THE UNIVERSITY OF NEWCASTLE UPON TYNE

DEPARTMENT OF COMPUTING SCIENCE

A Graphical System for Parallel Software Development

by

Robert Stephen Allen

PhD Thesis

25 March 1998

Abstract

Parallel architectures have become popular in the race to meet an increasing demand for computational power. While the benefits of parallel computing - the performance improvements due to simultaneous computations - are clear, achieving these benefits has proved difficult. The wide variety of parallel architectures has led to a similarly diverse range of parallel languages and methods for parallel programming, many of which feature complicated architecture-specific language mechanisms. The lack of good tools to assist in the development of parallel software has compounded the problem of parallel programming being limited to a field which is both specialist and fragmented.

This thesis investigates techniques for the graphical specification of parallel programs, using an architecture-independent graph-based notation representing the design of the program, combined with conventional sequential languages. Automatic code generation is used to translate the graph program into executable code suitable for different parallel architectures. To overcome the differing performance characteristics of parallel architectures, methods for the graphical adjustment of granularity are proposed and investigated, and an encompassing parallel design environment is presented.

Acknowledgements/Dedication

I would like to express my gratitude to a number of people who have provided support over the period of this work. Firstly my supervisor, Professor Peter Lee, for his continuous encouragement and guidance. Thanks also to Dr Chris Philips and Dr Paul Watson for their constructive input, and to Alex Semenov, Martin Hesketh, and others in the department for their help and support. Especial thanks to my parents, sister Jenny and the rest of the family for their love, care and unswerving belief in me through difficult times, and to all my friends old and new for their support, both moral and otherwise. A special mention must go to Nick McNamara for providing refuge, guidance and understanding, and to Jon Mace, Celia Donaghue, Nick Barnett and Louise Alder for sharing many good times and some sad times.

This work was funded for three years by the Engineering and Physical Sciences Research Council.

I would like to dedicate this thesis to the memory of Jon Mace, friend, flatmate and fellow PhD student, whose dedication to his work was an example I could never match. Without Jon's friendship and support this work might never have reached its conclusion, and I am grateful to him for helping me to find a way forward. Jon tragically died before completing his own work, but his memory lives on in the hearts and minds of all those who knew him.

Table of Contents

Abstract
Acknowledgements/Dedication
Table of Contents
Table of Figures
1. Introduction ... 1
1.1 Parallel Computing ... 1
1.2 Current Solutions to Parallel Programming ... 3
1.3 Dataflow ... 4
1.4 Graphical Programming ... 4
1.5 Introductory Work - MeDaL ... 5
1.6 Research Focus ... 6
2. Parallel Programming ... 9
2.1 The Nature of Parallelism ... 9
2.2 Types of Parallelism ... 10
2.3 Parallel Architectures ... 11
2.4 Techniques for Programming Parallel Architectures ... 13
2.5 Current Methods for Exploiting Parallelism ... 16
2.6 Problems with Existing Parallel Programming Methods ... 19
2.7 Novel Methods for Parallelism ... 21
2.8 MeDaL - Medium-Grained Dataflow Language ... 30
2.9 Parallel Programming Summary ... 34
3. Design Notations for Parallel Software ... 35
3.1 Introduction ... 35
3.2 Motivation ... 35
3.3 Different Notational Approaches ... 36
3.4 Initial Proposal of a Notation ... 38
3.5 Changes to MeDaL Notation ... 38
3.6 MeDaL Problems ... 43
3.7 Solutions to MeDaL Problems ... 47
3.8 Graphical Programming Environment ... 54
3.9 Summary of Design Issues ... 65
4. Problems and Solutions in Achieving Efficient Performance ... 67
4.1 Performance Features of Graphical Notations ... 67
4.2 Code Generation ... 69
4.3 Target Architectures ... 73
4.4 Performance Adjustment ... 74
4.5 Summary of Performance Issues ... 78
5. Results ... 79
5.1 Objectives ... 79
5.2 Implementation Environment ... 80
5.3 Measuring Performance ... 82
5.4 Experiment 1 - Matrix Multiplication with Data Partitioning ... 82
5.5 Experiment 2 - Multiple Matrix Calculation (Encore Multimax) ... 88
5.6 Experiment 3 - Multiple Matrix Calculation (HP Network) ... 101
5.7 Experiment 4 - LU Decomposition Notation ... 106
5.8 Results Summary ... 115

6. Conclusions 117
6.1 Overview 117
6.2 Results 118
6.3 Future Work 120
6.4 Closing Remarks 122
Appendix A Listings for Data Partitioning Example
Appendix B Listings for Actor Folding Example

Table of Figures

Figure 1-1 Shuffle Sort ... 6
Figure 2-1 Data Parallelism ... 11
Figure 2-2 Task Parallelism ... 11
Figure 2-3 Shared Memory ... 13
Figure 2-4 Distributed Memory ... 13
Figure 2-5 Data Partitioning ... 15
Figure 2-6 Data Reference Patterns ... 16
Figure 2-7 Dataflow Graph ... 22
Figure 2-8 Hence Node Icons ... 28
Figure 2-9 Hence Computation Node with Annotation ... 28
Figure 2-11 Computation Node in CODE with Annotations ... 28
Figure 2-10 Node Icons in CODE ... 28
Figure 2-12 E-type Datapath and F-type Datapath ... 31
Figure 2-13 General-purpose Actors ... 31
Figure 2-14 Source Actor and Sink Actor ... 32
Figure 2-15 Merge Actor and Replicator Actor ... 32
Figure 2-16 Company ... 32
Figure 2-17 Envisaged Programming System for MeDaL ... 33
Figure 3-2 Node Icons in CODE ... 37
Figure 3-1 Hence Node Icons ... 37
Figure 3-3 Depth Actor ... 39
Figure 3-4 Library Actors ... 40
Figure 3-5 Merge Node ... 40
Figure 3-6 Input Interface Node and Output Interface Node ... 41
Figure 3-7 Top-level Graph ... 42
Figure 3-8 Decomposition of an Actor into a Subgraph ... 43
Figure 3-9 Method Code using MeDaL Commands ... 44
Figure 3-10 Bitonic Merge Sort ... 46
Figure 3-11 Loop Actor ... 50
Figure 3-12 Sub-graph for Loop Actor ... 52
Figure 3-13 Loop Input ... 53
Figure 3-14 Loop Output at Termination ... 53
Figure 3-15 Loop Output Every Iteration ... 53
Figure 3-16 Graphical User Interface ... 55
Figure 3-17 Method Actor Attribute Form ... 56
Figure 3-18 Merge Node Attribute Form ... 56
Figure 3-19 Datapath Form ... 57
Figure 3-20 Method Code Window ... 58
Figure 3-21 Global Data Window ... 59
Figure 3-22 Template Selection for Data Partitioning ... 61
Figure 3-23 Alignment of Templates for Distribution ... 62
Figure 3-24 Method Code Using Partitioned Data ... 63
Figure 3-25 Template Patterns for Heat Flow ... 64
Figure 4-1 Translation Processes ... 70
Figure 4-2 Example Graph for Translation ... 72
Figure 4-3 Folding a Single Group of Actors ... 75

Figure 4-4 Partitioning of Data for Granularity Adjustment 77
Figure 5-1 Matrix Multiplication Graph.. 84
Figure 5-2 Matrix Multiplication Speedup (Shared Memory).................................... 85
Figure 5-3 Speedup Relative to Pure Sequential (Shared Memory)........................... 86
Figure 5-4 Matrix Multiplication Speedup (Network) 87
Figure 5-5 Speedup Relative to Pure Sequential (Network)....................................... 87
Figure 5-6 Matrix Calculation Program... 89
Figure 5-7 fold4 Folded Graph 91
Figure 5-8 fold5 Folded Graph 93
Figure 5-9 fold2 Folded Graph 94
Figure 5-10 Execution Time Graphs (Shared Memory) 96
Figure 5-11 Speedup for Different Folding Strategies (Shared Memory) 98
Figure 5-12 Speedup Relative to Pure Sequential (Shared Memory)....................... 100
Figure 5-13 Execution Time Graphs (Network)... 102
Figure 5-14 Speedup for Different Folding Strategies (Network)........................... 104
Figure 5-15 Speedup Relative to Pure Sequential (Network).................................. 105
Figure 5-16 Top-Level LU Decomposition Graph 108
Figure 5-17 Second-Level LU Decomposition Graph.. 109
Figure 5-18 LU Decomposition Level 3 111
Figure 5-19 Method Code for L Calculation 113
Figure 5-20 Method Code for U Calculation... 113
Figure 5-21 Speedup for LU Decomposition... 114

1. Introduction

1.1 Parallel Computing

Computing services a rapidly expanding demand for computational power over a diverse range of applications and user needs, from home entertainment to complex scientific experiments. The latter are an example of a group of problems which often require huge amounts of processing power, above the capabilities of a single computer. While the performance of individual processors continues to increase, technological limitations seem likely to restrict this increase in the future.

In response to the failure of single processor computers to satisfy user demands for power for certain computationally intensive problems, the field of parallel computing has developed. Parallel computing involves the simultaneous use of multiple processors to achieve a common goal, and it is the difficulty of programming multiple processors to operate in this manner - parallel programming - which forms the main motivation behind this thesis. In particular, the lack of good tools for developing parallel software is addressed.

There are many factors which make parallel programming difficult, not least that programming a single computer is itself a difficult task. Additional complexities introduced by the use of multiple processors include the problems of co-ordinating the multiple processors to exchange relevant pieces of data (**communication**) at an appropriate point in time (**synchronisation**). The difficulty of programming communication and synchronisation into software is compounded by the variety of ways in which they may be achieved on different parallel computers. The architectures of parallel machines have developed in a variety of directions, leading to a wide range of parallel computers, with significantly differing characteristics. Each has developed a style of programming suitable for efficient execution on the underlying hardware, and this is particularly crucial for programming communication methods. A common consequence of inadequate communication design is poor execution efficiency, and this consequence is highly significant when considering portable software - software which will execute efficiently on multiple architectures. Failure to achieve coherent and consistent communication could also lead to a failure of the computation, due to incorrect results or disruption of the processing, but efficiency is the main focus of this work.

The issue of efficient execution on parallel architectures is usually measured by the **speedup**. This is the factor by which performance increases over that of a single processor, and initial expectations would be that the speedup would directly related to the number of processors, for example a four processor machine would be expected to perform four times faster than a single processor machine. In practice however, this is not the case as theoretical speedup is restricted, as predicted by Amdahl's law [1] which states that sequential sections in a parallel algorithm will limit achievable speedup. In addition to theoretical limitations, practical factors which limit the speedup of a program running on a parallel machine include the overheads of communication and synchronisation - if one processor is kept waiting for some piece of data necessary to the calculation, but which has not yet been produced as a result from a previous calculation, or perhaps is delayed en route. These overheads may be

problems of the hardware architecture (slow communication routes), of the style of programming (transmitting data which could be processed locally), or of the algorithm (natural data dependencies - see Chapter 2). These problems are all connected in some way, as the style of programming could be adjusted to compensate for hardware inefficiencies, or the algorithm could be rewritten to take account of programming styles. Making adjustments in this manner however, is a dangerous approach, as it reinforces machine dependence - preventing efficient execution on a different parallel machine.

Machine-specific languages for parallel architectures are widely used, with some commonality and even portability between machines of similar types. However, across significantly different parallel machines, programs written for parallel execution are unlikely to execute efficiently, if supported at all. There is a need to provide software design and development tools at a higher abstraction level, above that requiring machine-specific mechanisms, which can produce efficient code for a wide range of parallel architectures. The lack of abstract techniques for developing parallel software can be seen to be similar to the so-called software crisis of the 1970s. At that time, the computing community realised that a lack of good tools for (sequential) software development was severely restricting programmer productivity. The low-level nature of programming techniques produced code which was difficult to read, debug and maintain, and was not easily portable. High-Level Languages (HLLs) emerged as a standard programming technique for general-purpose programming, offering an abstraction of the machine-specific details, and providing support for structuring programs such as loops, selection and functions. This structure helped bring the programming language nearer to the cognitive level of the programmer, and offered a more problem-oriented approach to programming. More recent developments, incorporating tools for design as well as implementation, include CASE (Computer Aided Software Engineering).

Parallel languages today have problems similar to those experienced in the software crisis for sequential languages. In parallel programming, the features of program structure are generally inherited from sequential languages, but the way in which communication and synchronisation are specified in a parallel program is often machine specific, or at least influenced by the underlying hardware. Parallel programs can be considered as a combination of sequential sections of code, or even complete sequential programs, where some of these sequential sections are able to execute simultaneously. Sequential languages (written for single processor machines) can adequately describe the computation of any particular section of sequential code. However, what sequential languages do not address (and do not need to in a sequential style of programming) is the co-ordination of these sections, i.e. the structure, ordering, and co-operation between sections, which is implicit in sequential programs. The significance of the co-ordination aspect in parallel programs is that it encapsulates the parallelism, and is the key target for abstraction to help move towards machine independence, and away from the current methods used in parallel programming. The added complexities of parallel programs, such as co-ordination and communication lead to an even greater need for design and development tools.

1.2 Current Solutions to Parallel Programming

Methods for parallel programming are many and varied, but fall into three main groups:

1. auto-parallelising compiler techniques,
2. new parallel languages,
3. extensions to sequential languages.

These will be considered in turn and expanded on in Chapter 2. The new parallel languages considered here are those which can be considered to have evolved from traditional sequential languages - the procedural or **imperative** style which are based around a sequence of commands to instruct the computer *how* to carry out the computation. **Declarative** languages, which concentrate on *what* the purpose of a program is, rather than *how* it is carried out, will be considered separately in the next section.

1. **Auto-parallelising compilers**

   One popular research area is into compiler techniques which can detect potential parallelism in sequential code. This frees the programmer from using explicit parallel constructs, as the compiler automatically generates the executable code for the target parallel architecture. The difficulty with this method is the limited extent to which a compiler can detect implicit parallelism in sequential code - often relying on the use of a specialised programming style or user-added compiler directives to give hints about parallelism. The resulting executable code usually offers far from optimal speedup, and cannot match the performance of specially designed explicitly parallel languages.

2. **New parallel languages**

   Languages designed specifically for parallel architectures generally require the explicit use of program statements to describe the communication and synchronisation of parallel sections of code. In Ada for example, this involves the use of a rendezvous - a point at which the co-operating processes meet to exchange data. Describing such parallel interactions is a difficult technique which programmers must master in order to become proficient in writing parallel software. Apart from the additional complexity, explicitly parallel languages tend to reflect the type of machine architecture they are used on, for example, Occam uses a message-passing approach and is designed for use on transputer arrays which are also based on a data passing model. Parallel languages written specifically for a certain type of parallel machine can exhibit efficient execution (achieve good speedup) on that machine if designed appropriately, but are unlikely to be efficient on a different type of architecture.

3. **Extensions to existing languages**

   The third approach to producing parallel programs is by adapting existing sequential languages. New language statements are introduced which extend the instruction set to cover parallel execution. For example, parallel extensions

to Fortran for shared memory environments use parbegin and parend around a section of code for parallel execution, or doall for parallel loop behaviour. Newer parallel models, including PVM and Linda, operate in a virtual parallel architecture, that is, one which is an abstraction of the actual architecture. These newer models will be examined in more detail later, but for now are considered alongside other language extension approaches as they provide library routines or parallel primitives which coexist with a host language. Whilst all these extension approaches give the programmer the flexibility to explicitly specify parallel execution, existing features in the host sequential language remain. The extent to which new skills are required is reduced, but with a corresponding uncomfortable union between language features which inherently conflict in a parallel environment. The program becomes difficult to read in terms of its parallelism - any natural parallelism obvious at the design stage is likely to become somewhat concealed within a sequential textual description.

The hidden parallelism indicated in the language extension approach is a key issue in the examination of parallel software tools in this thesis - the way in which parallelism is represented, and the features available to the user to specify parallel execution. Abstraction is the mechanism used later to exploit parallelism effectively whilst concealing complicated implementations. The theme of abstraction underpins many of the topics introduced here and is expanded on in Chapter 2. However, two abstractions are now considered: firstly *dataflow*, which addresses the semantic issues of a parallel specification - the features a user can exploit; and secondly, *graphical programming*, which is more concerned with the representation of parallelism.

1.3 Dataflow

Dataflow, belonging to the declarative style of languages, involves a computation controlled by the arrival of necessary data, and not dictated by any centralised flow of control. The dataflow style can be considered to be more abstract than imperative parallel languages as it evolves directly from the specification of the problem and is not forced into an unnatural form such as a single ordered set of instructions. Dataflow retains an implicit description of the parallelism, defined by data dependencies (see Chapter 2), which does not become hidden in the programming style as it may do so with other approaches.

While dataflow is traditionally associated with the flow of single data elements between simple operations, the same concepts can be applied at different levels of abstraction in a program, perhaps defining the relationships between sections of code such as functions or procedures. Using dataflow at a higher level avoids some problems associated with the excessive parallelism exposed by lower-level data dependencies (examined in Chapter 2) and provides a more suitable environment for the integration of dataflow concepts with other techniques such as graphical programming.

1.4 Graphical Programming

Graphical or visual programming is an unconventional style of programming using graphical objects to specify the computation and data relationships rather than text.

The advantage of this approach, when used in parallel software tools, is that parallelism is implicit within the structure of the connections between objects. However, one of the drawbacks of using graphics, especially when applied at a low level to sequential algorithms, is that it can often lead to a reduction in programmer understanding and an increase in complexity.

Using graphical objects to specify the *computation* is a difficult task, as programmers are trained to use sequential text and, at a program statement level, representation of a language construct by a visual icon is a difficult cognitive step for the programmer. Furthermore, many well established program components have no directly corresponding pictorial image, particularly where cultural boundaries are crossed.

As a technique for specifying the *co-ordination* model of a parallel program - encompassing the parallel structure and hence data dependencies - graphical programming is much more suitable. In many cases, diagrams are already used at a design stage to describe data relationships and the necessary ordering of program sections. Current programming practices usually cause this problem-oriented description to be forced into a sequential textual description. A more natural progression, adapted by some hybrid graphical/textual programming languages, tries to preserve the diagrammatic design (usually a graph), encompassing the parallel structure, and use text to refine the nodes of the graph. In this way, the precise and analytic properties of text are used to specify only the computation of the program section represented by the graph node. One such hybrid approach for parallel programming was MeDaL.

1.5 Introductory Work - MeDaL

MeDaL (**Me**dium Grained **Da**taflow **L**anguage) developed at Newcastle [2] used a graphical co-ordination model and a textual computation model. The graphical notation, based on the dataflow paradigm, was combined with a conventional programming language (C) to provide a program built up out of an interconnected set of sequential code segments related by data dependency. The graph structure specified the parallelism, with no requirement for explicit language constructs, and the graph and text combination was intended to be converted to executable parallel code by a translator, and supported by a run-time system unique to the shared-memory machine architecture.

The graphical part of the notation was based around a directed graph consisting of nodes which represented processing tasks, and arcs which represented the data dependencies between the tasks. The programmer inserted sequential code segments into the nodes to specify the computation behaviour of that task. The code segments for MeDaL corresponded to procedures or functions and were intended to approach parallel software design from a medium-grained level (explained further in Chapters 2 and 3). The graph structure, constructed from data relationships between nodes, held the crucial parallelism information, which could be extracted from the graph and translated into language constructs to implement the parallel behaviour. Figure 1-1 shows a MeDaL example describing a shuffle sort. The notation will be explained in detail in later chapters, but a significant point to note is that the node with the thick border has multiple instantiations, allowing a number of compare/exchanges to be carried out simultaneously.

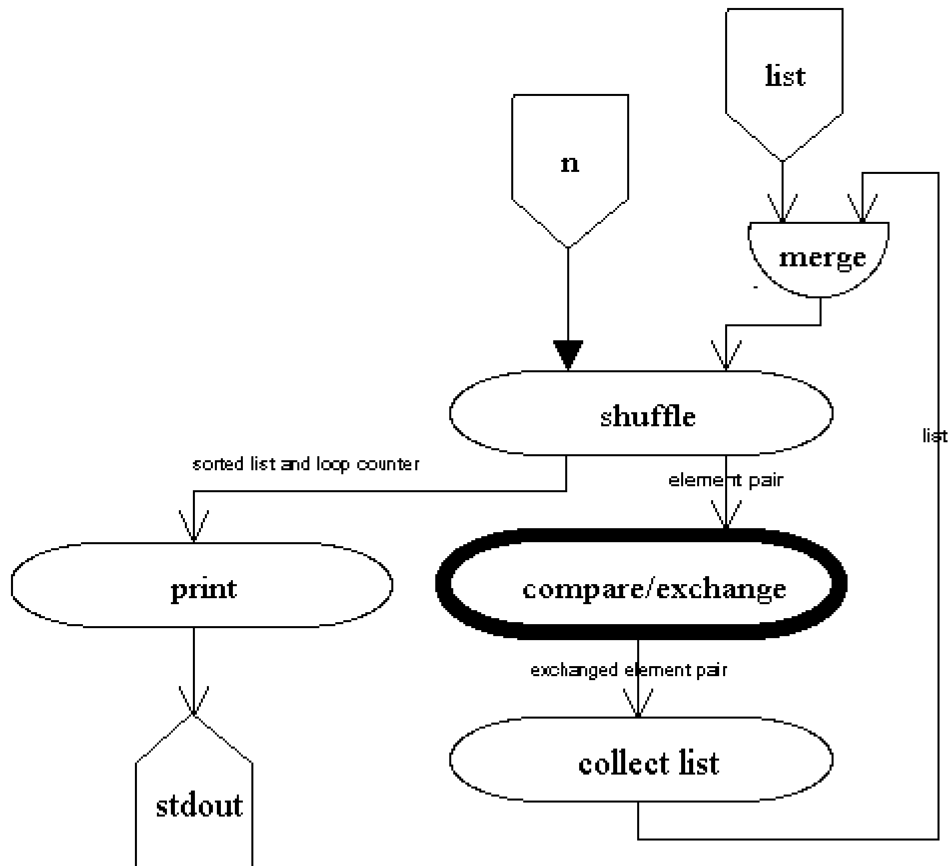


Figure 1-1 Shuffle Sort

Using MeDaL, low-level, machine-dependent mechanisms for implementing the parallel execution were abstracted away from the view of the programmer, replaced by a high-level graph ‘design’. This design was refined to a program statement level for the sequential components (nodes), but synchronisation and communication was intended to be generated automatically. Only the data specification for the arcs was needed, the actual transmission and receipt of the data was hidden from the programmer. These key features of MeDaL form the basis of the subsequent work detailed in this thesis. A detailed description of the MeDaL notation, its evaluation and modification, and the subsequent development of the encompassing environment is given in Chapter 3, and briefly introduced next.

1.6 Research Focus

The aim of this work is to investigate methods and tools for parallel software design, and to propose new abstract methods for developing parallel software, drawing on elements from dataflow and graphical programming. The objectives of these new techniques are:

1. to provide an architecture-independent parallel programming methodology,
2. to allow the flexible specification of different forms of parallelism,
3. to efficiently exploit parallelism on a variety of machine architectures,
4. to provide a support environment with facilities to encapsulate complex parallel mechanisms within graphical objects,
5. to allow access to parallel programming for the non-specialist programmer.

The remainder of this thesis addresses the shortcomings of current parallel programming techniques in Chapter 2, and proposes an alternative methodology for developing parallel software derived from MeDaL. New methods are introduced using graphical representations and graphical user interface objects to provide an abstraction of existing low-level, machine-dependent mechanisms for parallelism. A graphical parallel programming environment ParaDE (**Para**llel **D**esign **E**nvironment) is presented over the subsequent chapters, formed by software development tools based around a graphical user interface to implement the capture and development of graph programs based on a modified version of MeDaL. ParaDE provides the basis for an investigation of a number of key issues relevant to the ease of use (programmer productivity) and efficient implementation of parallel programs.

Chapter 3 examines the issues raised by the MeDaL research, and proposes the new graphical design system, ParaDE, which adapts and extends the scope of MeDaL. New techniques for providing abstractions of current program structures are explored, which address some of the problems of current methods. Additionally, the necessity for certain programming structures in a parallel programming language is questioned. However, there remains a focus on exploiting existing programmer skills such that a wide audience of programmers can take advantage of this design system, and the usefulness of program development aids such as automatic syntax checking is discussed.

The problems of providing an efficient implementation of the programming methodology in ParaDE over different architectures are the subject of Chapter 4. Automatic code generation is used to produce programs in an intermediate code which has run-time support on different parallel architectures. This code generation involves extracting the parallelism information from the graph structure, and in this chapter the techniques for adjusting performance of the programs without reference to the architecture are also explored. In particular, the graphical support for specifying and varying the different types of parallelism identified in Chapter 2 is developed. Not only does this provide a problem-oriented way of designing and implementing parallel programs, abstracting the low-level parallel mechanisms, but it also allows the fine-tuning of the performance of these programs. This fine-tuning is achieved using graphical techniques which are completely detached from the underlying architecture, but yet allow a precise optimisation of performance for different target machines.

Chapter 5 presents the results of a performance evaluation of programs produced using ParaDE. Two widely different parallel architectures are considered, to look at the issue of machine independence. The nature and extent of the causes of performance degradation are addressed, and the role of the intermediate system software - primarily based upon implementations of the Linda virtual shared memory model introduced in Chapter 2 - is investigated.

The conclusions of this work are given in Chapter 6, where an evaluation of the success of the ParaDE methodology is carried out. The achievements and problems of the methodology are summarised, and the usefulness of the graphical parallel programming environment is debated. Conclusions are drawn from the experiences gained from this work, and the significance of the findings in relation to the fragmented parallel programming community is discussed, together with a consideration of future work.

2. Parallel Programming

Chapter 1 introduced a number of topics which influenced the research into ParaDE. The purpose of this chapter is to more fully develop each of those topics in order to understand their relevance to the subsequent research. The discussion will be restricted to the subjects already introduced, as to cover the enormous and wide ranging number of branches of parallel computing in one chapter would be impossible.

