caching reduce bus traffic by one sixth according the manufacturer's estimates.

### 5.1.2 Operating System

The operating system run on the Multimax systems used in the experiments appears to a user as a conventional Unix 4.2 operating system. The overall structure is the same but much of the kernel was rewritten to allow concurrent accesses and several additional system calls were added to manage areas such as shared memory and System V shared segments. Unix processes are

multiprogrammed according to the current load and the number of available processors, and housekeeping jobs, such as daemons, are executed to perform system administration activities.

When a program is executed its performance is subject to numerous external factors common to all timesharing systems such as bus and memory contention, paging, and the anomalies of physical storage devices such as disk drives. The effects of these intrusions on the performance of test programs has been kept to a low level by running programs at quiet times to reduce interference from other activities. Due to the large amount of computing resources required for testing, however, some programs were run on busy systems with the proviso that the total number of active CPU-intensive processes did not exceed the number of processors. This yields acceptable performance results though the results are coloured somewhat by the other activities present on the system. For example, programs using a large block of shared memory may suffer adversely due to poorer caching and paging behaviour under a heavier system load than under a lighter load. In the case where there are more CPU-intensive processes than processors the interpretation is more open because of the large variations in timings. Some research has been carried out to examine the underlying trends here [Bert89], but in this thesis results of this type were discarded as they were thought too difficult to reason about convincingly.

### 5.1.3 Lightweight User Level Threads Library Model

All of the parallel programs considered in the next chapter have been implemented twice via two user level threads libraries. The explicitly parallel test programs (*gold parallel programs*) use the libraries directly, while shared value constructs use them implicitly, that is explicitly in their implementation. The purpose of a threads library is to act, in effect, as a small operating system formed out of lightweight calls and components. Hence, a library may offer services such as thread creation, thread manipulation, synchronisation, and memory allocation. Figure 5.1 shows a mapping between lightweight tasks, Umax processes and processors. At the user level many tasks can be created by a programmer and these are scheduled for execution by Umax processes via the task management portion of the lightweight threads library. Similarly, at the system level, Umax processes are scheduled for execution on real processors by the operating system.

The major advantage gained from using a user level threads library is that a programmer can easily produce a desired concurrent system, which can execute

very efficiently, and is fairly independent from the underlying operating system (therefore in principle, portable across machines with similar architectures). However, the big drawback is that there is no guaranteed way of enforcing protection between concurrently executing threads of control, so careful memory management is needed to stop threads from interfering with each other's data structures.

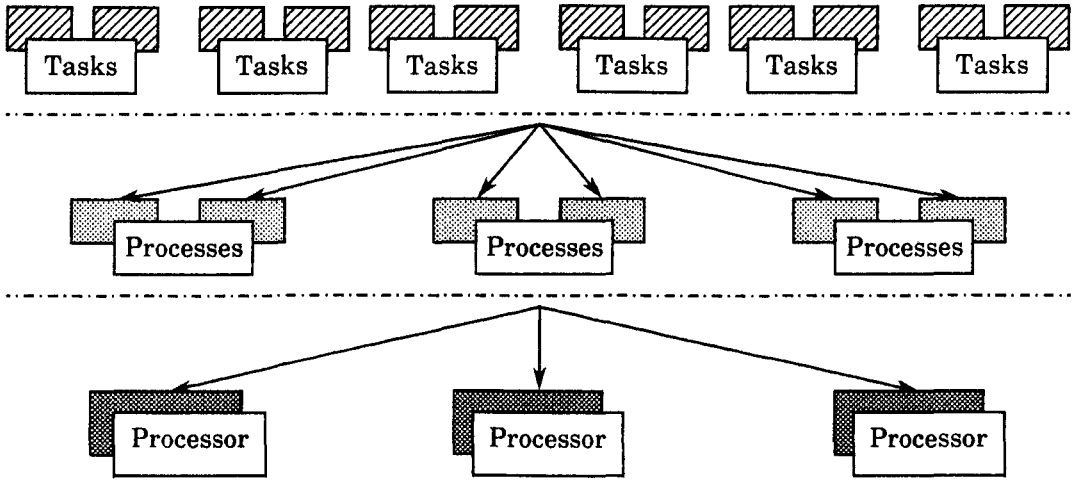



Figure 5.1 - Hierarchy of parallelism.

Two threads libraries were available for use, the Encore tasking library which uses threads of control called *tasks*, and the Encore Parallel Threads library (EPT, based on the Brown Threads package [Doep87]) which uses threads of control naturally called *threads*. Although shared value constructs have been implemented via both libraries, only the results for task based programs are presented. This is because the tasks library more closely resembles an ideal library capable of supporting shared values rather than the more heavyweight threads.

### 5.1.4 System Characteristics

When analysing the run time behaviour of programs it is useful to have order of magnitude estimates for the times taken to execute certain fundamental operations. Equipped with this knowledge it should be possible to predict the time overheads involved in say forking several processes at the start of a parallel computation.

| System (Processors) | Process Creation | Thread Creation | User Level Synch. | OS-Level Synch. |
|---|---|---|---|---|
| NS32032 | 600/50 ms | 1000/300 µs | 70 µs | - |
| NS32332 | 250/100 ms | 350/250 µs | 13 µs | 0.6 ms |
| NS32532 | 150/50 ms | 200/100 µs | 7 µs | 0.5 ms |

Figure 5.2 - Parallelism characteristics.

The figures presented in Figure 5.2 are mean sequential times to perform the specified operations, none of the operations themselves were internally parallelised. (One entry is omitted as data was not collected when the system was available.) Two figures are presented for process and thread creation times. The first figure represents the time to create an initial thread of control, which incurs an extra overhead for factors such as storage allocation. The second figure relates to the times for creating subsequent threads of control that do not have so great an overhead.

Each timing includes a component for a procedure call overhead which is significant only in the case of user level synchronisation. For the user level synchronisation, the fastest method of synchronisation (spinlocking) was measured. When operating a spinlock only one or two memory references have to be made in order to implement a lock operation (if the lock is acquired). For more elaborate user level mechanisms such as barriers, events and semaphores, several spinlock operations may be needed thereby increasing the synchronisation times accordingly. For the operating system synchronisation, System V semaphores were used as no suitable alternatives were available. These are quite a sophisticated mechanism, though the times recorded were for the simplest operations, such as signalling a semaphore, and do not represent the very much larger times that can occur if several processes have to be manipulated by a single call. For example, several processes could be waiting (blocked) on the value of a semaphore, and if it was necessary to unblock all these processes the total execution time of a semaphore operation would be very much longer in comparison to those for other operations on the semaphore.

The important observation from Figure 5.1 is that user level operations are on average two to three orders of magnitude faster than operating system level calls, therefore permitting more effective parallelisations of small data sizes or small grain sizes. Nonetheless, the initial one time cost of creating processes to run the

lightweight threads must always be taken into consideration. However, once the heavyweight processes have been created they can be reused throughout the duration of a program to execute lightweight threads at lightweight costs.

## 5.2 Implementing Shared Values

Most of the work in implementing the shared value constructs is straightforward though sometimes obtuse, but the process of designing and coding an efficient implementation for a Multimax multiprocessor is quite demanding. A shared value can be notionally represented as a *(value, synchronisation-check)* pair. The challenge that is set is how to physically store and intraconnect each shared value pair such that both the time taken to operate on the synchronisation check and the total storage requirement are minimised. The time constraint is quite natural in so much that the overheads of operating on shared values should be kept to a minimum to firstly, retain the ability to perform some fine grained work efficiently, and secondly not to add massive overheads when dealing with large numbers of shared values. The storage constraint only really applies to large problems, but it is these very problems that are likely to yield the most rewarding gains through the use of parallelism and should not be precluded because a shared value based program exceeds the amount of addressable storage whereas a conventional storage model may be accommodated. For example, consider a dusty deck style program that takes one week to run and needs 50 Mbytes of storage out of a system pool of 100 Mbytes. If this program were parallelised using poorly represented shared value constructs, then the program's memory requirement could potentially double or even triple making it unexecutable without memory extension. Unfortunately, buying extra memory is not a good solution to this problem because real applications are often scaled up to take advantage of any increase in hardware resources.

### 5.2.1 Representation of Shared Values

One of the important considerations in writing efficient programs is the real memory requirement. If a shared memory requirement is very high, excess paging may occur at run time and any benefits gained through the use of parallelism may be lost. Thus, a compact, yet easy to access representation for shared values must be chosen. Two representations come to mind,

(a) Bit representation. A shared value is stored as a loosely coupled pair consisting of a value and a remote tag. Each of the tags is represented by a

single bit with separate a single lock byte used for synchronisation. The advantage of this scheme is that with $n$ shared values, wordlength $w$, only $\lceil n/w \rceil+1$ extra bytes are needed. A modification to this scheme is to allocate more locks, up to one per value, to reduce contention on the lock.

(b) Byte representation. A shared value is stored as a tightly coupled pair consisting of a value and an adjacent tag, with each of the tags represented by a byte. This uses more storage than a bit based method, but consequentially has a shorter operating time as the seek time for an individual tag is shorter. Again separate storage is required for synchronisation variables.

Both methods were implemented on a trial basis and it was observed that a bit representation resulted in order of magnitude slower access times than a byte representation. Henceforth, a byte representation was chosen, although the underlying representation should really be parameterisable depending on a user's desire for storage or speed.

A concern that has been overlooked up until now is the time taken to create a shared value object when its constructor is invoked. Normally, when shared memory is allocated only a single call has to be made, which is albeit quite costly for large memory requests, but is nevertheless sufficient to provide a pool of shared memory for all a program's needs. If $n$ calls have to be made to a shared value constructor or $n*n$ for an array, the overhead of the procedure calls alone can be prohibitive in cases of limited parallelism. Fortunately, all of this extra work can be neatly side-stepped by taking advantage of the properties of the memory allocation system call. It returns zero-filled pages which, as it happens can be directly mapped onto initialised shared values without any additional processing.

### 5.2.2 Synchronisation for Shared Values

If shared values are actually used to communicate between threads of control, rather than being used for holding a shared result, appropriate synchronisation code must be used when referencing them. This synchronisation cannot be optimised away by a compiler because it is needed to enforce the correctness of a program. Thus, efficient implementation of shared value synchronisation is required.

A shared value has two states *full* and *empty*, implying that synchronisation must be provided so that the correct actions are taken when a shared value is read or written in a given state. The Encore Multimax provides hardware support in the form of a test-and-set operation which, it turns out, is sufficient to implement very efficient shared value synchronisation. The test-and-set operation works on variables that are regarded as locks as they also have two states *locked* and *unlocked*. Two implementation strategies come to mind for synchronisation based on locks which are termed *locked-first* and *unlocked-first* synchronisation.

Locked-first Synchronisation

Upon allocation, shared value synchronisation locks are set to the locked state. Threads that wish to read a shared value continually test the state of its lock until the lock is found to be in the unlocked state. A thread that wishes to write to a shared value firstly tries to set the corresponding lock to the locked state. After it fails, as the lock is already locked, it sets the value portion of the shared value and changes the state of the lock to be unlocked.

After a shared value has been written, its lock will be in the unlocked state. Threads wishing to read a shared value test its lock and upon finding the lock unlocked, read the value and continue normally. Now, a thread that wishes to write, incorrectly, to a shared value firstly tries to set the corresponding lock to the locked state. After it succeeds, that is lock was unlocked, it raises an exception indicating multiple write operations to a shared value.

Although the locked-first algorithm appears to satisfy shared value semantics there is in fact a fault whereby write operations to a shared value can be lost. Consider the case of multiple threads wishing to write to an empty shared value. One potential interleaving of events is that the writing threads test the lock and finding it locked, all set the value portion of the shared value. Hence, updates to the shared value are lost and what is worse, this is not detected by the final operation of unlocking the lock. The algorithm can be modified, however, by checking the state of the lock before terminating the write operation and raising an exception if the state of the lock is found to be unlocked. However, adding the extra check increases the time to operate on a shared value, and a small problem that still remains is that all but one of the values written in a multiple write are lost, meaning that potentially valuable debugging information can be destroyed.

Unlocked-first Synchronisation

Upon allocation shared value synchronisation locks are set to the unlocked state. Threads that wish to read a shared value continually test the state of the lock until the lock is found to be in the locked state. A thread that wishes to write to a shared value firstly sets the value portion of the shared value and then tries to set the lock to the locked state. If it succeeds, that is the lock was unlocked, it continues as normal. Otherwise if it failed, that is the locked was locked, it raises an exception indicating multiple write operations to a shared value.

After a shared value has been written, its lock will be in the locked state. Threads wishing to read a shared value test its lock and upon finding the lock locked, read the value and continue normally. Threads that wish to write, now incorrectly, proceed as before.

Once again, the values written by multiple write operations will be lost but an exception will be correctly raised indicating the presence of a multiple write. Unlocked-first synchronisation was implemented for the C++ version of shared values because it is a simpler, and therefore a quicker, method of synchronisation than locked-first.

## 5.3 Compile Time Optimisations

As a general comment on current microprocessor and compiler technology, it seems that some of the complexity that was previously supported in hardware is now being integrated into the optimising stages of modern compilers. This trend is typified by RISC microprocessors and their compilers. The kinds of optimisations that are performed by such compilers are on the whole language independent but the dependency information that they work with can be further exploited to enable tuning of specific language constructs.

One of the most well known compile time optimisations is the inlining of procedures. Although this technique has some drawbacks (mentioned in chapter two) very real performance benefits can be realised for shared values. In point of fact, consider calls to enforce shared value semantics on read and write operations to static shared values. Every reference to a shared value would involve a procedure call to check the synchronisation unless the synchronisation code could be inlined. As it turns out, the synchronisation code amounts to little more than a

few instructions which means that the procedure call alone amounts to fifty percent of the overall performance cost of implementing static shared values.

A more Tyger specific optimisation that can be accomplished, however, is the removal of unnecessary synchronisation. The most striking opportunity for this to occur is when threads read a group of shared values in an interval but do not write to them. Here, no synchronisation is necessary as there are no problems in maintaining consistency with parallel read operations. Other cases where synchronisation is not required can also be identified (e.g. a thread reading a value it has just written). Optimisations of this kind can be accomplished by performing dependency analysis (e.g. Kuck's dependency tests) of the form described in chapter two. However, problems can arise if multiple procedures have to be analysed. In the case of optimising read operations to shared values made from a procedure, all calling instances of that procedure must be checked before it can be determined that it is safe not to include synchronisation code. In general, program complexity makes it very difficult to do this effectively, implying that synchronisation code must always be generated. However, there are a number of compromise solutions. Firstly, code for two copies of a procedure could be generated with one containing synchronisation code and the other not. When a call of the procedure had been determined to be safe the unsynchronised procedure would be called, otherwise the synchronised procedure would used instead. As an alternative solution, in cases where no synchronisation is necessary the entire procedure (omitting synchronisation) could be inlined, though some analysis would have to be carried to assess the impact this would have on the performance of a program. The performance costs of not removing unnecessary synchronisation are described for several test problems in chapter six.

To reduce the time taken to locate and fix synchronisation faults in a program, some checking to detect multiple write operations to a shared value during a parallelism stripe can be carried out. This is merely a convenience for a programmer as this checking is performed at run time. In addition, other checking can be employed to detect the presence of indefinite postponement (e.g. a thread reads a value that is never written) and other related parallel programming faults such as deadlock, where threads wait for the other to write a value.

Further compile time optimisations can be made to check the Tyger constructs that manipulate dynamic shared values. Work of this kind is similar to that performed for Linda tuples described by Carriero [Carr87] and is not discussed further here.

# *Chapter 6*

# Performance Evaluation of Shared Values

This chapter explores the pragmatics of using an implementation of some of the static shared value constructs as a parallel programming model. The aim of this chapter is to establish that it is possible to produce an implementation of static shared values that can be used as a serious replacement, as regards efficiency, to imperative languages that explicitly exploit parallelism through the use of lightweight threads libraries. This chapter is a description of a series of four simple programming scenarios in which programs written using shared value constructs from Tyger-C++ are contrasted against similar C++ programs written using parallelism facilities from a user level threads library (gold programs), and sequential C++ programs.

Other parallel programming languages, such as Linda and Strand[88], were also considered for inclusion in the programming scenarios, but their inclusion was rejected for two reasons stemming from the nature of the scenarios. Essentially, in each scenario the efficiencies of several test programs are examined and the overheads that appear in the parallel programs are explained and countered where necessary. The explanation of program behaviours sometimes makes recourse to some low level implementation details, which in the case of the software mechanisms found in C++ and the threads libraries, can be done successfully on an intuitive level because of the low level of abstraction used in the implementation of these mechanisms. However, with programming models such as Linda and Strand[88], the analysis can only be done at the program level because the details of implementation are hidden. Thus, it is somewhat unfair to compare a model with an undisclosed implementation and potentially higher overhead costs to finely tuned shared value constructs. The other reason why a performance comparison between multiple languages might be somewhat unfair stems from the facts that shared values are aimed at exploiting medium grained data parallelism and the problems that are examined are prime candidates for data parallel execution. Other parallel programming approaches may well have the expressive power to produce elegant and succinct solutions to data parallel problems, but it is uncertain if the implementation of these solutions can really deliver optimal parallel performance. Of course, one could question the veracity of undertaking a performance based study rather than, for example, a usability based study. However, the usability of a programming mechanism is a somewhat

subjective quality and is coloured to some degree by the level of experience and degree of proficiency attained by a programmer. Moreover, the choice of suitable test problems is again important as no practical parallel programming mechanism can claim perfect solutions for all problems. Similar statements can be made about the expressiveness of a programming mechanism, though more formal analysis can be applied. Thus, while it would have been possible to investigate practically the limits to the expressiveness of shared values by programming a series of well known test problems, the limitations to shared values that exist have been described in chapter four and so are not discussed further here.

Parallel processing experiments were carried out on two similar shared memory multiprocessor systems (Multimax 320 and 520 systems), enabling an investigation into how much the architectural characteristics of a particular machine could influence the results. Components such as processor speed, the number of processors and the amount cache memory differed between the systems, as described in chapter five, making it possible to explore some of the time critical factors in the operation of shared values.

## 6.1 Measuring Parallel Processing Performance

There are several methods for measuring the performance of a parallel algorithm and its attendant parallelism constructs on a given parallel processor. This implies that no single measure can capture enough information to adequately describe the true *cost* of a computation, which is not surprising as the cost of a computation can have different meanings depending on its context. For example, the cost of a computation could refer to the time taken to run a job, or the resource utilisation, or even the financial cost. The most commonly used measures of performance include: the elapsed wall-time, the price/performance, the speedup, and the efficiency.

The *elapsed wall-time* of a job is the time taken to execute the job as measured, say, by a clock on the wall. This method of measurement can differ from the total elapsed time because the total time may be the sum of many partial times that originate from the concurrently executing subtasks of the job. From a system manager's (or computer scientist's) perspective it may be useful to know the total resource consumption of a job in terms of processor cycles, but in general the most useful measure of the effectiveness of a parallel algorithm on a given parallel processing system is simply the time it takes to run.

The *price/performance* characteristic of a program and system is the elapsed wall-time of a job divided by the cost of the system which ran the job. For instance, the price performance of a top of the range supercomputer may be relatively high despite the fact that the system may have a peak performance of several gigaflops. In contrast, however, the price/performance of a microprocessor based multiprocessor could be up to three orders of magnitude better. But, the crucial flaw in using this measure as an architectural guide comes if there is an upper bound on the actual elapsed run time of a job, such as in weather forecasting where currently the only tenable candidate systems are supercomputers.

One of the most frequently quoted measures of parallel program performance is the *speedup*, which is measured by recording the elapsed run times of a program while varying its number of threads of control. The speedup *s* is calculated from the formula $s = T_1/T_P$ where $T_1$ is the time taken for a single thread's run and $T_P$ is the time taken for a *p* thread run. There is some debate about whether $T_1$ is the time for a sequential program or the time for a parallel program running with only one thread. The difference between these two times comes from the overhead inherent in the parallel program. This overhead can be significant, leading to a misleading impression of a program when moderately parallel runs are slower than the sequential time, but still show a reasonable speedup with respect to parallel runs using only one or two threads. In this chapter, true sequential times are used for $T_1$ giving a fairer cost of the parallelism, with the exception of the last scenario which does not have a sequential version. Nevertheless, this is not quite the end of the story. The greatest point of contention in the interpretation of a program's speedup comes if $T_1$ is taken to be the time for the *best sequential program*. This may mean $T_2 .. T_P$ are generated by a parallel program based on a different algorithm to the sequential program, implying that the absolute time to process an application is being measured. Thus, as the goal of this chapter is to evaluate shared values, both the sequential and parallel algorithms for each problem are the same so that the effects of parallelising the sequential program can be examined, rather than the properties of the problem.

A related measure to the speedup is *efficiency* calculated from the formula e = s/p where *s* is the speedup and *p* is the number of processes. This metric gives a direct measure of how good a utilisation of the processing resources has taken place but again has some fallibilities similar to speedup. More precisely, with low numbers of processes high efficiency is essential for good performance but with very large numbers of processes a lesser efficiency may be acceptable.

Without doubt the most celebrated result in parallel program performance evaluation is Amdahl's Law [Amda67] which can be used to put an upper bound on the speedup a parallel program can hope to achieve. The law can be stated as

$$T(p) = T_S + T_P/p$$

where $T_S$ is the sequential time component and $T_P$ is the parallelisable part of a program. The exact interpretation of the law is sometimes seen as an open issue because for some problems quantities such as $T_S$ are not constant for a given $p$ and there is the assumption that $T_P$ is divided equally amongst the $p$ processes. These questions have given rise to a new metric called the *serial fraction* [Karp90] which examines the changes in $T_S$. The serial fraction can be computed from experimental figures using the formula

$$f = (1/s - 1/p) / (1 - 1/p)$$

where $s$ is the speedup for a given number of processes $p$. The intended use of this new metric is as a diagnostic tool in conjunction with, say, the speedup, to help attribute the cause of the lack of success of a particular parallel algorithm. For example, if the speedup of a given parallel program falls off while the serial fraction increases proportionally, one could argue that the extra work in managing the parallelism is responsible. Alternatively, if the speedup falls off and the serial fraction remains constant, the interpretation could be that the limit of parallelism in the program was being reached.

## 6.2 Primitive Shared Value Operations

The two intrinsic operations that can be performed on static shared values are the read operation and the write operation. The average times to perform these operations were measured together with the times taken to read and write conventional variables. The programs that were devised to measure these times were coded to allow for cached and non-cached effects. In addition, steps were taken to eliminate the effect of prefetching. This normally occurs on a Multimax system while reading consecutive locations from memory, when in the event of a cache miss, the next eight bytes of memory are fetched from shared memory. If four byte integers are read then only half the expected memory fetches are required with the other fetches coming from the fast cache memory, thus reducing the average fetch time. The results from programs written using static shared

values were compared against those produced by similar programs written to benchmark conventional variables and are summarised below.

On the 320 system the mean time to read a four byte integer from shared memory was approximately 1.7 microseconds, while the time to read the same value from cache memory was 0.8 microseconds. Write operations took 0.4 and 0.4 microseconds respectively. On the 520 system the times for reading were 2.0 and 0.2 microseconds, and 1.7 and 0.1 microseconds for writing. These results were obtained by assigning/reading from an array and were corrected to allow for the overhead of the array scanning loop and to negate the influence of prefetching. Having said this, the quoted values have small inaccuracies due to bus pipelining, which allows several memory operations to proceed in a partially overlapped manner. This means that the time to perform several consecutive operations was less than if they were performed in an independent piecemeal manner.