This chapter begins with a more in depth overview of the fundamental concepts involved in parallel programming: what constitutes parallelism, and how it is exploited in programming. The following sections describe some of the parallel architectures developed, and the general difficulties involved in exploiting parallelism on these machines. Next, the discussion moves on to the models traditionally adopted for parallel programming, and some alternative models which offer a new perspective on developing parallel programs. This chapter is concluded with a description of the MeDaL language which provides an introduction to the main work of this thesis, described in Chapters 3 and 4.

2.1 The Nature of Parallelism

The extent to which parallelism can be exploited in a program rests on the issue of **data dependency**. Data dependencies usually occur when some program operation is blocked (prevented from executing) because some or all of its input data is not available. The operation is thus forced to wait for some previous operation to complete and produce the missing data. These dependencies restrict the ability of the program to compute the operations in parallel, as a sequential ordering is imposed on the operations.

This is not the only way in which a data dependency can exist. Gajski [3] identifies three types of data dependency in a program - flow dependence, output dependence and antidependence, These three types constrain the achievable parallelism (the description in the previous paragraph corresponds to flow dependence) and are illustrated by the following example :

Given three statements:

1. $a = b + c$
2. $d = a + f$
3. $a = g$

it can be seen that none of these can be executed in parallel. Statement 2 has a flow dependence on statement 1 because the variable a cannot be used until its value has been computed in statement 1. Statement 3 is antidependent on 2 because the variable a must be used in 2 before its new value is computed in 3. Finally, statements 3 and 1 have an output dependence, as both attempt to compute a value for the same variable a.

The process of dependency analysis, which is necessary to identify parallelism [4] becomes more complicated the later in the software development cycle it is performed (in the same way that correcting design errors is more difficult at the coding stage than

earlier in the development). Ideally, all data dependencies would be identified at an early stage of the software development, and the algorithm describing the program should exhibit all possible parallelism. However, this is not as easy as it might first seem. Data dependencies can be determined at any number of levels in the program design and implementation. The examples above show what can be considered as a **low-level** examination of data dependencies, affecting the relationship between program operations. Such relationships probably involve single data elements, and correspond to a level approaching that of the binary machine code. A **high-level** data dependency might for example exist between two operating system processes, one of which is required to execute after the other, for one of the reasons given above. Two functions in a high-level language program (e.g. C) might exhibit a dependency which could be described as **medium-level**. The reasons for choosing one level over another will be examined in section 2.6.1 on granularity. Low-level dependencies are usually associated with dataflow languages, which are the subject of section 2.7.1. Some fairly simple rules have been developed in the dataflow language field which can drastically reduce the occurrences of some low-level data dependencies, particularly those of output and antidependence. This process of eliminating unnecessary dependencies is important in order that types of parallelism may be identified and exploited.

2.2 Types of Parallelism

Parts of a program which are not restricted by data dependency may exploit parallelism. There have been a number of categorisations of types of parallelism used in programming, one, by Carriero [5], describes three concepts: result parallelism, agenda parallelism and specialist parallelism.

1. **Result parallelism** is demonstrated where the *result* of the computation is achieved through the production of a number of components which make up the final result, for example, each element of a data structure. The component results are calculated simultaneously by a number of separate processes, up to the point at which one process must wait for a result from another process (data dependency).

2. **Agenda parallelism** can be seen when the same computation is applied to a number of data elements, or sections, simultaneously by a number of *worker* processes. The workers each continue to take another section of data when one is completed. In this way all of the workers are working together to complete each item (computation) on the *agenda*. Once this has been completed, the next computation is executed by the worker processes. This format is often referred to as a master-worker model, where the master issues the instructions (the agenda) to the workers.

3. The final type of parallelism in this categorisation is **specialist parallelism**. In this type, each process has a particular *specialist* function to carry out, and all of the functions are executed in parallel where this is possible in the computation. Specialist parallelism often occurs where a number of functions exist which are relatively autonomous. This would stem from an algorithm which comprised of a number of logically coherent and cohesive computations.

It is likely that a parallel algorithm exhibits more than one type of the parallelism categories outlined above. However, it has been common practice to write parallel programs exploiting only a single type. Rather than the categorisation proposed by

Carriero, parallel programs have tended to fall into one of two different forms, usually referred to as **task parallelism** and **data parallelism**. These correspond roughly to the specialist and agenda types respectively, where task parallelism involves a number of relatively independent processes which execute in parallel but sometimes communicate to exchange data. This is illustrated in Figure 2-2 where processes 2, 3 and 4 perform different functions simultaneously, receiving data from process 1. Data parallelism on the other hand is concerned with a single computation operating simultaneously over a number of parts of a data structure. Figure 2-1 shows columns a, b, c and d of a matrix sent to processors P1-4, which perform the same function on the different sections of data simultaneously. Scientific applications using techniques such as matrix operations are well suited to a data parallel model, as are database operations. Data parallelism is convenient for programmers and parallel systems in that it is usually regular and scaleable. Task parallelism, however, is often displayed in applications with irregular communication patterns or distinct multiple functions.

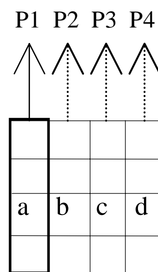


Figure 2-1 Data Parallelism

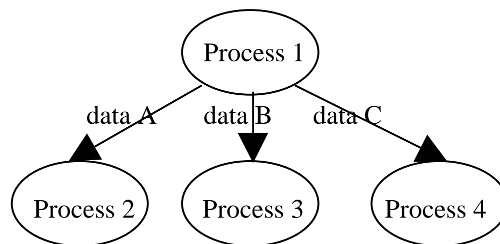


Figure 2-2 Task Parallelism

More recent work [6], [7], [8], investigates the exploitation of both task and data parallelism in parallel programming languages. Applications such as complex simulations may have a number of different simulation models - parallel tasks - but within each there may be large data structures suited to data parallel techniques. An example would be a simulation of an aircraft, involving models of fluid dynamics, structural mechanics etc. The exploitation of mixed parallelism, as with singular parallelism, is often limited to extensions of sequential programming languages which are frequently biased towards the architecture they are executed on. For this reason, a brief overview of parallel architectures follows before exploitation of parallelism on these architectures is described.

2.3 Parallel Architectures

Rather than try to describe parallel architectures in terms of the physical machines, which are many and diverse, it is useful to consider the characteristics of groups of machines by the use of computation models which represent them. Almasi [4] presents a computation model as having five main attributes:

1. primitive units of computation,
2. control mechanism,
3. data mechanism,
4. communication,

5. synchronisation.

Before describing different models for parallel computation, it is useful to first look at how the sequential model fits in with these five attributes. The sequential computation model, attributed to von Neumann, for the traditional single processor machine has dominated computing for decades. Von Neumann's model consisted of a processor for computing operations in a set of registers; a control mechanism for serially fetching instructions from memory and moving data between memory and the processor; and a data mechanism consisting of program and data stored indistinguishably in an ordered address space. The final two attributes, those of communication and synchronisation, were implicit within the model, as using a single processor and sequential execution removed such problems from the domain of the programmer.

The variety of parallel computation models are characterised largely by the last two attributes - communication and synchronisation. In more general terms, these features dictate the way in which the computation elements are co-ordinated, and relate to issues involving both data and instructions. The most commonly cited classification of computation models (covering both sequential and parallel forms) is that of Flynn [9]. Flynn classified models in terms of their ability to exploit single or multiple instruction streams across single or multiple data streams. A single instruction stream with a single data stream (SISD) corresponds to the sequential model of Von Neumann already described. Parallel computation is introduced in all of the other models in this classification, notably single instruction stream multiple data stream (SIMD) and multiple instruction multiple data streams (MIMD) which are discussed below. The remaining category (MISD) has not led to any useful architectures. Another classification by Kuck [10] enlarges Flynn's classification, but this only reinforces the diversity of parallel architectures, and the motivation here is merely to demonstrate some general differences between parallel machines.

SIMD machines in the Flynn classification include vector and array processors, where the same instruction stream operates on different sets of data simultaneously. This model maps closely to applications demonstrating data parallelism as introduced above. The alternative parallel model, MIMD, covers a range of architectures, including systolic arrays, dataflow architectures (examined in a later section), shared memory and distributed memory machines. Treleaven [11] offers a more detailed view of MIMD machines, presenting a further classification, but this section will only look at the two most common (and general-purpose) types of machines - **shared memory** and **distributed memory**.

Shared memory architectures generally comprise of a set of **tightly coupled** (closely connected) processors and a set of memory modules, as shown in Figure 2-3. The memory is shared in that any processor can access any memory location, through an interconnection network. Of the two types of architecture, the shared memory model is generally most desirable for programmers, giving a relatively simple model of computation (somewhat similar to programming a SISD machine). From a hardware point of view shared memory architectures are less attractive, as the cost of providing simultaneous access to the memory and the purchase cost involved in building a multiple processor machine (a **multiprocessor**) can be very high.

In contrast, distributed memory architectures can be easily and inexpensively implemented by connecting a set of stand-alone computers together. Distributed

machines feature a set of autonomous processors, each with its own private memory module, as illustrated in Figure 2-4. They are often referred to as **loosely coupled** architectures because of the more independent nature of each processor.

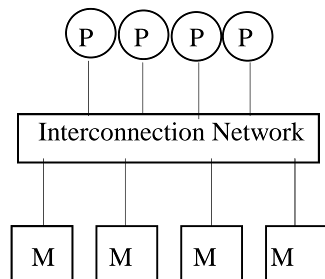


Figure 2-3 Shared Memory

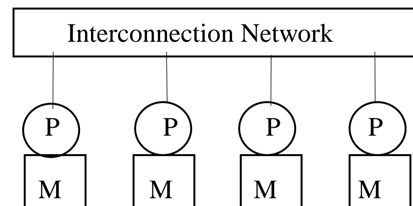


Figure 2-4 Distributed Memory

Communication is via the transmission of **messages** between processors along the interconnection network, which can be anything from a simple linear topology to a fully interconnected scheme. Distributed memory systems are more suited to applications that have largely independent computations which can be carried out simultaneously with a small amount of communication.

The suitability of applications for different architectures will be covered in section 2.6.1 on granularity, but it can be seen that distributed memory architectures more closely correspond to a task-parallel model introduced earlier due to higher communication costs, whereas shared memory systems do not match any particular parallel programming model as discussed so far. In fact shared memory systems are often viewed as a more general-purpose parallel programming platform. The different programming requirements and the ways in which parallel programming has traditionally exploited the parallelism of the various architectures introduced is now explored.

2.4 Techniques for Programming Parallel Architectures

The two main types of MIMD parallel architecture, shared memory and distributed memory, have different ways of implementing parallel behaviour, but the parallel requirements are similar. The first three attributes of a computation model introduced above, i.e. primitive units of computation, control mechanism and data mechanism (at least within the computation units), are generally achieved in much the same way as in sequential architectures. Programming for shared memory or distributed memory systems often uses modifications of existing sequential languages, or new languages in a similar style of computation behaviour. The remaining two attributes, communication and synchronisation, are the key features which allow parallelism, and which are dealt with differently in the two MIMD types of system. Additionally, data partitioning is an issue relating to communication and synchronisation, which will be shown later to heavily influence the choices of parallel programming techniques.

2.4.1 Communication and Synchronisation

Communication and synchronisation for shared memory machines is normally achieved through shared variables in memory, and the program control and co-ordination of computations is centralised. Programming shared memory machines has

a particular requirement in data communication - to allow only mutually exclusive access to the memory location being shared. Where multiple accesses to the same memory location occur, results can become inconsistent due to the unconstrained use of data being manipulated by multiple sources. This has to be resolved by the programmer, who inserts **locks** into the program to maintain the consistency of the data in memory, preventing a second process from accessing sections of memory which are in use [12]. The mechanisms used to ensure the consistency and correctness of shared variables include the **semaphore** [13]. Two operations, wait and signal (or P and V), decrease or increase the value of the semaphore. The value is prevented from going below zero by suspending any further processes until a corresponding signal causes a waiting process to be reactivated. Other locking mechanisms exist, including the higher-level **monitor** [14]. The monitor adopts an object style, containing both the data and the operations permitted on the data. Only these operations can be used on the data, and only a single process can enter the monitor at any one time.

Apart from the synchronisation issue, the programmer views a shared memory as normal data, in the same manner as the von Neumann model. Additionally, shared memory machines have the advantage that they can also implement a message-passing model, thus supporting both shared memory and message-passing based languages.

Distributed memory machines generally feature only the message-passing style of programming. This style requires the parallel processes to explicitly establish a communication route between them, and use this to transmit and receive **messages**. One example of a (relatively high-level) message-passing style is the **rendezvous** in Ada. A rendezvous is the meeting of two processes which exchange data before proceeding with their individual computations. Synchronisation is required between the two processes in this programming style, so if one reaches the rendezvous point before the other, then the first process must wait (suspend activity) until the second process arrives at the rendezvous point in its own code.

More recent parallel programming models such as PVM and Linda attempt to provide parallel mechanisms which operate on a *virtual* parallel architecture. This supports the notion that the characteristics of the parallel architecture are hidden from the programmer, who instead writes programs for a more abstract parallel model. In PVM [15], a large number of library routines (20) are provided for communication, process control and synchronisation. These routines are designed for a message-passing style of programming, and are similar in some ways to other message-passing based languages. For instance, PVM uses `snd` and `rcv` for communication and `barrier` and `waituntil` for synchronisation. The difference between PVM and other message-passing-styles described is that the underlying parallel architecture may be very much more complex, made up of multiple machines with differing characteristics.

Linda [16] adopts a different approach, offering a virtual shared memory, known as the *tuple space*. The Linda model still abstracts the underlying parallel architecture, which may even be distributed, but does so in a shared memory style, whereby programmers can output data *tuples* into the tuple space using `out`, input the data tuples using `in` or `rd`, and create processes using the `eval` statement. Linda uses only a small set of commands (6) which reside in a host language such as C, and offers a much higher level of interaction with the parallel environment. Whilst the

parallel mechanisms are much simpler than say PVM, which operates at a fairly low level of interaction, this may also limit the flexibility with which they are used.

2.4.2 Data Partitioning

While shared memory models adopt a single name space - a common shared memory - on which parallel versions of traditional languages map quite well, distributed memory models require the programmer to be responsible for the management of data moving between local name spaces. This data management is handled by the hardware in shared memory models and is hidden from the programmer. Often, where data parallelism is exploitable, the efficient execution of a parallel program hinges on the rapid availability of relevant data. A **data partitioning** scheme must be used to ensure that data is distributed in an efficient way to the correct memory units, as and when the data is required by the corresponding processor. This **temporal locality** is essential if communication costs are to be kept to a minimum, but conflicts with the desire to distribute data to maximise parallelism.

Methods for data partitioning can be quite complex, according to the intricacy of the data patterns in the algorithm, and characteristics of the compiler and the parallel architecture also influence the difficulty of finding an efficient data partitioning strategy [17]. Research has been carried out into methods of automating the data partitioning process [18], [19], with benefits of ease of programming, code maintainability, code modularity, and code portability [20]. However, the term automatic in this sense is usually referring to extraction of data characteristics by the compiler, from directives inserted by the programmer. Thus, the specification of data partitioning is still required, although using techniques like the *decomposition*, *distribute* and *align* commands in Fortran D [6] do help to reduce the intricacies of partitioning. The *decomposition* statement identifies a segment of data to be partitioned, *align* positions an array on this segment, and *distribute* provides a mapping to the machine architecture, using a block, cyclic, or block-cyclic distribution [17].

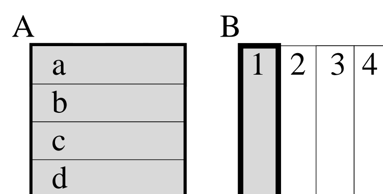


Figure 2-5 Data Partitioning

It is difficult to eliminate issues of partitioning from the domain of the programmer, as some are inherent in programming problems. In matrix multiplication, for example, the algorithm can be split into **unit computations** of the multiplication of each row of one matrix by successive columns of the other matrix. If this is the level at which parallelism is to be exploited, then the obvious data distribution would be the replication of the whole of one matrix over the set of processors, and the partition of the other matrix into columns, distributed over the processors. Figure 2-5 shows Matrix A consisting of four rows a,b,c and d which are distributed to all processors, and Matrix B which is partitioned into columns 1, 2, 3 and 4, which are distributed to different processors. The grey shaded parts show the data distributed to processor 1.

There are other ways of partitioning this problem, which might increase or decrease efficiency, depending on the target architecture, but from a problem-oriented point of view (i.e. relevant to the algorithm for matrix multiplication) this is one of the most natural partitioning schemes. It is not necessarily the most parallel, as all elements could be computed in parallel, but it is a common approach to this algorithm. This example shows another situation in which abstraction can be used to distinguish between problem-oriented issues, and those of machine architecture. It will be shown later that using abstract methods to specify a programming problem (including its data partitioning), machine dependence can be avoided while still retaining the facility to optimise performance.

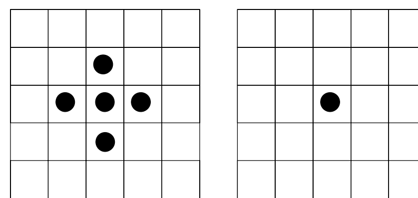


Figure 2-6 Data Reference Patterns

Studies of data distribution patterns [21] in programs using the Fortran D style partitioning statements [22] reveal that there are a number of common patterns which recur in general-purpose parallel programming. A set of **reference patterns** was developed [21] to give a graphical representation of the partitioning schemes.

Figure 2-6 shows two such reference patterns, which represent the input and output data patterns for the elements of a matrix used for example in a heat flow program, where the value of one element (shown on the right) is related to the values of its neighbours (shown on the left). While these patterns were developed from partitioning characteristics of existing programs, they could also be used as an abstract way to *specify* data partitioning. This technique will be explored later.

2.5 Current Methods for Exploiting Parallelism

The programming requirements for the different machine architectures introduced are satisfied in a number of ways, in different approaches to parallel programming. Gelernter [23] groups parallel programming models into two types. The first encompasses three broad classes of languages :

1. automatic parallelisation of sequential languages using compilation or run-time techniques,
2. new parallel languages, designed specifically for parallelism,
3. generalisations of some base language to create a new parallel language.

All of these three are identified by their approach of presenting a complete programming model, including whatever tools are required to prepare the programs for execution (e.g. compilers). The second type of programming model proposed by Gelernter is the concept of a separate co-ordination model which provides the means to create and manage processes, including any communication between them. This co-ordination model is distinct from, and works in co-operation with, the host

computation language. The co-ordination model approach (i.e. Linda) will be discussed in a later section. This section will concentrate on describing the first group of programming models, which have formed the basis for most of the parallel programming approaches used up to now.

2.5.1 Automatic Compiler Techniques

The ideal situation for parallel software would be to have the compiler detect tasks for parallelism, but while this would be useful for so-called *dusty deck* programs or *legacy software*, using the same approach for new software ignores the need to introduce parallelism at the design and algorithm stage. Much of the work on parallel programming has been for data parallel methods, using compiler automation primarily aimed at legacy software. While this is often of little interest to some computer scientists, who might prefer to develop new applications, the underlying theme is important. This theme is the possibility of producing parallel programs from sequential ones. If such a route was found, which produced efficient parallel code without ever having to consider parallelism, the benefits are clear.

Unfortunately, it is commonly accepted that as an overall aim the parallelisation of sequential programs cannot realistically be achieved. The main problem which stands in the way of this aim is the difficulty of identifying parallelism in sequential code. Dependency analysis is the technique of finding data dependencies in the code, in order that those program statements not affected by dependencies may be executed in parallel [4]. This technique is often associated with single data elements, but can be applied at a higher level, to identify independent *sections* of code rather than single statements. Many examples detected by dependency analysis are related to loop computations, and can be eliminated by altering variable names or rearranging array subscripts. However, even with suitable language features, the extent to which parallelism can be exploited by compilers is limited. Only those parts of the program conforming to the required format (e.g. loops using vectors) can be exploited, and it is likely that a programmer may not always use the necessary formats. Analysis of larger sections, such as *interprocedural* analysis, is normally even more difficult for compilers [24], as it is non-trivial to calculate the effects of a procedure, especially given the existence of aliases, references and pointers. Ascertaining independence is therefore almost impossible for compilers, particularly where procedures are separately compiled.

Another important consideration is the design of the algorithm from which a program is developed. Often there is no single correct way to solve a programming problem, and many different algorithms might reach the same result. Some of these algorithms may exhibit more parallelism than others, and some may show little or no parallel behaviour. Perhaps even more important is the fact that traditional methods encouraged, or dictated, a sequential design, from which the exploitation of parallelism is difficult if not impossible. In the development of the dusty deck programs, it is unlikely that parallelism was an issue, as the machines of the time were sequential. This means that the algorithm and the resulting program would be based on an efficient sequential design, but one which may not be efficient when executed in parallel, or may not be parallelisable at all. For new software and applications, designers would start with an efficient parallel algorithm, and this would often only

produce a corresponding efficient solution using a language developed for parallelism. Such languages are considered next.

2.5.2 New Parallel Languages

Programming languages can be categorised into one of two groups, **imperative** or **declarative**. The traditional sequential languages are imperative, in that they describe *how* a computation is carried out. The alternative declarative style of language concentrates on *what* the computation requires, and not on how to achieve it. Declarative languages are looked at with other styles of unconventional programming models in Section 2.7. Here the discussion centres on imperative parallel languages.

Two of the most well known languages written specifically for parallel execution (but still remaining within the imperative group of languages) are CSP [25] and Ada [26]. Both adopt message-passing styles of programming as do most specialised parallel languages (although Ada also supports a shared memory model). CSP was originally a theoretical model for concurrent processes, but has since been implemented as a language. It was very influential in message-passing styles of programming: the transputer language Occam [27] was based on CSP, and Ada took influences from both CSP and high-level language styles such as Pascal. Where Ada is more of a general-purpose language however, Occam is a very specialised language whose main use is for transputer arrays.

The Ada rendezvous, briefly described earlier, is a synchronous technique, whereby both processes must meet to exchange data. CSP, however, is asynchronous, requiring specified channels to act as communication routes between two processes. The receiver process does not have to be ready to receive the data, but can pick up the data from the channel at a later time.

While parallel languages such as Ada and CSP have proved very useful for certain types of software development, e.g. real time systems, the fact that they closely match the underlying message-passing machine architecture means that flexibility can be lost. Certain types of algorithms may be suited to such a style, but others may not. Some languages in this style have good implementations for shared memory architectures as well, such as Ada, and the rendezvous is fairly easy to implement on shared memory machines. However, the lack of design tools for developing programs in these languages limits their use to a small set of applications. In searching for a general-purpose parallel programming style, these parallel languages are often rejected in favour of extensions to existing sequential languages.

2.5.3 Generalisations of Sequential Languages

A number of parallel extensions to existing sequential languages have been developed to allow such languages to be used for parallel software. Some of these simply involve a few additional language constructs. A typical construct, *forall*, may allow a section of code to be performed on multiple sets of data simultaneously. This corresponds to the data parallel model. The alternative task parallel model can be seen in extensions such as the *fork* statement used to spawn child processes, which execute in parallel with the parent process then synchronise their control flow with the *join* statement. In some cases, such parallel extensions are merely compiler directives, indicating where parallelism *may* be exploited, and a compiler makes use of these to produce a parallel program suited to the underlying architecture. This method of explicit parallel

directives may be considered as somewhere between the automatic compiler approach, and that of true parallel extensions to existing languages, which specify parallelism directly.

HPF [28] is a development of a sequential language Fortran, with facilities added to allow the specification and exploitation of parallelism. HPF developed from the Fortran D language [22], which concentrated on the specification of data distribution and decomposition, aimed both at the problem level and at the machine mapping level. Some of the problem-oriented features of this data decomposition influenced the data partitioning features of ParaDE, and were introduced in Section 2.4.2.

Newer parallel models, such as PVM and Linda which were introduced earlier, can also be considered as parallel extensions to sequential languages. Linda (described in more detail in a later section) provides a small set of very simple parallel primitives which reside in a host language such as C. These primitives allow the manipulation of the shared tuple space and support process creation in an abstract high-level manner, without any need for specifying low-level parallel interactions. In contrast, PVM provides a large set of library routines which extend a host language, supporting facilities for communication, synchronisation and process control. These library routines give the programmer the flexibility of specifying detailed, low-level parallel interactions, whilst writing the computation in the conventional sequential language.

2.6 Problems with Existing Parallel Programming Methods

Chapter 1 introduced the concept of abstraction in software development, in particular the evolution of sequential languages to higher-level models, and the corresponding failure of parallel software models in this respect. Where sequential languages have drifted away from low-level, machine-dependent mechanisms, parallel languages in general have remained low-level with respect to the communication and synchronisation of parallel processes. This is not to be confused with the basic computation language, which remains based on sequential high-level languages in most cases. It is the additional requirements to co-ordinate the parallel sections of code which have been addressed using low-level techniques.

With the exception of compiler automation, which has had limited success, and some alternative models discussed below, the different approaches to parallel programming are influenced by the underlying machine architecture. The new parallel languages, usually displaying a message-passing approach, are typically based on distributed memory machines. CSP, for example, requires the existence of named communication channels, through which data (messages) are transmitted and received by the two communicating processes. Alternatively, the parallel extensions to existing languages, tending to be implemented on shared memory machines, use methods for implementing parallel behaviour which correspond to the shared memory model (although with some recent developments, such as HPF, this is becoming less significant). For example, the semaphore is used to ensure mutually exclusive access to a shared variable - the mechanism for communication and synchronisation for the shared memory approach. The main reason for machine dependence of parallel programming techniques is the desire to achieve efficient execution - i.e. a significant **speedup** over sequential solutions. Because the parallel architectures vary considerably, efficiency on one machine may be inefficiency on another. Usually the

main difference between efficient solutions on different parallel machines is the ratio of the time spent in computation to the time spent in communication - this is determined by the **granularity** of a program, i.e. the size of parallel computations.

2.6.1 Granularity

Granularity is simply the size of computations which execute in parallel. Fine granularity is observed when small sections of code form the parallel computation - this may be a few program statements or even a single operation. Coarse granularity refers to large sections of code, possibly even independent programs, running in parallel. The problem of granularity in parallel programming is concerned with choosing an appropriate size of computation which maximises parallelism whilst limiting the overheads of communication. Granularity is probably the key issue in parallel software design, and the efficiency, or otherwise, of parallel software often hinges on the appropriateness of the software grain-size to the target architecture. Critics of fine-grain programming claim that execution is costly in terms of communication and has, therefore, very high bandwidth requirements. The proponents of the approach still argue for fine granularity on several fronts:

1. easier compiler translation from implicitly parallel languages,
2. lower communication costs than generally believed (due to locality exploitation),
3. increased understanding of the limits of parallel machine behaviour,
4. suitability to many fine-grain applications [29].

Perhaps the most significant argument, however, is that fine granularity is good for understanding applications, and a subsequent coarsening of the granularity leads to reduced communication and synchronisation, while maintaining ease of programming. It is useful to draw a distinction between the granularity used to describe the application in design/implementation, and the granularity of the subsequent execution of this solution on a parallel architecture, or in fact the granularity used in design and in implementation. This is, again, an issue of abstraction, and is central to this work. Until programmers are able to differentiate between design, implementation and run-time behaviour, parallel programming will not overcome its problems of architecture dependence and unnecessary focus on low-level mechanisms. Of course, these distinctions are more difficult to make without good tools for designing and developing parallel software, incorporating useful abstractions of current parallel mechanisms. Many recent language designers recognise this, with the adoption of virtual machine architectures and abstract programming models as a basis for the underlying machine structure. There is a deliberate move away from the problems associated with architectural details, in fact these are handed over to the designers of the computation model on which the language sits. This model can then be portable to different parallel machines, requiring only the implementation of the system-level software support.

Mohr [30] discusses the difficulties of having the programmer define the granularity - the costs in terms of programmer effort and clarity of software, and the complexity of parameterising the program for different architectures. The conclusion drawn is that the *exposing* of parallelism should be the responsibility of the programmer, while the system takes on the task of *limiting* the parallelism; this being the emphasis in the

design of Mul-T [30], a parallel version of Scheme using the **future** construct of MultiLisp. This construct acts in a similar way to the compiler directives used in languages relying on compiler detected parallelism.

Other work has attempted to address the issues of multiple granularities [31] or heterogeneous parallelism [32]. The former is a compiler technique, not only exploiting fine-grain data parallelism and loop-level parallelism, but also parallelism at a function or procedure level. Alverson avoids reliance on compiler detection by exploiting parallelism at hardware, compiler, runtime and operating system levels [32]. It is claimed that the various components respond to varying parallel requirements. Alverson's system is based on a virtual shared memory model using conventional sequential languages, while the underlying machine uses **multithreading** on a large scale multiprocessor. Multithreading has become popular recently, with a number of implementations being developed [33], [32]. The benefit of threads is that they have lower overheads than context switching between processes in operating systems. Multithreading allows multiple simultaneous threads of control within an address space. The Distributed Computing Environment (DCE) [34], supports both multithreading within an address space and Remote Procedure Call (RPC) across address spaces. The latter is similar to the conventional procedure call mechanism and is thus a relatively easy transition for programmers to make.

Not only are the communication and synchronisation methods widely different between different parallel programming methods (and even between languages of similar programming styles), but the individual methods themselves are complicated. Additionally, it is difficult for a programmer to analyse the resulting behaviour without the use of complex theoretical models. Parallel programming has to a large extent remained an activity for researchers and specialist programmers, with the main body of conventional software developers unable to exploit the benefits of the new parallel architectures. For this reason, attempts have been made to question the fundamental characteristics of existing languages, and to develop novel approaches to parallelism. The issue of data dependency is addressed by one such novel approach, dataflow, whilst others address the problems of abstraction and architecture independence.

2.7 Novel Methods for Parallelism

Recent developments in parallel languages have shown a progression towards a more abstract model, trying to achieve the high-level approach already established for sequential software. The low-level mechanisms used in parallel languages for the co-ordination of parallel processes have begun to be included in languages in a more problem-oriented style, promoting portability and machine independence. Intermediate system software has been used in some cases to implement an additional level of abstraction encompassing both the computation and co-ordination aspects at a similar level. Before discussing one such model (the co-ordination language **Linda**) and some further abstraction methods for programming in the form of visual or graphical languages, the discussion focuses on some less conventional textual languages introduced in the new parallel languages section. These languages are the declarative languages, which adopt a more problem-oriented approach, concentrating on what the computation is trying to achieve, and not how it should go about it. Two

closely related declarative language styles are dataflow and functional languages, which are explained now.

2.7.1 Dataflow Concepts

The attractiveness of dataflow for parallel programming stems from its natural parallelism. The declarative approach abandons the traditional notions of control and instead reaches a solution following a set of relationships between computation nodes. For dataflow these computations were traditionally simple operations on single data values, but have since developed in a number of different directions. Dataflow has, over the years, been applied to software design, low-level and high-level languages, and even computer architectures. The concepts behind all of these however, are similar, only the objects to which they are applied, and the way in which they are implemented, varies. Figure 2-7 shows an example of a dataflow graph for the calculation of *(a+b)-(c*d).*

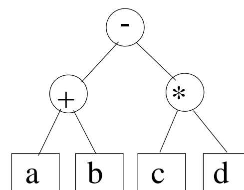


Figure 2-7 Dataflow Graph

The computation nodes (operations in the graph) follow a **firing** rule which defines the conditions under which execution occurs. Usually this requires that all input data should be available before execution of the operation. This rule is known as the **standard firing rule**. Sometimes, particularly in higher-level dataflow approaches, this rule is violated in certain circumstances. Examples of violating the standard firing rule will, in later sections, demonstrate the reasoning behind this behaviour.

The dataflow model views data as temporary values, produced by the source computation node and consumed by the destination computation node. The relationships between the computations are defined by the data dependencies between data inputs and outputs of connecting computations. We will assume, for simplicity, that the computation here is a simple operation, taking data values as input and producing data outputs. Such an operation in a dataflow model executes as and when its input data becomes available. A data dependency occurs when some operation is blocked because some or all of its input data is not available. It is thus forced to wait for some previous operation to complete and produce the missing data. Such restrictions on the parallelism occur naturally in the problem being developed, and it is likely that they cannot be overcome, unless the techniques described below can be applied to remove unnecessary dependencies. In fact, the problem with dataflow is not usually that parallelism is being restricted by such dependencies, but that the parallelism introduced by considering only data dependencies is far in excess of the machine's parallel capability. A later section will discuss how this granularity issue can be addressed.

The removal of data dependencies eases the task of the parallel software developer, and some fairly simple rules can drastically reduce the occurrences of output dependence and antidependence, as described earlier. One such rule is the single

assignment rule, adopted by most dataflow languages, which specifies that a variable can only be given a value once, each subsequent assignment must be to a new variable. This promotes clarity and ease of verification, although it is not a requirement for a dataflow language [35].

2.7.2 Dataflow Languages

Dataflow as a programming model has also influenced language development, with the emergence of a number of dataflow languages. Dataflow languages vary, with some being entirely functional (such as VAL), while others are single assignment languages (such as Id). Implicit and explicit parallelism are also a source of difference between dataflow languages in that while program statements specify parallel expressions explicitly, these are not commands but merely indications to the compiler that the potential for parallelism exists at that point (VAL claims to use both forms). SISAL [36] is a descendent of VAL, and was the first language of its type to match the performance of Fortran on a Cray Multiprocessor [37]. SISAL has no explicit parallel mechanisms, but data and operations are specified in a way that allows the exploitation of parallelism by the SISAL compiler. SISAL is more limited than other functional/dataflow languages including the restriction to first order functions. However, it has proved to be an efficient implementation of the dataflow concepts on a general-purpose machine.

The syntax of dataflow languages is similar to that of traditional high-level languages, from which a dataflow graph structure is generated automatically. This graph can be executed on a dataflow machine, discussed below. Token models which use dataflow graphs can verify the data behaviour by representing dataflow as a stream of tokens carrying data items [38]. Such models exhibit determinate execution, independence of system timing, and functions free of side effects. Locality of effects is feature of dataflow languages, resulting from the single assignment rule and value-based data passing. This leads to containment of the effects of data dependency.

Gajski, in a critical survey of dataflow techniques [3], claims that dataflow graphs are difficult to construct and manipulate by humans and are error prone. None of these points is necessarily true, given that it is well known that humans can deal more easily with the more natural graph based models than textual ones. However, this counter-argument usually refers to more high-level graph descriptions, offering a concise design-level description rather than a complex and detailed low-level data dependence graph. It is likely, though, that any bias towards textual descriptions is probably a result of conditioning and not human nature, as programmers have become accustomed to dealing with textual based software methods. The fine granularity of a dataflow language and its data dependencies would of course make graph generation more difficult, with the programmer working at a lower level of detail. High performance from such a language would not be achievable except using a specially designed machine architecture.

Gajski further claims that the properties of functionality and freedom from side effects are merely conveniences, and compiler techniques can exploit parallelism equally well. Clearly there are some similarities in this respect between parallelising compilers and dataflow languages, as both take a language with an imperative syntax and implicitly generate parallel executable code. This would seem to be the downfall of both, as it is the transition from sequential text to parallel code which is restricting

the explicit parallelism of the design. Dataflow languages are adopting a backwards approach to parallelism, generating a dataflow graph (essentially a refined design) from a textual implementation of a design, and attempting to automatically exploit the parallelism. The dataflow graph generated is then executed on a dataflow machine - a novel architecture developed some years ago which failed to achieve general acceptance. These architectures are described briefly next to highlight some problems associated with the direct implementation of the dataflow model. While dataflow machines have been largely unsuccessful and the dataflow languages have not achieved widespread acceptance as a parallel language, it will be shown later that dataflow as a concept can be usefully adopted in the design and development of parallel software, with conventional parallel architectures.

2.7.3 Dataflow Architectures

Machine architectures adopting the dataflow approach, such as the Manchester Data-Flow Machine (MDFM), have continued to attract attention, despite their limited acceptance. Work on dataflow architectures began in the late 1970s at Manchester [39] with subsequent projects at MIT [33] and the Electro-Technical Laboratory in Japan [40]. The basic components of the MDFM included a token queue, matching unit (pseudo-associative matching store), instruction fetching unit and execution unit. The latter was implemented by a parallel array of function units.

Early experience indicated that fine-grain dataflow parallelism exceeded the parallel capability of the hardware, and led to development of parallelism regulating techniques. A **Throttle Unit** was designed to hold back the amount of parallelism and thus reduce the data storage requirements of tokens [41]. This was achieved by the use of **Activation Names** in the tag field of a token, to distinguish tokens from different stages of re-entrant code. In this way, tokens could be prevented from entering sections of code, such as a later iteration of a loop. The throttle unit would not release the relevant activation name, to allow continuation, if the workload was deemed to be above some limit. There were problems with this design, that congestion was transferred to the throttle unit, the throttle algorithm was slow, and delays occurred in-between an evaluation of the workload and the effect of the throttle unit's release of an activation name. Variations on this model were adapted, including a *chunking* throttle, which moderated the size of iterative code affected by throttling [42]. Later developments included a revised algorithm for throttling, and implementation of the activity tokens by hardwired signals, feeding back system workload information directly to the throttle unit [43].

2.7.4 Functional Languages

Functional languages are a style of programming closely related to dataflow. In fact where only the fundamentals concepts of the two styles are adopted, their names are often used synonymously. Functional languages are briefly described here to provide the background to some textual, and later graphical, programming models based on the declarative style, which have been influenced by earlier pure dataflow and functional approaches.

A functional language uses **expressions** to describe computations, in the same way that mathematics uses expressions e.g. $a := f(x, y, z)$. Built-in operations together with function definitions make up a computation using input and output values.

Functional languages, like dataflow languages, exhibit freedom from side effects and use single assignment to prevent **aliasing**, i.e. the use of only a single name for each value. Higher-order functions are often supported in conventional functional languages, where functions are used as parameters, and produced as results, giving the concept of functions as values. **Type inference** is another common feature, meaning that the type of some values can be deduced from the input values and the operations applied on them.

The advantage of functional languages over their imperative counterparts is claimed [36] to be that they produce concise programs, reducing complexity and size, leading to easier development and maintenance. This stems mainly from the use of higher order functions and type inference. The formal basis of mathematical expressions, and the elimination of aliasing and side effects, allows reasoning about the programs, and correctness to be ascertained. This also potentially allows automatic parallelisation by compilers, extracting a very fine-grain data dependency parallelism.

Disadvantages of functional languages are that compiler exploitation of implicit parallelism is still far from efficient, and cannot compete with explicitly parallelised alternatives [44]. Additionally, modern functional programming languages feature higher-order functions and non-strict semantics. These features, while enhancing expressive power, increase the complexity of the compiler and add to the run-time overheads. There is also an argument that such novel features are too far removed from traditional programming styles to achieve wide scale acceptance.

There are clearly similarities between functional and dataflow languages, both adopting single assignment, and using data dependencies to define the parallelism. The primary difference between the two language styles is that while in dataflow the arrival of data triggers the operation (data driven), in functional languages the data is requested by the operation (demand driven).

Granular Lucid (GLU) is described as a coarse-grain dataflow language, but is also a first order functional language, indicating that higher order functions are not supported [45]. It combines a coarse-grain version of the dataflow/functional language Lucid with the imperative language C. User defined functions are written in C, as are the data types, and Lucid provides the specification of the composition of the C functions. Lucid is based on a structured set of equations, each defining a variable. A multidimensional space, defined by user specified dimensions, provides a representation of all of the values of the variables. Parallelism is implicit in Lucid, and found in a number of forms:

1. pipelined parallelism, where each function processes a different part of an input stream.

2. data parallelism, processing different data with the same function simultaneously.

3. tournament parallelism, a tree-based data parallelism, where the extent of the parallelism decreases logarithmically at each level.

All these parallel mechanisms were intended originally to be extracted by compilers, and are essentially fine-grain data dependency relationships. Where GLU differs from Lucid is that user-defined C functions become the elements which are executed using

the parallel forms described above. This not only increases the granularity, but also provides a basis for reuse of existing sequential software.

GLU programs are mapped to an intermediate abstract machine, and compiled from this form to a specific parallel architecture. The abstract machine uses a computation model known as **eduction** [46]. This is basically lazy evaluation executed in a demand-driven style (although data-driven can be partially used). This has the drawback of demand propagation which introduces extra overheads. However, in contrast to SISAL or Lucid alone, the implementation of coarser-grained program sections leads to a more efficient execution on conventional parallel computers (that is, not dataflow architectures). Coarser-grained approaches, and the application of dataflow/functional concepts to higher-level parallel models are becoming increasingly popular. One such higher-level model, whose use in a dataflow manner will be shown later, is the co-ordination language, Linda.

2.7.5 Co-ordination Languages - Linda

In addition to the language extensions which merely provide a number of extra parallel commands, some approaches to parallel software adopt a more complete parallel model. These, considering a more general philosophy to the use of parallelism, provide support for more than just parallel commands. A good example of this approach is Linda, which is described by its inventors as a co-ordination language [16].

Linda exists alongside the host language and not in place of it, for example, C-Linda, Prolog-Linda. Linda provides co-ordination mechanisms which allow programs written in the resulting language to express parallelism. The Linda model offers two components, a computation language and a co-ordination language, which operate at a similar abstraction level, promoting portability and also allowing exploitation of reusability and heterogeneity. Gelernter [23] describes different parallel programming styles in terms of their generalisation or specialisation of the base language. The Linda model shows parallelism as a specialisation of a more general co-ordination problem, whereas in traditional parallel extensions to sequential languages, parallelism is a generalisation of the underlying computation problem.

Linda, as well as being described as a co-ordination language, is often referred to as a parallel programming model in its own right. This is due to the virtual shared memory paradigm adopted by Linda, in the form of the tuplespace. This shared data area provides a means of process co-ordination by the exchange of data tuples. A tuple is a record of heterogeneously typed data, for example, `["name", x, 0]` comprises 3 fields, a literal string, a variable (which may be of type integer, say), and the literal constant 0. A tuple may consist of up to 16 fields of variously typed data, either as passive tuples - static data - or process (live) tuples which are evaluated by the `eval` command. Linda is based on a distributed data structures model [47] in the sense that the components that make up the memory may reside on different processors. However, to the programmer and the processes created in the program, the memory appears as a shared data space - the tuple space - with no regard to the physical location of the memory components. The tuple-space model is a convenient model for parallel programming, allowing the concepts of shared memory to be used as an abstraction of the underlying parallel architecture. This abstraction lends itself well to the implementation of some graphical parallel programming approaches which are

discussed in a later section. Before that, some of the broader concepts of graphical programming are explored.

2.7.6 Graphical Programming

Graphical, or visual, programming is an unconventional and abstract style of programming which uses graphical objects in place of text to specify either the computation, the data relationships, or sometimes both. Although its relevance to the preceding topics is not immediately apparent it will be shown that a combination of features of a number of diverse programming techniques, including graphical programming, can offer an interesting and useful new methodology for parallel programming. This section will first examine the ideas behind graphical programming, and then go on to investigate the application of graphical programming to dataflow and co-ordination languages. The following section will present the programming model MeDaL that adopted some of these features.

The main motivation behind graphical programming is the desire to raise the abstraction level of programming languages, and to try to capture the requirements of a programming problem at a level more in tune with the original problem rather than at the target architecture level. Graphics are a good way of achieving this, and in fact some languages influenced by graphical techniques have no textual component at all. Ichikawa, for example, proposes *iconic programming* as a collection of icons whose relationships are indicated by overlapping [48]. For instance, an accounting program may have a book icon and a calculator icon, and the interaction of these two, shown by placing the two icons so that they touch, might represent a function which performed calculations on the accounts book.

The advantage of the graphical approach is that pictorial representations are often easier for the human brain to identify. This leads to a quicker and increased comprehension of the specified program. However, using graphical objects at a program statement level actually increases the complexity, due to the difficulty of expressing fine detailed computations with pictures. It is difficult to find graphical representations of items that are not inherently visual, and sometimes pictures can be more ambiguous than text, open to a wider interpretation. Additionally, graphical representations tend to take up much more screen space, although adopting a hierarchical scheme can help this. The precise and analytic nature of a textual description seems more suited to a detailed computation specification, whereas graphical approaches have been used successfully for many design notations.

The advantage to parallel programming of graphical techniques is the implicit and easily distinguishable representation of parallelism. Whereas text follows a sequential pattern by convention, spatial arrangement of graphical objects does not have the same interpretation. Objects placed at an equal vertical position, but separated horizontally, imply independence until connections are added. This is why graphs such as dataflow graphs are often used to show data dependency (which implies parallelism). Traditionally, dataflow graphs also represent the computation at a fine granularity, indicating the dependency between data elements, but more recent approaches have adopted a higher-level graph, indicating dependencies between sections of code such as functions. The advantage here is that the useful properties of text can be exploited at a fine granularity within the function, while expressing parallelism at a higher granularity using a graph.

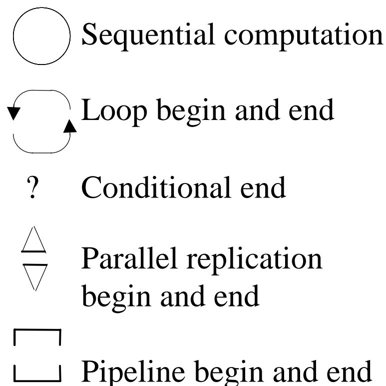
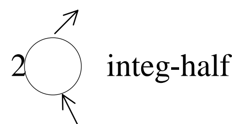


Figure 2-8 Hence Node Icons




```
NODE [118 257] 2
      < double a;
      < double mid;
      < int n;
      s = integ-half(a, mid, n);
      > double s;
```

Figure 2-9 Hence Computation Node with Annotation

A number of parallel software tools, based on a variety of programming styles, have adopted the use of a hybrid text and graphics notation, in a graph format. One, HeNCE (Heterogeneous Network Computing Environment) [49] [50], based the node connections on execution dependence, with some symbols for lower-level control features (e.g. looping, conditional branches). The *computation nodes* were refined by specifying a sequential subroutine, together with input and output declarations. The subroutine code was written in a variant of C or Fortran, and the resulting program was automatically translated into a parallel executable PVM program. Figure 2-8 shows the HeNCE node icons and Figure 2-9 gives an example node with its associated annotation.

CODE (Computation Oriented Display Environment) [51] [52] is an alternative parallel software development tool which also used a graph of nodes refined textually. CODE used dataflow to specify the relationships between nodes, although control flow was dictated by a set of different **firing rules** which indicate under what conditions a node can execute. Like HeNCE, CODE produced a PVM parallel program, in which the parallel mechanisms for process creation and message passing are contained. However, CODE required the use of annotations to name data ports, and to specify rules for firing and routing. In this sense, CODE is merely exchanging one form of parallel commands with another, although much of the low-level detail to actually transmit and receive messages is hidden from the programmer. Figure 2-11

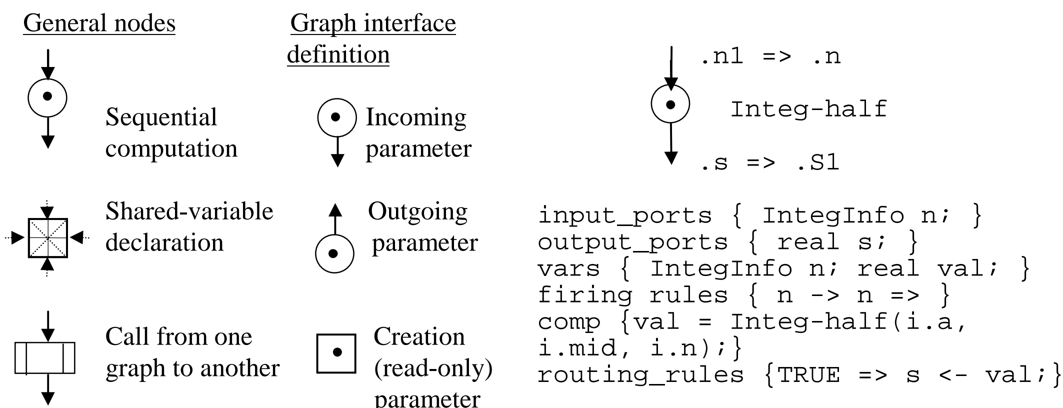


Figure 2-11 Node Icons in CODE

Figure 2-10 Computation Node in CODE with Annotations

and Figure 2-10 show the node icons and the node annotations for an example node in CODE.

Both CODE and HeNCE based their methodology on what can be considered to be a **graphical co-ordination model.** Where Linda separates the computation from the co-ordination of processes by providing an additional set of commands for process creation and data transfer, the two graphical approaches described replaced the co-ordination aspects with the graph structure. Neither CODE nor HeNCE completely replaced the textual co-ordination mechanisms, as both used additional text features to aid the description of data passing methods, but the concept of using graphics to define parallel structure was very successful.

The developers of CODE and HeNCE jointly examined the concepts of visual programming for parallel software development [53]. They recognised the benefit of graph structures for specifying parallelism by providing direct but abstract methods of expressing synchronisation and communication. However, the level of abstraction varied within the CODE and HeNCE language environments, and a coherent methodology for the design and implementation of parallel software was not clear. The joint investigation identified the sequential, textual components as the base of the design, and described a composition of these components into a parallel program. This represents a bottom-up rather than a top-down development, with the sequential components forming the building blocks of the parallel structure, rather than being refinements of this structure.

The use of abstraction in CODE and HeNCE, particularly of the data and computation interface, is an important feature which aids the programmer in developing parallel programs more easily. In addition, the usefulness of automatic translation of these abstract graphical mechanisms into low-level parallel code was recognised by CODE and HeNCE as a means of hiding the complex, architecture-dependent parallel mechanisms. However, they retained some elements of low-level detail in the description of the graph components which detract from the aim of a coherent abstraction level. Furthermore, issues such as data partitioning across multiple processes, or the replication of data or processes to exploit parallelism, were not considered. An assumption that the programmer would still need to describe these textually within the sequential subroutine was made, whereas issues of distribution or replication are relevant at the interfaces between subroutines (graph nodes) and not within them.

A coherent abstraction level and structured design methodology can be seen as two areas of inadequacy in the CODE and HeNCE developments. This thesis proposes a more structured top-down approach to the development of parallel programs, and the removal of all low-level specifications of data and computation interfaces. Subsequent chapters will address these issues in detail, but a brief outline of the proposed methodology is presented, before a detailed description of another graphical programming approach which more closely captures some of these ideals.

Software development must begin with an abstract description of the programming problem, and refine it in a set of steps to the final implementation. A graph describing the parallel structure of a set of smaller computations is the more abstract representation, and is comparable to a design of the parallel program. The textual description is a refinement of this design, where components are broken down into

more detailed computations. The interaction between the graph nodes at the higher level should be part of the design, specifying the interface between components. Rather than being a low-level detail, communication between nodes should be represented at a high level, and should only consider the source, destination and content of the data. Other details such as where and when data is transmitted is already implicit in the graph structure, and should need no further consideration. Each node (computation) should be treated as a *black box*, only concerned with its own inputs and outputs, and the graph should present a complete, but abstract definition of the program.

The following section describes a notation that could be adapted to support the methodology described to present a coherent and complete environment for parallel software development. Subsequent chapters will show how some of the features of the notation described below have been adopted in a new graphical design system, ParaDE.