Difficulties were encountered in the measurement of the times for operations on static shared values because of the detailed interpretation that was needed to unravel the low level implementation used to support the operations. The time to write to a shared value on the 320 system was 11.3 microseconds, with the initial fetch of the shared value being made from shared memory. (If a shared value constructor had just performed operations on a shared value, then it is likely that the shared value would have been fetched from the cache.) The time to read the value of a shared value on the 320 system was 11.1 microseconds, which is faster than expected as it is likely that the value was cached, after having previously been fetched and written. For the 520 systems the times were 5.1 and 3.3 microseconds for writing and reading respectively. (The times recorded for reading a shared value include a time component for a procedure call, of around one microsecond, that was not inlined; in a more complete implementation this overhead would not be present.)

In a perfect world it would be convenient to believe that the times reported for these basic references to shared data were more or less constant, but unfortunately, it was possible to observe a wide degree of variation in these times if the operations were invoked on data that was shared between several active processors. Some experiments were carried out to examine the worst case behaviour in which multiple processes competed to read and write every element of a shared array [Lee90]. In the slower 320 system, eight competing processes did not significantly alter the times for reading or writing to cache or to shared memory, but for the faster 520 system the picture was very different. Here,

processors were able to generate bus requests far more quickly so leading to bus (and memory) contention. This contention linearly increased the operating times for fetch and store operations to shared memory, which resulted in a more than trebling of the basic times when large numbers of processors were active. Clearly, behaviour of this kind can make the interpretation of the efficiency of the execution of parallel programs very difficult, though the use of shared values is thought not to make the situation any worse.

## 6.3 Programming Scenarios

The effectiveness of a new parallel programming language can best be gauged from experience drawn from its extensive use in programming sample and real applications. Part of this assessment process is a subjective criticism of how easy a language is to use and how well it can be applied to the expression of programmers' designs. Unfortunately, any subjective comments regarding the usability of Tyger-C++ made by the author may be biased, so these will be limited and instead some figures are presented to make a case for concluding that the implementation of shared values can be achieved successfully.

Although the choice of content for the scenarios was very large and varied, four numerically oriented programming examples that were thought amenable for parallel execution were examined. These were chosen because a large proportion of today's computing workload, that is suitable for parallel processing, can be linked to specific scenarios for the purposes of comparison in terms of speedup and efficiency. More specifically, each scenario examines the effectiveness of shared values at a different level of granularity of parallelism. Thus, the results provide insights into the best ways in which shared values can be used for delivering effective parallel processing performance.

The first scenario investigates the trivial array assignment operation. This represents O(n) processor operations with O(n) data elements. This kind of operation is easily parallelised by vector processors but is thought, in general, to be too fine grained for MIMD multiprocessors. The second scenario is a bit more appealing for parallelisation as it is matrix addition with $O(n^2)$ operations on $O(n^2)$ data. The third scenario is matrix multiplication and should provide the best improvements from the use of parallelism as its characteristics are $O(n^3)$ operations on $O(n^2)$ data. The fourth scenario, the generation of fractals, is somewhat different to the other scenarios in so much that the number of operations to compute a result cannot be accurately predicted, though, an upper

bound can be fixed. Difficulties arise with parallelising problems of this type because of the complexity of efficiently allocating work to threads.

All the programs in the scenarios were tested over the same range of data sizes to enable comparisons to be made between the different problems. (For the 320 system the data sizes were 50 to 500 in steps of 50 and for the 520 system the data sizes were 100 to 1000 in steps of 100.) Furthermore, only computation times were reported in each scenario, excluding time components for input/output operations. This means that total program run times were not considered, however, this was thought to be appropriate as it was the methods of parallelising the problems that were being measured and not the absolute properties of the problems themselves.

All the execution times presented for the test programs represent the observed minimum execution times in each case. Timings were selected from a series of trials and are therefore representative of the best performance figures that were obtained from the test programs. Mean values for timings are not presented because of the difficulty in calculating sensible values due to the variance in the observed timings. This variance arose because separate runs of the programs encountered different operating system states in which other programs were executing and taking up memory. Steps were taken to minimise the effects of other programs by executing the tests at quiet times, but due to the symmetric nature of the operating system it was not possible to exclude it altogether. As a consequence of being unable to calculate mean timings, no confidence intervals are presented to give an idea of the range of timings that were possible. However, it was observed that repeated minimum timing trials gave largely consistent answers, varying by less than one percent for midrange data values (e.g. about 250 for the 320 system and 500 for the 520 system), with slightly higher variations for very small data sets. Thus, as the degree of random variation in the minimum timings was very small, shared value and gold parallel programs that were within one percent were said to take the same time. Fortunately, where a noticeable variation in execution times did occur between the shared value and gold parallel programs, this variation was much bigger than one percent and so ultimately random variation did not have a significant impact upon the results.

### 6.3.1 Array Assignment

The explicit parallelisation of a collection of individual assignment statements is believed to be unsuitable as a parallel programming style for MIMD multiprocessors. Firstly, the overhead of creating even lightweight threads of

control is massively, typically 70 times, greater than the time taken to execute an assignment. Furthermore, advances in microprocessor design and optimising compilers have meant that parallelism of this kind can be exploited by modern *superscalar* microprocessors through instruction pipelining and the utilisation of multiple functional units. Hence the next level of granularity that one may consider parallelising is the domain of array and structure valued operations.

Method

Given a source array A[0:n-1] assign the values 0:n-1 to the corresponding elements of the array *A*. The following C++ code fragment states precisely what should happen

```
for (int value = 0; value < n; value++)
{
    A[value] = value;
}
```

The time taken for this loop accrues from executing the loop test and branch, plus the time taken to assign the values of A[i]. (It is assumed that the values of *i* and *n* are held in registers.) It is possible to try to factor out the loop overhead in this experiment leaving only the time for the assignment, but this is hard to do accurately as the overhead is very small and can vary appreciably from run to run due to operating system effects.

A shared value based program which used the block iterator (described in chapter four), and a gold parallel program which used an equivalent work allocation strategy with shared variables, were written to generate execution times to compare against those from the original loop.

Observations and Conclusions

Given that the time to store an integer was of the order of a microsecond, the execution times of the sequential program were expected to range over 100's of microseconds. From Figure 5.2 in chapter five it can be seen that the time to fork a process is O(100) milliseconds which leads to the conclusion that process creation is the overwhelming factor in this problem. Surprisingly, the timings for the parallel programs were not as bad as was expected as they peaked when three processes were created and did not increase linearly as one might have thought. The first reason behind this behaviour was that process forking could itself proceed in parallel, as a child process could carry on the activity of forking in parallel with its parent. The second reason was that one of the major time

components in forking a process was the setup time for the shared memory region. This time could only be incurred once, which explains the constancy over the times for the runs with more than three processes. In addition, it also accounts for superior times being obtained from the shared value program over the gold program. This observation was quite unexpected, but resulted from a policy decision in the implementation of shared values which dictated that the minimum amount of shared memory should be used when creating shared values. In contrast, quite by chance, the gold program used generous estimates for the amounts of shared memory it required, which meant that it had a higher initial overhead cost. Now, in real terms these overhead costs are very small 0(100) milliseconds, but due to the very small amount of useful work carried out in the array assignments, this overhead component is very large in proportion to the operation times being measured. As an final experiment a modified version of the gold parallel program was tested which confirmed this interpretation.

Some differences were noted in the relative performances between the 320 and the 520 systems. The original per system timing ratios of *sequential : gold : shared value* programs were 1 : 16 : 11 for the 520 system and 1 : 31 : 14 for the 320 system indicating that the extra processing associated with the introduction of parallelism could be more easily absorbed by the faster processors.

If the setup times for the Umax processes are factored out of the timings then the only major overhead is the time to create the lightweight threads of control. This is a significantly smaller overhead to be absorbed by a parallel execution but for the range of array lengths in this study no speedups could be obtained for either the gold or shared value programs. (As an aside, larger vector lengths were tried and speedups were obtained around O(10000) elements.) Thus, as expected no performance benefits could be derived from using heavyweight processes to exploit fine grained parallelism, and even lightweight threads of control had to be used with care.

### 6.3.2 Matrix Addition

The next level up after the linear array is the matrix which should provide more scope for effective parallelisation because $n$ times more work is involved in simply operating on a matrix than on a linear array.

Method

Given two sources matrices A[0:n-1][0:n-1], B[0:n-1][0:n-1] and a result matrix C[0:n-1][0:n-1] compute the sum of the two matrices as

```
for (int row = 0; row < n; row++)
    for (int column = 0; column < n; column++)
    {
        C[row][column] = A[row][column] + B[row][column];
    }
```

As there are two loops which can be parallelised two corresponding data parallelisation methods are clear, both of which can be blocked in order to tailor the number of threads precisely to the number of physical processors. Blocking obviates the need for multiprogramming by instead of creating one thread of control per unit of work, many of units of work are blocked together so that there is commonly one thread of control per physical processor.

(a) $N^2$ - fold parallelisation

For every ordered pair (i,j), where i,j are 0:n-1, create a thread of control to evaluate C[i,j]. This is, conceptually, the simplest method of computing the result matrix and leads to a highly parallel solution with $n^2$ threads being utilised. This method is the equivalent of replacing the two loops of the sequential version with a construct to create $n^2$ threads. The disadvantages of this approach are that, the lightweight thread creation time is relatively large compared to the computation time for an element, and it is in addition, wasteful to create so many threads of control as it is unlikely that a multiprocessor will have sufficient processors to support this method without some degree of multiprogramming.

(b) N- fold parallelisation

In this method a thread of control is created to evaluate C[i], where $i$ is 0:n-1 and C[i] represents either a row or a column of the result matrix. The advantage of this method over the previous one is that only $n$ threads are created leading to less creation and synchronisation overhead. In addition, it is more likely that a multiprocessor can allocate these threads to processors without having to multiprogram them for appropriately sized matrices.

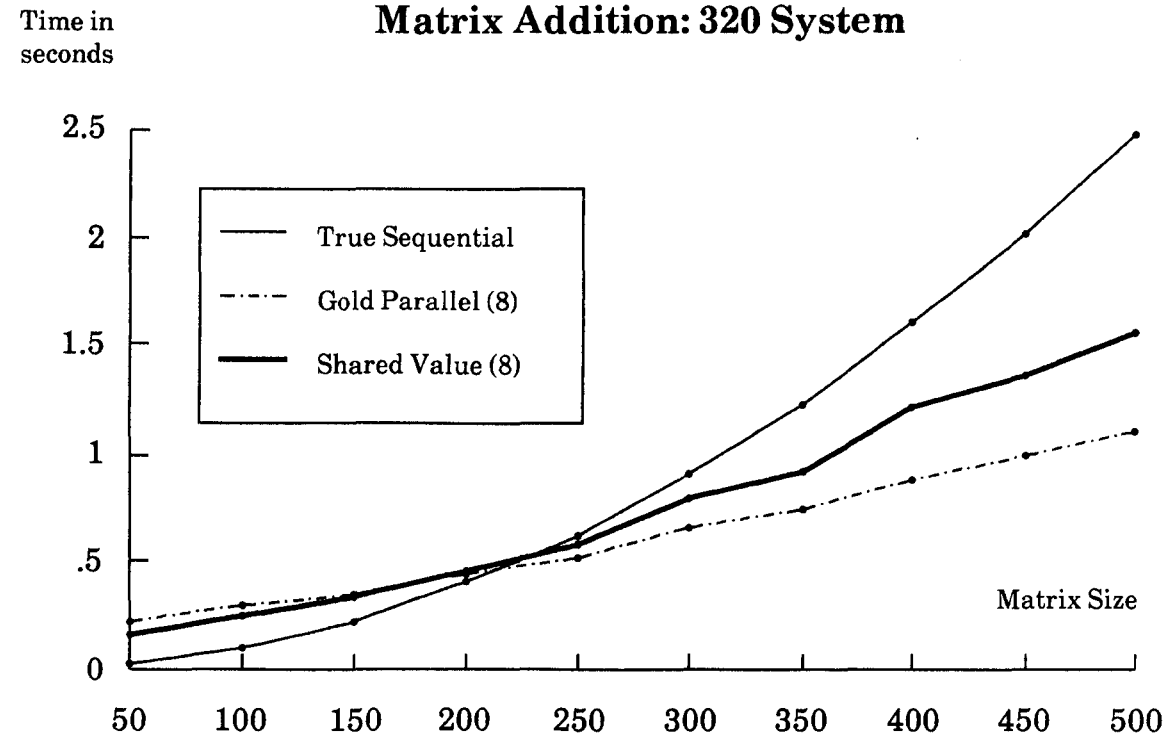
To reduce to a minimum the parallelism-associated overheads in this problem, a blocked version of strategy (b) was employed for both the gold and shared value programs. In the shared value program only the result matrix had to be declared as a matrix of shared values, the other two being stored in shared memory and regarded as read only during the parallelism stripe. Hence, the overheads that were measured in the shared value program were attributed to the operation of the shared value run time system and the times for each of the write operations to the elements of result matrix.

Observations and Conclusions

The characteristic times for matrix addition obtained from both systems are shown in Figure 6.1. In each of the system graphs the times for the sequential, gold parallel and shared value programs are illustrated. The data points for the parallel programs come from the maximally parallel runs for each system, which were not necessarily the fastest times observed. As it turned out the best times were observed when using around half the total number of available processors depending upon the problem size - as demonstrated later in Figure 6.2.

The interpretation of each of the system graphs is the same. The work for the sequential program increases in proportion to the square of the matrix size. For small matrix sizes the memory setup time component of the processes forking makes the shared value program more efficient than the gold parallel program (for the same reason as in the last scenario). But this situation changes as the matrix size increases and the overhead in the operation of the shared values increases proportionally.

Figure 6.2 shows a parallelism map obtained by varying both the problem size and the number of processors applied to computing a problem. The shape of the optimum parallelism map is a pyramid that steps up equally as processors are added, corresponding to a linear speedup. However, the map that was obtained illustrates that the overheads of thread creation eventually outweigh the gains made from the use of parallelism. The best results were obtained when around half the total number of processors were used, given sufficiently large matrices to overcome the initial overheads. Nevertheless, these results fall far short of linear speedup and highlight the unsuitability of parallelising this problem. Fortunately, though, the true picture is not quite so bad as in a real application many such array operations may be required, therefore offsetting the initial



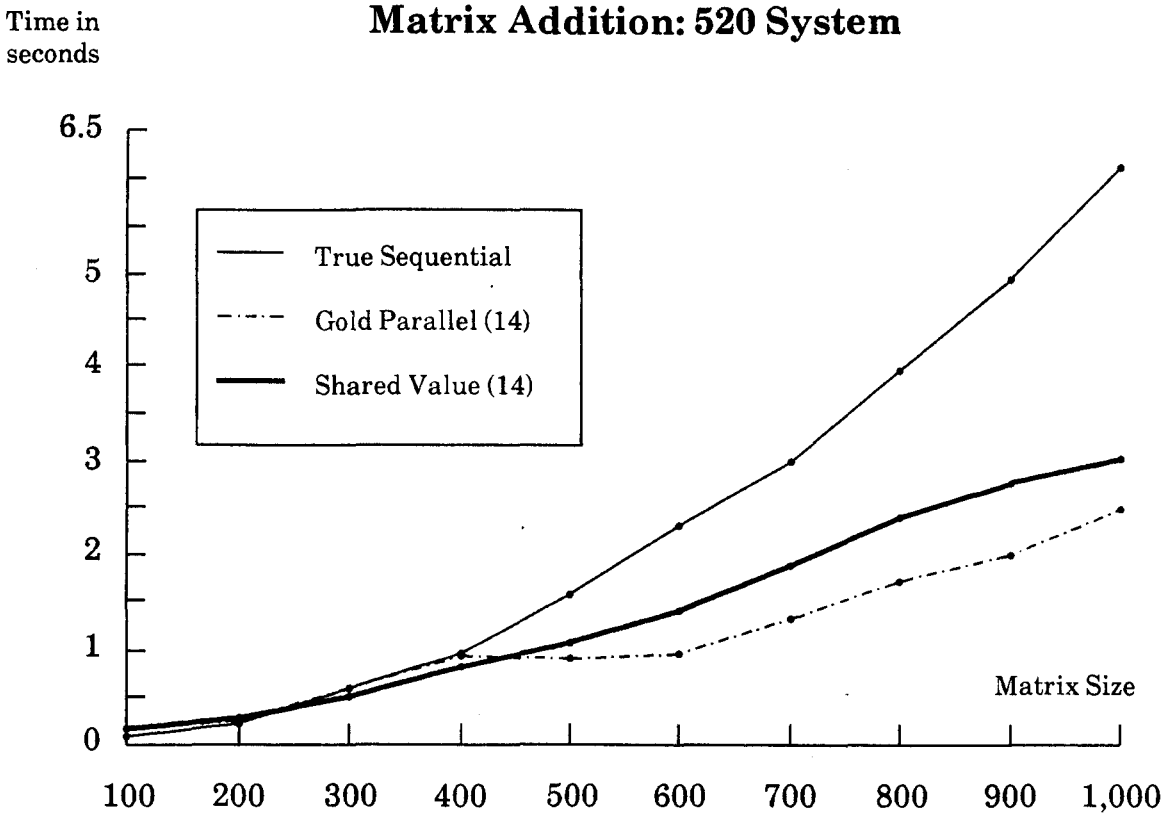
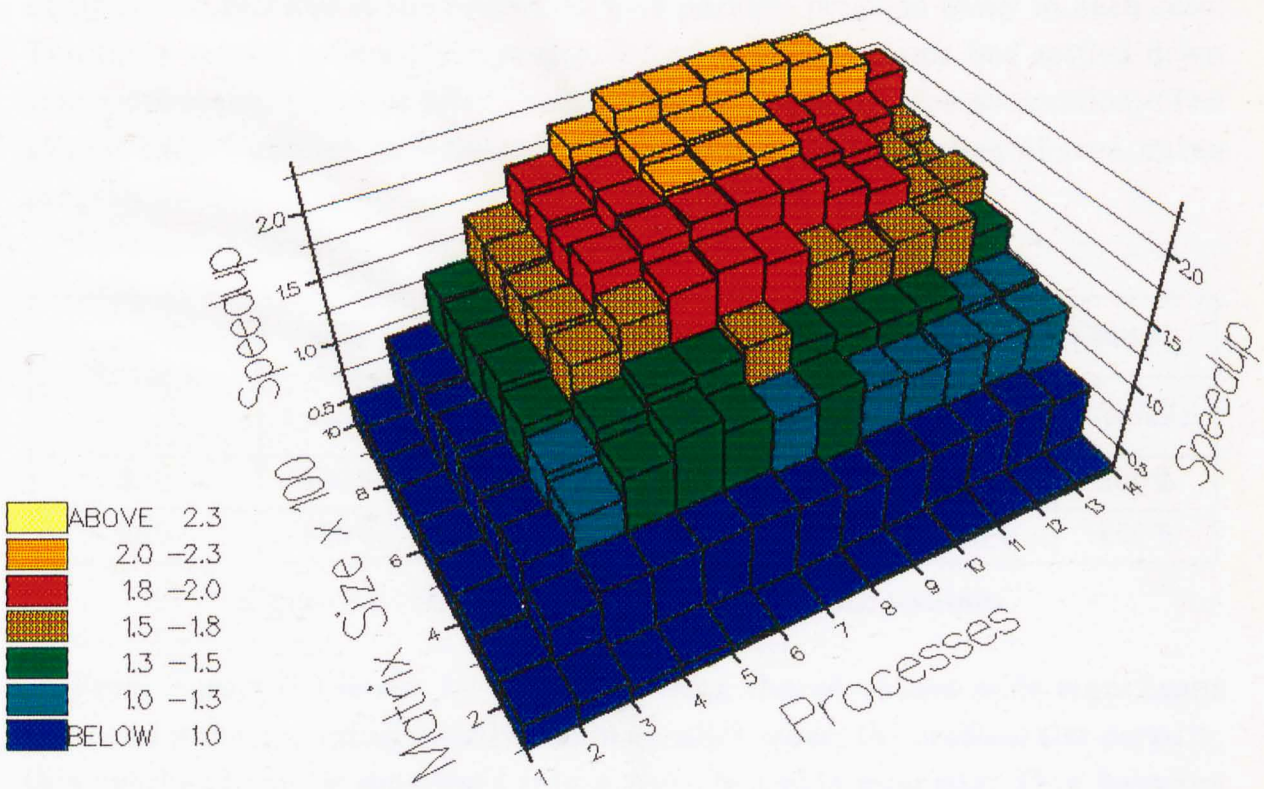




Figure 6.1 - Matrix addition: 320 and 520 systems.

Figure 6.2 – SV Matrix Addition.



overhead component resulting from the creation of Umax processes and leading to improved speedups overall.

To gain further insight into the operating characteristics of shared values the shared value program was modified so that all three of the matrices were declared as matrices of shared values. If only computation times are considered, as before, then three shared value operations would have to be performed per result element calculation (2 reads and 1 write).

A summary of the comparison between the gold program and shared value programs is presented in Figure 6.3. Both Multimax systems are covered with the entries for 1-parallel runs and N-parallel runs (N=No. of processors in system) being scaled relative to the respective gold parallel program entry in each case. The table entries reflect timings recorded after the programs had settled down into more stable patterns after overcoming the effects of process creation. (The *shared value** entries represent observations from the modified shared value program.)

| System | Gold Parallel | | Shared Value | | Shared Value* | |
|---|---|---|---|---|---|---|
| | 1-parallel | N-parallel | 1-parallel | N-parallel | 1-parallel | N-parallel |
| 320 | 100% | 100% | 200% | 130% | 350% | 200% |
| 520 | 100% | 100% | 200% | 130% | 330% | 150% |

Figure 6.3 - Scaled relative times for matrix addition.

From Figure 6.3 it can be seen that using shared values adds significant overhead when operating serially, but in parallel, where the problem size permits, this overhead can be amortised into a more bearable quantity. This happens because the overhead itself can be treated as parallelisable work and if sufficient processors are available it may be possible to reduce this overhead to the level of that associated with the gold parallel program.

### 6.3.3 Matrix Multiplication

One of the most fruitful areas of parallelism research is numerical analysis and in particular matrix computation. In order to absorb the overheads incurred in setting up multiple threads of control it is important that the work specified by an algorithm can be partitioned into appropriately sized portions. The conventional

method for multiplying two matrices A and B of rank n (n by n) is known to have a complexity of $O(n^3)$ operations and furthermore, there need be no read or write dependencies in calculating any of the elements of the result matrix, making it a good candidate for parallel execution.

Algorithm A

Given two source matrices A[0:n-1][0:n-1], B[0:n-1][0:n-1] and a result matrix C[0:n-1][0:n-1], C = AB can be calculated by the following algorithm

```
for (int row = 0; row < n; row++)
    for (int column = 0; column < n; column++)
    {
        ArrayType tmp = 0;
        for (int k = 0; k < n; k++)
        {
            tmp += A[row][k] * B[k][column]
        }
        C[row][column] = tmp;
    }
```

Once again the occurrence of nested loops permits several parallelisations of Algorithm A, all of which can be blocked as desired. Each of the parallelisation strategies for matrix addition can be extended to matrix multiplication, though this time there is an extra parallelisation that can be performed which pertains to the inner loop.

(a) $N^2$ - fold parallelisation

This is identical to the corresponding case for matrix addition with the clarification that the two outermost loops are the ones replaced by the parallelising construct. It is a more viable strategy than before, though, because of the greater amount of work involved in the computation of each element.

(b) N- fold parallelisation

This is similar to the corresponding case for matrix addition, except that once again more work is involved in computing each row or column.

(c) Scalar product parallelisation

In this strategy the inner loop is parallelised making it a less attractive proposition than the other strategies as there must be interthread synchronisation. Let $A[i] = a^T$ be a row from matrix A, and B[j] = b be a column

from matrix B, and C[i][j] = c be the corresponding element from C. The scalar product is calculated by c = ab, and can be decomposed such that $c = a_1b_1 + a_2b_2 + ... + a_nb_n$, which can be blocked to the required level of parallelism. Accesses to the shared sum *c* must be synchronised so that it remains consistent. In practice, this means using a mechanism in the gold program such as locking to enforce mutual exclusion with an additional variable acting as a lock. In the shared value program, either a number of static shared values can be used to represent *c* or a single dynamic shared value. Despite the fact that significant work can be associated with each blocked computation, experiments have indicated that it still remains insufficient to enable economical parallelisation to take place for the vector lengths in the study.