2.8 MeDaL - Medium-Grained Dataflow Language

A graphical notation for specifying parallel programs called MeDaL (**Me**dium-Grained **Da**taflow **L**anguage) was developed at Newcastle [2], with the aim of providing an abstract graphical notation at a medium-grained level. Features for exploiting the shared memory characteristics of the target architecture were developed, and a process for automatic generation of parallel mechanisms was proposed. The MeDaL notation adopted a dataflow-style graph at a medium-grained level, where nodes represent functions written in a sequential language, and arcs represent the flow of data between these functions. The graph notation captured the parallel structure of the program implicitly by specifying the data dependencies between the nodes. This section gives a brief overview of the MeDaL notation and concepts, before presenting ParaDE - a modified notation and graphical parallel programming environment - in the next chapter.

2.8.1 MeDaL Concepts

The MeDaL notation had two basic elements: **actors** represented by a small number of different nodes on a graph; and **datapaths** represented by directed arcs connecting the actors. The graph was read from top to bottom and allowed data to flow in one direction only, indicated by the arrowhead on the arc. Loops were permitted in the notation, and the graph could be developed in a hierarchical manner. This hierarchy was specified by the use of **companies** - nodes on the graph which represented an expandable subgraph.

Actors, with certain exceptions, **fired** (were executed) following the strict firing rule which states that all input data must be available. Actors continued to fire while data was available on their input datapaths, either consuming or reading the input data, depending on the type of datapath. An actor which had fired and consumed its input data would be disabled if there was no more input data, but on the arrival of new input data would be re-enabled.

The content of an actor - its **method** - was not described in the graph, but had an associated section of code, written as a function in a sequential programming language. The input and output datapaths corresponded to data variables within the

method code, providing input data values and producing output data values. A small set of extra commands provided the mechanisms to set up and transmit data on a datapath, implemented as functions of a class containing the output data as private variables. Certain special purpose actors had no method, but represented a specific predetermined function, e.g. to read data in from a file. Some of these special purpose actors also fired under slightly different rules, to allow their specific function to be carried out.

2.8.2 Datapath Notation

The MeDaL notation had two types of datapaths, called E-type and F-type paths (meaning empty and full). The datapath symbols are shown in Figure 2-12.

Figure 2-12 E-type Datapath and F-type Datapath

The difference between the two is that E-type paths - the most commonly used - provided data which was consumed by the actor, whereas the data in an F-type path was only read by the actor, and remained at the head of the datapath queue until overwritten by a new value. This was one mechanism for providing persistent data, as the actors, following the dataflow model, did not have any internal state. Datapaths were represented by a directed arc, with either an empty or filled arrow head, corresponding to E and F-type paths respectively, and were implemented as a FIFO queue. The data on a datapath was required to be of the same type, but structured data types were permitted to group different types together.

2.8.3 Actor Notation

MeDaL provided six types of basic actors: two general-purpose actors, and four special purpose actors. The general-purpose actors had their behaviour specified by the user-supplied method code, and had one or more inputs and one or more outputs. The standard general-purpose actor fired whenever data became available at the head of the datapath queue. The **deep** actor, however, could fire multiple times to execute several copies of the method simultaneously. The notation for a general-purpose actor was a round-edged box, with a thicker border to represent the deep actor. The general-purpose actors are shown in Figure 2-13.

Figure 2-13 General-purpose Actors

Special purpose actors were provided to represent a small number of commonly used features with special behaviour. The **source** and **sink** actors provided a facility for introducing data into the start of the program and ending the program respectively. A source actor had no inputs and fired once at the start of the program, either executing a supplied method or producing a Boolean value true if no method was supplied. A MeDaL program needed at least one source actor in order that the program could start

execution. A sink actor had a single input which was consumed by the actor and discarded, unless a method was provided to carry out some termination behaviour, perhaps to release memory allocated for the data item. Figure 2-14 shows the symbols for the source and sink actor.

Figure 2-14 Source Actor and Sink Actor

The other two special purpose actors were concerned with the combination or replication of datapaths. The merge actor had at least two inputs, and fired when data was available on any one of the inputs. This single input value caused the merge actor to fire, transferring the data to its output path. This provided a means of combining datapaths, so that the output datapath could carry data from a number of sources. If more than one input path carried data to the merge actor at the same time, the merge actor would fire once for each input, in an undefined order. The replicator implemented the inverse behaviour, copying the data on its single input datapath to all of the output paths, of which there had to be at least two. For both of these actors, the input and output datapaths were required to carry the same type of data. The symbols are shown in Figure 2-15.

Figure 2-15 Merge Actor and Replicator Actor

2.8.4 Company Notation

The company symbol, shown in Figure 2-16, represented a subgraph which could be expanded to show all the actors in the subgraph. It could be used in place of a part of the main graph, to allow a hierarchical view. Company inputs and outputs must match the corresponding datapath connections to the subgraph.

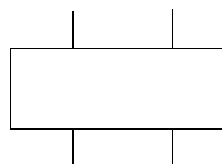


Figure 2-16 Company

2.8.5 Persistent Memory

The use of the F-type data path for persistent memory has already been discussed. MeDaL provided another form of persistent memory within the method code. In place of the *send* function to transmit the data, a different function *sendSticky* could be used

which placed the data on the output datapath, but didn't transmit it until a *send* call was made. This allowed the formation of a data structure from data components which may have arrived at different times. The provision of persistent data was an attempt to provide the useful feature of persistent state within a dataflow notation which would not normally allow persistence, to preserve freedom from side-effects. This issue will be returned to later.

2.8.6 Implementation of the MeDaL Notation

The MeDaL notation was intended to be part of a programming system for parallel software development. The structure of such a system was envisaged by the designer of MeDaL [2] to have a number of components, as shown below in Figure 2-17.

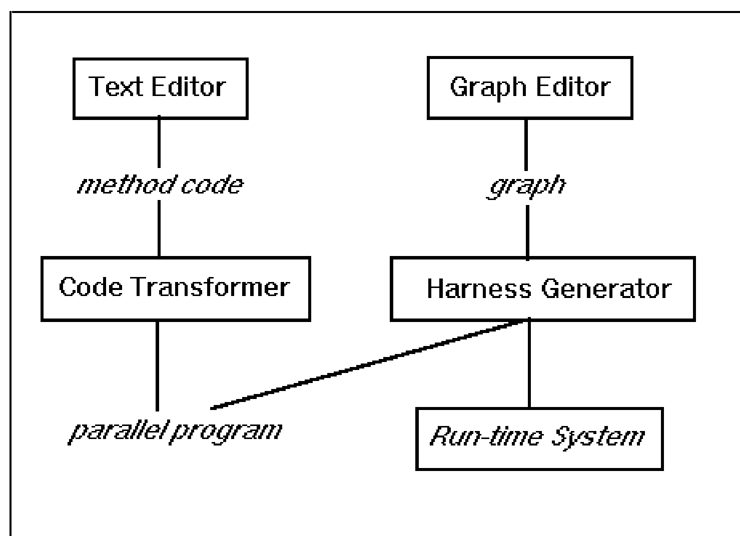


Figure 2-17 Envisaged Programming System for MeDaL

Of these, only the run-time system was implemented, with the graph, method code, and parallel program being produced by hand, but in a regular form that could be automated. The main focus of that work was the MeDaL notation for describing parallel programs in the form of a graph with corresponding text for the method code, and the run-time system for supporting the execution of such a program.

While the concepts for automatic generation were considered, and ideas were proposed for the production of portable parallel code, the actual implementation of MeDaL programs still required the use of several special commands to facilitate the setting up, transmission and receipt of data between the actors. Additionally, the user was required to use a set of specially designed data types, based around those of the host sequential language. These were implemented as classes in the C++ language, enabling the use of object-based functions to manipulate the encompassed data types. The content of the *wrapper code* which encased the method code written by the user was considered, requiring the provision of the means to retrieve input data from datapaths and pass this data as input parameters to the function containing the method code. A similar mechanism for the output of data was required and this was produced by hand in the MeDaL prototype system. These facilities were intended to be part of the harness generator and/or code transformer, for the run-time and compile-time support respectively. The implemented run-time system recognised a given format of program statements produced manually from the graph notation and input and output interface specifications. These program statements invoked run-time functions

implemented using parallel programming mechanisms provided by the target architecture. This run-time system was machine dependent, implemented on an Encore Multimax shared memory machine, based on a threads style of programming.

2.9 Parallel Programming Summary

The early part of this chapter introduced some key concepts and issues in parallel programming - the nature of parallelism and the techniques required for programming in a parallel style of programming. Subsequent sections have described the variety of traditional methods of parallel programming. A number of important problems were identified in these methods, and some novel approaches to programming, particularly graphical programming and dataflow were introduced as significant elements in a new era of programming styles. The MeDaL notation and implementation environment has been introduced as an important step in the evolution of graphical programming languages for parallel programming, and some key features of MeDaL have been described. However, MeDaL is by no means an ideal solution to the problems addressed in this chapter, and the next two chapters explore ways in which the MeDaL work has been adapted and transformed to provide ParaDE, a practical design system for parallel programming.

3. Design Notations for Parallel Software

3.1 Introduction

Chapter 2 described a number of topics related to parallel programming, leading up to recent methods using graphical programming and dataflow. In particular the MeDaL parallel programming notation and two similar notations, HeNCE and CODE were discussed. This chapter re-examines the philosophy behind MeDaL, HeNCE and CODE, and evaluates the success of MeDaL with comparisons to the other two notations. Based on the experience of using MeDaL to develop parallel software, the problems of the MeDaL notation are identified, and alternative solutions are discussed. This chapter then introduces ParaDE, a modified notation based on MeDaL, which has been developed through the experience of practical use of the MeDaL notation in designing parallel programs. Differences to the MeDaL approach are described, and the significance of these changes to parallel software design is discussed.

This chapter begins with a discussion of the motivations behind the development of graphical programming notations, and outlines the main aims which influenced this research. These aims are based on an evaluation of current software development tools, and experience gained through the use of such tools in parallel program development. A number of key factors, important in the design of graphical notations for parallel programming are identified. The way in which different notations address these factors is explored in Section 3.2, before Section 3.3 outlines the initial features of the modified notation and Section 3.4 proposes changes to MeDaL features. Section 3.5 addresses specific problems with the use of the MeDaL notation in developing parallel programs and Section 3.6 proposes alternative features for ParaDE. Finally, Section 3.7 describes the new graphical environment developed to encompass ParaDE, and how this environment complements the notation to support an abstract methodology for parallel software development.

3.2 Motivation

Notations for parallel software development in the graphical style of MeDaL, such as HeNCE and CODE introduced in Chapter 2, have a number of common aims, although the approaches to satisfying them may be different. This section examines the aims of this style of programming and determines the success with which each notation has achieved them.

Perhaps the most important aim is to capture the parallel structure of the program - the process co-ordination - in an abstract graphical form. The concept of a graph with nodes and arcs is widely used in a number of fields in computer science, and is adopted by graphical parallel programming notations. The benefits of the graph style have been discussed in the previous chapter, but the primary advantages are the implicit description of parallelism through node layout and the clear representation of communication routes by the use of arcs joining nodes.

The suitability of the graphical form chosen to represent parallel programs depends on a number of issues :

1. the level of abstraction,
2. the type of programs to be developed,
3. the programming model adopted,
4. the target architecture,
5. the criticality of efficient execution,
6. the experience of the intended user.

It is assumed that efficient execution is always critical, given that speed is the motivation for using parallel programs, and as this study is concerned with machine independent software, point four will also be largely ignored at the notational stage analysed in this chapter, but investigated in detail in the next chapter. A further assumption is made that the intended user is experienced in programming, but not with parallel programming, and that general-purpose programs are to be developed. This fits in with the notion of providing a general, abstract notation, avoiding current complexities with parallel languages, as described in Chapter 2, and opening up the field of parallel computing to programmers in general.

The issues of programming model and level of abstraction remain, and these are key to the design of a parallel programming notation. The way in which different notations address these factors is now explored.

3.3 Different Notational Approaches

MeDaL set out to provide a modular notation suitable for describing parallel programs at a medium-grained (sub-program) level, where a program was composed of tasks (actors) interconnected by arcs (datapaths), the latter representing the inter-task data dependencies. The notation did not attempt to address the functionality of the actors themselves. Instead this was left to the textual description of the actors. A small set of actor types was defined, providing a representation of both general-purpose actors, and some special purpose functions. The latter, including the merge and replicator actors, were intended to allow the user access to a library of useful functions, whilst maintaining efficient implementation and remaining within the dataflow firing rules (which require the presence of data on all input datapaths for general-purpose actors). The functions provided were chosen as a trade-off between efficiency and generality, allowing the user enough flexibility to describe a general-purpose program without over-complicating the notation with unnecessary detail, that is, choosing an appropriate level of abstraction for the medium-grain programs which were targetted by MeDaL.

The differences between MeDaL, HeNCE and CODE is in the interpretation of the connecting arcs and the provision of special purpose nodes to represent program features commonly available in more conventional styles of programming. These relate to the two issues identified in Section 3.2 of programming model - the general semantics of the graph, and level of abstraction - generality versus speciality of the functions provided. The latter are the most obvious of the differences between notations and will be examined first.

1. HeNCE provides loop begin and end node icons, as well as conditional, pipeline and parallel replication, as shown in Figure 3-1. These features operate in pairs, to provide control structures that would be present in a textual language.

   In contrast, CODE has no control nodes, its only additional nodes are involved with data and hierarchical graphs, shown in Figure 3-18.

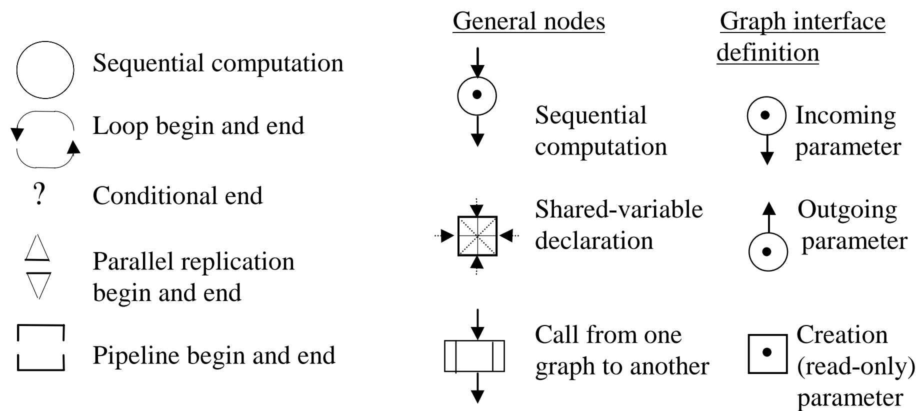


Figure 3-1 Hence Node Icons

Figure 3-18 Node Icons in CODE

   MeDaL offers a limited set of control nodes, primarily merge and replicator actors, which combine or split datapaths to allow the strict firing rule to be maintained at the actor inputs. CODE adopts a different approach to firing, supporting the specification of different firing rules, which encapsulate control features represented by the merge and replicator actors of MeDaL within the node interface. This may be seen as incorporating parts of the parallel co-ordination within the nodes, rather than MeDaL's approach of keeping the co-ordination structure distinct from the computations within the nodes. HeNCE nodes have no firing rule as such, as explained below.

2. The issue of where to include control features is intertwined with the semantics of the arcs connecting the nodes. CODE and MeDaL both operate under a dataflow scheme, where arrival of data on the input arcs of a node causes it to execute. CODE differs from MeDaL in providing flexibility in the firing rules, where MeDaL adopts only a strict firing rule. HeNCE arcs, in contrast, do not represent dataflow. They indicate an execution ordering between nodes, so that a node can fire when all its predecessors have fired. There is no data involvement with the arcs in HeNCE; they represent purely control features, and thus the control nodes provided by HeNCE fit in with the control-based view of the program.

   The control nodes in MeDaL are provided partly out of necessity to allow certain behaviour within the program, but also provide a convenient way to allow optimisation of the implementation of useful program features. In other words, abstraction can be used to provide common functions which may be complicated to implement manually. The provision of appropriate useful functions would seem to be a more important aim than adding extra "patches" to maintain the firing rule. However, by doing so, MeDaL can manage with a single, straightforward firing rule, leading to an easier understanding of the behaviour of a graph program. In contrast, the firing of a node in CODE cannot be determined at the graph level, as the rule is specified within the textual interface to the node. HeNCE graphs are not

concerned with dataflow, but merely an execution ordering, so in this sense their behaviour is clear at the graph level, although the graph becomes purely a control description of the program, as data details are hidden within the node. In addition, MeDaL was aimed particularly at a design level, incorporating hierarchy with the company feature, whereas the other notations were largely concerned with a lower-level, implementation specification.

3.4 Initial Proposal of a Notation

This thesis supports the MeDaL approach of providing a simple behavioural description using a single firing rule and limited set of control nodes. Behind this stance is the desire to present the graph by itself as a design-level description of the program, with the textual description serving as a refinement of parts of the design. In order to support this concept of a design, the notation must be able to provide a clear behavioural description, without having to resort to low-level (program-statement-level) programming details. Additionally, such a design description would involve both data and control features, although data would be the primary factor, as control issues are often concerned with implementing the data behaviour in a given programming language. The intention is to concentrate on pure design details at the graph level, and try to avoid consideration of issues which fall naturally into later stages of the software development cycle.

In keeping with the design concept, the separation of the computation details from the co-ordination details is important. The computation description of a node can be viewed as an implementation, or further refinement, of an object in the graph, whereas the graph itself encapsulates purely the co-ordination of computations. In order to isolate the co-ordination, the firing rule must be predefined, or implicit in the graph structure, and not part of the node description.

While the co-ordination of computations may be considered to be a control issue, in design notations it is often the flow of data between computations which defines the co-ordination, and in fact this approach retains the *natural* co-ordination behaviour, implicit in the algorithm. For this reason, dataflow is adopted as the behavioural model. This might lead to questioning the necessity for any control nodes, but in fact the merge and replicator actors in MeDaL are merely graphical conveniences for manipulating the junction of datapaths of equal type, and therefore can be redefined as **data junctions**, and distinguished from other nodes (actors) by their lack of computation specification (method code).

3.5 Changes to MeDaL Notation

Most of the basic MeDaL notation is adopted in the ParaDE notation, but several important changes have been made. These changes, and their justification, are described below.

3.5.1 General Visual Syntax

1. The deep actor has been renamed the depth actor and its symbol has been replaced with a three-dimensional symbol as shown in Figure 3-2, to illustrate more clearly that this actor is replicated and operates in parallel. The “depth” of the actor - its parallelism - is indicated by the multiple layers of the actor in the third dimension.

This symbol can be visualised as a number of general-purpose actors layered together, to create an associated group of actors. Only a single method code can be supplied, and datapath input and output is connected to the group as a whole, reinforcing the idea that the code is replicated over all the instances of the depth actor. Datapaths which have partitioned data (described later) are shown with multiple arrowheads, illustrated by the sole input datapath in Figure 3-2. Other, non-partitioned, datapaths can also be connected to the depth actor.

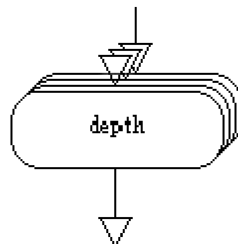
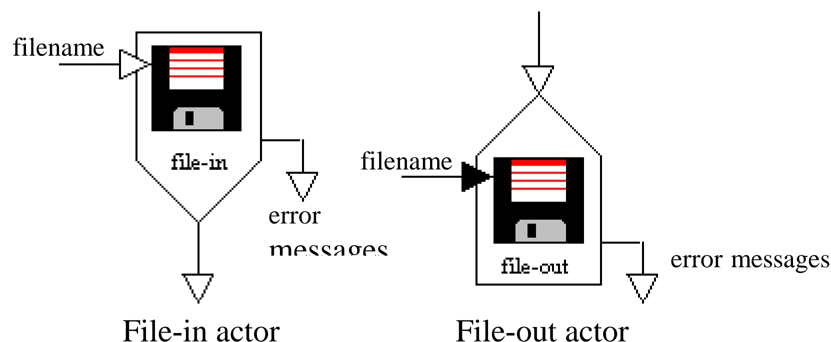


Figure 3-2 Depth Actor

2. The company symbol has been abandoned, in favour of a more general decomposition concept, whereby any actor which can have method code supplied can also be decomposed into a sub-graph (details are discussed later). This is seen as a simplification of the notation, as the question of what the contents of an actor are - whether method code or sub-graph - should not be an issue at the level of the actor symbol. This concept corresponds to a "black box" approach, where each actor (box) is considered as a single object with inputs and outputs, until the lower level is entered.

3. Icons have been added to some of the library actors, to give a better indication of their function to the user. For example, the Standard Output actor has an icon of a computer monitor to represent output to the screen. The file input and output actors have been redesigned as specialisations of the source and sink actors, rather than general-purpose actors to reflect their input and output behaviour. In addition, the filename input datapath, although maintaining the same semantics as in MeDaL, has been redesigned to point to the label of the floppy disc icon, representing the place where a filename would be written. Similarly the output datapath for error messages is now shown connected to the side of the actor. This position of the connection indicates that it is a datapath carrying exceptional information, rather than the normal data input/output to/from the tip of the actor - the standard point of connection for a source/sink actor. These changes have been made to visually clarify the characteristics of each of the library actors. The modified library actors are shown below in Figure 3-3.



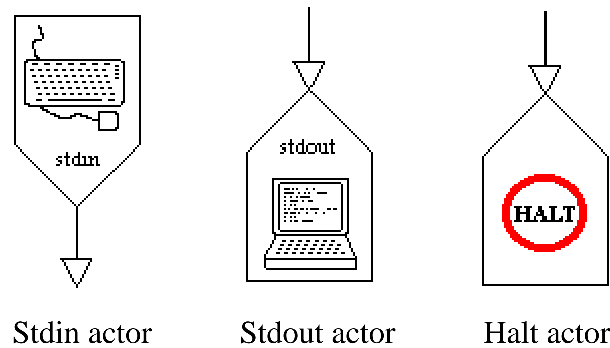


Figure 3-3 Library Actors

4. The Merge symbol has been modified to avoid the problems of the old symbol being associated with a logical AND symbol, implying significantly different semantics. The new symbol, shown below in Figure 3-4, is more akin to a logical OR symbol, which more closely corresponds to the behaviour of the merge node.

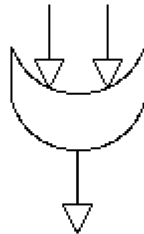


Figure 3-4 Merge Node

3.5.2 Naming Changes

1. The names of the datapath types have been changed from E-type and F-type to Discrete and Continuous respectively. These new names are adopted from Real Time Structured Analysis notations, and are more descriptive and representative of the character of the datapaths. A discrete datapath carries a single (discrete) data item to its destination actor, whereas a continuous datapath holds a data item (continuously) at the head of its queue, to allow the destination actor to repeatedly access it.

2. The merge and replicator actors have been redefined as data junctions to distinguish them from other actors, as described in Section 3.3.

3.5.3 Decomposition Details

The company concept is replaced by actor decomposition, which can apply to any actor allowing method code. Whereas inputs and outputs to a company were required notationally to match the inputs and outputs to the exploded graph both in position and type, actor decomposition offers a more flexible approach, better suited to the graphical programming environment.

An actor which is decomposed into a sub-graph has a set of interface nodes provided, corresponding to the input and output datapaths connected to the actor.

Figure 3-5 Input Interface Node and Output Interface Node

Input and output interface nodes have no special characteristics of their own, but serve merely as a notational connecting device from datapaths on the current graph to datapaths in a sub-graph. Interface nodes are generated automatically by the system when an actor is decomposed. An input interface node is created on the sub-graph for each input datapath connected to the actor, and an output interface node created for each output datapath. Interface nodes do not fire, and add no new behaviour to the program. They do not interrupt or affect the flow of data. Input and output interface nodes look like source and sink nodes respectively, except that they are smaller and are a solid black colour (see Figure 3-5). The reason for the interface node design is that they represent input and output, in the same way that source and sink actors do, but do not justify a complete actor definition (full sized node with labels and picture) as they are merely a connecting device and introduce no new input or output. An example of a sub-graph is shown in Figure 3-7, where a method actor *actor1* has been expanded into a sub-graph containing four actors, with an interface node created for its single input and output connection to the parent graph. The parent graph is shown in Figure 3-6. These diagrams, and those that follow, are screen-shots captured from the prototype implementation of ParaDE that is described in Section 3.8.