Observations

To compute Algorithm A, a sequential program was written together with a gold parallel program that employed a blocked version of strategy (b). When the computation times for the gold parallel program runs were compared to those for the true sequential runs, near linear speedups were obtained in the computation of suitably sized matrices. The startup times to create multiple threads of control (Umax processes and user level tasks) were, in general, very much smaller than the times taken to multiply matrices and very much smaller than the naturally occurring random timing variations (e.g. the fluctuations in the times of reading and writing the data values). In terms of total program execution time, however, linear speedups were not possible because of the large proportion of a program's time spent in serial activity such as reading and writing the $O(n^2)$ data values.

In a comparative program written using shared values the two source matrices A and B, as in the previous scenario, were not declared as matrices of shared values because they were not changed during the parallelism stripe. Thus, the only overheads that the shared value program incurred over the gold program were those of the shared value run time system and the final (single) assignment of each element of the result matrix. In practice this meant that the shared value program ran on average between 6% slower for 1-parallel runs and 0% slower for $N_r$-parallel runs (where N was 8 or 14).

Although the parallel execution of Algorithm A meant that large matrix products could be computed in much reduced times from the sequential times, the basic times for the algorithm were still quite high. So that these times might be

reduced some algorithmic tuning was performed to yield an algorithm that was more suitable for execution on the shared memory multiprocessors.

Algorithm B

Given two sources matrices A[0:n-1][0:n-1], B[0:n-1][0:n-1] and a result matrix C[0:n-1][0:n-1], transpose B giving $B^T$ then calculate $C = AB^T$ using the following algorithm

```
for (int row = 0; i < n; row++)
{
    for (int column = 0; column < n; column++)
    {
        ArrayType tmp = 0;
        for (int k = 0; k < n; k++ )
        {
            tmp += A[row][k] * B[column][k]
        }
        C[row][column] = tmp;
    }
}
```

Observations and Conclusions

As matrix B was stored in transpose format a new method of indexing its elements was required in Algorithm B. The new method of indexing meant that elements from matrix B were fetched by row rather than by column, reaping the benefits of simplified address calculation, reduced paging, and maximised cache prefetching. The net effects of these performance advantages were that sequential execution times were on average 30% faster than before, even taking in account the time taken to first transpose matrix B. Two corresponding parallelised versions of Algorithm B was also coded and the experiments repeated. Fortunately, similar, but not quite as impressive gains were made by the shared value and gold parallel versions, with there once again being almost no difference in performance between the shared value and gold parallel programs.

Although the switch from Algorithm A to Algorithm B yielded a noteworthy increase in performance, further program tuning was possible because of the way in which C++ could be used to represent arrays. In the first two algorithms array elements are referenced by subscripts (e.g C[i][j]), but alternatively, pointers can be used to directly reference array elements (e.g. *C). A sequential version of a program using this direct accessing method based on *Algorithm B* executed around 15% faster with similar improvements observed for a gold parallel program. Unfortunately, the current implementation of shared values did not

support such direct access to elements so no comparative shared value program was written. (This limitation was a side-effect of the implementation and was not a fundamental limitation imposed by the shared value constructs.)

To continue the search for a faster method of multiplying matrices, Strassen's method for fast matrix multiplication was investigated [High89]. The idea behind the algorithm is to decompose the source matrices into submatrices and use these submatrices in several equations to compute the submatrices of the product matrix. As it turned out, the sequential implementation of the algorithm proved to be bounded by the time taken for the matrix decomposition and the allocation of temporary matrices to hold intermediate results. Thus, the problem was more one of storage management rather than anything else, so no parallel versions were coded. In addition, the algorithm makes use of many submatrix addition, subtraction and multiplication operations which if parallelised, would result in many (though quite small) overheads in setting up and synchronising all the necessary threads of control. Nevertheless, it seems likely that a parallel implementation Strassen's method could be of benefit for very large matrices though testing this hypothesis fell outside the scope of test data for this scenario.

The performance tuning of data referencing is endemic to high performance computing as it is often memory accessing which accounts for a large percentage of the total computation time. It is hoped that shared value based programs can be tuned in similar fashion to conventional programs to allow some continuity of a programmer's knowledge and experience. The results from this scenario seem to lend some justification to this claim.

Figure 6.4 shows the characteristic times of the 520 system for sequential program using Algorithms A and B, together with shared value based versions of these two algorithms. The corresponding gold parallel program curves are not shown on the graph because these curves would have been coincident with the shared value curves. Similar results were obtained from the 320 system. The parallelism map of Figure 6.5 shows the almost linear speedup of the shared value program using Algorithm B. The data in the map was scaled to the best sequential version of Algorithm B, but if the speedups had been scaled relative to the results from a 1-parallel shared value run, even better speedups would have been obtained.

To conclude the experiments with matrix multiplication, the same technique of declaring all the matrices as matrices of shared values was employed again. Here, 2N read operations and one write operation were needed to produce every element

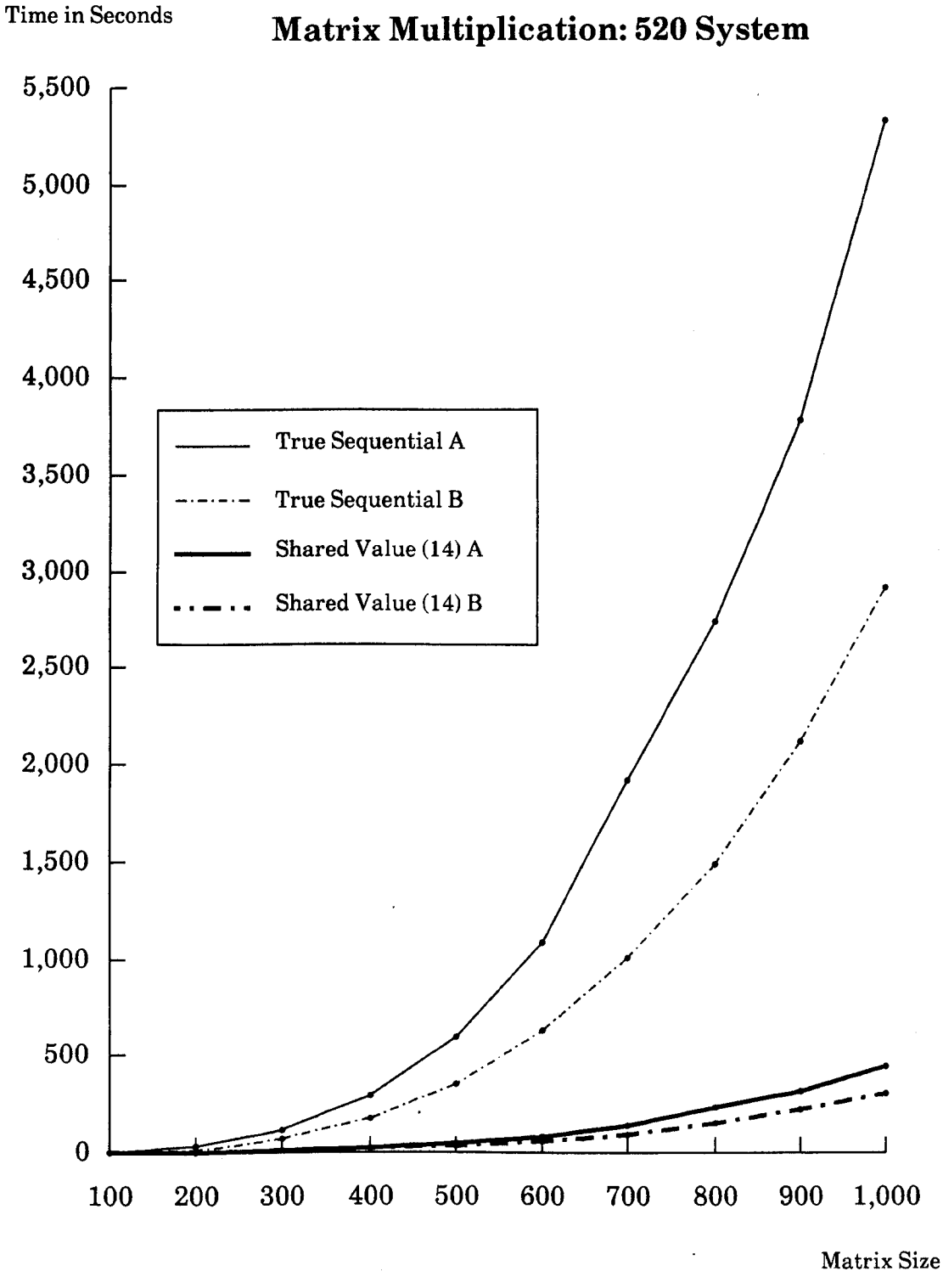
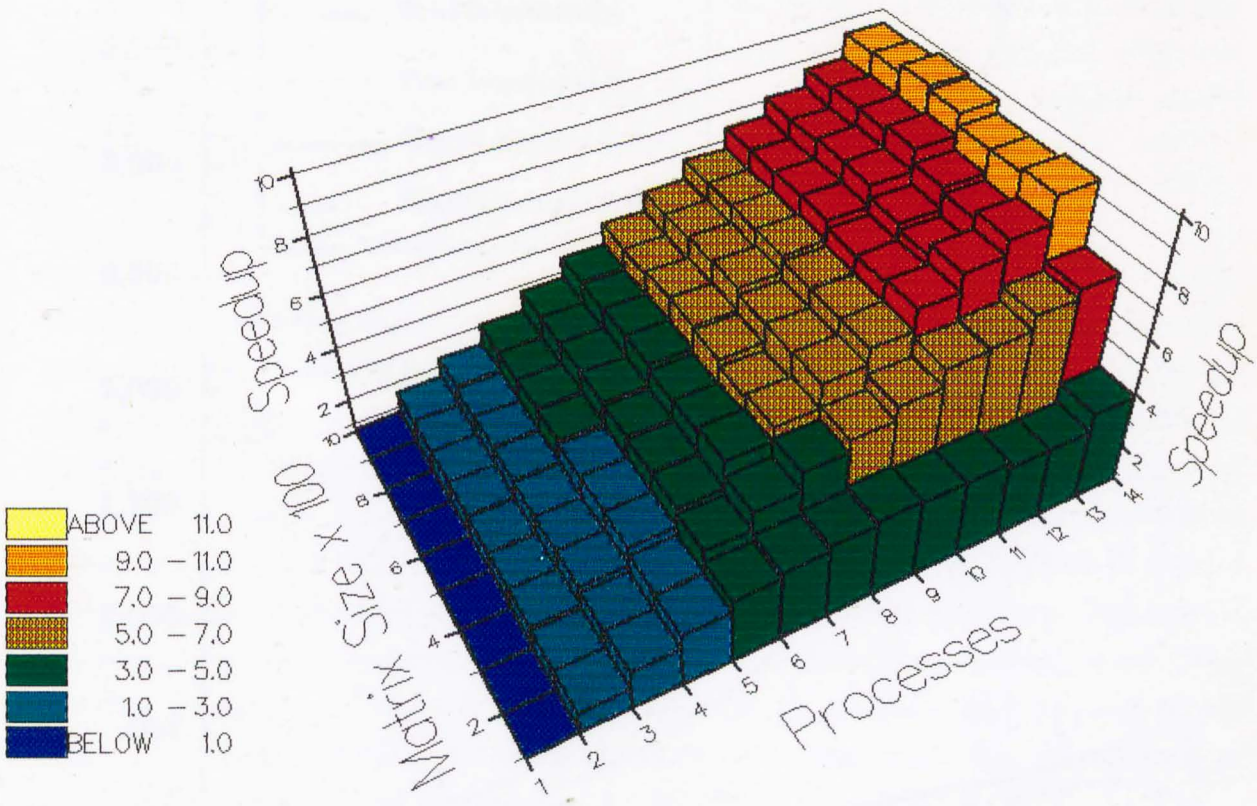



Figure 6.4 - Matrix multiplication: 520 system.

Figure 6.5 – SV Matrix Multiplication.



of the result matrix. The comparison of the gold program and the two shared value programs for Algorithm A is shown in Figure 6.6. Once again the results were scaled relative to the gold results for both 1-parallel and N-parallel runs.

| System | Gold Parallel | | Shared Value | | Shared Value* | |
|---|---|---|---|---|---|---|
| | 1-parallel | N-parallel | 1-parallel | N-parallel | 1-parallel | N-parallel |
| 320 | 100% | 100% | 106% | 101% | 260% | 280% |
| 520 | 100% | 100% | 105% | 100% | 340% | 290% |

Figure 6.6 - Scaled relative times for matrix multiplication (A).

As mentioned earlier, the basic shared value program performed excellently only incurring very small overheads with respect to the gold parallel program. Unfortunately, as expected, the modified shared value program did rather poorly resulting from all of the shared value read operations it had to perform.

### 6.3.4 Fractal Generation

Many interesting problems relating to real world situations, such as weather and economic modelling, can be represented by dynamic systems. Although programs based on dynamic systems have the potential to allow the exploitation of massive parallelism, implementation problems such as work partitioning and thread scheduling can become prohibitive. Some of the most well known work in dynamic systems has been done by Mandelbrot and it is the calculation of fractal images that provides the stimulus for the final suite of test programs. The fractal examples and methods examined here are taken from Mandelbrot's book "The Fractal Geometry of Nature" [Mand82]. Due to the scalable and unpredictable nature of fractal images, other researchers have also used the calculation of fractals for parallel processing research [Vaug89].

Algorithm

The essential process that is being followed in the computation of a fractal image is a feedback process where the $k$-th point $(x_k, y_k)$ generates the $(k+1)$-st point $(x_{k+1}, y_{k+1})$ by means of a given law:

$$x_{k+1} = f(x_k, y_k; p)$$
$$y_{k+1} = g(x_k, y_k; q)$$

where $p$ and $q$ are parameters held constant during each iteration,

$$(x_0, y_0) \rightarrow (x_1, y_1) \rightarrow \ldots \rightarrow (x_k, y_k) \rightarrow \ldots$$

The actual fractals produced in the experiments, Julia sets, were computed from the complex feedback process $z \rightarrow z^2 + c$. A decomposition of the complex numbers $z$ and $c$ into their real and imaginary parts gives $z = x + iy$, $c = p + iq$. Hence the process can be described by:

$$x_{k+1} = x^2_k - y^2_k + p$$
$$y_{k+1} = 2x_k y_k + q$$

A coloured image can be produced by plotting a pixel in a colour corresponding to the number of iterations it takes for that point (x,y) to escape towards infinity.

Method

As the calculation of each of the pixels can be done independently several parallel execution strategies are possible. However, a condition that was imposed was that the resulting image had to be able to be displayed as it was being calculated to provide a form of program visualisation. That is to say, the progress of the program could be monitored by looking at the current state of completion of the displayed image. In practice this meant that the image had to be sent a row at a time to the display device to reduce the amount of data being transferred between the host and the device, and therefore the latency in the time taken to display the image. Since the amount of work required to compute a row could vary dramatically from row to row and this work could be non-trivial, it was decided to partition each row equally amongst the threads, so ideally, the time to compute a row with $t$ threads is $T_{row}/t$. The form of a worker thread was as follows

```
for (int row = 0; row < N; row++)
{
    for (int column = lowerBound; column < upperBound; column++)
    {
            // calculate value for pixel X[row][column]
    }
}
```

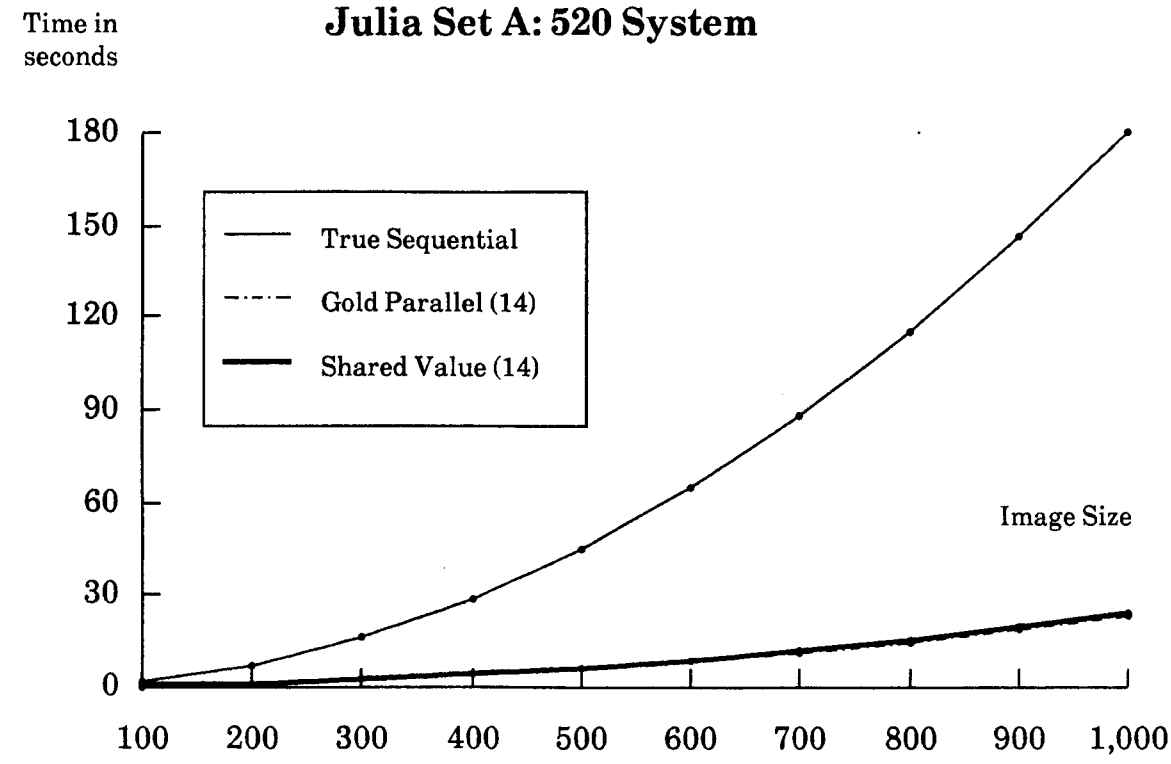
A gold parallel program and shared value based program were written following the above outline. No sequential program was written for this problem as the pixels were displayed upon their calculation by a separate display thread. (When the experimental timings were carried out the display thread performed no operations so that the timings were not influenced by external factors.) The gold

program needed some synchronisation code so that display thread could tell when a row of pixels had been completed, no extra synchronisation code was needed for the shared value program.

Observations and Conclusions

One of the features that differentiates this scenario from its predecessors in this chapter is that the number of calculations required to compute a result is not fixed for a given problem size. Alterations to the parameters of the problem can significantly alter the work involved in computing an image so to ensure a representative study, two distinctly different data sets were examined. The first image (A) appears as a classic Julia set image, but the second image (B), appears as a series of coloured bands (see Appendix A). The characteristic times to compute each Julia set (A and B) on the 520 system are shown in Figure 6.7. As before the times for the sequential, gold parallel, and shared value runs are illustrated using the maximally parallel cases for the latter two programs. (Note that the gold parallel and shared value curves are almost coincident.) Even though the two images are completely different, the characteristic work curves for each image have the same shape, but with smaller times for the second image. Similar graphs were obtained for the 320 system though these are not shown for brevity.

For Julia set A, the speedups obtained from parallel programs using the static block iterator for work partitioning, while being useful were rather disappointing, given the highly parallel nature of the problem. This stemmed from an unequal distribution of work over the parallel threads due to the inability of the allocation strategy to adapt to the different work concentrations in the fractal. In Figure 6.8 the speedup map for the shared value program is displayed. One point to note is that the speedup is actually slightly less with three processes than with two processes and this poor performance is repeated for odd numbers of processes until the number of processes become large. The reason for this is that when the fractal is divided amongst the threads, if there is an odd number of threads, then the middle thread ends up executing the largest amount of work because of the form of the particular fractal image. When an even number of threads is used, the most computationally intensive part of the fractal is split over two threads, giving a better overall work distribution. Nevertheless, the speedups shown on the map are quite poor, being around half the maximum theoretical values. Other allocation strategies were tested and the one which came out the best was the hunk iterator.



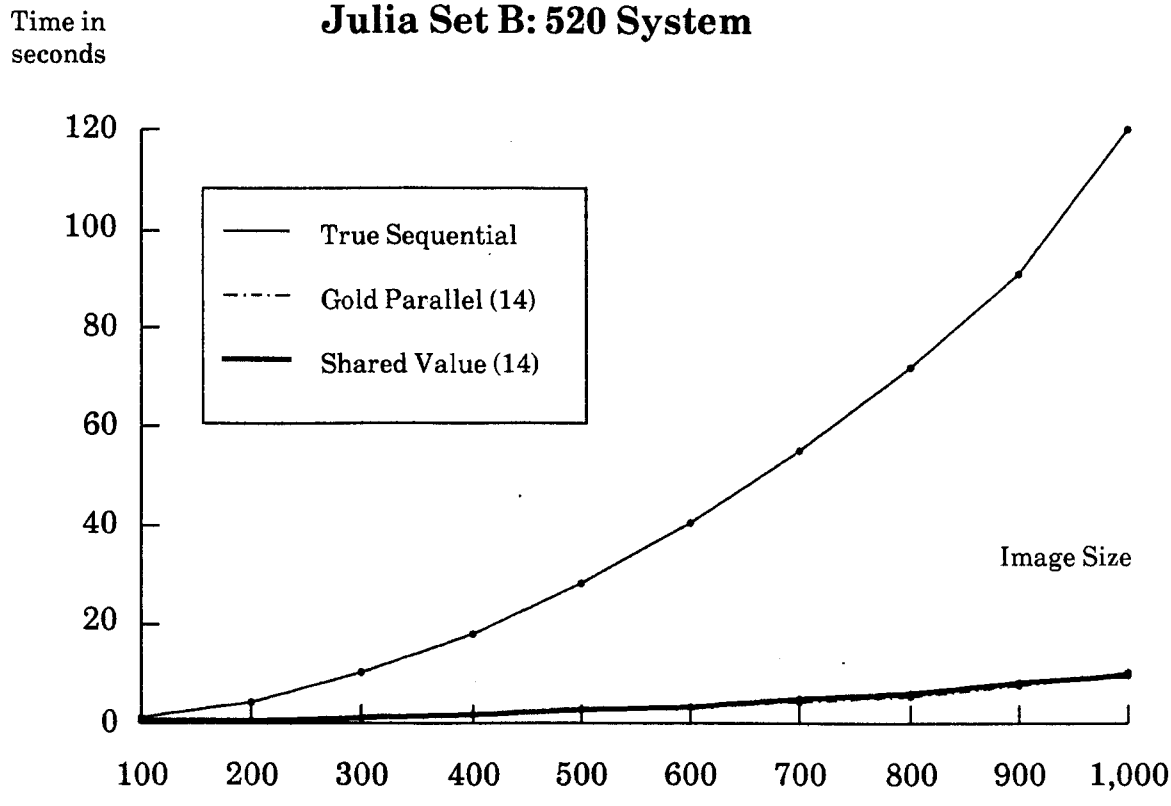




Figure 6.7- Julia sets (A and B images).

Figure 6.9 shows the speedup map for a shared value program using this iterator, which shows very respectable speedups across the board.

For Julia set B, the work distribution is quite regular, so it was possible to get very good speedups using only the block iterator. Figure 6.10 shows the parallelism map for a shared value program computing this fractal. The speedups indicated are much better than those for Julia set A using the same iterator.

Figure 6.11 compares the relative performances of the gold parallel program against a shared value program using two different iterators on Julia set A. In the case of the block iterator the shared value program performs quite well, only incurring small additional overheads. However, merely by using a different iterator (and no other code changes) runs of the program using the hunk iterator yielded much better parallel results.

An optimisation in the computation of a Julia Set that could have been made was to take advantage of the symmetry of an image. This would have meant that only 50% of the original computation would have been necessary, with the remaining percentage of the image being computed by a rotation. As it happens, with the entire image being computed each time, a form of symmetric load balancing can sometimes take place. It occurs when a thread starts by computing a block with high/low time complexity say B and another thread at the opposite end of the image computes a block with time complexity B* low/high respectively. After completing its block the first thread then computes a block of complexity B* and likewise the other thread computes a block with complexity B after completing its initial block. Hence, the total computation time for the first thread is B+B* which equals the time for the other thread's computation B*+B. This behaviour became clear in the results for the block iterator, as superior speedups were obtained for runs with even numbers of threads over those with odd numbers (as mentioned earlier).

## 6.4 Summary

As might of been expected from the discussion of grain size in chapter three, shared memory multiprocessors need quite a large grain size of parallel computation before appreciable gains can be realised from parallel processing. This is true irrespective of whether shared values are used as a parallel processing mechanism or if hand coded parallelism constructs are used instead.

Figure 6.8 – SV (Block) Fractal A.

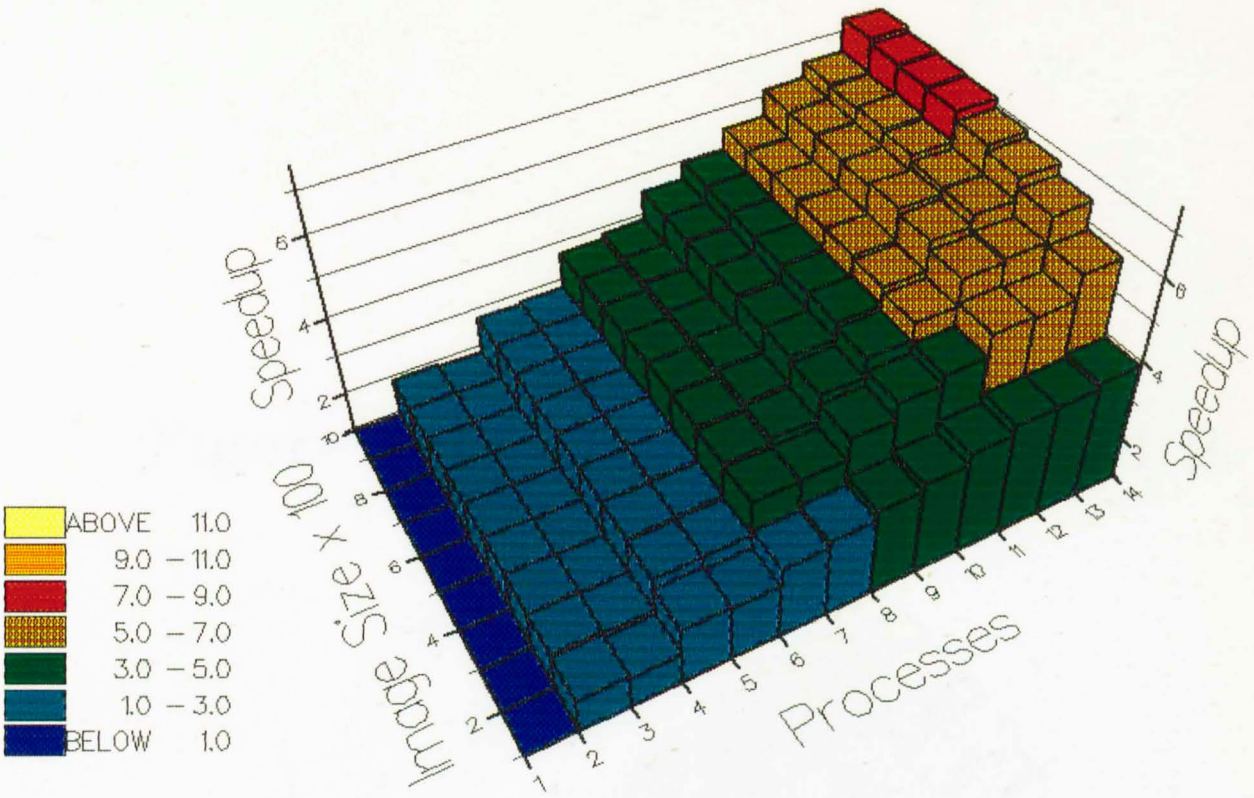



Figure 6.9 – SV (Hunk) Fractal A.

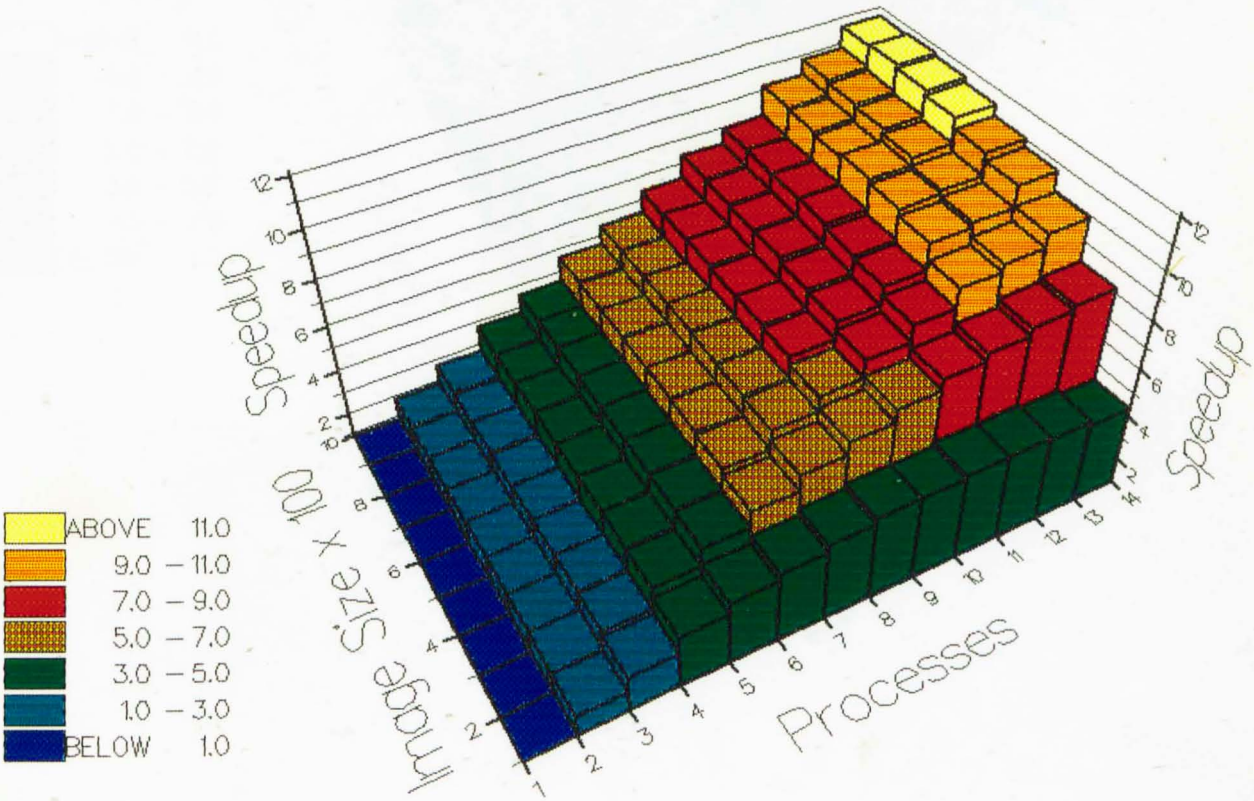
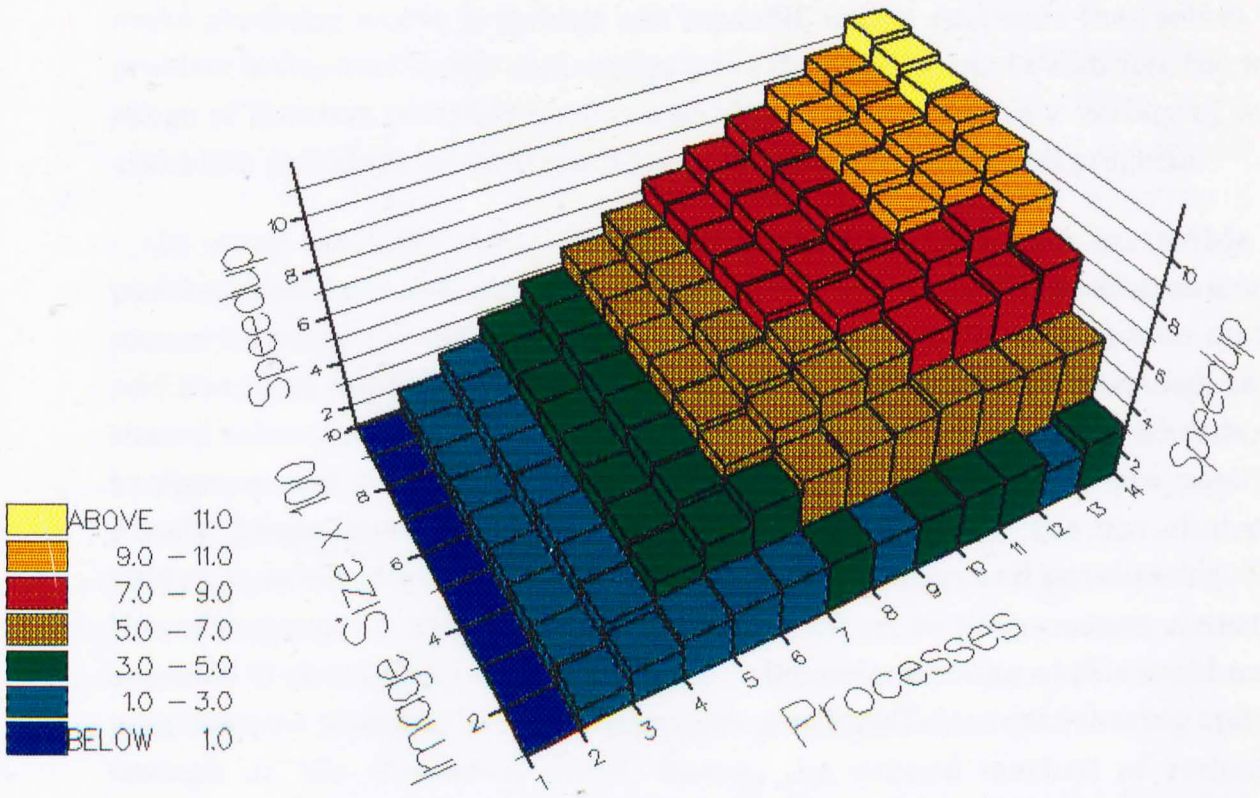


Figure 6.10 – SV (Block) Fractal B.



| System | Gold Parallel | | Shared Value (Block) | | Shared Value (Hunk) | |
|---|---|---|---|---|---|---|
| | 1-parallel | N-parallel | 1-parallel | N-parallel | 1-parallel | N-parallel |
| 320 | 100% | 100% | 102% | 104% | 104% | 65% |
| 520 | 100% | 100% | 104% | 104% | 104% | 58% |

Figure 6.11 - Scaled relative times for fractal (A).

From the viewpoint of ease of use, shared values are a tidy method for coding the simple programming scenarios explored in this chapter. Their use is much preferable to using explicit constructs which require detailed specification and make resulting source programs less readable as the real code that solves the problem is obscured by the code used to drive the parallelism. In addition, the wide range of iterators provided for use with shared values allows a variety of work allocation policies to be evaluated by only trivially recoding a test program.

In many cases the extra overhead of using shared values is acceptable for problems that have sufficient work to enable efficient parallel execution on shared memory multiprocessors. From the supplementary tests carried out with matrix additions and multiplications, however, it was noted that unconstrained use of shared values could lead to parallel programs that had quite large overheads due to unnecessary synchronisation. Two solutions to this problem are possible. Firstly, programmers could be urged to be more careful in the use of shared values, so as not to introduce unnecessary synchronisation and serialisation. This line of argument is only a partial solution, however, as programmers cannot be expected to shoulder all the burden for parallel programming as this could mean programmers spending too much time arranging for efficient data sharing and not enough on the algorithm itself. Hence, the second method of reducing synchronisation overhead is for a compiler or preprocessor to simply eliminate shared value synchronisation where it is not needed.

# *Chapter 7*

# Conclusions

From the outset the aims of this thesis have been to examine why the parallelism that is offered by shared memory multiprocessors has not been effectively exploited by mainstream computing and to suggest ways of ameliorating this situation. In the first part of this thesis evidence was presented to argue that multiprocessors are an economically viable architecture. Moreover, it was suggested that mainstream computing should be interested in multiprocessors because they hold the promise of high performance and high reliability for relatively low cost.

The next part of the thesis went on to describe the types of multiprocessor architecture that have been constructed and their operating characteristics, as the SIMD approach to parallelism at first seems to be very different from the MIMD approach. Having defined the base parallel architectures, an extensive survey of the software used to program these systems was presented. This was divided into two sections covering both the system software which operates the hardware, and the user software which uses the hardware (via the system software) to execute applications. One important point that comes out of this survey is that good system software is a key component in effective multiprocessing. This is because good system software can simplify the job of implementing and executing user level parallel software by simulating an abstract multiprocessor that is easier to use than the underlying hardware. Other researchers have recognised this process of defining a hierarchy of virtual machines so this observation is not too surprising [Tane87]. The other important point that comes from the survey is that there are essentially three main approaches to exploiting parallelism:

- In auto-parallelism, software tools are used to generate parallel programs from imperative sequential programs (e.g. "dusty decks") either by using a parallelising compiler or through the use of some parallel program construction tools.

- In natural parallelism, an abstract (and possibly parallel) model of computation is used which is completely divorced from any underlying hardware details. Parallelism is extracted and implemented implicitly by an interpreter or compiler.

- In explicit parallelism, specially designed parallelism constructs specify precise details about parallel execution, possibly making use of specific hardware capabilities (e.g. test-and-set operations).

Thus, there is a wide range of approaches available for parallel programming but the question that arises is which approach is the best and why? Clearly, there is no short answer to this question because various degrees of success have been achieved using each approach. Nevertheless, one can outline briefly the problems that each approach faces.

To produce an effective auto-parallelising compiler, one must restrict the semantic capabilities of source language constructs to minimise the effects of phenomena such as aliasing, which can ultimately prevent a compiler from constructing a model of the inter-statement dependencies within a program. In practice, source programs are written in dialects of primitive languages such as Fortran, splitting off this form of language research from more mainstream work (e.g. object-oriented techniques). While this may not be perceived as a serious disadvantage there is another problem. Basically, the effectiveness of the parallelisation techniques employed by compilers is dependent upon the amount of recoverable parallelism in a source program and how that parallelism can be mapped to the underlying hardware. In some cases these tasks are not easy to accomplish automatically, so some interaction with users is required. Interactions can sometimes be simple for a user, however, on other occasions detailed working knowledge of the operating system and the underlying hardware may be needed to solve a problem. Thus, auto-parallelisation is not a panacea for parallel programming because while it can lead to some effective portable parallel software, it can also have a stifling effect on the development of programming languages and can require expert programming which belies its boast of automatic parallel processing.

In natural parallelism, abstract models of computation are used for programming, which stimulate programming language development and redress one of the problems of auto-parallelisation. However, the problem of mapping potential parallelism to underlying hardware still remains and even takes on a more disturbing form. As there is little linkage between source programs and their underlying parallel hardware it can be difficult to control and monitor the parallel execution of a program without adding a few extra constructs or commands, or by providing some software tools. Even so, it may turn out that the effective execution of a particular programming model cannot be accomplished on

a given architecture because of the top-down nature of this route to parallelism. Thus, the problem with this route to parallelism is that it is not easy to select the *best* model (because best cannot easily be defined) from the many abstract programming models, and then derive an implementation for this model that is very efficient for shared memory multiprocessors.

Thus the stage is set for the explicit route to parallelism. Here research commenced by developing constructs that operated at the basic hardware level (e.g. test-and-set) but rapidly started to evolve constructs that did not reference specific machine features and were instead more abstract and portable; though not with too great a loss in efficiency. Unfortunately, the case for adopting the explicit route to parallelism is not cut and dried because explicit parallel programming has, in some cases, deservedly attracted a reputation for being complicated, with programmers being forced to carry out lots of work managing details such as data access control and control flow. Moreover, controversy has raged about which of the information sharing paradigms of shared memory or message passing is the better. Fortunately, as advances have been made in both hardware and software technologies it is now possible to hide away many of the underlying hardware characteristics, and blur the distinction between physically and virtually shared memory. Hence, this bottom-up approach to parallelism was the one followed in this thesis because it allows for the evolution of parallel programming constructs towards the goals set by natural parallelism, but at the same time does not lose sight of what is practical on a given hardware base. Of course, there are other important factors in the choice of programming approach, such as ease of use, so these issues were tackled in the next part of the thesis in chapter three.

When designing a parallel program that is expected to execute efficiently in parallel on a multiprocessor it seems reasonable to work with a parallel or potentially parallel algorithm. The way in which these algorithms are designed and expressed is influenced to some degree by the final type of programming model. In a naturally parallel model, one can design algorithms that can be executed in parallel, but it may be difficult to accomplish this without the aid of some programming paradigms. In an explicit programming model, help is available in the form of programming constructs as well as through the use of parallelism paradigms. Hence, by using an explicit approach to parallelism a programmer can purposefully direct his programming efforts towards producing effective parallel software.

To state clearly then, the problems addressed by this thesis are those that are responsible for the current lack of effective parallel software for shared memory multiprocessors. These problems are:

- programs have not been designed to follow suitable parallel algorithms,
- there is no clear choice for the best parallel language (or style of language) to implement such algorithms,
- it has been difficult to efficiently match algorithms and parallel programming techniques to the underlying parallel hardware.

It is easy to lay the blame for the lack of effective parallel software on these three factors but to find a remedy one must examine these issues more carefully.

To produce a parallel program, one must arrive at an algorithm that can be partitioned such that the separate parts can be executed by separate processors. The partitioning does not necessarily have to be explicit, if the inclusion of parallelism constructs detracts from the clarity of the algorithm, but in some cases explicit partitioning is useful for modelling concurrent behaviour. To partition an algorithm two broadly based techniques have emerged (though some algorithms exploit both techniques simultaneously): (i) the functional aspects of an algorithm are partitioned, and (ii) the data values operated on by an algorithm are partitioned.

Once an algorithm has been partitioned it can be executed in parallel but there are no guarantees that any performance benefits will be derived from this; other factors govern its efficient parallel execution. Experimental work in chapter six confirms the belief that the grain size of the parallelism that is exploited by a parallel program is a key factor. In addition, algorithmic factors such as intercomputation dependency and hardware related factors such as patterns of memory usage are also important. Hence, a parallel processing mechanism must support the expression of a parallel algorithm that is appropriate for the application being programmed and must also make provision for such an algorithm to be executed efficiently. Furthermore, when a program is partitioned and executed in parallel, it is desirable that many operations execute simultaneously so that a large proportion of the useful parallelism is utilised. However, with many operations executing in parallel the complexity of a program can grow dramatically making it difficult to write, debug and monitor. Hence, a parallel processing mechanism must also address these factors of usability as well as issues of expressiveness.

To this end the Tyger parallel programming model, discussed in chapter four, was envisaged as being part of some larger software environment that consisted of an operating system, a compiler, and a set of tools for program design and monitoring. Moreover, the Tyger model and program design tools were aimed at promoting a particular style of parallel program design, with emphasis on regularity and modularity (e.g. that commonly found in data partitioning). This approach is somewhat similar to a large grain SIMD model of parallelism, though the intention was also to support MIMD computation in the form of functional partitioning as well. Undoubtedly, some applications may give rise to intricate patterns of interaction between concurrent threads of control, but in these cases the complexities and subsequent programming difficulties originate from the algorithms themselves. Therefore, the best that an explicit parallel programming mechanism can do is not to make a programmer's task any harder by making its facilities difficult to use.

The Tyger model presents a programmer with a simple, fixed, view of parallel computation in which a program consists of a sequence of alternate intervals (stripes) of sequential or parallel activity. Inside a parallelism stripe, threads of control execute in a shared address space communicating and synchronising by using shared values. Shared values themselves are a relatively simple mechanism so this encourages programmers to have simple regular interactions between threads of control, to help manage the complexity inherent in parallel activity. The Tyger model is reasonably easy to visualise but is also powerful enough to express arbitrarily complex forms of parallel activity. Hence it gives a programmer a starting point from which to design a parallel program. Either the program will execute largely in a single parallelism stripe where threads of control are spatially distributed, or will consist of a series of disjoint parallelism stripes where threads of control are temporally distributed. Shared values permit the communication of data between threads of control in each case by firstly exploiting single assignment semantics and secondly by the use of a history facility.

Shared values are a double edged mechanism for parallel programming. On one hand static shared values are a simple, neat and code-efficient way of expressing data parallel algorithms and simple functionally parallel algorithms. More complicated interactions between threads of control can be modelled by custom built data abstractions descended from static shared values, or by dynamic shared values. On the other hand, the implementation of shared values can take advantage of the underlying parallel hardware so that performance-efficient

programs can be written. For instance, with the single assignment property maximal locality of reference can be exploited as local copies of variables can be maintained in cache memories. Moreover, contention for shared memory can be minimised by working out of a cache memory, so obviating the need to always get the current value of a shared variable from shared memory. Further to this, shared values raise the level of memory abstraction in programs by decoupling references to variables from actual physically shared memory and replacing them by references to shared value objects such as static shared values, dynamic shared values, and higher level objects like streams. Hence, this raises the possibility that shared values could also be implemented efficiently on a distributed memory multiprocessor, with the cache memories mentioned before being replaced by local memories.

In chapter six, an implementation of static shared values was measured against lower level parallel programming facilities to gauge the efficiency of shared values. For certain types of problem shared values and their iterators are a good mechanism for parallel programming, leading to clear programs that incur only slight overheads over more explicit approaches. To add to this, though, it was also noted that the use of shared values could prove costly as in the case of matrix multiplication where the source matrices were only shared in a parallelism stripe and not written. Hopefully, difficulties such as these can be overcome by the use of optimisation techniques in source program analysis, or perhaps even by hardware solutions at the level of the cache or local memory.

## 7.1 Future Work

Although the Tyger model is capable of expressing many forms of interaction between parallel threads of control there remain several areas where additional research could be usefully undertaken. These areas include:

- new control constructs,
- an extended Tyger model,
- a Tyger-C++ compiler and accompanying software tools.

Shared value semantics dictate that when a thread tries to read the value of a shared value that thread will be blocked until a value is made available. Using a non-blocking strategy has long been recognised as being useful in message passing, for amongst other things, as a way of increasing the concurrency in programs. More specifically, if a value is not immediately available for use by a thread, that thread may be able to look elsewhere for another value it can use, or it

may perform some other useful work until the requested value is made available. Non-blocking behaviour could be incorporated into the Tyger model by the introduction of guarded commands, where the guard tests are non-blocking. Using the guarded command mechanism obviates the need for special non-blocking versions of existing shared values constructs and side steps many of the problems of using non-blocking primitives (e.g. their ability to give rise to race conditions).