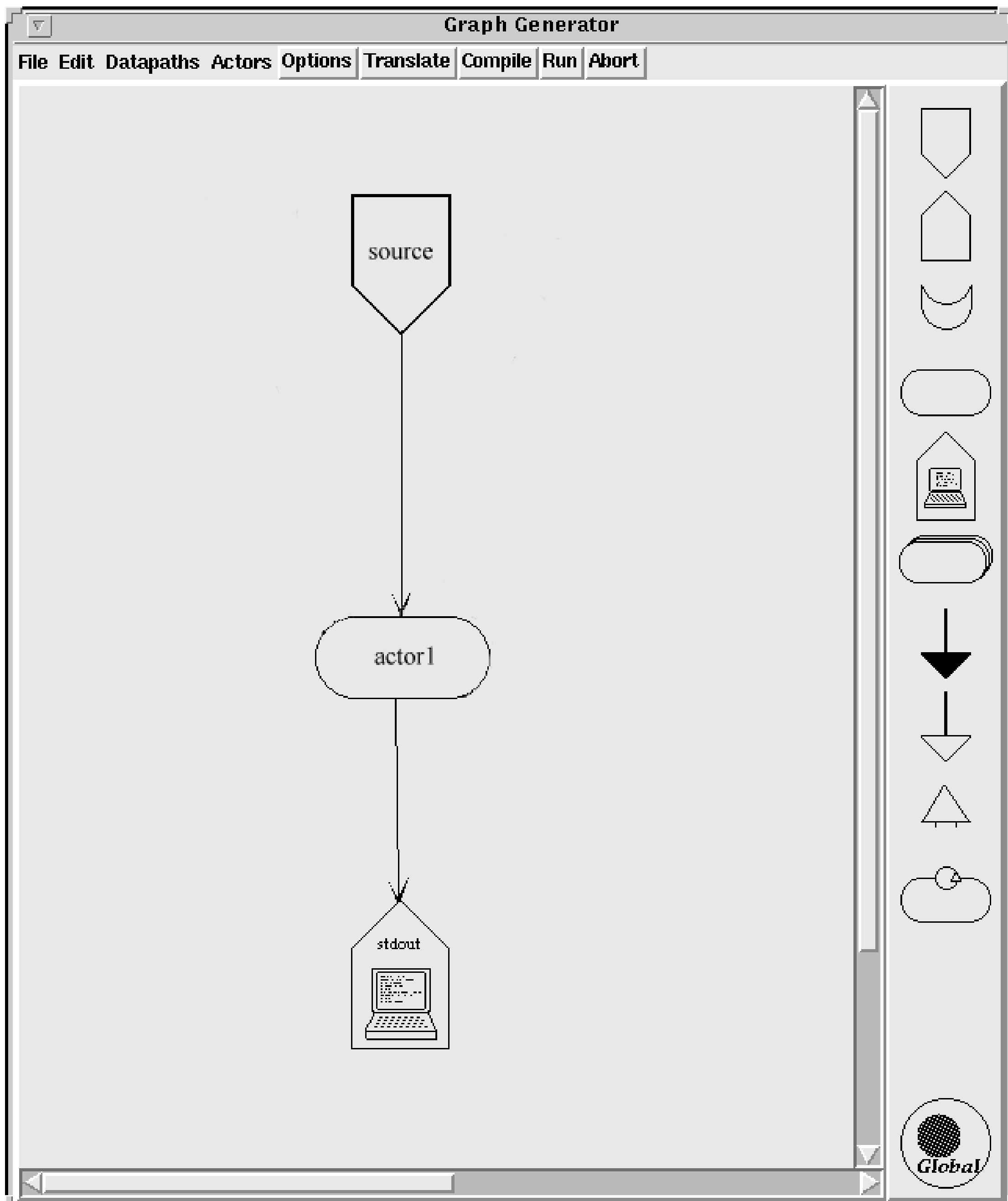


Figure 3-6 Top-level Graph

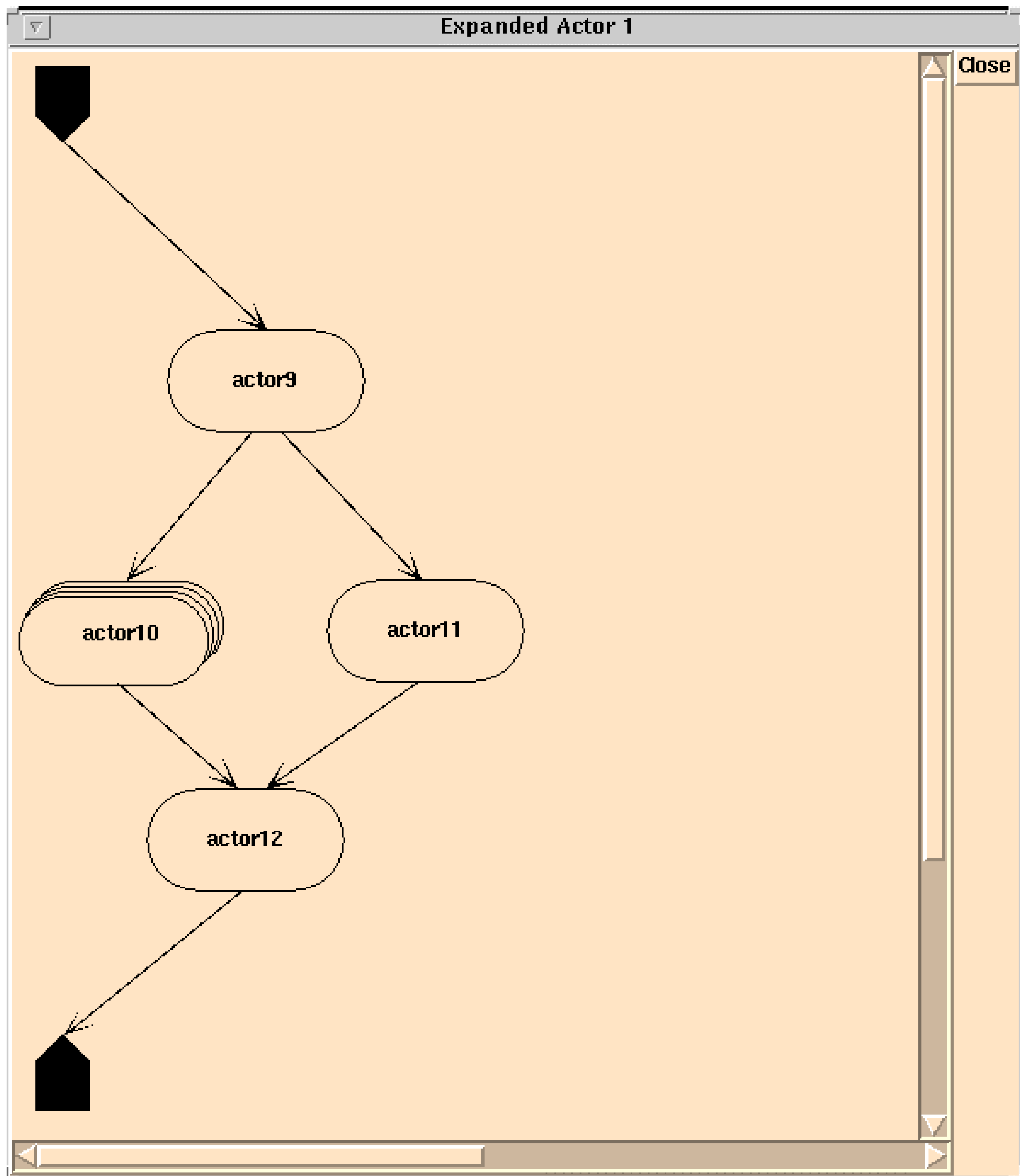


Figure 3-7 Decomposition of an Actor into a Subgraph

3.6 MeDaL Problems

3.6.1 MeDaL-Specific Language Extensions

The MeDaL notation was intended to support machine independence, although the implementation was targeted specifically at a shared memory architecture with some features specific to this architecture. Apart from these particular features, machine independence can be seen to have been broadly achieved, encapsulating representations of the mechanisms required for parallel execution within the graph structure. Automatic code generation was intended to produce executable parallel

code for the given target architecture, and this thesis examines the issue of automatic code generation in Chapter 4, which was simulated by hand coding in MeDaL. This hand coding showed that a parallel program specified in the MeDaL notation could be matched with a corresponding program executed in association with the MeDaL run-time system to give parallel behaviour.

MeDaL provided abstractions of the parallel mechanisms, avoiding the necessity for machine-specific features to exploit parallelism, and in this sense machine independence was achieved. However, to achieve true portability, the program specification must not only be independent of the underlying machine architecture, but also independent of any machine-specific run-time system. If this is not the case, then machine dependence is not avoided, but merely hidden within an extra layer of software. MeDaL was seen to be strongly linked to its run-time system (which was written in using a threads package specifically for the Encore Multimax), through the use of special MeDaL data types and language extensions. The data types, restricting the flexibility of the sequential textual language used to specify the method code, consisted of two basic types for Boolean and integer types respectively, and a single template type for a generic array type. These data types were implemented as C++ classes, and only within these classes could the user gain access to the required commands to facilitate the set up, transmission, and receipt of data between actors and datapaths. The commands provided for the MeDaL data types, and requiring user access, included *extend* to initialise the size of an array type, and *send* and *sendSticky* to cause the transmission of the data along the datapath. These commands are additional to the basic language adopted for the specification of the method code, and can be seen as language extensions, in a similar way, for example, to the use of *forall* in parallel Fortran implementations.

Figure 3-8 shows an example of some of the MeDaL commands used to interface method code and datapaths. An output datapath, `out0`, is declared as an integer array (using the special MeDaL datatypes) in the heading of the method code. Within the code body, the first line `out0.extend(100)` makes a call to a run-time system function which initialises an appropriate array of size 100. The variable `out0` is then used conventionally, in this case just assigning 0 to each element, then a call is made to another run-time system command, `send`. This final command performs the transmission of the data contained in `out0` conceptually travelling along the specified datapath to the destination actor.

```
void company0_actor0(Mary<int> & out0)
{
  out0.extend(100);
  for(i=0;i<100;i++)
    out0[i]=0;
  out0.send();
}
```

Figure 3-8 Method Code using MeDaL Commands

The disadvantages seen in this example are that the user has to understand and use the additional commands `extend` and `send` to use the datapath interface, and is also restricted by the datatypes implemented by MeDaL. These language extensions are merely practical devices concerned with convenient implementation, and not closely related to the parallel execution mechanisms, so their existence is tolerable in what was a prototype development tool. However, any such language extension is undesirable in a software tool aiming at portability, and this thesis proposes that the method code should be specified purely in terms of the chosen sequential language, using all of the conventional facilities for data types and computation expected by a typical "sequential" programmer. The ParaDE design system avoids all use of extensions, hiding the actual data transmission commands from the user. Instead, these commands are generated automatically by the translation tool, which also extracts information about the datapaths from the graph in order to automatically insert declaration and initialisation code.

3.6.2 Data Persistence

One of the major concerns during the development of MeDaL was the representation and implementation of shared data within a program. This is a difficult issue, as the dataflow paradigm does not strictly support such a feature, partly because it would violate the freedom from side effects. However, its presence in general programming is essential, and, particularly with the shared memory platform on which the MeDaL research was based, could be exploited to great effect.

As already described, MeDaL offered two forms of so called persistent memory, the first being the full or F-type datapath (renamed continuous), which is a graphical feature enabling persistence of a data input into an actor. The second is incorporated into actor method code in the form of the send-sticky command. This retains the state of an output data path, in order that complex data can be built up from simpler data types. The choice of these two is well justified, and seem to provide the necessary features. However, in using the full path and send-sticky in experimental programs described below, it was not immediately apparent as to how to implement the program's algorithm with the given features. Although the MeDaL notation, at first sight, seems to offer an intuitive design methodology, when it came to implement the design of the test program (which was done by hand originally, as no automatic generator existed), it took a number of attempts to produce a working solution of even a very simple example. It was clear that a good understanding of both the persistent memory features, and the wider dataflow principles, such as firing rules, was required before software development using MeDaL became a comfortable task. It is possible that the encompassing graphical programming environment subsequently developed (described in Section 3.7) could give assistance to the programmer, reporting for example on misuse of datapaths which may prevent actors firing. However, some issues remain which still leave software development using MeDaL less than easy.

An example was developed, using the MeDaL notation, which implemented a bitonic merge sort [54]. This involved sorting a list of n numbers, by repeating a process of shuffling the list and a pairwise compare and swap between neighbouring elements. This was repeated $log(n)$ times to produce a sorted list. Although not a complicated example, it required the use of many of MeDaL's features. The pairwise comparison and exchange was carried out in parallel using a depth actor, and the iteration required

a cyclic graph, using a merge node and both forms of persistent memory. The final MeDaL graph is shown below in Figure 3-9.

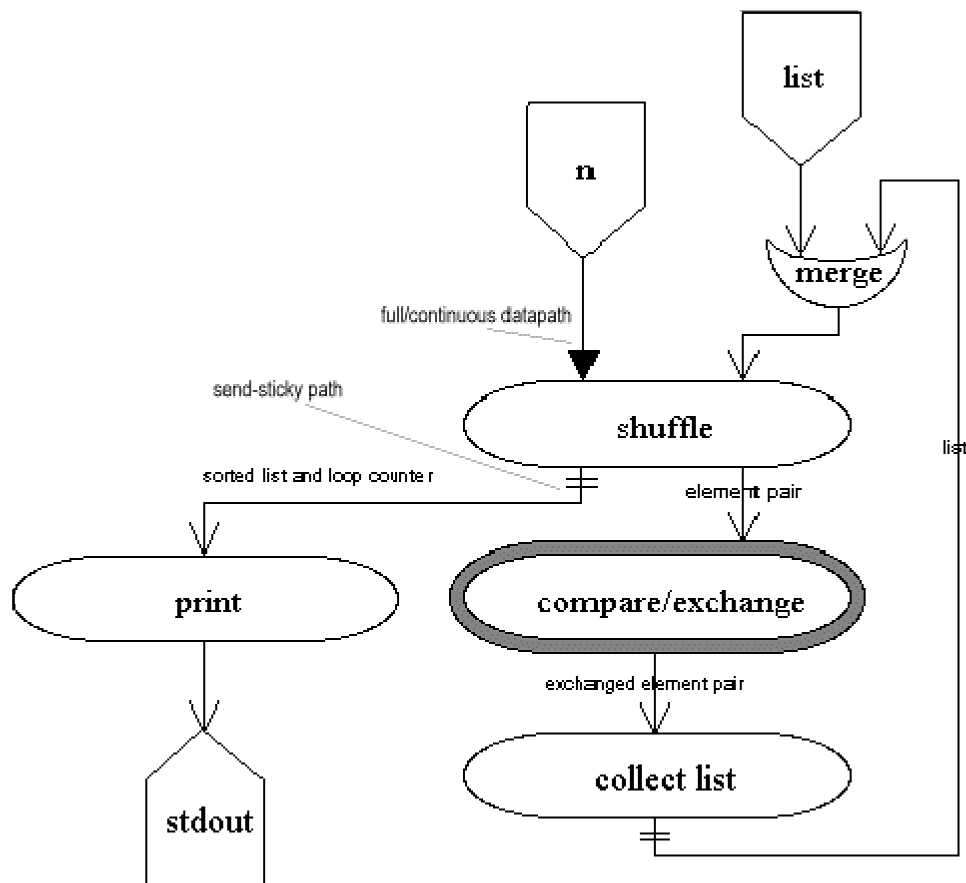


Figure 3-9 Bitonic Merge Sort

The first point to note is that a temporary addition has been made to the notation in order to visualise the existence of persistent memory of the send-sticky form. This was felt to be necessary in order to understand the working of the algorithm. A double horizontal bar is used here, to indicate that the data may not flow directly through the datapath, but may wait upon some condition, e.g. all the list elements having been processed. In addition, the modified merge actor symbol is used to minimise confusion of the dataflow properties of the graph. The list length source actor uses a full datapath, which, as well as providing the size of future output arrays, is used for a test against the loop counter for the loop termination condition.

An initial attempt at the design, with only a surface knowledge of the notation, immediately violated the firing rule, as without a merge node, both the list source and loop feed back path provided input to the shuffle actor. This is perhaps obvious, as only the source list, or the fedback list would be needed at any iteration. However, the same problem, and others, occurred in implementing the loop counter, which was far less obvious. In keeping with the semantics of a for loop, the loop termination test was carried out at the start of the loop, in the shuffle actor. But, as persistent memory is only available in datapaths, a path was needed to send-sticky the loop counter. This had no apparent destination actor, as the final value was of no importance, just the event. It was decided to attach the datapath to the print actor, along with a second

datapath carrying the sorted list. The latter was not sent until the loop terminated, and the loop counter was discarded when sent to the print actor. This solution posed great problems, eventually tracked down to a firing violation at the print actor. Although both input datapaths had data on them at the end of the loop, the very act of sequentially calling send twice meant that at the destination actor only one input was valid at any time. It became necessary to incorporate both loop counter and sorted list into one array, which was sent-sticky until the end of the loop. Whilst providing a solution, this combination of two completely independent data items (which could be different types) is far from desirable.

Further misunderstandings of the notation led to trying to collate the element pairs using send-sticky within the depth actor, whereas a separate actor is required especially for this purpose. Perhaps some graphical device, better illustrating the parallelism and indicating the flow of multiple datapaths, might have pre-empted this. Another problem with using send-sticky is that it is only possible with a MeDaL array type, so single element types, such as the integer loop counter, must be used in an array. However, MeDaL array types could not be used on a full datapath. Clearly there are great difficulties, mainly with the persistent data in MeDaL, which had to be addressed before the methodology can be said to be a better alternative to current techniques.

As already described, the MeDaL notation and prototype run-time system had a number of key limitations :

1. no graphical design environment,
2. manual translation of the graph objects into code,
3. notation features specific to shared memory architectures,
4. language extensions (such as send, extend) for implementing the datapath interface,
5. a machine-dependent run-time system,
6. problems with data persistence,
7. difficulties implementing high-level control structures such as loops.

These limitations are now addressed, and solutions proposed, as part of the ParaDE design system.

3.7 Solutions to MeDaL Problems

It was clear from the use of MeDaL that the problems identified above required further investigation if a viable design system was to be produced. Thus, initial research centred on addressing these problems, categorised in three main areas :

1. machine independence,
2. data persistence,
3. high-level control structures.

Other factors, such as implementing a graphical design environment and automatic generation of code are discussed later.

3.7.1 Machine Independence

This thesis proposes to remove all language extensions, such as send and extend, used to implement the datapath interface within the method code. To achieve this it is necessary to hide or abstract whatever functions are required for setting up, transmission and receipt of data. Two main concepts are presented for the solution:

1. A machine-independent run-time system is adopted, using implementations of the Linda parallel programming model to decrease the reliance on specialist functions and data types. The choice of Linda represents a move towards a highly portable and widely implemented run-time system, whose programming model fits in well with the proposed notation. Chapter 4 addresses further implementation issues regarding Linda.
2. Graphical user interface objects are developed to provide graphical abstractions of all issues involved with the interfacing of the actors and datapaths. The use of graphics removes all dependence on languages and machines, and provides a simple but effective means of capturing user-supplied data, or automatically generating names and initial values (e.g. data variable names for datapaths). Section 3.8 examines the graphical programming environment in more detail.

3.7.2 Data Persistence

The problems identified in Section 3.6 led to two main observations on the use of data persistence with the send-sticky mechanism in MeDaL, and the replacement of this mechanism by the facilities described below.

1. The main use of the send-sticky mechanism was to collect together pieces of a larger data structure that had been produced, for example, by different instances of a depth actor. This was the intended use for the send-sticky facility, but can be seen to be part of the wider issue of data partitioning whose aim is to maximise locality of data across the actors, that is, provide only what data is necessary for that specific computation. Data partitioning should include both the dividing and regrouping of a data structure, of which only the latter is addressed by send-sticky, the former being left to the user code in the preceding actor. The partitioning of data was identified as a crucial element in many algorithms involving data parallelism, and the development of a simple, abstract method for specifying the partitioning and regrouping of data was seen as significant in order to contain the complexity of user code and allow flexibility in the level of parallelism. This flexibility was considered as part of the performance adjustment techniques investigated in this thesis, and will be discussed in Chapter 4 which deals with these issues. The initial specification of the data partitioning is considered with the depth actor details in Section 3.8.2.
2. The send-sticky facility was sometimes used to implement control behaviour at the graph level, for example, there were no means of specifying loop behaviour in the graph except by using a cyclic graph (with a merge node to connect with the original datapath input to the loop) and implementing the loop control variable as a persistent data element within the main data array. As described earlier, this proved cumbersome and difficult both for program development and understanding. It was recognised that this type of control did not easily fit into the dataflow model, and ways of abstracting this control from the current level were investigated. These

investigations concentrated on the loop structure, and led to a number of conclusions :

- The loop is used frequently enough in general-purpose programming to justify special support for this programming form.
- In parallel programming the loop is often used in two different ways:
  - To implement an iterative algorithm, or part of an algorithm. That is, to support the repeated use of the same piece of code.
  - To force sequentiality on a set of independent calculations, for reasons of convenience, or performance (to increase grain-size).
- Clearly, the first of these two is desirable, whilst the second is an unnecessary use of the loop, developed through sequential programming practices or required by current parallel programming tools.
- Cyclic graphs in a dataflow model are difficult to understand behaviourally, and may violate firing rules due to conflict of data items from different iterations.
- The loop is very difficult to implement in the original MeDaL notation, as there is no obvious place for maintaining the loop control variable.

The solution developed was to make the notation acyclic and abandon the send-sticky mechanism, in favour of a graphical data partitioning facility and a new actor, the **loop actor**, described below. The data partitioning feature is used in conjunction with the depth actor, specifying how the data is distributed across the multiple instantiations of the actor method code (discussed in Section 3.8.2 and Chapter 4). The depth actor can be seen as one area in which MeDaL developed control structures, although companies were the only hierarchical node, meaning that the depth actor only applied to nodes at the lowest level of the graph.

3.7.3 High-Level Control Structures

In designing and implementing programs in a high-level language, a user is encouraged to structure the program in a hierarchical way, with one level specifying a general method or algorithm, and subsequent levels refining this into greater detail. One algorithm feature commonly used in programming is iteration, as many programming problems have iterative solutions. Most high-level languages feature a loop operator to implement iteration, and this control structure was not provided in MeDaL, with the resulting problems discussed in the previous section.

Investigation into the requirement for, and use of, the loop structure in parallel programs identified a significant conflict between the conceptual and practical use of loops. Whereas conceptually the purpose of the loop is clear - to implement iterative behaviour - in practice two distinct uses of the loop were observed in parallel programs:

1. to implement iterative behaviour,

2. to implement parallel behaviour.

An example of the latter is in parallel versions of Fortran, where a `forall` statement is used as a parallel extension to the loop structure. Whilst the use of the `forall` statement is very similar to the tradition `for` statement for iteration, the purpose is significantly different. The `forall` structure implements a parallel behaviour,

sometimes called loop-spreading, replicating the computation in the body of the loop over all values of the loop variable. In this parallel version, the sequence of the loop is replaced by simultaneous concurrent computation of each loop value, and in fact iteration plays no part.

These two uses are clearly different, as iterative behaviour is, by definition, sequential, and the use of the loop to implement parallel behaviour is confusing and counter-intuitive. A clear distinction is drawn in the proposed design system, ParaDE, between these two uses, and the loop structure is used strictly for sequential, iterative behaviour. Parallel behaviour, including the so called loop-spreading behaviour, is implemented solely by the depth actor, described in a later section. Here, the discussion centres on the use of a loop structure for sequential iteration.

The ParaDE notation does not allow cyclic graphs, instead a special purpose loop actor can be used to implement higher-level looping structures. The loop actor can be used to support the top-down design methodology by specifying control at a level above the user code, that is, at a design level, adding structure and removing complications associated with the dataflow rules described above. Abstraction of the details of the loop body supports the top-down design approach, and is achieved in the same way as hierarchy in all actors, by decomposing the actor into a sub-graph.

In order to specify an iterative program, certain details must be supplied :

1. a variable representing the loop counter,
2. a boolean expression defining the termination condition,
3. specification of the feedback path and its data,
4. program behaviour on termination of the loop.

To permit some of these features to be expressed graphically at the top level, the notation illustrated in Figure 3-10 was developed. Other features, more suited to the decomposed level of the loop body are handled in ParaDE as follows.

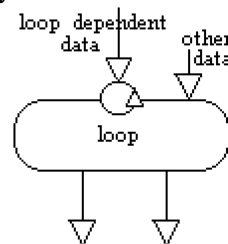


Figure 3-10 Loop Actor

The user can specify the condition expression for the loop to continue using a text entry box on the graph window representing the decomposition of the loop actor. A loop variable is provided in the same window for use in the condition. This variable is initialised to zero and incremented every iteration. It is also available to be used as a read only variable within the method code of the actors on the sub-graph. Loop-dependent data (i.e. where the input to the next iteration of the loop is dependent on its previous output) is identified as an input datapath connected to the circular directed path at the top of the actor, as illustrated in Figure 3-10. This data takes its initial value in the normal dataflow manner from the preceding actor, hence no initialisation is carried out within the loop actor method code. Any number of datapaths can be

connected in this manner, to allow different data to be passed around the loop. Alternatively, a struct data type (in C) could be used to aggregate data on the same datapath. Any additional input datapaths connected to other parts of the loop actor will behave in their normal manner each iteration. Thus, a continuous datapath input will preserve its value over all iterations of the loop, whereas a discrete datapath input will be required to produce a new data item which is consumed by the loop actor on every iteration.

The body of the loop is specified in the same way as a sub-graph with the exception that the output interface nodes which connect to the top-level graph can be identified as giving output every iteration, or only when the termination condition is satisfied, described below. The loop actor can therefore have multiple output datapaths, which produce output as directed by the user on the graph. Figure 3-11 shows a decomposition of the loop actor in Figure 3-10, with the loop variable *loopCount* (generated automatically) and continuation expression *loopCount < 10*.