The capabilities for parallel programming provided by static and dynamic shared values allow the natural expression of many parallel algorithms. However, there is a class of methods that is not supported particularly well by either shared value mechanism. Such methods involve parallelising algorithms that perform in-place manipulation of data, in which a final result data structure is obtained by performing a series of in-place transformations to some or all of the original data structure. For example, in linear algebra an algorithm may conceptually partition a matrix into submatrices and operate separately on each submatrix, before using the entire matrix in a completely different way in a later process. The most awkward algorithms of this type to handle are those that are expressed recursively (e.g. Strassen's method for matrix multiplication extended to rectangular matrices [High89] and symbolic applications such as quicksort [Hoar62]). Here, only limited parallelism can be applied initially at the top level of the recursion, constrained by the number of top level recursive calls. As the recursion develops, however, there is scope for farming out recursive calls to separate threads. Nevertheless, using static or even dynamic shared values can lead to much copying of data when each new thread is started and when the results are collected. Not surprisingly, the program code to perform this data movement can look rather unwieldy as well. The key observation regarding the efficient implementation of in-place methods, that can be used to ease the situation, is that often threads of control need exclusive access to contiguous and logically separate parts of a data structure to perform their transformations. A brief outline of a mechanism that could be used to support in-place transformations now follows.

When a thread of control is started it is given ownership of the static shared values it ultimately wishes to write. During the course of its execution a thread may write many times to a shared value that it owns to perform in-place transformations. When a thread terminates it relinquishes the ownership of its shared values, which then become unowned. If a thread wishes to read the value of a shared value, it is suspended until that shared value is no longer owned by another thread and the shared value has been bound to value, i.e. there is no

possibility of there being a thread waiting to write to it. Hence, to an observing thread that reads a shared value (previously owned by another thread) there is only one assignment ever made to the shared value - its final value.

The shared value ownership scheme could also be used to extend the expressive power of the Tyger model by enabling child threads in a parallelism strip to create their own threads of control. When a new thread of control is created during an existing parallelism stripe, this thread of control executes in the same stripe as its parent thread of control and its parent's peer threads. When a thread creates a new thread of control, it can pass on the ownership of the shared values that it owns to the child or can terminate and lose the ownership as before. Thus, while a thread has ownership of a shared value it can write to the shared value as desired. Alternatively, a thread can create a child thread and selectively transfer ownership of its shared values to the child, for child to perform further write operations. The transfer of value ownership should be regarded as an indivisible operation, and furthermore, it should not be possible for multiple threads to own the same shared value simultaneously. The idea of shared value ownership is essentially an application of atomic actions described in chapter two.

As was mentioned in chapter five, although a large proportion of the Tyger model can be implemented at run time, several performance benefits can be gained by carrying out as much work as possible at compile time. The future implementation of a Tyger compiler would be a boon to the production of high quality code, though, it is dispensable when working in a research environment. In chapter four, a brief overview was presented of other software tools that would be beneficial to programmers working with the Tyger model. Future work could be devoted to implementing such tools which could be used most aptly in development of large pieces of parallel software.

## 7.2 Closing Summary

The original goal of this thesis was to discover ways in which shared memory multiprocessors could deliver the power of parallel processing to users. The conclusion of this work is that effective exploitation of parallelism is dependent on several factors with the most important being a notation to express parallel activity. To this end a model of programming, the Tyger model, was developed to enable the specification of parallel activity in a precise yet uncluttered way. The Tyger model builds on the notion of there being some shared name space within a parallel program, but separates this idea from the underlying memory model.

Moreover, the Tyger model emphasises locality of effect, to reduce interthread dependencies and also to retain efficiency by operating out of fast memory when possible. This decision is fortuitous in that emerging and future multiprocessors will probably be built along of the lines of hybrid multiprocessors (distributed memory simulating shared memory), because of the rapid increase in the ratio of processor to communication speed.

Many of the ideas put forward in the Tyger model are in use in other research efforts into parallel programming. The idea of using single assignment variables for communication and synchronisation has been taken up by many languages in both the imperative and declarative fields. For example, PCN [Chan90] uses the single assignment rule when writing to shared data but allows multiple assignments to local data. In addition, tuple based communication and synchronisation has been around for a number of years (e.g. Linda), but research is still ongoing. Distributed data structures have also been recognised as an important parallelism metaphor and a new language has been developed to support them [Zeni90]. Finally, in one extreme, a research language has three separate types of parallel programming mechanism to support different programming styles. These mechanisms are: (i) single assignment variables, (ii) tuple communication, and (iii) object based concurrency. Thus, it seems that no single definitive method of parallel programming has yet emerged.

The view of control flow during the parallel execution of a program developed under the Tyger model is similar to that in other parallel imperative languages. For instance, the idea of sticking to a fixed number of threads of control that are reused during the execution of a program is exploited in the Force [Jako90] parallel programming language and the successor to Fortran 8X - Fortran 90 [Lytt90]. Moreover, Fortran 90 contains many synchronisation constructs, such as events, to enable the expression of crude dataflow synchronisation. The Tyger model approaches parallel programming in much the same way as these languages but does not suffer from their main disadvantage. More specifically, the Tyger model is not locked into a particular language (e.g. Fortran) and is therefore free to be implemented in much higher level languages to give superior levels of abstraction and ease of use. This leads to more readable, but not necessarily less efficient, source program code.

When research was directed away from using C as an implementation language for shared value constructs to C++, the advantages of using carefully chosen data abstractions became clear. The immediate advantage was that data

abstraction could be used to hide implementation details and so help combat the complexity associated with parallel programming. Hence, a programmer could design an algorithm, code it using shared values, and not be too concerned over its efficiency compared to using more primitive parallel programming methods. This came about because the synchronisation built into shared values could be directly mapped into efficient low level synchronisation constructs (for shared memory multiprocessors) by the Tyger run time system. However, perhaps the major advantage of using data abstraction was to allow the concept of a shared value to be applied as a property to a given data structure. This allowed the construction of data structures with type-specific synchronisation properties. For example, instead of waiting for every element in each row of a matrix, a thread can wait for an entire row of the matrix. This has a twofold effect of making such programs more understandable, and making them more efficient, as synchronisation is applied at the row level (increasing the granularity), rather at the element level. One point to notice, however, is that similar structured application of shared value synchronisation can also be achieved by the use of the shared value ownership scheme proposed in the previous section.

Thus, shared values seem to be a useful mechanism for parallel programming as they retain the notion and the efficiency of information sharing, while liberating programmers from an actual need to use physically shared memory. Shared value constructs such as iterators are a good way for coding parallelism in the form of data partitioning and simple functional partitioning. Experimentation with object-oriented techniques has given Tyger-C++ the capability to define new data objects that possess inherent shared value semantics, giving added flexibility to the shared value method of interthread communication and synchronisation. In addition, the Tyger model itself can be seen as a way of again extending the power of shared values by allowing them to be reused over the course of a parallel program, but the model also serves as a useful framework for parallel program design. So, while the last word in shared values may not have been said, the existing ideas have proved to be an excellent starting point to further parallel programming research.

# *Bibliography*


[Acce86] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A.Tevanian, and M. Young, "Mach: A New Kernel Foundation of UNIX Development," *Summer Usenix Conference Proceedings*, pp. 93-112, 1986.

[Agha86] Agha, G.A., *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[Aho74] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Aho86] Aho, A.V., R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[Ahuj86] Ahuja, S.R., N.J. Carriero, D. Gelernter, and V. Krishnaswamy, "Progress Towards a Linda Machine," *Proceedings Conference on Computer Design*, pp. 97-101, October 1986.

[Alle88] Allen, F.E., "Compiling for Parallelism," *Proceedings 1986 IBM Europe Institute course on Parallel Processing*, North-Holland, 1988.

[Alle84] Allen, J.R. and K. Kennedy, "PSC: A Program to Convert FORTRAN to Parallel Form," *Supercomputers: Design and Application*, IEEE Computer Science Press, 1984.

[Alli86] *FX/Series Product Summary*, Alliant, October 1986.

[Alma89] Almasi, G.S. and A. Gottlieb, *Highly Parallel Computing*, Benjamin Cummings, 1989.

[Alme85] Almes, G.T., A.P. Black, E.D. Lazowska, and J.D. Noe, "The Eden System: A Technical Review," *IEEE Transactions on Software Engineering*, pp. 43-59, January 1985.

[Amda67] Amdahl, G.M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.," *AFIPS Conference Proceedings*, vol. 30, pp. 483-487, 1967.

[Amer87] America, P., "POOL-T: A parallel object-oriented language.," *Object-Oriented Concurrent Programming*, pp. 199-220, The MIT Press, 1987.

[Andl77] Andler, S., *Synchronization Primitives and the Verification of Concurrent Programs*, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, May 1977.

[Andl79] Andler, S., "Predicate Path Expressions," *Proc. 6th ACM Symposium on Principles of Programming Languages*, pp. 226-236, San Antonio, January 1979.

[Andr76] Andrews, G.R. and J.R. McGraw, "Language Features for Process Interactions," *Report TR-76-290*, Cornell University, Department of Computer Science, 1976.

[Andr82a] Andrews, G.R., "The distributed programming language SR - mechanisms, design and implementation," *Software -- Practice and Experience*, vol. 12, no. 8, pp. 719-754, August 1982.

[Andr82b] Andrews, G.R. and Fred B. Schneider, "Concepts and Notations for Concurrent Programming," *Tech. Rep. TR 82-150*, Cornell University, September 1982.

[Aral88] Aral, Z. and I. Gertner, "Non-Intrusive and Interactive Profiling in Parasight," *Proceedings of ACM/SIGPLAN PPEALS*, pp. 21-30, New Haven, Connecticut, July 1988.

[Arch86] Archibald, J. and J.L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, vol. 4, no. 4, pp. 273-298, November 1986.

[Arvi80] Arvind, V. Kathail, and K. Pingali, "A Data Flow Architecture with Tagged Tokens," *Technical Memo 174*, Laboratory for Computer Science, MIT, September 1980.

[Arvi83] Arvind, and R.A. Iannucci, "A Critique of Multiprocessing von Neumann Style," *Proceedings 10th International Symposium on Computer Architecture*, pp. 426-436, Stockholm, 1983.

[Arvi88] Arvind, and K. Ekanadham, "Future Scientific Programming on Parallel Machines," *Journal of Parallel and Distributed Computing*, vol. 5, no. 5, pp. 460-493, October 88.

[Atki77] Atkinson, R. and C. Hewitt, "Synchronization in Actor Systems," *4th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 267-280, January 1977.

[Baer68] Baer, J.E. and D.P. Bovet, "Compilation of Arithmetic Expressions for Parallel Computations," *Proceedings of IFIP Congress*, pp. 340-246, 1968.

[Bal88] Bal, H.E. and A.S. Tanenbaum, "Distributed Programming with shared data," *Proceedings of the IEEE CS International Conference on Computer Languages*, pp. 82-91, 1988.

[Bal89] Bal, H.E., J.G. Steiner, and A.S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, vol. 21, no. 3, pp. 261-322, September 1989.

[Bald87a] Baldwin, D., "Why We Can't Program Multiprocessors the Way We're Trying to Do It Now," *Technical Report 224*, University of Rochester, August 1987.

[Bald87b] Baldwin, D. and C. Quiroz, "Parallel Programming and the CONSUL Language," *Proceedings of the 1987 International Conference on Parallel Processing*, 1987.

[Balz71] Balzer, R.M., "PORTS: a method for dynamic interprogram communication and job control," *Proc. AFIPS SJCC Computer Conf.*, vol. 39, pp. 485-489, 1971.

[Bane88] Banerjee, U., *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

[Barn68] Barnes, G.H. *et. al.*, "The Illiac IV Computer," *IEEE Trans. on Computers*, vol. C-17, pp. 746-757, 1968.

[Batc80] Batcher, K.E., "Design of a Massively Parallel Processor (MPP)," *IEEE Transactions on Computers*, vol. C-29, pp. 836-840, 1980.

[Bate88] Bates, P., "Debugging heterogeneous distributed systems using event-based models of behaviour," *Proceedings of Workshop on Parallel and Distributed Debugging*, pp. 11-22, ACM, 1988.

[Beet87] Beetem, J., M. Denneau, and D. Weingarten, "The GF11 Parallel Computer," *Experimental Parallel Computing Architectures*, North-Holland, Amsterdam, 1987.

[Bent86] Bently, J., *Programming Pearls*, Addison-Wesley, 1986.

[Bern88] Bernstein, D. and K. So, "PREFACE - A Fortran Preprocessor for Parallel Workstation Systems," *IBM Research RC 136000*, March 1988.

[Bern89] Bernstein, D., A. Bolmarcich, and K. So, "Performance Visualization of Parallel Programs on a Shared Memory Multiprocessor System," *International Conference on Parallel Processing*, vol. 2, pp. 1-10, 1989.

[Bers88] Bershad, B.N., E.D. Lazowska, and H.M. Levy, "PRESTO: A System for Object-Oriented Parallel Programming," *Technical Report 87-09-01*, Department of Computer Science, University of Washington, January 1988.

[Bert89] Berten, M.S. and H.F. Jordan, "Multiprogramming and the Performance of Parallel Programs," *Parallel Processing For Scientific Computing*, pp. 374-383, SIAM, 1989.

[Birm85] Birman, K., "Replication and Fault Tolerance in the ISIS System," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pp. 79-86, December 1985.

[Bisi88] Bisiani, R. and A. Forin, "Multilanguage Parallel Programming of Heterogeneous Machines," *IEEE Transactions on Computers*,vol. 37, no. 8, August 1988.

[Bloo79] Bloom, T., "Synchronization mechanisms for modular programming languages," *MSc. dissertation, MIT/LCS/TR-211*, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1979.

[Boro82] Borodin, A. and J.E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation," *Proceedings Fourteenth ACM Symposium on Theory of Computing*, pp. 338-344, 1982.

[Breu88] Breuel, T.M., "Data Level Parallel Programming in C++," *Proceedings USENIX C++ Workshop*, pp. 153-167, 1988.

[Brew88] Brewer, O., J. Dongarra, and D. Sorensen, "Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs," *Technical Memorandum*, Mathematical and Computer Science Division, Argonne National Laboratory, June 1988.

[Brin72] Brinch Hansen, P., "Structured Multiprogramming," *Comm. ACM*, vol. 15, no. 7, pp. 574-578, July 1972.

[Brin73a] Brinch Hansen, P., *Operating Systems Principles*, Prentice-Hall, New Jersey, 1973.

[Brin73b] Brinch Hansen, P., "Concurrent programming concepts," *ACM Computing Surveys*, vol. 5, no. 4, pp. 223-245, Dec. 1973.

[Brin75] Brinch Hansen, P., "The programming language Concurrent Pascal," *IEEE Trans. on Software Engineering*, vol. SE-1, no. 2, pp. 199-206, June 1975.

[Brin78] Brinch Hansen, P., "Distributed processes: A concurrent programming concept," *Comm. ACM*, vol. 21, no. 11, pp. 934-941, Nov. 1978.

[Brin81] Brinch Hansen, P., "Edison: a multiprocessor language," *Software -- Practice and Experience*, vol. 11, no. 4, pp. 325-361, April 1981.

[Brod81] Brode, B., "Precompilation of Fortran Programs to Facilitate Array Processing," *Computer*, vol. 14, pp. 46-51, September 1981.

[Camp74] Campbell, R.H. and A.N. Habermann, "The specification of process synchronization by path expressions," *Lecture Notes in Computer Science*, vol. 16, pp. 89-102, Springer-Verlag, Heidelberg, 1974.

[Camp76] Campbell, R.H., "Path Expressions: a technique for specifying process synchronization," *PhD. Dissertation, Computing Laboratory*, University of Newcastle upon Tyne, August 1976.

[Camp79] Campbell, R.H. and R.B. Kolstad, *Path Expressions in Pascal*, Dept. Computer Science University of Illinois at Urbana, Urbana, Illinois, January 1979.

[Carr85] Carriero, N. and D. Gelernter, "The S/Net's Linda Kernel," *10th ACM Symposium on Operating System Principles*, December 1985.

[Carr87] N.J. Carriero, "Implementing Tuple Space Machines," *Research Report 567*, vol. C-36, no. 12, pp. 1425-1439, Yale University, Department of Computer Science, December 1987.

[Carr89] Carriero, N. and D. Gelernter, "How to Write Parallel Program: A Guide to the Perplexed," *ACM Computing Surveys*, vol. 21, no. 3, pp. 323-358, September 1989.

[Chan90] Chandy, K.M., S. Taylor, C. Kesselman, and I. Foster, *The Program Composition Project*, February 1990.

[Cher85] Cheriton, D.R. and W. Zwaenepoel, "Distributed Process Groups in the V Kernel," *TOCS*, vol. 3, no. 2, pp. 77-107, May 1985.

[Chin86] Ching, W-M., "Program analysis and code generation in an APL/370 compiler," *IBM Journal of Research & Development*, vol. 30, no. 6, pp. 594-602, 1986.

[Chin88] Ching, W-M. and A. Xu, "A Vector Code Back End of the APL370 Compiler on IBM 3090 and some Performance Comparisons," *Proceedings of the APL-88 Conference*, pp. 69-76, Sydney, Australia, February 1988.

[Chur41] Church, A., "The Calculi of Lambda-Conversion," *Annals of Mathematics Studies*, no. 6, Princeton University Press, 1941.

[Coff71] Coffman, E.G., M. Elphick, and A. Shoshani, "System Deadlocks ," *Computing Surveys*, vol. 3, no. 2, pp. 67-78, June 71.

[Conw63a] Conway, M.E., "A multiprocessor system design," *AFIPS Conf. Proc.*, vol. 24, pp. 139-146, Spartan Books, Baltimore, 1963.

[Conw63b] Conway, M.E., "The Design of a separable transition-diagram compiler," *Comm. ACM*, vol. 6, no. 7, pp. 396-408, July 1963.

[Cook80] Cook, R.P., "*MOD - A language for distributed programming," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 563-571, Nov. 1980.

[Crow85] Crowther, W., J. Goodhue, R. Gurwitz, R. Rettberg, and R. Thomas, The Butterfly Parallel Processor™, *IEEE Computer Architecture Technical Committee Newsletter*, pp. 18-45, September-December 1985.

[Cvet87] Cvetanovic, Z., "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems," *IEEE Transactions on Computers*, vol. C-36, no. 4,pp. 421-432, April 1987.

[Davi86] Davies, J., C. Hudson, T. Macke, B. Leasure, and M. Wolfe, "The KAP/S-1: An Advanced Source-to-Source Vectoriser for the S-1 Mark IIa Supercomputer," *Proceedings 1986 International Conference on Parallel Processing*, pp. 833-835, IEEE Comp. Soc. Press, 1986.

[Deit84] Deitel, H.M., *An Introduction to Operating Systems*, Addison-Wesley, 1984.

[Demm87] Demmel, J.W., J.J. Dongara, J.J. Du Croz, A. Greenbaum, S.J. Hammarling, and D.C. Sorensen, "Prospectus for the development of a linear algebra library for high-performance computers," *Technical Memorandum No 97*, Mathematics and Computer Science Division, Argonne National Laboratory, Illinois, 1987.

[Denn66] Dennis, J.B. and E.C. Van Horn, "Programming semantics for multiprogrammed computations," *Comm. ACM*, vol. 9, no. 3, pp. 143-155, March 1966.

[Denn79] Dennis, J.B., C.K. Leung, and D.P. Misunas, "A Highly Parallel Processor Using a Data Flow Machine Language," *CSG Memo 134-1*, Laboratory for Computer Science, MIT, June 1979.

[Detl88] Detlefs, D.L., M.P. Herlihy, and J.M. Wing, "Inheritance of synchronisation and recovery properties in Avalon/C++," *Computer*, vol. 21, no. 12, pp. 57-69, December 1988.

[Dijk65a] Dijkstra, E.W., *Cooperating Sequential Processes*, Academic Press, 1968, Also in *Programming Languages*, Academic Press, pp. 43-112, 1968.

[Dijk65b] Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control," *CACM*, vol. 18, no. 5, pp. 341-346, May 1965.

[Dijk68] Dijkstra, E.W., "The Structure of "THE" multiprogramming system," *Communications ACM*, vol. 11, no. 5, pp. 341-346, 1968.

[Dijk75] Dijkstra, E.W., "Guarded Commands, Non-determinacy, and Formal Derivation of Programs," *Communications of ACM*, vol. 18, no. 8, pp. 453-457, August 1975.

[DoD80] *Reference Manual for the Ada Programming Language*, U.S. Dept. of Defence, July 1980.

[Doep87] Doeppner, T.W., "Threads: A System for the Support of Concurrent Programming," *Technical Report CS-87-11*, Department of Computer Science Brown University, June 1987.

[Dong85] Dongarra, J.J. and I.S. Duff, "Advanced Computer Architectures," *Mathematics and Computer Science Div. Report TM-57*, Argonne National Laboratory, 1985.

[Dong86] Dongara, J.J. and D.C. Sorenson, "Schedule: Tools for Developing and Analyzing Parallel Fortran Programs," *Technical Memorandum No 86*, Mathematics and Computer Science Division, Argonne National Laboratory, Illinois, November 1986.

[Dunc90] Duncan, R., "A Survey of Parallel Computer Architectures," *Computer*, vol. 23, no. 2, IEEE, February 1990.

[Elli85] Ellis, J., "Bulldog: A Compiler for VLIW Architectures," *Ph.D. Dissertation*, Department of Computer Science, Yale University, February 1985.

[Enco87] *Multimax Technical Summary*, Encore Computer Corporation, 1987.

[Ensl77] Enslow, P.H., "Multiprocessor Organisations - A Survey," *Computing Surveys*, vol. 9, pp. 103-129, March 1977.

[Feld79] Feldman, J.A., "High level programming for distributed computing," *Comm. ACM*, vol. 22, no. 6, pp. 353-368, June 1979.

[Feng72] Feng, T.Y., "Some Characteristics of Associative/Parallel Processing," *Proceedings 1972 Sagamore Computing Conference*, pp. 5-16, Syracuse University.

[Feng81] Feng, T., "A Survey of Interconnection Networks," *Computer*, pp. 12-27, IEEE, Dec. 1981.

[Feo90] Feo, J.T., D.C. Cann, and R.R. Oldehoeft, A Report on the Sisal Language Project, *Journal of Parallel and Distributed Computing*, 10, pp. 349-366, December 1990.

[Fern81] Fernbach, S., "Parallelism in Computing," *Proc. Int. Conf. Parallel Processing*, pp. 1-4, August 1981.

[Fiel84] Fielland, G. and D. Rogers, "32-bit Computer System Shares Load Equally Among up to 12 Processors," *Electronic Design*, pp. 153-168, January 1984.

[Fink86] Finkel, R.A., *An Operating Systems Vade Mecum*, Prentice-Hall, 1986.

[Flon76] Flon, L. and A.N. Habermann, "Towards the construction of verifiable software systems," *Proc. ACM Conference on Data*, SIGPLAN Notices, vol. 8, no. 2, pp. 141-148, March 1976.

[Floy67] Floyd, R.W., "Assigning meanings to programs," *Proc. Amer. Math Society Symp in Applied Mathematics*, vol. 19, 1967.

[Flyn66] Flynn, M.J., "Very High-Speed Computing Systems," *Proceedings IEEE*, vol. 54, pp. 1901-1909, December 1966.

[Fort78] Fortune, S. and J. Wyllie, "Parallelism in Random Access Machines," *Proceedings Tenth ACM Symposium on Theory of Computing*, pp. 114-118, 1978.

[Fost90] Foster, I. and S. Taylor, *Strand New Concepts in Parallel Programming*, Prentice Hall, 1990.

[Fran84] Frank, S.J., "A Tightly Coupled Multiprocessor System Speeds Memory-Access Times," *Electronics*, vol. 57, no. 1, pp. 164-169, January 1984.

[Gajs84] Gajski, D.D., D.H. Lawrie, D.J. Kuck, and A.H. Sameh, "CEDAR," *Proceedings IEEE COMPCON'84*, pp. 306-309, March 1984.

[Gall88] Gallivan, K., D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, *Behavioral characterization of multiprocessor memory systems: a case study*, 1988.

[Gard70] Gardner, M., "Articles on Life," *Scientific American*, no. 4, p.120, April 1970.

[Gehr88] Gehringer, Edward F., Janne Abullarade, and Michael H. Gulyn, "A Survey of Commercial Parallel Processors," *Computer Architecture News*, vol. 16, no. 4, pp. 75-107, ACM, September 1988.

[Gele82] Gelernter, D. and A.J. Bernstein, "Distributed communication via global buffer," *Proc. Symposium on Principles of Distributed Computing*, pp. 10-18, Ottawa, Canada, August 1982.

[Ghez85] Ghezzi, C., "Concurrency in programming languages," *Parallel Computing*, pp. 229-241, November 1985.

[Gold88] Goldman, R. and R.P. Gabriel, "Preliminary Results with the Initial Implementation of Qlisp," *Proceedings of 1988 Conference on Lisp and Functional Programming*, pp. 143-152, July 1988.

[Good79] Good, D.I., R.M. Cohen, and J. Keeton-Williams, "Principles of proving concurrent programs in Gypsy," *Proc. Sixth ACM Symposium on Principles of Programming Languages*, pp. 42-52, San Antonio, Texas, January 1979.

[Good83] Goodman, J.R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings 10th Annual International Symposium on Computer Architecture*, 1983.

[Gott83] Gottlieb, A., B.D. Lubachevshy, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 2, pp. 164-189, April 1983.

[Gray78] Gray, J.N., "Notes on Data Base Operating Systems," *Lecture Notes in Computer Science: Operating Systems and Advanced Course*, vol. 60, pp. 393-481, Springer-Verlag, 1978.

[Greg63] Gregory, J. and R. McReynolds, The Solomon Computer, EC-12, pp. 774-781, *IEEE Transactions on Electronic Computers*, December 1963.

[Gurd85] Gurd, J.R., C.C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34-52, January 1985.

[Habe75] Habermann, A.N., *Path Expressions*, Dept. of Computer Science Carnegie-Mellon University, Pittsburgh, Pennsylvannia, June 1975.

[Hadd77] Haddon, B.K., "Nested monitor calls," *Operating Systems Review*, vol. 11, no. 4, pp. 18-23, Oct. 1977.

[Hall90] Hall, J. G., R.P. Hopkins, O. Botti, and F. De Cindio, "A Petri Net Semantics of OCCAM 2.," *Technical Report*, Newcastle University Computing Laboratory, 1990.

[Hals87] Halstead, Robert H. Jr., "Parallel Computing Using Multilisp," *in Parallel Computation and Computers for Artificial Intelligence*, ed. Janusz S. Kowalik, pp. 21-50, Kluwer Academic Publishers, New York, 1987.

[Herl90] Herlihy, M.P. and J.M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 463-492, July 1990.

[High89] Higham, N.J., "Exploiting Fast Matrix Multiplication Within the Level 3 BLAS," *ACM Transactions on Mathematical Software*, September 1989.

[Hill85] Hillis, W. Daniel, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.

[Hoar62] Hoare, C.A.R., "Quicksort," *Computer Journal*, vol. 5, pp. 10-15, 1962.

[Hoar69] Hoare, C.A.R., "An axiomatic basis for computer programming," *Comm. ACM*, vol. 12, no. 10, pp. 576-580,583, Oct. 1969.

[Hoar72] Hoare, C.A.R., "Towards a theory of parallel programming," *Operating Systems Techniques*, Academic Press, New York, 1972.

[Hoar74] Hoare, C.A.R., "Monitors: an operating system structuring concept," *Communications ACM*, vol. 17, no. 10, pp. 549-557, Oct. 1974.

[Hoar78] Hoare, C.A.R., "Communicating sequential processes," *Comm. ACM*, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[Hock81] Hockney, R.W. and C.R. Jesshope, *Parallel Computers*, Adam Hilger Ltd., 1981.

[Holt81] Holt, R.C., D.B. Wortmann, J.R. Cordy, D.R. Crowe, and I.H. Griggs, "Euclid: A language for producing quality software," *Proceedings of National Computer Conference*, Chicago, May 1981.

[Holt83] Holt, R.C., *Concurrent Euclid, The Unix system and Tunis*, Addison-Wesley, 1983.

[Hoog83] Hoogendoorn, C.H., "On the use of spin locks in multiprocessors," *Parallel Computing83*, pp. 413-417, North-Holland.

[Huda89] Hudak, P., "Conception, Evolution and Application of Functional Programming Languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359-411, September 1989.

[Hünd77] Hündler, W., "The Impact of Classification Schemes on Computer Architectures," *Proceedings 1977 International Conference on Parallel Processing*, pp. 7-15.

[Hutc87] Hutchinson, N.C., "Emerald: A Language to Support Distributed Programming," *Proceedings from the Second Workshop on Large-Grained Parallelism*, pp. 45-47, Carnegie-Mellon University Software Engineering Institute, Pittsburgh, PA, November 1987.

[Hwan85] Hwang, K. and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1985.

[IBM85] *IBM 3090 System Summary - Engineering/Scientific announcement letter*, pp. 185-220, IBM Corporation, 1985.

[Jako90] Jakob, R. and H.F. Jordan, "An MIMD Execution Environment with a Fixed Number of Processes," *Proceedings VAPP/CONPAR*, September 1990.

[Jone80] Jones, A.K. and P. Schwarz, "Experience using multiprocessor systems - a status report," *ACM Computing Surveys*, vol. 12, no. 2, pp. 121-165, June 1980.

[Jord84] Jordan, H.F., "Structuring Parallel Algorithms in an MIMD, Shared Memory Environment," *Report Number CSDG 84-2*, Dept. of Electrical and Computer Engineering, University of Colorado, 1984.

[Jord85] Jordan, H.F., "Parallel Computation with the Force," *ICASE Report Number 85-45*, NASA Langley, 1985.

[Kaln79] Kalney-Rivas, E. and D. Hoitsma, "Documentation of the Fourth Order Band Model," *NASA Technical Memorandum 80608*, December 1979.

[Karp87] Karp, Alan H., "Programming for Parallelism," *Computer*, vol. 20, no. 5, pp. 43-57, IEEE, May 1987.

[Karp90] Karp, A.H. and H.P. Flatt, "Measuring Parallel Processor Performance," *Communications of the ACM*, vol. 33, no. 5, May 1990.

[Katz78] Katzmann, J.A., "A Fault-Tolerant Computing System," *Proceedings 11th Hawaii International Conference on System Sciences*, pp. 85-117, IEEE Computer Society, 1978.

[Katz85] Katz, R. *et al.*, "Implementing a Cache Consistency Protocol," *Proceedings of 12th Annual International Symposium on Computer Architecture* , pp. 276-283, New York, June 1985.

[Kell84] Keller, R.M., F.C.H. Lin, and J. Tanaka, "Rediflow Multiprocessing," *Proceedings IEEE COMPCON'84*, pp. 410-417, March 1984.

[Knig66] Knight, K.E., "Changes in Computer Performance," *Datamation*, vol. 12, pp. 40-54, September 1966.

[Knut71] Knuth, D.E., "An Empirical Study of Fortran Programs," *Software -- Practice and Experience*, vol. 1, pp. 105-133, 1971.

[Kohl81] Kohler, W.H., "A survey of techniques for synchronization and recovery in decentralized computer systems," *ACM Computing Surveys*, vol. 13, no. 2, pp. 149-183, June 1981.

[Konn89] Konno, C., M. Yamabe, M. Saji, and Y. Umetari, "The BF Coordinate Transformation Technique DEQSOL," *Parallel Processing For Scientific Computing*, SIAM, 1989.

[Korn82] Kornfeld, W.A., "Combinatorially Implosive Algorithms," *Communications of the ACM*, vol. 25, no. 10, October 1982.

[Krist87] Kristensen, B.B., O.L. Madsen, B. Moller-Pedersen, and K. Nygaard, "The BETA Programming Language," *Research Directions in Object-Oriented Programming*, pp. 7-48, The MIT Press, 1987.

[Kuck72] Kuck, D. *et al.*, "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," *IEEE Transactions on Computers*, vol. C-21, no. 12, pp. 1293-1310, December 1972.

[Kuck74] Kuck, D. *et al.*, "Measurements of Parallelism on Ordinary FORTRAN Programs," *Computer*, pp. 37-46, January 1974.

[Kuck78] Kuck, D.J., *The Structure of Computers and Computations*, John-Wiley, 1978.

[Kuck84] Kuck, D.J., R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Retargetable Vectorizer," *Tutorial on Supercomputers*, pp. 163-178, IEEE Press, 1984.

[Kung78] Kung, H.T. and C.E. Leiserson, "Systolic Arrays for VLSI," *Proceedings of the Sparse Matrix Symposium (SIAM)*, 1978.

[Lamp80] Lampson, B.W. and D.D. Redell, "Experience with processes and monitors in Mesa.," *Comm. ACM*, vol. 23, no. 2, pp. 105-117,Feb. 1980.

[Lamp81] Lampson, B.W., "Atomic transactions," *Distributed Systems - Architecture and Implementation*, Lecture Notes in Computer Science, vol. 105, Springer-Verlag, New York, 1981.

[Lamp74] Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem," *CACM*, vol. 17, pp. 453-455, 1974.

[Lars84] Larson, J.L., "An introduction to multitasking on the Cray X-MP-2 multiprocessor," *Computer*, pp. 62-69, July 1984.

[Laue75] Lauer, P.E. and R.H. Campbell, "Formal Semantics of a class of high level primitives for coordinating concurrent processes," *Acta Informatica*, vol. 5, pp. 297-332, 1975.

[Laue78] Lauer, P.E. and M.W. Shields, "Abstract specification of resource accessing disciplines: adequacy, starvation, priority and interrupts.," *SIGPLAN Notices*, vol. 13, no. 12, pp. 41-59, December 1978.

[Laue79] Lauer, H.C. and R.M. Needham, "On the duality of operating system structures," *Proc. Second Int. Symp. on Operating Systems, reprinted in Operating Systems Review*, vol. 13, no. 2, pp.3-19, April 1979.

[LeBl90] LeBlanc, T.J., J.M. Mellor-Crummey, and R.J. Fowler, "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 203-217, June 1990.

[Lee89] Lee, P.A., "Shared Memory Multiprocessors: A Cost Effective Architecture For Parallel Processors," *Software For Parallel Computers*, UNICOM, June 1989.

[Lee90] Lee, P.A. and M.A. Stoker, "Mechanisms For Parallel Programming on Unix Multiprocessor Systems," *Proceedings van de NLUUG Voorjaarsconferentie 1990*, pp. 55-66.

[Lipo87] Lipovski, G.J. and M. Malek, *Parallel Computing: Theory and Comparisons*, John Wiley & Sons, 1987.

[Lisk77] Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, vol. 20, no. 8, August 1977.

[Lisk82] Liskov, B. and R. Scheifler, "Guardians and actions: linguistic support for robust, distributed programs," *Proc. Ninth ACM Symp. on Principles of Programming Languages*, pp. 7-19, Albuquerque, New Mexico, January 1982.

[List77] Lister, A., "The problem of nested monitor calls," *Operating Systems Review*, vol. 11, no. 3, pp. 5-7, July 1977.

[Lytt90] Lyttle, R.W., "Fortran For Shared Memory Parallel Processors," *Parallel Update*, no. 13, pp. 36-46, BCS Parallel Processing Specialist Group, 1990

[Mand82] Mandelbrot, B., *The fractal geometry of nature*, W.H. Freeman, San Francisco, 1982.

[McDo89] McDowell, C.E. and D.P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, pp. 593-622, December 1989.

[McGr82] McGraw, J.R., "The VAL Language: Description and Analysis," *ACM TOPLAS*, vol. 4, no. 1, January 1982.

[McGr83] McGraw, J.R. *et al.*, "SISAL: Streams and Iteration in a Single-Assignment Language," *Language Reference Manual Version 1.1, Tech Report M-146*, 1983, Lawrence Livermore National Laboratory.

[McJo87] McJones, P. and A. Hisgen, "The Topaz System: Distributed Multiprocessor Personal Computing," *Proceedings: Workshop on Workstation Operating Systems*, IEEE Computer Society Technical Committee on Operating Systems, Cambridge, MA, November 1987.

[Mead80] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison Wesley, 1980.

[Meil81] Meilander, W.C., "History of Parallel Processing at Goodyear Aerospace," Proc. 1981 *Int. Conf on Parallel Processing*, IEEE Computer Society, August 1981.

[Meth85] Methrotra, P. and J. Van Rosendale, "The BLAZE Language: A Parallel Language for Scientific Programming," *ICASE Report Number 85-29*, NASA Langley Research Center, 1985.

[Mitc79] Mitchell, J.G., W. Maybury, and R. Sweet, "Mesa language manual version 5.0," *Xerox Palo Alto Research Center Report CSL-79-3*, April 1979.

[Mitz86] Mitzell, D., "PROLOG AND PARALLELISM: The Inherently Sequential Nature of Unification," *Naval Research Reviews*, vol. 1, pp. 22-24, 1986.

[Mull86] Mullender, S.J., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, 1986, pp. 289-299, 1986.

[Mura71] Muraoka, Y., "Parallelism Exposure and Exploitation in Programs," *PhD. dissertation*, Report 71-424, Dept. of Computer Science, University of Illinois, Urbana-Champaign, February 1971.

[Myer86] Myers, Ware, "Getting the Cycles out of a Supercomputer," *Computer*, vol. 19, no. 3, pp. 89-92, IEEE, March 1986.

[Nico88] Nicol, D.M., "Parallel Algorithms for Mapping Pipelined and Parallel Computations," *ICASE Report No 88-2*, April 1988.

[Nils80] Nilsson, N.J., *Principles of Artificial Intelligence*, 1980.

[Noga85] Noga, M.T. and D.C.S Allison, "Sorting in Linear Expected Time," *BIT*, vol. 25, pp. 451-465, 1985.

[Nyga78] Nygarrd, K. and O.J. Dahl, "The development of the SIMULA languages," *Preprints ACM SIGPLAN History of Programming Languages Conference SIGPLAN Notices*, vol. 13, no. 8, pp. 245-272, Aug. 1978.

[Orga82] Organick, E., *A programmer's Guide to the Intel 432*, McGraw-Hill, 1982.

[Oste86] Osterhaug, A., *Guide to Parallel Programming*, Sequent Computer Systems Inc., 1986.

[Paal89] Paalvast, E.M. and H.J. Sips, "A High-Level Language for the Description of Parallel Algorithms," *Parallel Computing89*, North Holland.

[Papa84] Papamarcos, M.S. and J.H. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings 11th Annual International Symposium on Computer Architecture*, pp. 348-354, Ann Arbor MI, June 1984.

[Paul75] Paul, G. and M.W. Wilson, "The Vectran Language: An Experimental Language for Vector/Matrix Array Processing," *IBM Palo Alto Scientific Center Report 6320-3334*, August 1975.

[Paul82] Paul, G., "Vectran and the Proposed Vector/Array Extensions to ANSI Fortran for Scientific and Engineering Computation.," *Proceedings IBM Conference on Parallel Computers*, 1982.

[Perr86] Perron, R. and C. Mundie, "The Architecture of the Alliant FX/8 Computer," *Digest of papers Compcon86*, IEEE Computer Society Press, Spring 1986.

[Perr79] Perrott, R.H., "A Language for Array and Vector Processors," *ACM Transaction on Programming Languages and Systems*, vol. 1, no. 2, pp. 177-195, October 1979.

[Perr87] Perrott, R.H., *Parallel Programming*, Addison-Wesley, 1987.

[Peyt86] Peyton-Jones, S.L., *The implementation of functional programming languages*, Prentice-Hall, 1986.

[Peyt90] Peyton-Jones, S., "Parallel Programming@Glasgow," *Parallelogram International*, vol. 2, no. 27, pp. 16-22, June 1990.

[Pfis85] Pfister, G.F., W.C. Brantely, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3)," *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[Poly87] Polychronopoulos, C.D. and D.J. Kuck, "Guided Self-Scheduling: A Practical Sceduling Scheme for Parallel SuperComputers," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp.1425-1439, December 1987.

[Poly89] Polychronopoulos, C.D., M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," *1989 International Conference on Parallel Processing*, vol. 2, pp. 39-48.

[Prat84] Pratt, T.W, *Programming Languages - Design and Implementation*, Prentice-Hall, 1984.

[Rayn86] Raynal, M., *Algorithms For Mutual Exclusion*, pp. 18-22, North Oxford Academic, 1986.

[Ratt85] Rattner, J., "Concurrent Processing: A new direction in scientific computing," *AFIPS Conference Proceedings 54*, pp. 159-166, AFIPS Press, Reston Va., 1985.

[Reed79] Reed, D.P, "Implementing atomic actions on decentralized data," *Preprints for Seventh Symposium on Operating System Principles*, pp. 66-74, Pacific Grove, California, December 1979.

[Ritc74] Ritchie, D.M. and K. Thompson, "The UNIX timesharing system," *Comm. ACM*, vol. 17, no. 7, pp. 365-375, July 1974.

[Rose87] Rose, J.R. and G.L. Steele, "C*: An Extended C Language," *Proceedings USENIX C++ Workshop*, pp. 361-398, 1987.

[Rosi89] Rosing, M. and R.B. Schnabel, "An Overview of Dino - A New Language for Numerical Computation on Distributed Memory Multiprocessors," *Parallel Processing For Scientific Computing*, pp. 312-316, SIAM, 1989.

[Rozi88] Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "CHORUS Distributed Operating Systems," *USENIX Computing Systems*, vol. 1, no. 4, pp. 301-370, University of Califorina Press, 1988.

[Rudo84] Rudolph, L. and Z. Segall, "Dynamic Decentralised Cache Schemes for MIMD Parallel Processors," *Proceedings 11th Annual International Symposium on Computer Architecture*, pp. 340- 347, Ann Arbor MI, June 1984.

[Russ69] Russell, E.C., "Automatic Program Analysis," *PhD. dissertation*, Dep. Engineering, University of California, Los Angeles, 1969.

[Sarg87] Sargeant, J., "Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines," *Internal Report UMCS-86-11-5*, Department of Computer Science, University of Manchester, January 1987.

[Sche77] Scheifler, R.W., "An analysis of inline substitution for a structured programming language," *Communications of the ACM*, 1977.

[Seit85] Seitz, C.L., "The Cosmic Cube," *Comm. ACM*, vol. 28, pp. 22-33, 1985.

[Shap89] Shapiro, E., "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 412- 510, September 1989.

[Shaw84] Shaw, D.E., "SIMD and MSIMD Variants of the NON-VON Supercomputer," *IEEE COMPCON'84 Proceedings*, pp. 360-363, March 1984.

[Shen90] Shen, V.Y., C. Richter, M.K. Graf, and J.A. Brumfield, "VERDI: A Visual Environment for Designing Distributed Systems," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 128-137, June 1990.

[Shor73] Shore, J.E., "Second thoughts on parallel processing," *Computer and Electrical Engineering*, vol. 1, pp. 95-109, 1973.

[Shri89] Shrivastava, S.K., G.N. Dixon, G.D. Parrington, F. Hedayati, S.M. Wheater, and M. Little, "The Design and Implementation of Arjuna," *Technical Report*, University of Newcastle upon Tyne, 1989.

[Shri90] Shrivastava, S.K. and S.M. Wheater, "Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions," *Proceedings of Tenth International Conference on Distributed Computing Systems*, Paris, France, May 1990.

[Skil88] Skillicorn, D.B., "A Taxonomy for Computer Architectures," *Computer*, vol. 21, no. 11, pp. 46-47, November 1988.

[Slad56] Slade, A.E. and H.O. McMahon, "A Cryoton Catalog Memory System," *Proc. EJCC*, vol. 10, pp. 115-120, December 1956.

[Smit81] Smith, B.J., "Architecture and applications of the HEP multiprocessor computer system," *SPIE (Real-Time Signal Processing IV)*, vol. 298, pp. 241-248, Society of Photo-Optical Instrument Engineers, Bellingham, Washington, 1981.

[Snir82] Snir, M., "On Parallel Search," *Proceedings Principles of Distributed Computing Conference*, pp. 242-253, August 1982.

[Snyd82] Snyder, L., "Introduction to the Configurable, Highly Parallel Computer," *Computer*, vol. 15, pp. 47-56, January 1982.

[Snyd84] Snyder, L., "Parallel Programming and the Poker Programming Environment," *Computer*, vol. 17, no. 7, July 1984.

[Snyd87] Snyder, A., "Inheritance and the Development of Encapsulated Software Systems," *Research Directions in Object-Oriented Programming*, pp. 165-188, The MIT Press, 1987.

[Ster87] Sterling, L. and Ehud Shapiro, *The Art of Prolog*, The MIT Press,March 1987.

[Stok81] Stokes, R. and R. Cantarella, "The history of Parallel Processing at Burroughs," *Proc. 1981 Int. Conf on Parallel Processing*, IEEE Computer Society, August 1981.

[Tane87] Tanenbaum, A.S., *Operating Systems - Design and Implementation*, Prentice-Hall, 1987.

[Thom88] Thomas, Robert H. and Will Crowther, "The Uniform System: An approach to runtime support for large scale shared memory parallel processors," *Proceedings of the 1988 International Conference on Parallel Processing*, vol. II, Software, pp.245-254, Penn State, University Park, Penn, August 1988.

[Tink88] Tinker, P. and M. Katz, "Parallel Execution of Sequential Scheme with ParaTran," *Proceedings of 1988 ACM Conference on LISP and Functional Programming*, pp. 28-39, 1988.

[Thor81] Thornton, J.E., "Control Data 6600 and STAR-100," *Proc. 1981 Int. Conf on Parallel Processing*, IEEE Computer Society, August 1981.

[Trel82] Treleaven, P.C., D.R. Brownbridge, and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys*, vol. 14, no. 1, pp. 95-143, March 1982.

[Vaug89] Vaughan, J.R., "The Mandelbrot set as a parallel processing benchmark," *University Computing*, vol. ii, no. 4, pp. 193-7,December 1989.

[vonN58] Neumann, J. von, *The Computer and the Brain*, Yale University Press, New Haven, 1958.

[Wadg85] Wadge, W.W. and E.A. Ashcroft, *Lucid, The Dataflow Programming Language*, Academic Press, 1985.

[Wads71] Wadsworth, C.P., "Semantics and Pragmatics of the lambda calculus," *PhD. Thesis*, Oxford, 1971.

[Weih80] Weihl, E.W., "Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables," *Seventh Annual ACM Symposium on Principles of Programming Languages*, pp. 83-94, 1980.

[Wels79] Welsch, J. and D.W. Bustard, "Pascal-Plus another language for modular multiprogramming," *Software -- Practice and Experience*, vol. 9, pp. 127-130, 1979.

[Wirt77] Wirth, N., "Modula: a language for modular multiprogramming," *Software -- Practice and Experience*, vol. 7, pp. 3-35, 1977.

[Wirt80] Wirth, N., "Modula-2," *Technical Report 36*, Institut fuer Informatik E.T.H. Zurich, March 1980.

[Wolf89] Wolfe, M., *Optimizing Supercompilers for Supercomputers*, Pitman, 1989.

[Wrig90] Wright, K., "Parallel Algorithms for QR Decomposition on a Shared Memory Multiprocessor," *Parallel Processing Memoranda*, no. 7, Computing Labroratory, University of Newcastle upon Tyne, December 1990.

[Wulf71] Wulf, W.A., D.B. Russell, and A.N. Habermann, "BLISS: A language for systems programming," *Comm. ACM*, vol. 14, no. 12, pp. 780-790, December 1971.