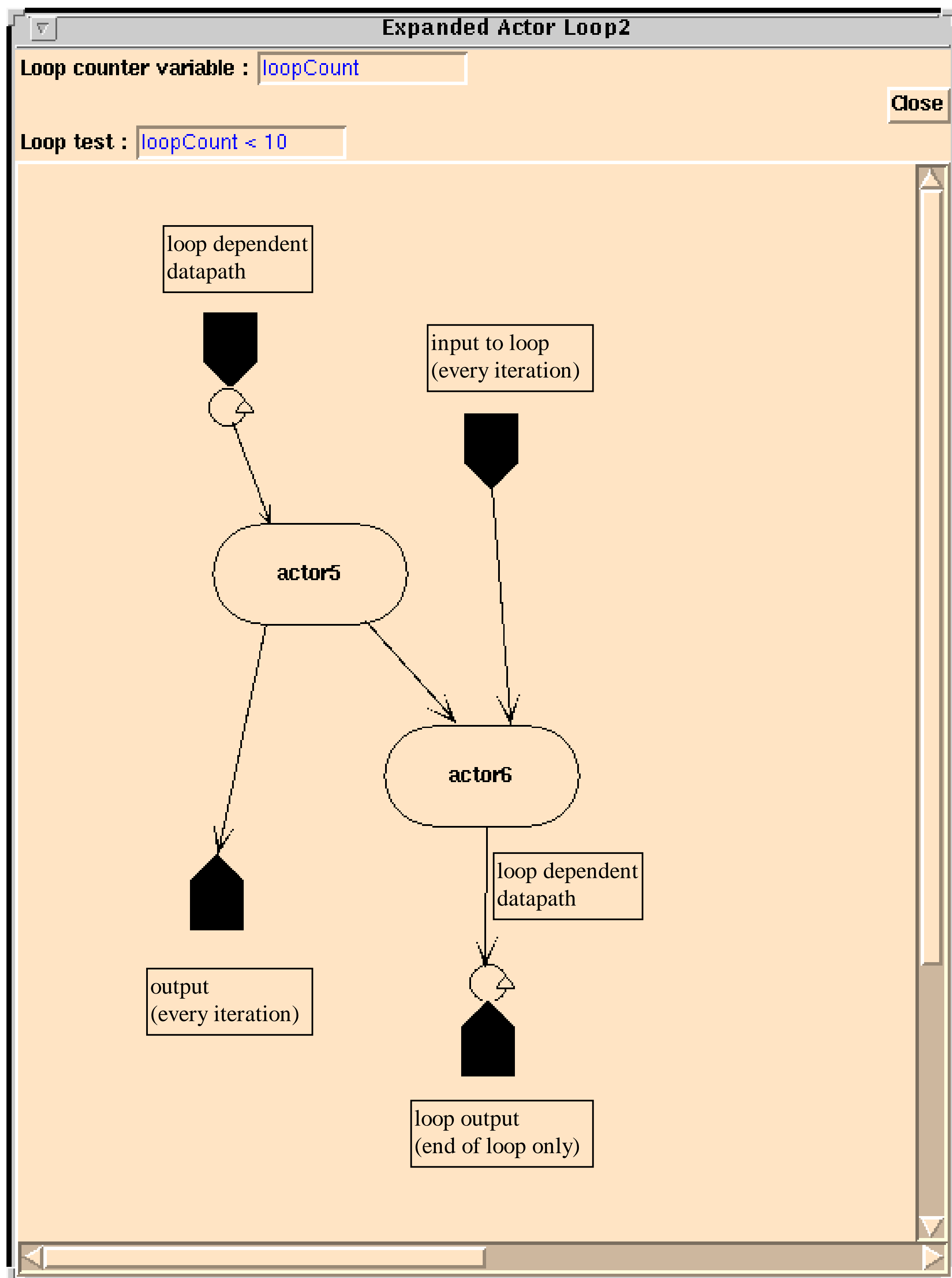


Figure 3-11 Sub-graph for Loop Actor

The special case of the output datapath used as a feedback path to the next iteration of the loop is not represented as a normal output datapath at the top level, except conceptually by the directed circular path notation at the top of the actor. At the decomposed level, however, an interface node is generated and identified for this purpose, allowing the user to connect data-dependent output datapaths from the loop body. An extra symbol is used to represent the point at which the loop condition is tested, and the loop-dependent data fed back to the start of the loop. This symbol is a directed circle, similar to that on the loop actor symbol. It is used in two ways :

1. A loop circle symbol is automatically connected to the base of the input interface node for the datapath in question (Figure 3-12). This represents the point at which the data is received into the loop, and its connection to the input interface node is symbolic of the initial input to the loop from the preceding actor.

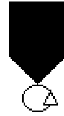


Figure 3-12 Loop Input

2. The corresponding symbol for the loop data at the end of the loop is identical, but has no connection initially, that is, it is placed on the graph without touching or being fixed to any other actor or datapath. The user can choose how to connect output nodes to this symbol, to achieve the desired form of output, using the mouse. There are two such forms of output from the loop :

   - The first, where the loop circle symbol is connected at the top of an output interface node (see Figure 3-13) indicates that within the loop, the data on this path is fed back to the loop input point. In effect the two identical symbols represent the same point. At the end of the loop, when the termination condition is satisfied, the loop output is transmitted down the output interface node, and thus to the actor following the loop actor at the higher level.

Figure 3-13 Loop Output at Termination

   - The second configuration involves connecting the loop circle symbol to the base of the output interface node (see Figure 3-14). This represents output from the loop every iteration. As with the first method, the loop circle symbol acts as the point at which the condition is tested, but as it now resides after the output node, then the output node transmits data on this path regardless of the result of the test condition.

Figure 3-14 Loop Output Every Iteration

The input datapaths which caused the loop to fire initially, and which represent the loop-dependent data, will be disabled from firing under the influence of the preceding actors until the loop has satisfied its termination condition. These input datapaths are identified by their connection to the directed circle notation at the top of the loop actor, and annotated for clarity in Figure 3-11.

During the execution of the loop body, the value of the loop variable is made available to the method code of the lower-level actors through a local variable, in a similar way to the accessing of input and output variables demonstrated in the next section. This

loop variable has a name automatically generated. The system takes care of the declaration of the variable, and the user simply uses the name of the variable within the method code as if it were a locally declared variable. The loop variable is under the control of, and maintained by, the system. Thus, no explicit datapaths are required for control purposes, and all datapaths associated with the loop actor and its decomposition carry only data necessary for the computation.

Although loop behaviour could be implemented in the MeDaL notation, the difficulties of providing persistence to control the loop counter in a strict dataflow environment led to very cumbersome and poorly understood graphs. The loop actor has been shown to provide a structured and well-defined representation of iterative algorithms, whilst maintaining the dataflow principles at the graph level. In addition, the role of the loop in parallel programs is more clearly defined in ParaDE, and the distinction between true iteration and repetition for performance has been clarified. The use of decomposition in the graphical environment allowed the loop structure to be abstractly represented as a graph node and refined in a structured manner. This graphical environment is now described in more detail.

3.8 Graphical Programming Environment

A graphical programming environment, built around the ParaDE notation, is more than just an attractive user interface. In fact it is the key to a successful parallel programming development system. The graphical environment provides a number of facilities:

1. a graph editor for constructing program designs,
2. abstractions of complex and architecture-dependent features,
3. facilities for refining the design and specifying attributes,
4. tools for the automatic generation of code,
5. an interface to external tools for compilation and debugging,
6. an interface to target architectures for executing the programs,
7. graphical features for adjusting the performance of programs.

This section describes the environment in general before examining some specific features which provide graphical abstractions of critical elements of a parallel program. The features of the design system are described, including the novel ways in which separation of data and computation elements is achieved, the new methods for the partitioning of data over depth actors, and a discussion of the general approach to developing programs using this design system. The section concludes with a description of the graph checking tool and an introduction to the code generator, both important tools provided in the graphical programming environment.

3.8.1 General Details

The programming environment is based around a graph editor, with which users create and modify a design of their parallel program. Actors from the set described

earlier can be selected and placed on the graph and connected with datapaths drawn using a mouse. Figure 3-15 shows the graphical user interface with a top-level design displayed. The icons on the right of the screen represent the objects (actors, datapaths) which can be selected to be added to the graph.

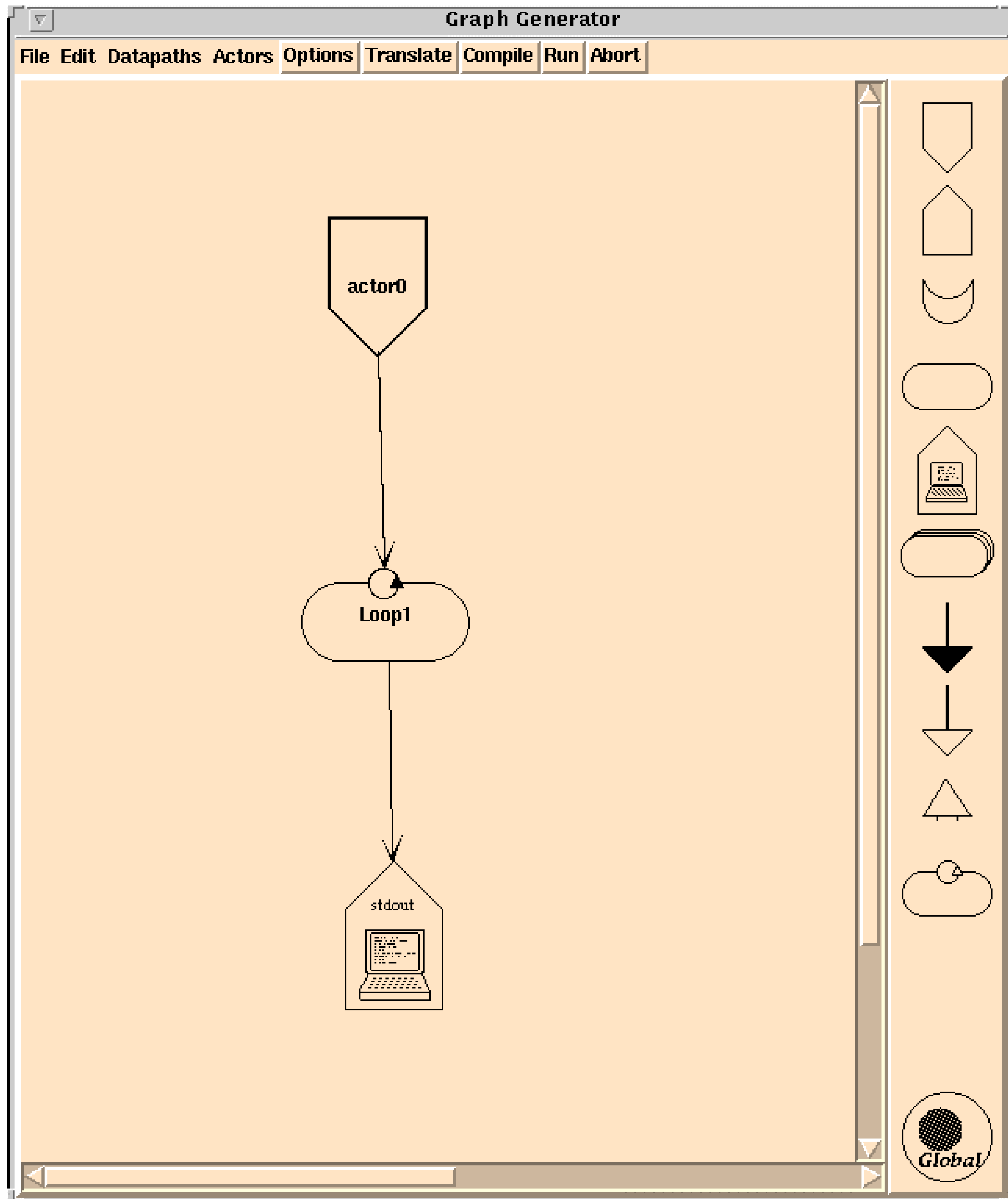


Figure 3-15 Graphical User Interface

Attribute Forms

Each actor has associated with it an attribute form, in which information about the actor type and input/output datapaths (also shown in the method code window) is displayed. The user can, where appropriate, add details such as a label for the actor. All actors have an attribute form, although the form will vary according to the type of actor. A merge node for instance will show information about its input and output

connections, but will not allow any user input or further refinement of the actor. A depth actor form has special facilities to enable the specification of the partitioning of the data between the instances of the depth actor (see Section 3.8.2 and Chapter 4). Figure 3-16 shows an actor form for a method actor *actor2* with a single input *dp0* and output *dp1* (recognised by the system as the datapath connections to the actor at the graph level) and which has a corresponding method code window as described above in Figure 3-10. The actor name *actor2* has been generated automatically, but can be edited by the user. Figure 3-17 shows an attribute form for a merge node which merges input datapaths *dp3, dp10* and *dp11* into a single output datapath dp5. There is no method code associated with the merge node.

Actor Form
Actor Name : actor2
Actor Type : Method
Input datapaths : dp0
Output datapaths : dp1
Method Code Expand
OK

Figure 3-16 Method Actor Attribute Form

Actor Form
Actor Name :
Actor Type : Merge
Input datapaths : dp3 dp10 dp11
Output datapaths : dp5
Method Code Expand
OK

Figure 3-17 Merge Node Attribute Form

Each datapath also has an attribute form associated with it. The interfaces between the actors are specified by filling in entry boxes and selecting items in the datapath form to define the type and size of the data. The interfaces should be specified at this design level, as the information supplied is carried through to the method code level of refinement. The use of attribute forms provides a means of specifying the details of the elements in the graph (actors, datapaths) without cluttering up the graph display. It is also useful to separate the structure of the program, in the form of the graph, from the detail of names, sizes and types which can be considered as a refinement of the interfaces between graph elements, or of the elements themselves. For example, a datapath connection between two actors indicates a flow of data, and therefore a data dependency between the actors, but this is all the information conveyed by the graph initially. Further refinement of the program would lead to the definition of the data carried by this datapath, and then its use in the computation specification (method code).

Figure 3-18 shows a datapath form for datapath *dp0* which is a discrete datapath carrying an array of 9 integers from *actor0* to *actor2*. These actor and datapath names

are generated automatically when the graph is created, but the user can change them if desired to a more meaningful label in the appropriate actor attribute form. The data type can be changed by selecting from the drop down menu labelled *Data type*. This menu contains a list of data types, which do not necessarily need to be supported as standard types in the host language. Instead, they are meant as a language-independent list of data types, which the code generator can translate into actual data types or aggregates. Of course, as the code is written purely in the host language, there must be a recognised way of using these types in the language. Currently, standard types are supported - Integer, Float, Double, Char, and arrays up to two dimensions for these types. Where appropriate, the form will provide entries for the size of the data which the user can edit. The datapath label *dp0* has been generated automatically, but can also be edited by the user.

Datapath Form
Datapath label : dp0
Source : actor0
Destination : actor2
Datapath type : Discrete
Data type : array<int>
Array size : 9
OK

Figure 3-18 Datapath Form

Method Code Window

The user is encouraged to develop the program in a top-down manner, decomposing the graph and refining the design until the actors represent "primitive" functions. These are functions which the user does not wish to refine further, and which represent cohesive units of code with a clear input and output interface (defined by the datapaths). At this primitive level, the user enters the sequential code for the function, using a specified programming language such as C, into a method code window associated with the actor. At the top of this window, information is supplied about the actor's input and output interface and local data. Figure 3-19 shows a method code window where *actor2* has a trivial function - passing the data from the input array to the output array, using two local variables *i* and *j*. Notice how the local variables and input and output arays are contained within text entry boxes, separated from the actual code. This allows the translation tool to insert code to manipulate the data types in whatever way is necessary for parallel execution, perhaps involving special set up code as used in MeDaL. Whilst being separate from the code, all necessary information concerning the type size and names of the data variables are provided to allow the user to make use of them in the normal manner. The main text area contains the user supplied code, written purely in the host language (in this case C). The representation of the data types in the information section above the main text area is a design system feature, and does not use the same format as array declarations in C. The data type information is extracted from the attribute forms of datapaths on the graph, and the choice of representation of these types is not significant, but is intended to be independent of the method code language (which may not necessarily be C).

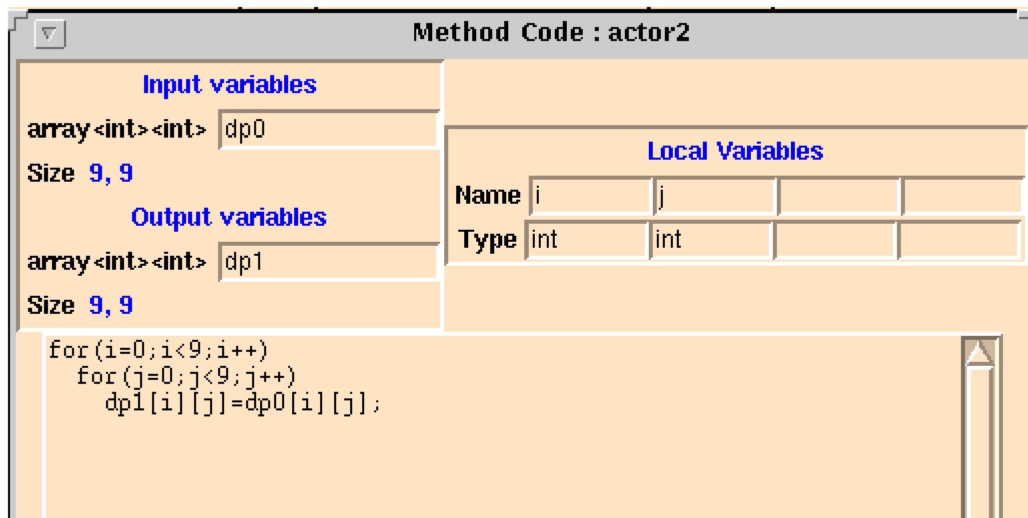


Figure 3-19 Method Code Window

Actor Input and Output Data

The labels specified by the user in the datapaths' attribute forms (e.g. *dp0* in Figure 3-18) can be used in the method code as local variable names representing the input and output variables, as illustrated in Figure 3-19. At the top of the method code window are displayed the names, types and, where appropriate, sizes of the datapaths connected to the actor. This information is extracted from, and cross referenced with, the datapath forms. The actual transmission of data to and from an actor is handled automatically, and there is no requirement for any special language constructs to facilitate the sending of data. By providing the user with input and output variables associated with the datapaths, the system removes the need for the user to get involved with low-level mechanisms (e.g. message-passing constructs packing data and calling transmission functions to send the data to a named port). Given the strict firing rule, all inputs to an actor must have data available on them before the actor fires, and hence when the actor's method code is executed, the input variables must have a value, and can be accessed within the code. Similarly, values written to output variables are dispatched along the datapath without any explicit command from the user.

The default action of an actor is to send all its valid output (i.e. all output variables that have been assigned a value) at the end of the method code. There are, however, circumstances where it may be desirable, or even essential for the output to occur at some earlier point in the method code. For example, if the code section is large, and one of the output variables is assigned its value near the start of the method code, it may be efficient to transmit the output variable as soon as it becomes valid. This behaviour can be achieved by the use of draggable output bars. These are horizontal lines labelled with the output variables' names, which would normally reside at the bottom of the method code window. The user can drag a copy of the output bar to the position in the method code which is required, and optionally remove the original bar from the end of the code. This is a simple graphical mechanism which again obviates the need for special language constructs to achieve exceptional behaviour. This feature is not implemented in the current prototype of ParaDE.

Where an output variable has not been assigned a value in the method code, there will be no corresponding data transmitted along the datapath (the existence of values on a

datapath is checked by the system before transmission). In this way, the control of if-and-when data flows through a particular datapath can be specified by the user at the code level, and the graph depicts all the possible data flows between actors. This no-value datapath situation might occur for example in a conditional statement, where one of two datapaths carries the data, depending on the outcome of the condition. In general, however, there is no need for the user to be concerned with dataflow control within the method code, as the graph specifies the algorithmic control structure.

Global Data

The graph editor has a special icon labelled "Global" (e.g. see Figure 3-15) which, when selected, brings up a text edit window. This area can be used by the programmer to put #include files required for the program, and to define global read-only data constants which then become available for use within the method code of any actor. Figure 3-20 shows a global data window.

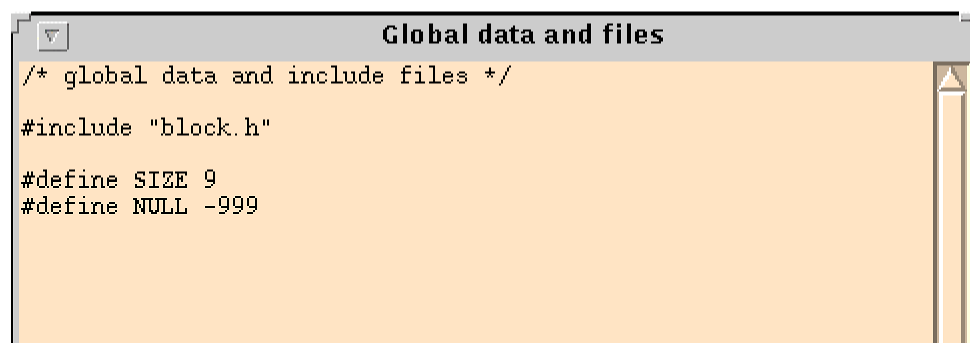


Figure 3-20 Global Data Window

3.8.2 Depth Actor - Data Distribution

The depth actor is a special case of the general-purpose actor in that it has multiple instantiations to implement parallel behaviour. Only a single method code is specified, but this code is replicated to create concurrency. This actor was part of the original MeDaL work, but an investigation into the use of high-level structures including parallel mechanisms and looping led to a redefinition of the use of the depth actor. The discussion of the use of looping structures for parallel behaviour was described earlier in Section 3.7.3, but the key point here is that the depth actor is used uniquely for specifying data-parallel behaviour. This behaviour is where the same computation can be applied to multiple data values concurrently. The actual level of parallelism obtained is dependent on the underlying architecture and the run-time conditions dictated by that architecture and its use. However, the data distribution facility described in this section indicates the order of parallelism desired by the user. The depth actor can be decomposed in the same way as the loop actor, but it also has an extra button on its attribute form which allows the specification of a data distribution across the instances of the depth actor, that is, the data on the input datapath can be divided up and distributed between the depth actor instances according to the data patterns of the algorithm.

The use of this data partitioning isolates the issues of data management from computation, where previously in MeDaL designs, an extra actor was required before and after a depth actor, to split up the data structure and distribute it to each instance of the depth actor and afterwards to re-assemble the part-results from the depth actors

into a composite data structure. With MeDaL, the details of the data distribution had to be written by the user as part of the method code. This was undesirable as it brought issues of data management into the method code, and could lead to complex data partitioning code, obscuring the actual computation and giving greater scope for user errors.

The graphical data distribution facility provided by the ParaDE programming environment described in this section provides an abstraction of the potentially complex matter of data partitioning, and maintains the separation between computation and data. The data distribution window is accessed from the attribute form of a datapath connected to a depth actor, and the attributes of the data on this datapath - its type and size - are recognised by the system and carried forward into the data distribution window.

The emphasis of the data partitioning facility is on providing graphical descriptions of commonly used partitioning strategies (influenced by research into data patterns used in common programs [21]), from which the user selects features appropriate to their program. The actual code for the data distribution is then automatically provided by the code generator - again supporting the concept of machine portability, as data partitioning techniques may vary from machine to machine. Of course the method code written for the depth actor is required to be generic in terms of the data structure it processes. This means that the method code takes a section of data as input, regardless of the positioning of the data section within the larger data structure. An investigation into the application of data partitioning in parallel programs led to the design of a graphical distribution specification which corresponds in some ways to the use of *Align, Processors, Distribute* and *Template* statements in HPF [28]. The distribution procedure developed involves four stages:

1. Selection of a template, representing the size and structure of the data block (part of the overall data structure arriving on the parallel datapath - identified by its multiple arrowhead) required by one instance of the depth actor.
2. Alignment of this template with the original data structure, identifying the relationship between the template and the data structure.
3. Alignment of a second template to specify the pattern of replication across the data structure.
4. Grouping of templates into a single computation.

All of these stages are achieved graphically in ParaDE, to support the abstract methodology for specifying parallelism. The last stage is concerned with performance optimisation, and will be addressed in Chapter 4. The other three stages are now described in more detail.

The selection of a template represents the specification of a section of data required as input to a *unit computation*. This computation is a section of code which is logically cohesive and which can be applied over all such data blocks throughout the whole data structure. The code is, therefore, generic and may be small in terms of grain-size. Its size is not important at this stage, here the concern is with the algorithmic nature, or design, of the program. This focus on the algorithm, although a common sense approach, is often lost in parallel program design, where performance issues invade

the program development early on, leading to solutions which perform efficiently on a limited set of architectures, and which confuse the design and implementation stages. It is important to fully specify the design independently of performance issues, and disregarding the computation size at this stage is part of this aim.

The template selection stage is not as complicated as it might sound, but merely requires a step back from implementation details, and some thought about how the algorithm is intended to work. For example, matrix multiplication is a simple, but common, data parallel computation which would use the depth actor. The algorithm for matrix multiplication performs a summation of element-pair multiplications for a given row and column from the two source matrices. This summation of multiplications is clearly the unit computation, which is replicated over all row-column pairs. The computation takes as input one row and one column and calculates the summation of row-element and column-element multiplications. A generic computation has been chosen, as the input and output data is independent of position in the matrix. This input data is a single row from one source matrix and a single column from the other, and these one-dimensional arrays, with the same size as the original matrices, are the templates that would be chosen for this computation.

The data distribution window that appears when selected from the depth actor's attribute form shows a number of templates of different types of data structure, e.g. a single element, an element pair or a one dimensional array (row or column). Figure 3-21 shows a distribution window, with a template selected.

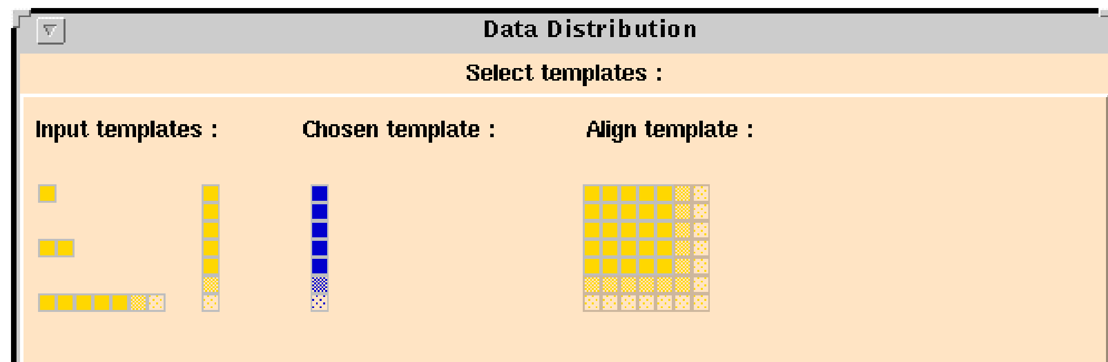


Figure 3-21 Template Selection for Data Partitioning

The data templates displayed were the only types supported in the prototype design system, but in a full system, a comprehensive set of data templates would be provided, supporting all common partitioning strategies, as defined by [21] and described in Chapter 2. Less common partitioning strategies can still be implemented in the traditional manner, with the user supplying the partitioning algorithm within the method code.

Having chosen a data template which forms the input to the unit computation specified by the depth actor method code, the next stage is to specify the relationship between one template and another on the original data structure, that is, the non-partitioned data which formed the input datapath to the depth actor at the top-level graph. This process identifies the way in which the computation is replicated over the whole data structure. The chosen template is aligned with the original data structure by dragging the template, using the mouse, on top of the partial data structure pictured to the right in Figure 3-21. This specifies how the position of the template shape

relates to the whole data structure - covering issues such as border conditions, where perhaps the outermost elements are kept at a constant value, and so are not calculated using the method code.

The alignment of the first template identifies the starting point for the replication. The same procedure is followed for a second template, selecting a template type and aligning it on the original data structure. This second alignment specifies the pattern of replication across the whole data structure. For example, selecting a column template, aligned to the first column of a two dimensional array, the second template could be aligned to the second column of the array for a matrix multiplication where each column is computed identically, but could be aligned to the third column if odd and even columns were to be computed differently. Figure 3-22 shows the first template aligned to element 0 of the array and the second aligned to element 1.

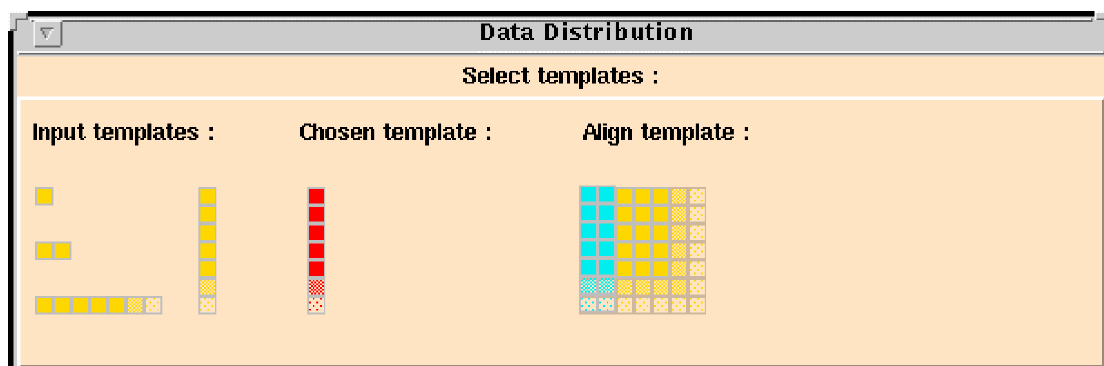


Figure 3-22 Alignment of Templates for Distribution

A piece of method code for the depth actor is written by the user referring to the data variables associated with the selected templates. These variables show the type and size of the actual input and output data required for a single computation, rather than the original data type and size of the non-partitioned data structure, but otherwise the procedure is the same as writing method code for any other actor. Figure 3-23 shows a method code window where one input datapath carries a whole 4x4 element matrix, dp6, but two other inputs, dp8 and dp10, carry partitioned data. The 4x4 element matrix has been partitioned into columns to give a one-dimensional data structure, of size 4 for the actor to work with.

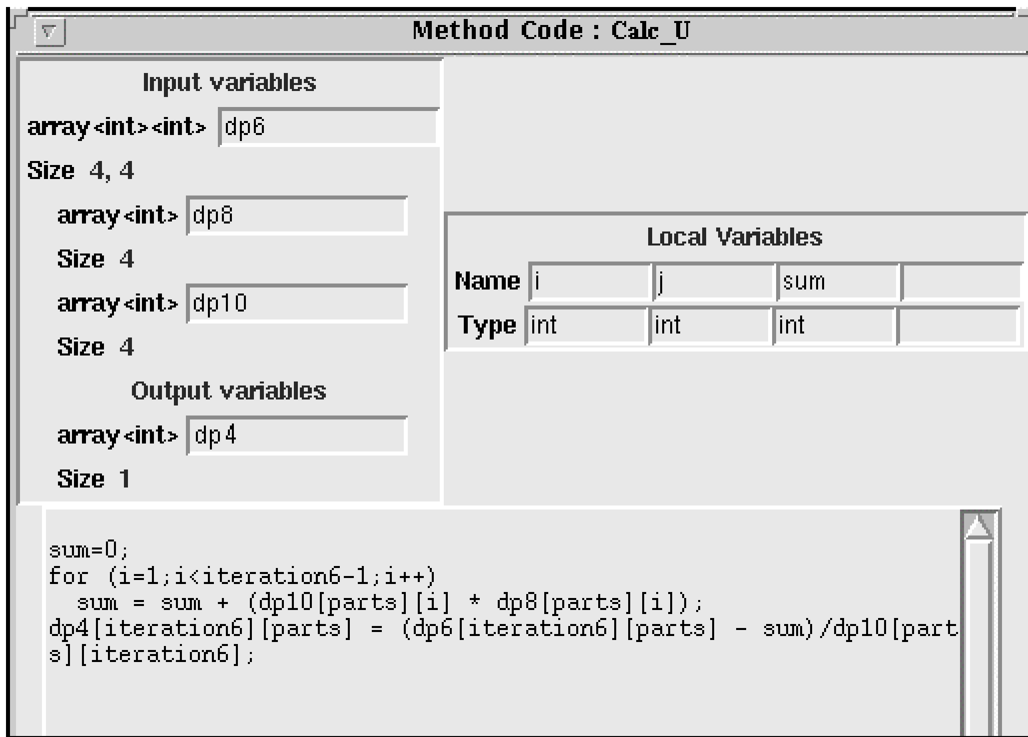


Figure 3-23 Method Code Using Partitioned Data

Output datapaths are specified in the same way as the input datapaths, with template selection and alignment for partitioned data. Data that is not to be partitioned is used in the usual way, missing out the partitioning window. Figure 3-23 shows a partitioned output datapath, dp4, which is a single element - the result of the unit computation specified by the method code. The type is given as an integer array of one element rather than a single integer, but this is merely a detail of the prototype implementation, and in future the data would be recognised as a single integer. Similarly, the method code uses a second dimension for the columns dp8 and dp10, and system-defined indexing variables, but again these are temporary implementation details to support partitioning and would be hidden from the user.

3.8.3 Approaches to Program Design

The depth actor with data partitioning, and the loop actor, have been presented as useful individual features for designing aspects of a parallel program and identifying truly parallel behaviour. However, to fully support a structured top-down approach to designing parallel programs, a more holistic approach is required. A discussion of the issues of how to approach parallel program design is a lengthy and complicated topic, but this section puts into perspective some of the techniques proposed in this chapter, shows how they are used in a practical example, and describes how ParaDE provides improved support for parallel program design.

It is sometimes the case that the original program being designed would not occur in a sequential manner in its natural environment. A common example of this is where a model of some situation over time is required. The computer algorithm to describe the model needs to take account of the time change, as well as any changes to the model within or across the time intervals. Taking heat flow through a solid bar as an example, it is necessary to model the conduction of the heat through the bar over time.

The computer model would need to make some assumptions, such as dividing the bar into regular size pieces, and calculating the heat transmission between the pieces. As well as the partition over the length of the bar, the solution also needs to divide the time up into intervals. So, for a given time instant, the heat transmission between pieces of the bar is calculated. One way to model this would be to have a loop actor, where each iteration represents a time slice, and within each time slice, the pieces of the bar transmit their heat to neighbour elements. The body of the loop would traditionally be represented as a second loop, nested within the first, with the heat transmission modelled as a sequence of transmissions from the top row of elements touching the heat source, down through the bar. It is this inner loop that can be expressed differently in the ParaDE design system.

Adopting a top design methodology, the process is begun by selecting a loop actor at the top level, representing a time snapshot of the whole data structure used to model the bar. The loop termination condition is convergence of the algorithm, in this case the point of temperature equilibrium (although it may in other examples be some time limit, or a fixed number of iterations). In each iteration of the loop there is a calculation of the heat transmission across the whole data structure representing the bar, a check for convergence, then the loop proceeds to the next iteration for a new calculation.

The conventional parallel loop, as used in parallel extensions to traditional sequential languages, has been described earlier, and rejected as a counter-intuitive approach to specifying parallelism. An alternative is to define the “loop body” as a section of code executed independently in parallel, using the depth actor, with no connection between the executions of replicated code. If a partitioning of the data over these replicated code sections is then specified, a solution to the “loop” section has been provided, without recourse to any loop features which inherently imply an ordering.

In the heat flow example, the problem of calculating the heat transmission can be broken down into one of individual elements passing heat values to their neighbours. In this example, the input template is the single element and its four direct neighbour elements, forming a cross shape. The output template is a single element, in the central position of the original element. Figure 3-24 shows the template patterns.

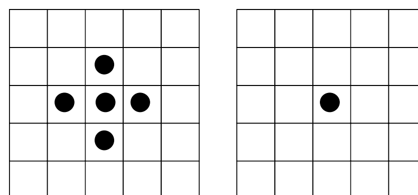


Figure 3-24 Template Patterns for Heat Flow

The next stage in the problem solution is to identify the relationship, if any, *between* the unit computations. It may be that each computation is independent of the neighbouring computations, in which case this step is redundant. In the current version of ParaDE, the assumption is made that there is no dependency between neighbouring unit computations, so this step is not described further here.

The steps described in this section outline how a user of this design system would begin to approach the design of parallel programs, using the features introduced in this chapter such as the loop actor, the depth actor and data partitioning. An example of a

real-life problem has been described whose complex behaviour can be represented easily using these features. The use of the loop actor and depth actor in conjunction has shown how the iterative and parallel behaviour is distinguished and how it is modelled in ParaDE, providing a clear and concise design approach. The remaining stages in program development are concerned with tuning the design to the architecture on which it is to be executed, which is discussed in detail in Chapter 4, with graph checking and translation to complete the implementation.

3.8.4 Graph Checking and Translation

Program graphs, as used in this design system, aid the user's understanding of parallel programming, improve the process of design, and reduce the number of errors in design by limiting the ways in which graph objects can be connected. However, there is still scope for user error in producing parallel programs, and ParaDE aims to eliminate as many errors as possible before execution time. To this end, a graph checking facility is provided to establish validity of the graph before translation and compilation into a parallel executable code. The graph checking tool will search for any inconsistencies in the graph such as unconnected or incorrectly connected actors, and will also detect cycles and missing attributes. The benefits of having a graph checking tool include :

1. errors in the graph can be detected before compilation of the textual program,
2. the user can be guided into correct and appropriate use of the notation,
3. graph errors can be addressed at the graph level,
4. errors involving co-ordination can be isolated from those involving computation,
5. a correct "design" leads to higher confidence in attaining a correct program implementation.

The translation (discussed further in the next chapter) includes a number of optimising techniques which flatten the hierarchy and generate efficient implementations of special graph features, e.g. the merge node. Information about the parallel structure of the program and the data flows between actors will be extracted from the graph and used in conjunction with the user-supplied method code to automatically generate a parallel program in an architecture-independent language. Compilation follows to produce an executable program for the target machine or network of machines. Compilation errors are reported back to the user showing which actor the error was detected in.

3.9 Summary of Design Issues

This chapter has investigated a number of issues concerned with designing parallel software, and proposed solutions to some of the problems uncovered. Other graphical programming environments have been considered, and their approaches to parallel software design discussed. Based on an evaluation of these approaches, and an investigation of the motivations behind graphical parallel programming environments, changes to the MeDaL notation have been proposed, both in terms of visual appearance to improve understanding of parallel program design, but also in the semantics of some of the features provided by MeDaL, to provide greater clarity in the

specification of parallel programs. Some fundamental problems with the MeDaL approach were identified, in particular the issues of architecture dependence, data persistence and lack of high-level control structures. To address these problems, features of a new design system, ParaDE, were proposed, including graphical abstractions of low-level detail and architecture-dependent features and new control structures such as the loop actor and data partitioning over the depth actor. The effect of these new techniques was investigated, and the encompassing graphical design system was described to provide a complete environment for the design and development of parallel software in a structured and well disciplined manner. Particular emphasis was given to the new structures for specifying parallelism, such as the data partitioning technique, and it was shown how these techniques improved conventional approaches. This emphasis provided a clearer distinction between the definition of parallelism as part of the program's algorithm, and the sequential aspects of a program, previously used to incorporate performance optimisation into a parallel program. The focus on parallel design, rather than implementation and performance, was maintained throughout this chapter, and the separate issue of performance is now explored in the next chapter.

4. Problems and Solutions in Achieving Efficient Performance

The previous chapter explored the issues involved in notations for parallel programming, and presented the ParaDE graphical notation. This chapter investigates the problems of achieving efficient performance in graphical programming environments, and proposes techniques for improving performance whilst maintaining the abstraction level of the graphical environment. One technique investigated in this chapter is the automatic generation of efficient executable code on different parallel architectures from a program developed in the ParaDE notation. Another technique explored is the user-guided, architecture-independent optimisation of the performance of a program at the graph level.

In order to fully investigate the methods used to address the issue of performance in graphical programming environments, some details of the implementation of the proposed techniques for performance adjustment are described, in addition to the user-level view of how the techniques are put into practice in developing parallel programs. Before these techniques are described, it is useful to consider the performance requirements of notations for parallel architectures.

Chapter 2 identified two performance related factors which influence the choice and suitability of a graphical notation for parallel programming - the nature of the target architecture, and the criticality of efficient execution. It is assumed in this work that efficient execution is paramount, as the nature of parallel programming suggests, and that the target architecture is one of a variety of common types. In addition, experience in parallel programming languages is not a requirement, so these factors must be addressed in an architecture-independent manner.

Traditionally, this simple requirements statement is contradictory, as performance has always been addressed with respect to the individual architecture, and has generally required in-depth knowledge of architecture-specific parallel programming techniques. This chapter proposes techniques for exploring performance in an architecture-independent manner.

Section 4.1 introduces the issue of performance with a look at ways in which current graphical programming notations address implementation and performance of programs, and whether or not they do so in a machine-independent manner. The next two sections outline the methods used in ParaDE to generate programs for different architectures and describe the target architectures used for translation in the ParaDE prototype system. Section 4.4 then describes two specific techniques for adjusting performance in ParaDE - actor folding and data partitioning. A summary of the performance issues addressed in this chapter is given in Section 4.5. Chapter 5 presents the results of experiments developed to assess the success of the techniques described in this chapter.

4.1 Performance Features of Graphical Notations

This section will consider the implementation and performance features of three alternative approaches in the graphical parallel programming area: CODE and HeNCE, which have been widely used and reported upon, and an implementation of

MeDaL, the notation from which parts of ParaDE draw influence. Some performance figures are given to illustrate that whilst these three graphical programming approaches offer improvements over traditional approaches in terms of the abstraction level and design environment, they fail to address the performance demands convincingly. This demand for improved performance strengthens the argument for the need for techniques to adjust grain-size to suit the target architecture, and hence optimise performance, without losing the benefits of an abstract design notation.

CODE was originally targeted at the Sequent Symmetry shared memory machine, exploiting features in the FastThreads package (University of Washington) for the creation and manipulation of parallelism. More recent developments have used compilation into PVM code for a network of workstations [53]. CODE uses a node and arc graph representation to provide an abstraction of parallel structure and process co-ordination, as described previously. However, some of the benefits of using this abstraction are lost in the manner in which the data interface to the nodes is implemented. The specified data ports associate directly with communication channels in the PVM mechanisms, and the user-supplied firing rules and routing rules are used to control the low-level behaviour of the data across these channels. The transmission and receipt of data is managed using these specified parameters, which gives the user flexibility and accurate control over data flow, but requires a detailed and low-level approach to data communication.

HeNCE also uses PVM as its intermediate code, generated from the subroutine (computation node) specification using the node parameters supplied by the user. HeNCE has, from the start, been targeted towards heterogeneous networks, and the target architecture is described as "a network of Unix machines of various types and capabilities" [53]. HeNCE requires the specification of a *Cost Matrix* - a list of the characteristics of the machines comprising the target "abstract parallel machine" [50]. The cost matrix is a table with a row for each host machine. The columns in the table show subroutines used in the HeNCE graph, with the relative cost of running the subroutine on each machine. No guidance is given as to the determination of cost values, nor any support provided for the measurement of execution costs - this is left entirely up to the user, but if no cost matrix is supplied HeNCE assumes a single host and sequential operation.

Neither CODE nor HeNCE have provided much documentation on the performance of programs developed in their model. A joint internal technical report by both developers mentions briefly that a Block Triangular Solver example developed in CODE using a 420x420 element matrix shows a speedup of 3.5 with 14 processors relative to a sequential program. This is compared to a theoretical maximum speedup of 4.9 and a hand-written parallel program with speedup of 3.7 [55]. It will be demonstrated in the next chapter that ParaDE achieves speedups across two widely different parallel architectures of at least 1.6 and 2.35 (relative to sequential programs) for parallel algorithms with theoretical speedups of 1.67 and 2.5 respectively. No figures were given for the Block Triangular Solver example in HeNCE, although the implementation of blocked arrays in HeNCE was described as not as efficient as CODE. A shorter, published version of this report omits performance measures entirely [53], and other reports by the same authors concede that portability "has not been demonstrated by implementation" [56]. Both developments, while acknowledging the importance of efficiency, seem to have concentrated on the

concepts and features of their model, particularly the aspects of heterogeneity, code reuse, and debugging.

The development of the MeDaL system was on a much smaller scale than CODE and HeNCE, and had only produced initial performance results for a small number of simple examples. MeDaL, as described earlier, was targeted at a run-time system developed specially for the Encore Multimax, using the Encore Parallel Threads library. Programs were intended to be medium-grained, and some features in the notation were designed to exploit shared memory behaviour. Early results showed promising performance for a matrix multiplication example - speedup of over 7 for 8 processors using matrices of above 100x100 elements, relative to the same program run on a single processor. The same experiment run relative to a sequential program demonstrated significantly worse speedup of just over 2. However, the parallel execution times were measured for a program whose algorithm was not optimised for parallelism, and incorporated a sequential data partitioning stage prior to parallel computation. This partitioning stage also failed to take advantage of basic data manipulation optimisations such as using pointers.

Two of the graphical environments discussed, CODE and HeNCE, use automatic code generation to implement the mechanisms required to exploit the parallelism specified in the graph notation. MeDaL was intended to follow the same approach, but the prototype system did not include code generation, instead relying on hand-translation. Code generation is an important issue in the performance of these graphical environments as the abstract notations, by definition, hide the lower-level details of process and data management. It is these lower-level details which are often optimised to achieve the desired performance by programmers using traditional parallel programming methods. This optimisation stage becomes the responsibility of the code generation tool, but it will be demonstrated in this chapter that code generation in ParaDE is merely one step in an iterative process to achieve efficient performance.

4.2 Code Generation

Code generation in ParaDE is fully automatic, once the graph program is created and attributes associated with actors and datapaths. From the graph program specified by the user, and the method code supplied for each node in the graph, an executable program is generated by a translation tool. This tool involves a number of processes to first translate the graph structure into an intermediate language, and then compile and link the resulting code into an executable parallel program. The first of these steps, the translation, is the most novel stage, the subsequent steps involve standard compilation tools.

Figure 4-1 shows the processes from graph creation through to execution.

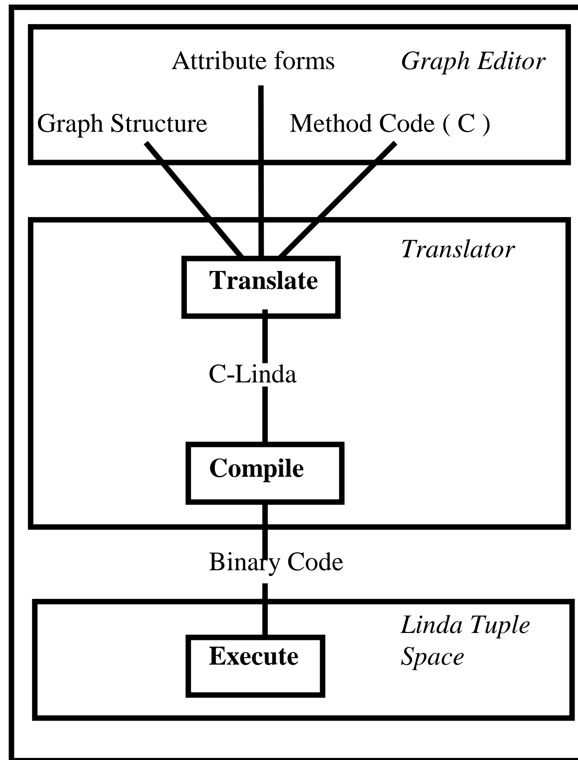


Figure 4-1 Translation Processes

The translator tool, written in *incr Tcl*, the object oriented extension to the Tool Control Language [57], follows a series of stages in order to produce the C-Linda code equivalent of the graph program.

Figure 4-1 shows three inputs to the translator - method code, which contains the sequential code segments for each actor, the graph structure, representing the parallel co-ordination between the actors, and attribute forms, giving the details of the interactions between actors.

In the first stage of translation, a *roadmap* is generated to formulate the execution ordering of the nodes in the graph. The execution ordering is based on the data dependencies between nodes, and is thus a static representation of the possible execution order - the actual order of execution at runtime may differ as a certain path in the node graph will not fire if the necessary data is not present. The roadmap concept was introduced in the manual implementation of MeDaL [2] although the use of the roadmap in execution in ParaDE is fundamentally different. MeDaL adopted a dynamic process creation approach at runtime, whereby the roadmap was used to invoke the execution of processes. As grain-size can be adjusted in ParaDE, as

described later, dynamic process creation, and the overheads associated with it, becomes unnecessary, as the number of (statically generated) processes is entirely manageable and efficient. Thus the roadmap represents only the conceptual ordering of actors by data dependency, and actual processes may comprise multiple actors in the granularity adjustment phase.

The roadmap is formed by a set of firing levels of nodes, where all nodes within one firing level may execute in parallel, but the execution of nodes in one level may depend on the completion of nodes in a previous level. The nodes at the first level form the set of source actors, and these actors become processes with no input, such that they fire immediately when the program starts. For each of the remaining actors in the roadmap, input and output paths are traced by extracting the attributes associated with each actor object, until the end of that path is reached - a termination or output node. This process of determining precedence is important to establish the flow of data through the graph - the source and destination of data. Linda communication mechanisms are then inserted into the processes representing each actor (the *wrapper code* encompassing the method code), using the attributes of the input/output data previously entered into attribute forms. It will be shown later that this execution ordering dictated by data flow is maintained when processes are merged during actor folding to improve performance. Initial generation, however, establishes one Linda process for each actor, one **`in`** statement for each input datapath, and one **`out`** statement for each output datapath. The granularity of the program implied by the size of these processes is not critical at this stage - subsequent techniques are used to investigate and improve performance by varying granularity.

Figure 4-2 shows an example graph for translation. The roadmap generated for this graph would be as follows (where the numbers are the internally generated actor id - in this case used as the actor names too for clarity):

```
0 1 2 3 4 5 6 7
8 10 9 11
12
13
14 (stdout)
```

In this roadmap, ordering is significant, although actors in one level are not necessarily dependent on all the actors in the previous level. Level 2 for example contains actors 8, 10, 9 and 11, and the following level contains actor 12. Actor 12 is dependent on actors 8 and 10, but not 9 and 11. What the roadmap represents is the earliest level at which actors can execute, so actors 9 and 11 may fire at level 2, but could also fire at the next level, as there is no dependency to actor 12.