[Wulf72] Wulf, W.A. and C.G. Bell, "C.mmp - A multi-mini-processor," *Proceedings AFIPS 1972 Fall Joint Computer Conference*, vol. 41, pp. 765-777, 1972.

[Yoko86] Yokote, Y. and M. Tokoro, "The design and implementation of ConcurrentSmalltalk," *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, 1986.

[Zeni90] Zenith, S.E., "Programming with Ease: Semiotic definition of the language," *Yale Technical Report TR-809*, Yale University, Department of Computer Science, New Haven, Connecticut, July 1990.

# *Appendix A*

## Julia Set Fractals

To give some impression of the difficulties encountered in partitioning the work for the fractal parallel programs both the Julia sets used in chapter six are displayed on the next page. The upper image is the A image, which having a high concentration of work in its middle group of columns helps to explain why statically allocating blocks of columns to workers leads to poor speedups being obtained. The lower image is the B image, which has the same amount of work per column and therefore gives good speedups when blocks of columns are computed in parallel. However, this is not necessarily true if rows are allocated to threads instead of columns, because some rows take significantly more iterations to complete than others, although this is not obvious from the diagram.

Sequential Julia Set Algorithm

This algorithm is taken from Mandelbrot's book 'The Fractal Geometry of Nature' [Mand82]. An image has *a* times *b* points and 0 to K possible colours. Given values for *p* and *q* where c = p + iq, *x* where $x_{min} \le x \le x_{max}$, *y* where $y_{min} \le y \le y_{max}$ and *M* is 100,

(0) Set $\Delta x = (x_{max} - x_{min})/(a - 1)$, $\Delta y = (y_{max} - y_{min})/(b - 1)$, for all the pixels $(n_x, n_y)$ with $n_x = 0, ..., a - 1$ and $n_y = 0, ..., b - 1$ do the following,

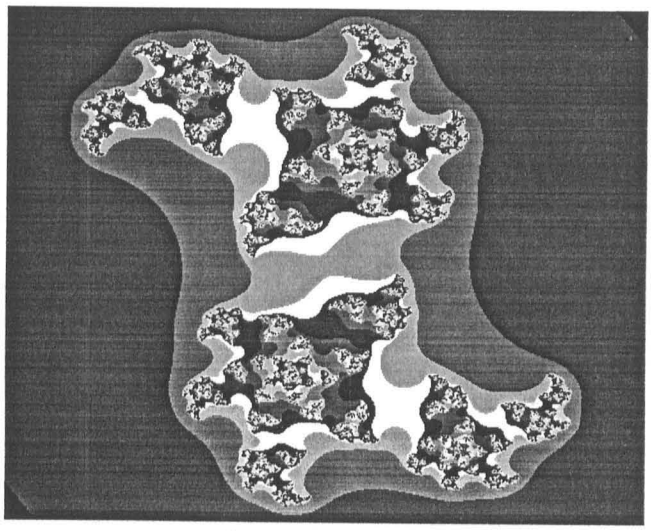
(1) Set $x_0 = x_{min} + n_x.\Delta x$, $y_0 = y_{min} + n_y.\Delta y$, k = 0.

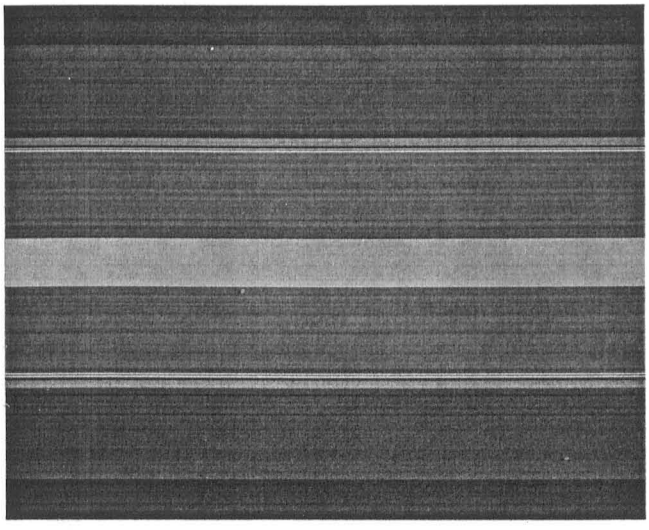
(2) Calculate $(x_{k+1}, y_{k+1})$ from $(x_k, y_k)$ using the law in chapter six, k += 1.

(3) Calculate $r = x_k^2 + y_k^2$
if (r ≤ M) and (k < K) go to step (1)
else if (r > M) then choose colour k
else choose colour 0 (black) ( k = K ).

(4) Assign colour k to the pixel $(n_x, n_y)$, compute next pixel starting from (1).

A image: **p** = 0.4, **q** = 0.4i, **x** = -1.25 to 1.25, y = -1.25 to 1.25, M = 100, K = 100.
B image: **p** = -0.257, **q** = -0.89i, **x** = 0.0 to 0.0, y = -2.0 to 2.0, M = 100, K = 100.





# *Appendix B*

## Test Program Listings

The programs listed in this appendix are some of the Tyger-C++ programs that were used for the testing reported on in chapter six. Four programs are presented:

(i) the vector assignment program,
(ii) the matrix addition program,
(iii) the matrix multiplication program (with transpose),
(iv) the fractal program (in three parts).

Some of the programs reference program text in the form of header files via the "#include" preprocessor directive. Where appropriate these header files have been shown either in this appendix or in the next, though with the exceptions of system header files, the timing header file, and the graphics header file which are not shown for brevity.

```
// parallel vector assignment program

#include "sv.h"

extern "C" {
#include "timelib.h"
}

void    printvector(SV& vector, int n)
{   cout << "\n";
    for (int i = 0; i < n; i++)
        cout << vector[i] << " ";
    cout << "\n";
}

PAR   dostore(struct list* prms)
{   int     lower =   prms->index;
    int     upper =   prms->ceiling;
    SV&     vector =  *(prms->host);

    for (int i = lower; i < upper; i++)
        vector[i] = (SVTYPE) i;
}

int main(int argc, char* argv[])
{   int         n;          // size of vector
    int         procs;      // number of processes
    rawTime     start;      // first timing point
    rawTime     secs;       // second timing point

    if (argc == 3)
    {   n = atoi(argv[1]);
        procs = atoi(argv[2]); }
    else
    {   cerr << "Usage: " << argv[0] << " size procs\n" << "\n";
        exit(1); }

    start = start_timer();
    SV vector(n);
    secs = raw_ticks(start);
    cerr << "Result after" << raw_micros(secs) << " microsecs\n";
    start = elapsed_ticks();
    vector.Clones(procs);
    vector.DoAll(0,n-1,dostore);
    secs = raw_ticks(start);
    cerr << "Ending time " << raw_micros(secs) << " microsecs\n";

    printvector(vector,n);
}
```

---

```
// parallel square matrix addition program

#include "sv.h"

int       n         ;   // the order of the matrices
SVTYPE    *matrix1  ;   // pointer to 1st value of 1st matrix
SVTYPE    *matrix2  ;   // pointer to 1st value of 2nd matrix

void    getmatrix(SVTYPE* matrix, int n)
{   int row;

    for (int i = 0; i < n; i++)
    {   row = i*n;
        for (int j = 0; j < n; j++)
            cin >> matrix[row + j];
    }
}

void    printmatrix(SV& matrix, int n)
{   int row;

    cout << "\n";
    for (int i = 0; i < n; i++)
    {   row = i*n;
        for (int j = 0; j < n; j++)
            cout << matrix[row+j] << " ";
        cout << "\n";
    }
    cout << "\n";
}
```

```
PAR   dosum(struct list *prms)
{   int     lower     =   prms->index;
    int     upper     =   prms->ceiling;
    int     row       =   lower*n;
    SV&     matrix3   =   *(prms->host);

    for (int i = lower; i < upper; i++)
    {   for (int j = 0; j < n; j++)
            matrix3[row+j] = matrix1[row+j] + matrix2[row+j];
        row += n;
    }
}

extern "C" {
#include "timelib.h"
}

int main(int argc, char *argv[])
{   int             procs ;
    timeUnits       start, secs;

    if (argc == 3)
    {   n = atoi(argv[1]);
        procs = atoi(argv[2]); }
    else
    {   cerr << "Usage: " << argv[0] << " size procs\n" << "\n";
        exit(1); }

    matrix1 = new SVTYPE[n*n];
    matrix2 = new SVTYPE[n*n];

    start = start_timer();
    SV matrix3(n*n);
    secs = seconds(start);
    cerr << "Result matrix created in " << secs << " s\n";

    getmatrix(matrix1,n);
    getmatrix(matrix2,n);
    secs = seconds(start);
    cerr << "Data read in after " << secs << " s\n";

    start = elapsed_time();
    matrix3.Clones(procs);
    matrix3.DoAll(0,n-1,dosum);
    secs = seconds(start);
    cerr << "Ending time " << secs << " s\n";

    printmatrix(matrix3,n);
}
```

---

```
// parallel square matrix multiplication program

#include "sv.h"

extern "C" {
#include "timelib.h"
}

int       n         ;   // the order of the matrices
SVTYPE    *matrix1  ;   // pointer to 1st value of 1st matrix
SVTYPE    *matrix2  ;   // pointer to 1st value of 2nd matrix

void   getmatrix(SVTYPE* matrix, int n)
{   int row;

    for (int i = 0; i < n; i++)
    {   row = i*n;
        for (int j = 0; j < n; j++)
            cin >> matrix[row + j];
    }
}

void   printmatrix(SV& matrix, int n)
{   int row;

    cout << "\n";
    for (int i = 0; i < n; i++)
    {   row = i*n;
        for (int j = 0; j < n; j++)
            cout << matrix[row+j] << " ";
        cout << "\n";
```

```
    }
    cout << "\n";
}

void   transpose(SVTYPE *matrix, int n)
{   SVTYPE  *ptr1, *ptr2, *cPtr, *rPtr, *ePtr;
    SVTYPE  swap;
    int     offset = n + 1;

    ePtr = ptr2 = matrix + n;
    for (ptr1 = matrix+1; ptr1 < matrix + n*n; ptr1 += offset)
    {   for (cPtr = ptr1, rPtr = ptr2; cPtr < ePtr; cPtr++, rPtr += n)
        {   swap = *cPtr;
            *cPtr = *rPtr;
            *rPtr = swap;
        }
        ptr2 += offset;
        ePtr += n;
    }
}

PAR   doproduct(struct list *prms)
{   SVTYPE  sum;      // working value of sum
    int     row, col; // offsets for array indexing
    int     lower    =  prms->index;
    int     upper    =  prms->ceiling;
    int     step     =  prms->stride;
    SV&     matrix3  =  *(prms->host);

    for (int i = lower; i < upper; i += step)
    {   row = i*n;
        for (int j = 0; j < n; j++)
        {   col = j*n;
            sum = (SVTYPE) 0;
            for (int k = 0; k < n; k++)
                sum += matrix1[row+k] * matrix2[col+k];
            matrix3[row+j] = sum;
        }
    }
}

int main(int argc, char *argv[])
{   int         procs ;
    timeUnits   time, secs;

    if (argc == 3)
    {   n = atoi(argv[1]);
        procs = atoi(argv[2]); }
    else
    {   cerr << "Usage: " << argv[0] << " size procs\n" << "\n";
        exit(1); }

    matrix1 = new SVTYPE[n*n];
    matrix2 = new SVTYPE[n*n];

    start = start_timer();
    SV matrix3(n*n);
    secs = seconds(start);
    cerr << "Result matrix created in " << secs << " s\n";

    getmatrix(matrix1,n);
    getmatrix(matrix2,n);
    secs = seconds(start;
    cerr << "Data read in after " << secs << " s\n";

    start = elapsed_time();
    transpose(matrix2, n);
    matrix3.Clones(procs);
    matrix3.ForEachPoint(0,n-1,doproduct);
    secs = seconds(start);
    cerr << "Ending time " << secs << " s\n";

    printmatrix(matrix3,n);
}
```

```
// Julia Set program: fractalMain.C, fractal.C, fractal.h.
// fractalMain.C

#include "fractal.h"

Julia   *drawing;      // image to compute

int main(int argc, char* argv[])
{   extern      PAR   JuliaCompute(struct list *Prms);

    drawing = new Julia(argc,argv);
    drawing->Picture->DrawImage(JuliaCompute);
}
```

```
// fractal.C

#include "fractal.h"

extern "C" {
#include "timelib.h"
}

void    JuliaCompute(struct list *Prms)
{   extern      Julia   *drawing;
    (void) drawing->Compute(Prms);
}

// Constructor for FractalImage and attendant variables
FractalImage::FractalImage(int argc, char* argv[], int workers,
                                    int columns, int rows)
{   // create new drawing surface
    host = new WorkStation(argc, argv);

    // save away the dimensions of the image
    Rows = rows;
    Columns = columns;
    MaxColours = 15;
    // scale for pixels
    RowSize = MaxRows/Rows;
    ColSize = MaxCols/Columns;

    // create shared values to represent image
    Image = new SV(Rows*Columns);
    Image->Clones(workers);
}

PAR   FractalImage::Display()
{   if (RowSize == 1)
        PixelPlot();
    else
        RectPlot();
}

void    FractalImage::RectPlot()
{   int p           ;   // current position in line
    int sp          ;   // start of plateau
    int value       ;   // colour of plateau
    int row = 0     ;   // current row offset
    int y = 0       ;   // position of row on screen

    for (int rows = 0; rows < Rows; rows++)
    {   row += Columns;
        y += RowSize;
        p = 0;
        while (p < Columns)
        {   value = (*Image)[row+p];
            sp = p;
            while ((p < Columns) && ((*Image)[row+p] == value))
                ++p;
            host->Rectangle(sp*ColSize, y, p*ColSize-1,
                                y+RowSize-1, value);
        }
    }
    host->Refresh();
}
```

```
void    FractalImage::PixelPlot()
{   int value      ;   // colour of plateau
    int sp         ;   // start of plateau
    int p          ;   // current position in line
    int row = 0    ;   // current row offset

    for (int rows = 0; rows < Rows; rows++)
    {   row += Columns;
        p = 0;
        while (p < Columns)
        {   value = (*Image)[row+p];
            sp = p;
            while ((p < Columns) && ((*Image)[row+p] == value))
                ++p;
            host->Line(sp, rows, p, rows, value);
        }
    }
    host->Refresh();
}

void    FractalImage::DrawImage(void (*w)(struct list *Prms))
{   timeUnits    start  ;   // first timing point
    timeUnits    end    ;   // second timing point

    start = start_timer();
    Image->DoAllBlock(0, Columns-1, w);
    end = seconds(start);
    cout << "Computed Image in " << end << "\n" << flush;
    Display();
}

Julia::Julia(int argc, char **argv)
{   if (argc < 9)
    {   cerr << "Usage: " << argv[0] << " workers x y p q x y k\n";
        exit(1);
    }

    Picture = new FractalImage(argc, argv, atoi(argv[1]),
                                    atoi(argv[2]), atoi(argv[3]));
    sscanf(argv[4],"%F",&P);
    sscanf(argv[5],"%F",&Q);
    sscanf(argv[6],"%F",&X_Min);
    sscanf(argv[7],"%F",&Y_Min);
    MaxK = atoi(argv[8]);
    X_Max = -X_Min;
    Y_Max = -Y_Min;
    Delta_X = (X_Max - X_Min)/(Picture->Columns - 1);
    Delta_Y = (Y_Max - Y_Min)/(Picture->Rows - 1);
    M = 100;
}

PAR   Julia::Compute(struct list *Prms)
{   int i, j, k;
    double x, y, x1, y1, r, yvalue;
    int row;

    for (j = 0; j < Picture->Rows; j++)
    {   row = Picture->Columns*j;
        yvalue = Y_Min + j * Delta_Y;
        for (i = Prms->index; i < Prms->ceiling; i += Prms->stride)
        {   x = X_Min + i * Delta_X;
            y = yvalue;
            k = 0;
            do
            {   x1 = x * x - y * y + P;
                y1 = 2 * x * y + Q;
                x = x1;
                y = y1;
                k++;
                r = x * x + y * y;
            }
            while ((r <= M) && (k < MaxK));
            (*(Picture->Image))[row+i] = (r >M)?((k %
                (Picture->MaxColours - 1)) + 1):(0);
        }
    }
}
```

```
// fractal.h

#ifndef _fractal_h_
#define _fractal_h_

#include "graphics.h"

#include "sv.h"

const  int MaxRows = 480; // on screen
const  int MaxCols  = 640; // on screen

class  FractalImage
{   public:

    // constructor for image
    FractalImage(int argc, char* argv[], int w, int c, int r);

    // procedure to drive computation
    void   DrawImage(void (*workercode)(struct list *));

    SV *Image           ;   // final image
    int Rows            ;   // no. of image rows
    int Columns         ;   // no. of image columns
    int MaxColours      ;   // no. of colours
    int RowSize         ;   // length of screen point (pixels)
    int ColSize         ;   // height of screen point (pixels)

    private:

    WorkStation  *host  ;   // drawing surface
    void   PixelPlot()  ;   // drawing operation
    void   RectPlot()   ;   // drawing operation
    PAR    Display()    ;   // begin drawing
};
```

```
class  Julia
{   public:

    // constructor for fractal
    Julia(int argc, char **argv);

    // routine to compute points of fractal
    PAR   Compute(struct list * Prms);

    FractalImage *Picture  ;   // image data structure

    private:

    int        M        ;      // parameters of fractal
    double     X_Min    ;
    double     X_Max    ;
    double     Y_Min    ;
    double     Y_Max    ;
    double     Delta_X  ;
    double     Delta_Y  ;
    double     P        ;
    double     Q        ;
    int        MaxK     ;
};

#endif !_fractal_h_
```

# *Appendix C*

## Shared Value Classes

The listings presented in this section represent the program code used to implement static shared values in Tyger-C++. The files presented are:

| | | | |
|---|---|---|---|
| (i) | sv.h v(1) | - | header file for shared values used during the testing. |
| (ii) | sv.C v(1) | - | implementation file for v(1) shared values. |
| (iii) | sv.h v(2) | - | header file for more advanced shared values. |
| (iv) | sv.C v(2) | - | implementation file for v(2) shared values. |
| (v) | MultiTask.h | - | header of thread library to support shared values. |
| (vi) | MultiTask.C | - | implementation file for thread library. |
| (vii) | SVStream.h | - | an example stream class based on shared values. |
| (viii) | tStream.C | - | an example program showing the use of SVStreams. |

All of this code has been written in the form of straight C++ classes, however, subsequent implementations of Tyger-C++ may be a combination of classes and a source program preprocessor.

A similar combined approach to language implementation was the one followed in the implementation of Tyger-C, based around the Unix compiler writing utilities of *yacc* and *lex*. Although the Tyger-C files are not included here, the freely available yacc and lex files for a C grammar (around 800 lines of C) formed the basis of a preprocessor that outputed C code containing Tyger library calls to implement the parallelism. The preprocessor itself was quite small, around 500 lines of C, with the Tyger library being somewhat larger at around 1300 lines of C.

```
// sv.h v(1), single assignment, no multiple stripes

#ifndef _sv_h_
#define _sv_h_

#define   SVTYPE  int
#include "MultiTask.h"

extern    void   spin_idle(LOCK *lock);

const  int SVsvALLOC = 1;     // failed to get array of sv
const  int svBAD = 999;       // multiple write to location

class sv
{   friend ostream& operator << (ostream& outS, sv& data);
    friend istream& operator >> (istream& inS, sv& data);

    public:

        sv()   {}      // null body for constructor
        ~sv()  {}      // null body for destructor

        // overload assignment from a 'sv' to a SVTYPE
        SVTYPE& operator = (SVTYPE value)
        {   svData = value;
            if (spin_condlock(&svSync) != PAR_ACQUIRED)
                svError(svBAD);
            return svData; }

        operator SVTYPE()
        {   spin_idle(&svSync);
            return svData; }

        void*  operator new (size_t size) { return (void *) NULL; }
        void*  operator new (size_t size, sv* allocAddress);
        void   operator delete (void *ptr, size_t size) {}

    private:

        LOCK    svSync    ;      // synchronisation for 'sv'
        SVTYPE  svData    ;      // actual data value for 'sv'
        void    svError(int error_code);
};
```

```
class SV
{   friend ostream& operator << (ostream& outS, SV& data);
    friend istream& operator >> (istream& inS, SV& data);

    public:

        SV (int array_size = 1);

        sv& operator [] (int index)
        {   return SVData[index]; }

        SVTYPE& operator = (SVTYPE value)
        {   return SVData[0] = value; }

        operator SVTYPE()
        {   return SVData[0]; }

        int Clones()
        {   return SVWorkers; }

        void Clones(int n)
        {   SVWorkers = n; }

        void   DoAll(int lb, int ub, void (*worker) (struct list * ),
                   doAllCodes patt = svBlock)
        {   SVWorld->mtDoAll(lb, ub, worker, this, patt); }

        void   ForEach(int lb, int ub, void (*worker) (struct list * ),
                   forEachCodes patt = svHunk)
        {   SVWorld->mtForEach(lb, ub, worker, this, patt); }

    private:

        sv          *SVData      ;   // array of 'sv'
        int         SVDataSize   ;   // size of array
        int         SVWorkers    ;   // number of workers for SV
        MultiTask*  SVWorld      ;   // tasking environment
        void        SVError(int error_type);
};

#endif !_sv_h_
```

```
// sv.C v(1), single assignment, no multiple stripes

#include "sv.h"

// allow 'sv' to be printed out by using <<
ostream& operator << (ostream& outStream, sv& data)
{   spin_idle(&data.svSync);
    return outStream << data.svData;
}

// allow 'sv' to be read in by using >>
istream& operator >> (istream& inStream, sv& data)
{   if (spin_condlock(&(data.svSync)) != PAR_ACQUIRED)
        data.svError(svBAD);
    return inStream >> data.svData;
}

// overloaded new operator for efficient storage
void*  sv::operator new (size_t size, sv* Address)
{   Address += 1;
    Address->svSync.field[0] = (unsigned char) PAR_UNLOCKED;
    return Address;
}

void   sv::svError(int error_code)
{   switch(error_code)
    {   case svBAD : fprintf(stderr, "Multiple write to address
                        %d\n", &svData); break;
        default : fprintf(stderr, "svError: caught internal error\n");
    }
    fflush(stderr);
    exit(error_code);
}
```

```
// constructor used by a user
SV::SV(int array_size)
{   sv  *lastUsed;

    // attach tasking environment
    SVWorld = new MultiTask(array_size*sizeof(sv));

    // allocate storage for array of sv
    SVDataSize = array_size;
    if ((SVData = (sv *) malloc(sizeof(sv)*SVDataSize)) == NULL)
        SVError(SVsvALLOC);

    // initialise array of sv if necessary
    if (PAR_UNLOCKED != 0)
    {   lastUsed = SVData - 1;
        for (int i = 0; i < SVDataSize; i++)
            lastUsed = new (lastUsed) sv;
    }
}

void   SV::SVError(int error_code)
{   switch (error_code)
    {   case SVsvALLOC:fprintf(stderr, "Failed to allocate SV\n");
                                break;
        default : fprintf(stderr, "SVError: caught internal error\n");
                                break;
    }
    fflush(stderr);
    exit(error_code);
}
```

```
// sv.h v(2), multiple stripes, one level of history

#ifndef _sv_h_
#define  _sv_h_

#define  SVTYPE  int

#include "MultiTask.h"

// level of history storage
const  int svHistory = 2;

// error code for class SV, failed to get array of sv
const  int SVsvALLOC = 1;

// error code for class sv, multiple write to location
const  int svBAD = 999;

// values for svThreshold
const  int svParallel = 1;
const  int svSequential = 0;

// internal shared value class never seen by a user
class sv
{  // dec. of friend class ostream to overload stream output
   friend ostream& operator << (ostream& outStream, sv& data);

   // dec. of friend class istream to overload stream input
   friend istream& operator >> (istream& inStream, sv& data);

   public:

      sv() {}     // null body for constructor
      ~sv() {}// null body for destructor

      // overload assignment from a 'sv' to a SVTYPE
      SVTYPE& operator = (SVTYPE value)
      {  svData[svCurrent] = value;
         if (spin_condlock(&svSync) < svThreshold)
            svError(svBAD);
         return svData[svCurrent];
      }

      // explicit type conversion from a 'sv' to a SVTYPE
      operator SVTYPE()
      {  while (svSync.field[0] < svThreshold);
         return svData[svCurrent];
      }

      SVTYPE& old()
      {  return svData[svOld]; }

      // reset synchronisation tag for next stripe
      void   clear();

      // setup for parallelism stripe
      void   parallel()
      {  svThreshold = svParallel;
         svOld = svCurrent;
         svCurrent = (svCurrent + 1) % svHistory;
      }

      // setup for sequential stripe
      void   sequential()
      {  svThreshold = svSequential; }

      // overload 'new'
      void*  operator new (size_t size) { return (void *) NULL; }
      void*  operator new (size_t size, sv* allocAddress);

      // overload 'delete' operator with a null operation
      void   operator delete (void *ptr, size_t size) {}

   private:

      LOCK     svSync;              // synchronisation for 'sv'
      SVTYPE   svData[svHistory];   // actual data values for 'sv'
      static   int svCurrent;       // index to current 'sv'
      static   int svOld;           // index to last 'sv'
      static   int svThreshold;     // toggle single assignment

      void     svError(int error_code);
};
```

```
// shared value class seen and used by a user
class SV
{   // dec. of friend class ostream to overload stream output
    friend ostream& operator << (ostream& outS, SV& data)
    {   return outS << data.SVData[0]; }

    // dec. of friend class istream to overload stream input
    friend istream& operator >> (istream& inS, SV& data)
    {   return inS >> data.SVData[0]; }

    public:

        // default parameter allows implicit size of 1
        SV (int array_size = 1);

        // overload [] operator
        sv& operator [] (int index)
        {   return SVData[index]; }

        // overload assignment operator
        SVTYPE& operator = (SVTYPE value)
        {   return SVData[0] = value; }

        // overload reading by defining type conversion
        operator SVTYPE()
        {   return SVData[0]; }

        // number of workers that an iterator will use
        int Clones()
        {   return SVWorkers; }

        void    Clones(int n)
        {   SVWorkers = n; }

        // entry procedure to parallelism stripe (reenterant)
        void    beginStripe();

        // exit procedure for parallelism stripe (reenterant)
        void    endStripe();

        // iterators
        void    DoAll(int lb, int ub, void (*worker) (struct list * ),
                        doAllCodes patt = svBlock)
        {   beginStripe();
            SVWorld->mtDoAll(lb, ub, worker, this, patt);
            endStripe();
        }

        void    ForEach(int lb, int ub, void (*worker) (struct list * ),
                        forEachCodes patt = svHunk)
        {   beginStripe();
            SVWorld->mtForEach(lb, ub, worker, this, patt);
            endStripe();
        }

        // cheat routine to start stripe explicitly
        void    Start(void* worker)
        {   beginStripe();
            SVWorld->Unbound(worker, SVWorkers);
        }

        // cheat routine to end stripe explicitly
        void    End()
        {   SVWorld->Bind();
            endStripe();
        }

    private:

        sv          *SVData;        // array of 'sv'
        int         SVDataSize;     // size of array
        int         SVWorkers;      // number of workers for SV
        MultiTask*  SVWorld;        // tasking environment

        void        SVError(int error_type);
};

#endif !_sv_h_
```

```
//sv.C

#include "sv.h"

// static initialisations
int     sv::svCurrent = 0;
int     sv::svOld = 0;
int     sv::svThreshold = svSequential;

ostream& operator << (ostream& outStream, sv& data)
{   while (data.svSync.field[0] < data.svThreshold);
    return outStream << data.svData[data.svCurrent]; }

istream& operator >> (istream& inStream, sv& data)
{   if (spin_condlock(&(data.svSync)) < data.svThreshold)
        data.svError(svBAD);
    return inStream >> data.svData[data.svCurrent]; }

void    sv::clear()
{   svSync.field[0] = PAR_UNLOCKED; }

// overloaded new operator for efficient storage
void*  sv::operator new (size_t size, sv* allocAddress)
{   allocAddress += 1;
    allocAddress->clear();
    return allocAddress; }

void    sv::svError(int error_code)
{   switch(error_code)
    {   case svBAD : fprintf(stderr, "Multiple write to address
                        %d\n", &svData); break;
        default : fprintf(stderr, "svError: caught internal error\n");
    }
    fflush(stderr);
    exit(error_code);
}
```

```
// constructor used by a user
SV::SV(int array_size)
{   sv  *lastUsed;

    SVWorld = new MultiTask(array_size*sizeof(sv));
    SVDataSize = array_size;

    if ((SVData = (sv *) malloc(sizeof(sv)*SVDataSize)) == NULL)
        SVError(SVsvALLOC);

    if (PAR_UNLOCKED != 0)
    {   lastUsed = SVData - 1;
        for (int i = 0; i < SVDataSize; i++)
            lastUsed = new (lastUsed) sv;
    }
}

void    SV::beginStripe()
{   if (SVWorld->Parallel == 0)
    {   for (int i = 0; i < SVDataSize; i++)
            SVData[i].clear();
        SVData[0].parallel();
    }
}

void    SV::endStripe()
{   if (SVWorld->Parallel == 0)
        SVData[0].sequential();
}

void    SV::SVError(int error_code)
{   switch (error_code)
    {   case SVsvALLOC:fprintf(stderr, "Failed to allocate SV\n");
                        break;
        default : fprintf(stderr, "SVError: caught internal error\n");
    }
    fflush(stderr);
    exit(error_code);
}
```

```
// MultiTask.h

#ifndef _MultiTask_h_
#define _MultiTask_h_

#define   PAR   void

#include <stdlib.h>
#include <iostream.h>
#include <malloc.h>
#include <stdio.h>

// compatibility between threads libraries
#ifdef TASKS
#define   malloc                  share_malloc
#endif

// hack for Encore C compiler to generate inline spinlocks
extern "C" {
#include "parallel.h
int     cspinlock(LOCK *lock);
void    spinlock(LOCK *lock);
void    spinunlock(LOCK *lock);
}

#define   spin_condlock           cspinlock
#define   spin_lock               spinlock
#define   spin_unlock             spinunlock

// front-ends to iterators
#define   DoAllBlock(a, b, c)     DoAll(a, b, c, svBlock)
#define   DoAllStride(a, b, c)    DoAll(a, b, c, svStream)
#define   ForEachHunk(a, b, c)    ForEach(a, b, c, svHunk)
#define   ForEachPoint(a, b, c)   ForEach(a, b, c, svPoint)

// work distribution parameters
typedef   int strtng_ndx;
typedef   int clsing_ndx;
typedef   int strd_ndx;

// work allocation codes
enum  forEachCodes    { svHunk = 0, svPoint = 1, svStream = 3 };
enum  doAllCodes      { svBlock = 4, svStride = 5 };

class  SV;    // forward reference

// Structure template used to pass parameters to a master thread
struct masterStruct
{   int     start                   ;   // lower bound
    int     finish                  ;   // upper bound
    int     workers                 ;   // no of threads to use
    SV      *host                   ;   // SV to operate on
    void    (*proc)(struct list *)  ;   // worker body
    union
    {   doAllCodes      stag;
        forEachCodes    dtag;
    };
};

// structure for passing parameters to worker thread
struct list
{   strtng_ndx    index             ;   // starting point (inclusive)
    clsing_ndx    ceiling           ;   // ending point (exclusive)
    strd_ndx      stride            ;   // stride between values
    SV            *host             ;   // calling variable
    void          **args            ;   // additional arguments
};

class MultiTask
{   friend class SV;
    private:

        MultiTask(int valueSize);

        void    Task(void (*w), int w, struct masterStruct* prms);

        void    Unbound(void (*w), int nw);

        void    Bind();
```

```
        void    mtForEach(int lb, int ub, void (*w)(struct list *),
                    SV *host, forEachCodes patt = svHunk);

        void    mtDoAll(int lb, int ub, void (*w)(struct list *),
                    SV *host, doAllCodes patt = svBlock);

    private:

        static int DataSize     ;   // total memory used
        static int Operational  ;   // tasking status
        static int Parallel     ;   // stripe status

        void    CleanUp(char *message);
};

// amount of system and spare shared memory for tasking env.
extern    int     UNIVERSE_SIZE;

// macro to allocate shared memory and create tasking env.
#define SHARE(X)  (void) new MultiTask(UNIVERSE_SIZE = X);

#endif !_MultiTask_h_
```

```
// MultiTask.C

#include "MultiTask.h"
#include "sv.h"

// static member initialisations
int     UNIVERSE_SIZE         = 5*1024*1024;
int     MultiTask::Operational = 0;
int     MultiTask::Parallel    = 0;

// Implementation declarations
const  int mStack             = 1024*20;
const  int wStack             = 1024*20;
#ifdef TASKS
// determines the size of an object in 4-byte words
#define   nw(x)  (int) sizeof(x)/sizeof(int) + 1
const  int mtError            = -1;
const  int goFailed           = -1;
#else
extern "C" {
#include "thread.h"
}
const  int tP                 = 2;
#endif

// *******************************************************
// Non member functions of class MultiTask used to implement
// the parallelism - no implicit class data parameter to pass

// tidyThreads - non member function called to shut down threads
void   tidyThreads(char *message)
{   cout << flush;
    cerr << "MultiTask: " << message << "\n" << flush;
#ifdef TASKS
    task_stop();
#else
    THREADkill(THREADcurrent);
#endif
}
```

```
// staticMaster - Driver thread for the static work distribution.
static void   staticMaster(struct masterStruct *mstrPrms)
{   int     wrkrs     = mstrPrms->workers;
    int     top       = mstrPrms->start;
    int     ld        = (mstrPrms->finish - top + 1) / wrkrs;
    int     xcss      = (mstrPrms->finish - top + 1) % wrkrs;
    SV      *hst      = mstrPrms->host;
    int     nwthrds   = wrkrs - 1;
    void    (*proc)(struct list *) = mstrPrms->proc;

    // create an array of parameter structures for children
    struct list *plist;
#ifdef TASKS
    if ((plist = (list *) malloc(sizeof(list)*wrkrs)) == NULL)
        tidyThreads("Failed to allocate static work list");
#else
    plist = new struct list[wrkrs];
#endif

    // initialise parameter array and fork children
    for (int tc = 0; tc < wrkrs; tc++)
    {   switch(mstrPrms->stag)
        {   case svBlock :
            {   plist[tc].index = top;
                plist[tc].ceiling = top += ((xcss > tc) ? (ld + 1 ) : (ld));
                plist[tc].stride = 1;
                plist[tc].host = hst;
                break;
            }
            case svStride :
            {   plist[tc].index = tc + top;
                plist[tc].ceiling = mstrPrms->finish+1;
                plist[tc].stride = wrkrs;
                plist[tc].host = hst;
                break;
            }
            default: tidyThreads("error in work specifier"); break;
        };
```

```
        if (tc < nwthrds )
        {
#ifdef TASKS
            if ((task_start(proc, wStack, 1, &plist[tc])) == NULL)
#else
            if (THREADcreate(proc, &plist[tc], 0, ATTACHED,
                wStack, tP) == NULL)
#endif
            {   cerr << "MultiTask: thread " << tc;
                tidyThreads("failed to start thread");
            }
        }
        else // master allocation
            (*proc)(&plist[tc]);
    }

    // wait until children have terminated
#ifdef TASKS
    task_join();
#else
    THREAD handle;
    do
    {   handle = THREADjoin(); }
    while (handle != NULL && THREADfree(handle));
#endif
}

// structure used to pass information to hunk processing workers
// lock must be declared as the first field for rapid access
struct wBoard
{   LOCK lock                   ;   // lock field for wBoard
    int     workload            ;   // work remaining
    int     start               ;   // next item to process
    SV      *host               ;   // host variable
    int     workers             ;   // no of workers
    void    (*proc)(struct list *)  ;   // worker code
};

// structure used to pass information to point processing workers
// lock must be declared as the first field for rapid access
struct bBoard
{   LOCK lock                   ;   // lock field for bBoard
    int     head                ;   // first item to process
    int     tail                ;   // last item to process
    int     top                 ;   // next item to process
    SV      *host               ;   // host variable
    int     workers             ;   // no of workers
    int     id                  ;   // worker id
    void    (*proc)(struct list *)  ;   // worker code
};

PAR   hunkDriver(struct wBoard *Brd)
{   struct list mylst;

    mylst.stride = 1;
    mylst.host = Brd->host;
    do
    {   spin_lock((LOCK *) Brd);
        if (Brd->workload <= 0)
        {   spin_unlock((LOCK *) Brd);
            break;
        }
        else
        {   mylst.index = Brd->start;
            if ((mylst.ceiling = Brd->workload/Brd->workers) == 0)
                mylst.ceiling = 1;
            Brd->workload -= mylst.ceiling;
            mylst.ceiling += mylst.index;
            Brd->start = mylst.ceiling;
            spin_unlock((LOCK *) Brd);
            (*(Brd->proc))(&mylst);
        }
    }
    while (1);
}
```

```
PAR   streamDriver(struct wBoard *Brd)
{   struct list mylst;

    mylst.stride = 1;
    mylst.host = Brd->host;
    do
    {   spin_lock((LOCK *) Brd);
        if (Brd->start >= Brd->workload)
        {   spin_unlock((LOCK *) Brd);
            break;
        }
        else
        {   mylst.index = Brd->start;
            Brd->start += 1;
            spin_unlock((LOCK *) Brd);
            mylst.ceiling = mylst.index + 1;
            (*(Brd->proc))(&mylst);
        }
    }
    while (1);
}

// Function for allocation of points -  it is inlined for speed
inline  int pointAllocate(struct bBoard* Brd, struct list* prms)
{   spin_lock((LOCK *) Brd);
    if (Brd->top < Brd->tail)
    {   prms->index = Brd->top;
        Brd->top += 1;
        spin_unlock((LOCK *) Brd);
        prms->ceiling = prms->index+1;
        return 1;
    }
    else
    {   spin_unlock((LOCK *) Brd);
        return 0;
    }
}
```

```
static PAR   pointDriver(struct bBoard *Brd)
{   int     nid = Brd->id;
    struct list mylst;

    mylst.stride = 1;
    mylst.host = Brd->host;
    do
    {   if (pointAllocate(Brd, &mylst) != 0 )
            (*(Brd->proc))(&mylst);
        else
            break;
    }
    while (1);

    do
    {   nid = (nid + 1) % (Brd->workers);
        while (pointAllocate(Brd + (nid - Brd->id), &mylst) != 0)
            (*(Brd->proc))(&(mylst));
    }
    while (nid != blckBrd->id);
}

static PAR   dynamicMaster(struct masterStruct *Prms)
{   // copy parameters out of masterStruct
    int     wks         =   Prms->workers;
    int     top         =   Prms->start;
    int     wrk         =   (Prms->finish - top + 1);
    int     ld          =   wrk / wks;
    int     xcss        =   wrk % wks;
    int     nwthrds     =   wks - 1;
    SV      *hst        =   Prms->host;
    void    (*proc)(struct wBoard *);

    if (Prms->dtag == svPoint)
    {   // create an array of parameter structures for children
        struct bBoard *BBrd;
#ifdef TASKS
    if ((BBrd = (bBoard *) malloc(sizeof(bBoard)* wks)) == NULL)
        tidyThreads("Failed to allocate bBoard");
```

```
#else
        BBrd = new struct bBoard[wks];
#endif
        // initialise parameter array and fork children
        for (int tc = 0; tc < wks; tc++)
        {
            BBrd[tc].top = BBrd[tc].head = top;
            BBrd[tc].tail = top += ((xcss > tc) ? ( ld + 1 ) : ( ld ));
#ifdef TASKS
        if (spin_init(&(BBrd[tc].lock),PAR_UNLOCKED) == NULL)
            tidyThreads("failed to initialise lock");
#endif

            BBrd[tc].proc = Prms->proc;
            BBrd[tc].id = tc;
            BBrd[tc].workers = wks;
            BBrd[tc].host = hst;

            if (tc < nwthrds)
            {
#ifdef TASKS
        if ((task_start(pointDriver, wStack, 1,&BBrd[tc])) == NULL)
#else
        if (THREADcreate(pointDriver, &(BBrd[tc]),0, ATTACHED,
            wStack, tP) == NULL)
#endif
                {   cerr << "MultiTask: thread " << tc;
                    tidyThreads("failed to start thread");
                }
            }
            else // master allocation
                pointDriver(&BBrd[tc]);
            }

// wait until children have terminated and delete params
#ifdef TASKS
        task_join();
        share_free(BBrd);
#else
        while (THREADjoin());
        delete [wks] BBrd;
#endif
    }
    else
    if (Prms->dtag == svHunk || Prms->dtag == svStream)
    {
#ifdef TASKS
        struct wBoard    *WBrd;
    if ((WBrd = (wBoard *) malloc(sizeof(struct wBoard)))==NULL)
            tidyThreads("Failed to allocate wBoard");

        if (spin_init(&(WBrd->lock), PAR_UNLOCKED) == NULL)
            tidyThreads("failed to create lock");
#else
        struct wBoard* WBrd = new wBoard;
#endif

        WBrd->proc = Prms->proc;
        WBrd->start = top;
        WBrd->host = hst;
        WBrd->workers = wks;

        if (Prms->dtag == svHunk)
        {   WBrd->workload = wrk;
            proc = hunkDriver;
        }
        else    // Prms->dtag == svStream
        {   WBrd->workload = Prms->finish+1;
            proc = streamDriver;
        }

        for (int tc = 0; tc < nwthrds; tc++)
#ifdef TASKS
            if ((task_start(proc, wStack, 1, WBrd)) == NULL)
#else
            if (THREADcreate(proc, WBrd, 0, ATTACHED, wStack, \
```

```
                tP) == NULL)
#endif
        {   cerr << "MultiTask: thread " << tc;
            tidyThreads("failed to start thread");
        }

    // do own allocation of work
    (*proc)(WBrd);

#ifdef TASKS
    // wait until children have terminated
    task_join();
    share_free(WBrd);
#else
    while (THREADjoin());
    delete WBrd;
#endif
  }
  else
    tidyThreads("dynamicMaster invalid work specifier");
}

// ********************************************************
// Member functions of class MultiTask

// Constructor for class MultiTask
MultiTask::MultiTask(int valueSize)
{  if (!Operational)
   {  DataSize = UNIVERSE_SIZE + valueSize;
#ifdef TASKS
    if (malloc_init(DataSize) == -1)
      CleanUp("Failed to allocate shared memory");
#endif
    Operational = 1;
   }
}

#ifdef TASKS
// Procedure Cleanup the forced exit function
void  MultiTask::CleanUp(char *message)
{  cout << flush;
   cerr << "MultiTask: " << message << "\n" << flush;
   exit(mtError);
}
#endif

void  MultiTask::Task(void (*w), int  p, struct masterStruct* ps)
{  if (Parallel == 0)
   {  Parallel = 1;
#ifdef TASKS
    if ((task_init(DataSize, p, w, mStack, nw(ps), ps))
        == goFailed)
      CleanUp("Failed to create dynamic master thread");
#else
    THREADgo(p, DataSize, w, ps, sizeof(ps), mStack, tP);
#endif
    Parallel = 0;
   }
   else
   {
#ifdef TASKS
    Parallel += 1;
    if (ps == NULL)
    {
      if (task_start(w, wStack, 0) == NULL)
        tidyThreads("failed to start sole thread\n");
    }
    else
      if (task_start(w, wStack, nw(ps), ps) == NULL)
#else
      if (THREADcreate(w, ps, 0, ATTACHED, wStack, tP)
        == NULL)
#endif
    tidyThreads("failed to start sole thread\n");
```

```
    }
}

void    MultiTask::Unbound(void (*workercode), int w)
{   Task(workercode, w+1, NULL); }

void    MultiTask::Bind()
{
#ifdef TASKS
        task_join();
#else
        while(THREAD_join());
#endif
}

// Procedure Foreach the dynamic paritioning procedure
void    MultiTask::mtForEach(int lb, int ub, void
(*workercode)(struct list *),  SV *host, forEachCodes patt)
{   // Allocated on stack but no problems should arise - not shared
    struct masterStruct mstrStrct;

    mstrStrct.start      = lb;
    mstrStrct.finish     = ub;
    mstrStrct.workers    = host->Clones();
    mstrStrct.proc       = workercode;
    mstrStrct.host       = host;
    mstrStrct.dtag       = patt;

    Task(dynamicMaster, host->Clones(), &mstrStrct);
}
```

```
// Procedure DoAll the static partitioning procedure
void    MultiTask::mtDoAll(int lb, int ub, void
(*workercode)(struct list *),  SV *host, doAllCodes patt)
{   // Allocated on stack but no problems should arise - not shared
    struct masterStruct mstrStrct;

    mstrStrct.start      = lb;
    mstrStrct.finish     = ub;
    mstrStrct.workers    = host->Clones();
    mstrStrct.proc       = workercode;
    mstrStrct.host       = host;
    mstrStrct.stag       = patt;

    Task(staticMaster, host->Clones(), &mstrStrct);
}
```

```
// SVStream.h, implements single writer multiple reader

#include "sv.h"

// actual data elements in stream
class SVStreamElement
{   public:

        SVStreamElement()
        { next = (SVStreamElement *) NULL; }

        ~SVStreamElement() {}

        SVStreamElement*  next   ;
        SV                data   ;
};

// method by which a user can read stream elements
class SVSHandle
{   // overload << operator to allow 'cout' to work
    friend ostream& operator << (ostream& outS, SVSHandle& s)
    {   s.current = s.ptr;
        while (s.ptr == (SVStreamElement*) NULL);
        s.ptr = s.ptr->next;
        return outS << s.current->data;
    }

    public:

        SVSHandle(SVStreamElement* head)
        { ptr = head; }

        ~SVSHandle() {}

        // type conversion to SVTYPE (reading operation)
        operator SVTYPE()
        {   current = ptr;
            while (ptr == (SVStreamElement*) NULL);
            ptr = ptr->next;
            return current->data;
        }

    private:
        SVStreamElement*  ptr;
        SVStreamElement*  current;
};

// stream class used by a user
class SVStream
{   // overload >> to allow 'cin' to work
    friend istream& operator >> (istream& inS, SVStream& s)
    {   s.tmp = s.tail;
        s.tail = s.tail->next;
        s.tail = new SVStreamElement();
        s.tmp->next = s.tail;
        return inS >> s.tmp->data;
    }

    public:
        SVStream()
        {   head = new SVStreamElement();
            tail = head;
        }

        ~SVStream()
        {   while (head != (SVStreamElement *) NULL)
            {   tmp = head;
                head = head->next;
                delete tmp;
            }
        }

        // overload assignment operation
        SV& operator = (SVTYPE value)
        {   tmp = tail;
            tail = tail->next;
            tail = new SVStreamElement();
            tmp->next = tail;
            tmp->data = value;
            return tmp->data;
        }
```

```
        // type conversion to SVSHandle (reading operation)
        operator SVSHandle()
        { return *(new SVSHandle(head)); }

    private:

        SVStreamElement*  head;
        SVStreamElement*  tail;
        SVStreamElement*  tmp;
};
```

---

```
// tStream.C - sample program using SVStream

#include "SVStream.h"

int main()
{   int           n          = 10   ;
    SVStream      X                 ;
    SVSHandle     XValues    = X    ;

    for (int i = 0; i < n; i++)
        cin >> X;

    for (int j = 0; j < n; j++)
        cout << XValues << " " ;
    cout << "\n";
}
```