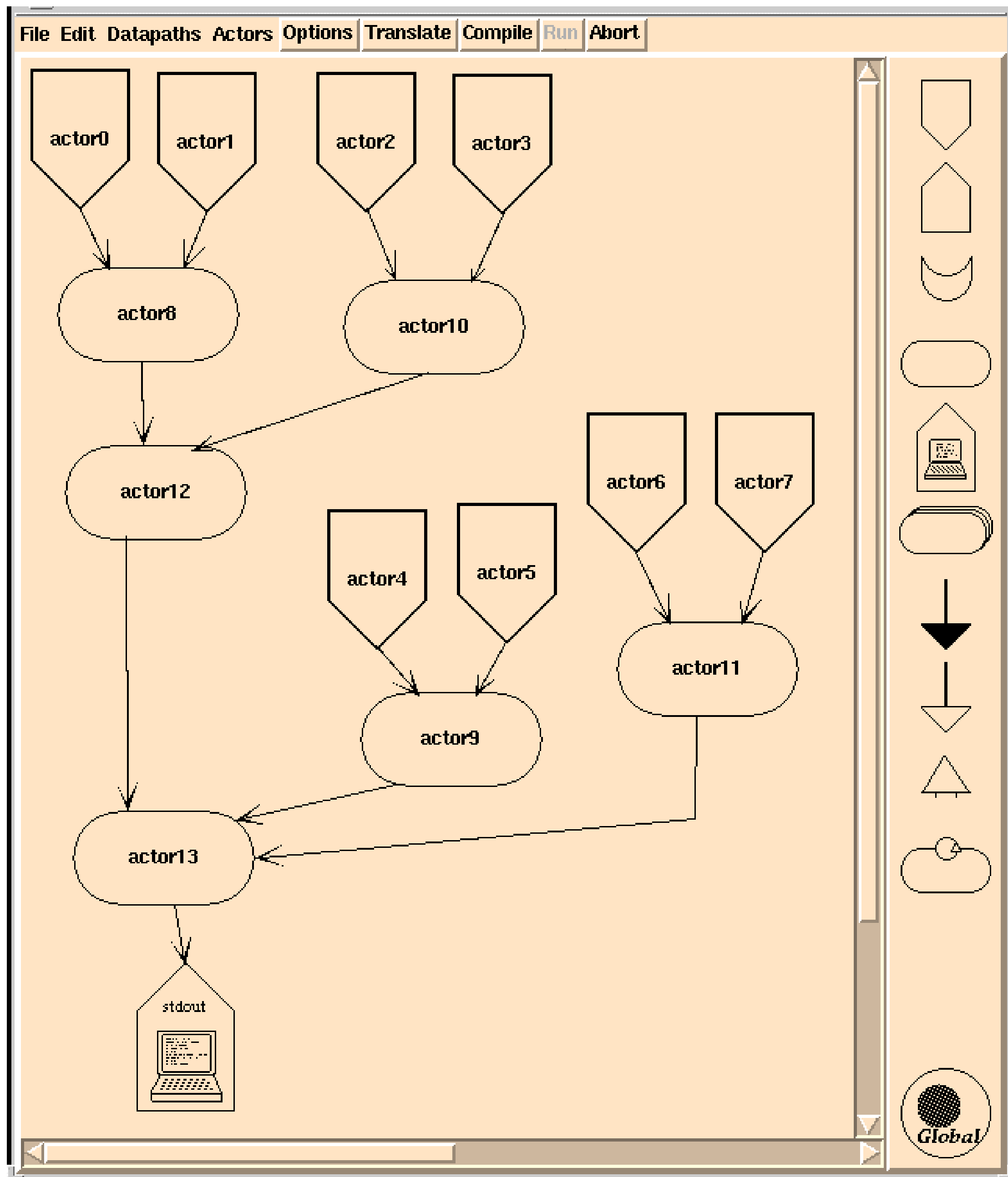


Figure 4-2 Example Graph for Translation

The roadmap is used to specify the execution ordering of actors (dictated by dataflow), and this information is used by the translation tool to generate process invocation commands in the intermediate-level code, Linda, which is based on a model adopting a process and data style at a similar level to the graphs in ParaDE. Alternative models to Linda adopting much lower-level parallel mechanisms, such as PVM, have been used in systems such as CODE and HeNCE, on the basis that these lower-level models are more efficient. However, the Linda model is, conceptually, much more suited to the graph program approach, with simple and distinct process, input and output features. The relatively high-level model is easier to understand and use, and more closely fits the concepts of the actor/datapath model. This makes it more suited to the development of a ParaDE prototype, particularly one which features granularity adjustment and which is aimed at investigating relative

performance not absolute performance. Other graphical programming environments have a strict mapping between graph node and parallel process, making the efficiency (and granularity) of the process model highly critical. With the ability to map multiple actors to a single process however, the granularity of the graph and underlying processes can be different. For this reason, the Linda model is a suitable implementation layer for a prototype of ParaDE. Future developments may lead to new or improved parallel models, and ParaDE may take advantage of these. The user-visible environment and notation have no obvious ties to the Linda model, as the underlying execution environment is transparent to the user. This means that only parts of the translator tool would need to be changed to exploit new parallel language technologies, and these changes would also be transparent to the user.

4.3 Target Architectures

The initial prototype translator for ParaDE was geared towards reusing the run-time system developed for MeDaL [2]. This used a specialised execution environment based on the Threads package available for the Encore Multimax, and included dynamic process creation and scheduling facilities. The reason for adopting this system was the existence of facilities specially designed for the MeDaL notation, and the need to investigate the feasibility of automatic code generation without substantial programming effort.

The code generation experiments proved successful, and example graph programs created using a specially designed graph editor were translated into executable code suitable for the MeDaL run-time system. However, the MeDaL run-time system was only partially developed, without support for hierarchical graph features, and problems were encountered at execution time which prevented comprehensive performance evaluation of the code generation tools. Additionally, the need to investigate multiple platforms was recognised, and so a change was made to adopt the Linda model for the underlying run-time support. The MeDaL run time system was aimed at medium-grained computations, and supported features specifically exploiting the shared memory architecture of the Encore Multimax.

Two implementations of Linda were used, for different target architectures. One was a commercial product for a network of workstations [47], the other a beta version developed for the Encore Multimax shared memory machine. These two architectures have widely different characteristics with respect to efficiency in parallel programs, and offered a sample of architectures on which to investigate architecture independence using performance tuning techniques. Conventional parallel programs written specifically for one of these machines are unlikely to run efficiently on the other machine, due to the different granularity characteristics. The same efficiency differences would apply if the programs were hand-written using a more abstract model such as Linda, because the size of the code or data for each Linda process would be significantly different between implementations for the two architectures. However, the parallel mechanisms would be similar (at the Linda model level), and it is this feature which is exploited by the translator tool developed in this work.

Both Linda implementations use the same primitives - **`in`** and **`out`** for data input and output, and **`eval`** for process creation. The translator, therefore, in principle produces the same *wrapper code* around the method code for an actor, in terms of the process

and data management. In practice there are some differences between the Linda implementations, such as the mechanisms for matching tuples consisting of partial arrays or pointers. Appendix A shows code generated for the different Linda implementations. It may be possible to use a common set of tuple matching rules, which would make the translation entirely portable between the Linda implementations, but efficiency would be lost, as array and pointer optimisations contribute significantly to performance in realistic examples. The translator tool was designed to take advantage of these optimisations, generating slightly different solutions for each target architecture.

4.4 Performance Adjustment

The performance adjustment features offered by ParaDE focus around the issue of granularity. Granularity is reflected in the amount of parallelism exploited in a program, and two main approaches to exploiting parallelism can be adopted - task or data parallelism. These approaches to parallelism were described earlier, in Chapter 2, but broadly correspond to the division of computation by distinct functionality (task parallelism) or by sections of a set of data operated on by the same function (data parallelism). Two techniques proposed here for performance adjustment address each of these parallelism types. Actor folding is a technique for gathering tasks together to increase grain-size, while data partitioning concentrates on the division of data across identical tasks and the subsequent grouping of data to reduce the number of tasks actually executed. The specification of the size of data block being partitioned and the number of blocks grouped together for a single process are the granularity adjusters for this second technique. For both actor folding and data partitioning the focus is on specifying the design to maximise parallelism, and then reducing the parallelism to improve performance on a particular architecture. The two performance adjustment techniques are now described in more detail.

4.4.1 Actor Folding

The concept of adjusting granularity by changing the computation time of a process is not new - in fact it is an integral part of producing efficient solutions in any parallel language. However, conventionally, the execution time of processes executing in parallel was adjusted by rewriting major sections of the code, combining or splitting sections of code by hand, with all the complicated changes to input and output data to compensate for the new process structure.

The key point behind the actor folding technique is that the user has the ability to adjust granularity without changing the design graph or any of the code written. Instead, a graphical facility is offered whereby the user simply selects a group of actors using the mouse, and clicks on the translate button to produce a new code translation. Figure 4-3 shows a graph with a group of actors selected for folding - actors 0, 1 and 8 are highlighted after being selected by dragging a rectangle around the group using the mouse.

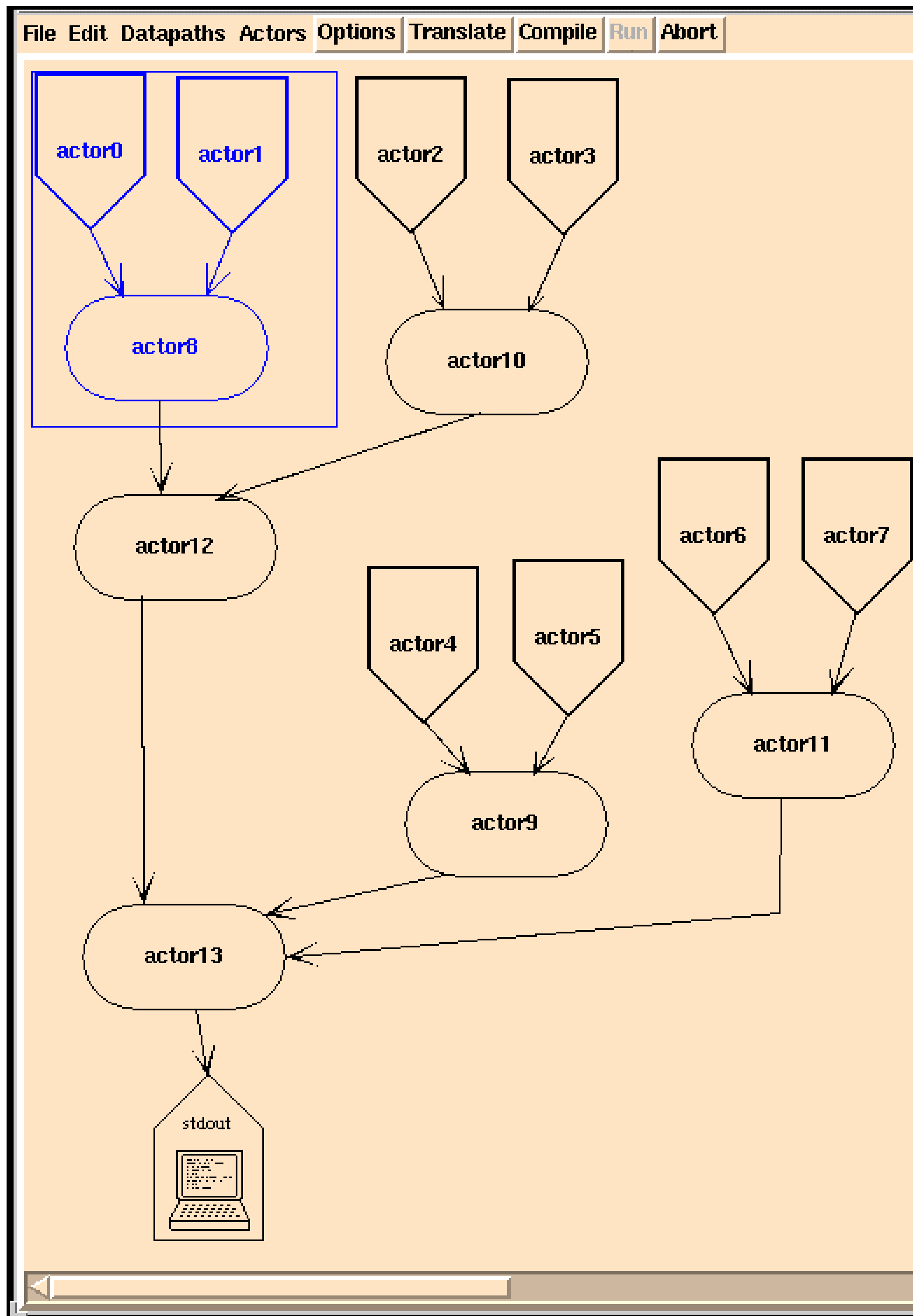


Figure 4-3 Folding a Single Group of Actors

All merging of method code and adjustment of data input and output is handled automatically by the translator tool. The actual process of translating folded actors is quite complicated, involving the checking of data dependencies and careful ordering of the sequential sections of code and input and output data to ensure semantic consistency and prevent deadlocks which might otherwise be introduced. Following folding, the roadmap is regenerated, removing all but one of the actors in the folded group, and the method code of the eliminated actors is placed into the single process.

A forward and backward pass of the original roadmap checks the data connections within the new process, and between the sections of the new process and remaining actors in the new roadmap. Data connections between actors folded into the same group are optimised out, removing the Linda communication mechanisms, but leaving the data variables associated with each datapath. In this way, data communication is localised, with input and output data being passed via variables local to the new process (in fact often the data movement is as trivial as associating a new pointer to an array). This data optimisation can cause a significant reduction in the overheads caused by communication between processes, in addition to the improvement in performance due to the general change in the computation time of a process .

One problem introduced by merging code sections, and relying on data variables to manage the data flows, is the conflict between identical variable names. For automatically generated names this is not a problem, in fact it is a benefit, as the same data variable name is used for the output from one actor and the input to the next - in effect the data “communication” has occurred transparently. Where it may become a problem is in the multiple use of a user-defined local variable. The effects of the variable’s use in the first method code may influence its value for the second method code. In practice, mechanisms have been built in to minimise these effects - checking for duplicate names, enforcing single declaration, and an automatic initialisation-before-use policy. As a safeguard, the user is warned of duplicate names, and advised to make variable names unique. An alternative approach, whereby each actor’s method code is placed into a compound statement to isolate the scope of the variables, was rejected for the ParaDE prototype as the host language (C) did not support the declaration of variables between program statements.

4.4.2 Data Partitioning

Chapter 3 described how data distribution was achieved for the depth actor, by providing a set of commonly used partitioning strategies. The user selects a template of a data pattern, aligns this with the original data structure then indicates the replication pattern across the data structure. These steps can be considered as algorithm specific - they are fundamental to the design of the algorithm and its data, but take no account of the performance of a solution. A final stage incorporates granularity adjustment which, either alone or in conjunction with actor folding, dictates the efficiency of the executable code generated by the translator tool.

In the same way that a group of actors is selected for folding into a single process, a group of templates can be selected, using the mouse, to be grouped together. This group of templates represents the section of data which is to be processed by a single copy of the actor performing the computation. Figure 4-4 shows, in the bottom half of the window, how a group of column-size templates has been selected to form a data block for each copy of the depth actor. In this example, the matrix representing the whole data structure is displaying a small section of a much larger structure, over which the data partitioning would be replicated.

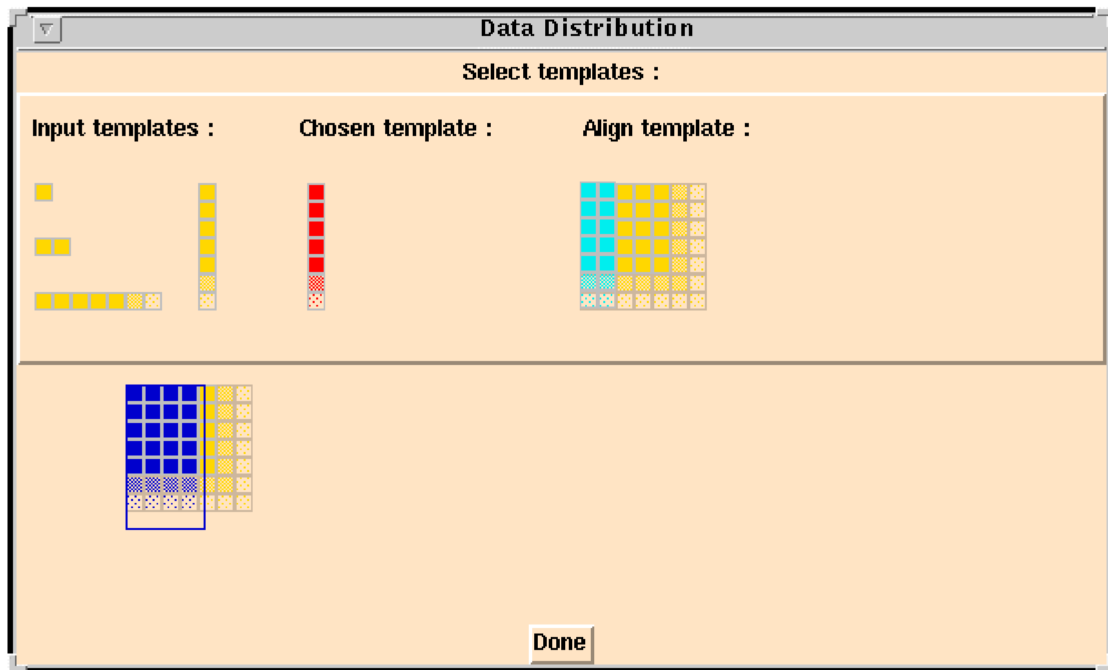


Figure 4-4 Partitioning of Data for Granularity Adjustment

Each copy of the (depth) actor computes a set of result data for the block of input data supplied. The computation - specified by the method code for the actor - remains the same. The difference occurs in the translation procedure, where the method code becomes enclosed in a loop, with each iteration of the loop processing a single unit segment (template-size block) of data. As with the actor folding technique, the inter-process communication is optimised out, with the same process performing multiple computations without multiple input and output of data. The data is supplied as a single block, and partitioned at the start of the process. Similarly, each block of result data is collated into a single output data block. A final data gathering step occurs at the beginning of the actor following the partitioned section, to bring together all of the blocks of result data. These partitioning and gathering stages occur transparently, and are hidden within the translated code. This means that the user effort required for adjusting the size of the data blocks for each actor is minimal, in fact a simple mouse-guided selection on the graphical representation of the whole data structure is all that is required. The user-written code and the program graph remain unchanged, with the translation tool automatically re-generating the new solution based on the graphical selection of the data blocks.

Optimisations for the different Linda implementations can be exploited to take advantage of the features of the target architecture. In particular, the mechanisms for handling large arrays vary, and slightly different tuple matching rules apply. These differences have a significant effect on the data partitioning technique, as very large data structures may need to be manipulated without increasing the overheads in data management. For example, one data manipulation optimisation is where different sections of an array can be extracted using offsets from a pointer to the head of the array. This style was used to implement some of the data partitioning strategies, and can be extended to handle whole dimensions of multi-dimensional arrays.

4.5 Summary of Performance Issues

This chapter has outlined a number of issues concerned with the efficient implementation of parallel programs over different target architectures. The important factors in achieving performance using abstract notations were introduced, before a discussion of methods used in recent developments in the graphical programming field. The inadequacies of current graphical systems was explored and put into perspective by some actual performance facts and figures. The problems identified were then addressed by introducing features of ParaDE designed to overcome limitations in performance. Automatic code generation was described as a method for making architectural/language differences transparent to the user, combined with the use of ParaDE's architecture-independent notation described in earlier chapters. The problems of efficiency associated with portable solutions were addressed by introducing two new graphical techniques for adjusting performance - actor folding and data partitioning. These two techniques were shown to adopt graph-level manipulations of the program to vary the granularity of the program in two distinct ways. The first, adjusting computation time, adopted a task-level parallelism approach and was addressed by the actor folding technique. The second, adjusting data size, adopted the alternative parallel model - data parallelism - and was addressed using strategies for data partitioning and grouping.

This chapter has demonstrated how these granularity adjustment techniques require no knowledge of the intricacies of the target architecture, and no skills in architecture-specific language mechanisms. In this sense, although performance tuning is by definition machine specific, the techniques introduced are themselves independent of language or architecture. The next chapter will investigate the success, or otherwise, of the performance of solutions developed in ParaDE, and the effects of using granularity adjustment techniques to fine-tune performance.

5. Results

Previous chapters have described the parallel design system, ParaDE, developed in this research for the purpose of improving techniques for parallel software design. This chapter evaluates the new methods used in ParaDE against the initial objectives proposed in Chapter 1, and presents the findings of a range of experiments used to determine the effectiveness of ParaDE.

As this research covers a number of different areas of concern in parallel programming, this chapter will begin with an overview of the initial objectives in Section 5.1. This is followed by a description of the implementation environment in Section 5.2, and an overview of issues in performance measurement relating to this work in Section 5.3. The main body of the chapter proceeds in Sections 5.4 to 5.7 with an analysis of the findings in each of the areas addressed, and a summary of the results in Section 5.8.

5.1 Objectives

Earlier chapters identified three major areas of concern to developers of parallel software, which will be addressed by the results in this chapter :

1. the ease of use and suitability of ParaDE in developing parallel software,
2. the performance of the parallel software developed using ParaDE,
3. the portability of the software generated by ParaDE for different parallel platforms.

The ease of use of ParaDE and its suitability for parallel software design are the most subjective and most difficult to define of the three areas listed. In particular, there are two trade-offs which require special consideration:

- between developing new parallel programming constructs or languages to implement parallel behaviour and taking advantage of existing programmer skills in commonly used languages. (Both points of view have influenced parallel software - the former leading to specially designed parallel languages such as Occam, the latter leading to parallel extensions of existing languages such as Fortran).
- between abstract, high-level program descriptions easy to grasp by users, and detailed, low-level program descriptions useful for achieving good performance on a particular architecture.

The purpose for which a design system is intended affects the balance of these trade-offs, and Section 5.7 reiterates the purpose of the ParaDE notation and graphical emvironment. The section then describes the results of investigations into the suitability of ParaDE, and presents a complex problem to demonstrate the practicality of the tools and features comprising ParaDE.

Performance is the issue most prominent in parallel software research, as it is the key motivation behind the use of parallel software. Most of this chapter therefore concentrates on the performance achieved by parallel software produced using ParaDE.

Often, parallel software evaluations focus purely on the performance of a piece of software on a given architecture. Alternatively, they may present the performance of different software solutions developed for different architectures. These approaches, whilst addressing the direct performance issue noted in point two, fail to take account of the third point - the portability problem. In contrast to this, Sections 5.4, 5.5, and 5.6 will demonstrate how ParaDE can exploit the same user-solution across different architectures. Furthermore, it will be demonstrated that the graphical programming environment can be used to *optimise* the performance of a given program. This is achieved using ParaDE, without changing the original design or user implementation. It will be shown that by providing mechanisms for fine-tuning the performance, the abstract notation proposed can give rise to both portable *and* efficient solutions.

Section 5.3 describes the main issues involved in determining efficient performance, before Sections 5.4 to 5.6 focus on particular features of ParaDE, namely data partitioning and actor folding, and demonstrate how these features have been used in experiments to adjust the performance of two test programs involving matrix calculations.

5.2 Implementation Environment

With architecture independence being one of the key objectives of this research, two different parallel architectures were chosen to form the basis of a comparison of performance on different architectures, and the implications of performance adjustment in significantly different environments. The two architectures used are now described.

5.2.1 Encore Multimax

The Encore Multimax 520 is a shared-memory multiprocessor, with 12 fully symmetric processors and a shared memory. The processors are 30MHz NS32532s, and are connected to the memory by a common bus architecture with a typical transfer rate of 100MBytes/second. Two processors reside on each processor card, sharing 256KBytes of cache memory (which is transparent to the user).

The Multimax runs UMAX4.3 operating system, a Unix variant, which, as well as supporting Unix process-level parallelism, also provides a finer granularity parallelism using threads. The Encore Parallel Threads library supports a lightweight process scheme, running within Unix processes, and provides scheduling of threads within processes. The Linda run time system on the Multimax uses threads, although as a Beta version performance is probably not optimal. To help minimise performance degradation due to external loads, experiments were run at times of minimal usage - during the night and at weekends, as the machine is in general use in the Computing Science department, providing the main multi-user environment for staff and students.

5.2.2 Cluster of Workstations

A cluster of 10 Hewlett Packard Unix workstations provides the alternative architecture, with both processing power and memory distributed among the workstations. The ten workstations have a license to run Network Linda, produced by Scientific Computing Associates, allowing the user to treat the distributed resources as a single entity. These machines are actually part of a larger cluster in general use by

the Medical School and other departments, part of the University Computing Service. Again, the load from other users was minimised by judicious scheduling and repetition of experiments.

Apart from small differences in syntax for array handling between the two Linda systems used, the intermediate programs generated for the two different architectures are broadly equivalent. Thus, comparisons between results on the two architectures can be made, both with respect to the overheads of the Linda system, and the efficiency of a given program on each system after the ParaDE tools have been used to adjust the performance.

5.3 Measuring Performance

The performance of a parallel program is usually judged by **speedup**. Chapter 1 introduced speedup, described further in Chapter 2 as a measure of the increased performance of a parallel program over its sequential counterpart. However, a number of alternative approaches to determining the sequential element have been presented, namely:

1. the parallel program executed on a single processor of the parallel architecture,
2. the parallel program with the parallel mechanisms removed, executed on a single processor of the parallel architecture,
3. a different program, using a sequential algorithm, run on one processor of the parallel architecture,
4. a version of the parallel algorithm optimised for sequential execution, executed on a single processor of the parallel architecture.

One of the main objectives of this work is to produce architecture-independent parallel software, and to this end, the Linda parallel programming model, described in Chapter 2, was chosen as an intermediate software model to which parallel programs are transformed by ParaDE. ParaDE, therefore, has limited control over the performance of either a sequential program running on the Linda run-time system, or the overheads introduced by Linda constructs implementing process management and communication between processes. For this reason, it was deemed appropriate for most experiments to base the speedup measurement on the first option - keeping parallel constructs in the program, but running on a single processor (although some granularity adjustments do optimise out certain parallel mechanisms). Any improvements in the basic performance, or reductions in communication overheads, due to future developments by the designers of the Linda system would be reflected directly in programs developed using ParaDE. What it is necessary to demonstrate, however, is that the transformation from design notation to intermediate code in Linda does not introduce significant additional overheads which degrade performance. This will be considered in the analysis of the results presented in subsequent sections. For completeness, comparisons are made to speedup results relative to pure sequential (hand-written) programs, described by option three in the above list, as this is a measure often demanded to show absolute performance.

5.4 Experiment 1 - Matrix Multiplication with Data Partitioning

5.4.1 Aims

This first experiment aims to demonstrate the effectiveness of the data partitioning technique (described in Chapter 4), showing how data partitioning can be used to adjust grain-size and improve performance. Comparisons between a Shared Memory implementation and a Workstation Network implementation will be made to identify the performance benefits available from each architecture, and so demonstrate how a program developed with ParaDE can be executed efficiently on different architectures.

5.4.2 Description

Matrix multiplication is commonly used as a test program for parallel programming systems. The reason for this is that it is a relatively simple programming problem, which is scaleable - i.e. the matrix size can be increased to investigate the effect of performance for increasing computation size. Of course, communication overheads could also increase with matrix size, but with an appropriate algorithm (described below), the computation-to-communication time ratio should be large enough to achieve efficient parallel execution. Furthermore, ParaDE can (transparently) take advantage of optimisations for each architecture. For example, one optimisation uses pointers to reduce the effect of communication costs for large data structures, by eliminating some data copying in tuple space operations.

Figure 5-1 shows the program graph for matrix multiplication, which consists simply of two source actors, supplying matrix A and matrix B to a depth actor, MatMult. The depth actor method code performs the multiplication using the standard algorithm: for each row of Matrix A, sum the products of each element in the row multiplied by the corresponding element in the column of Matrix B. An obvious data partitioning strategy therefore, is to split the Matrix A into rows that can be processed in parallel by a depth actor. Matrix B is supplied whole as an initial granularity choice appropriate to the algorithm. The partitioning of Matrix A is indicated on the datapath by the multiple arrowhead, whereas Matrix B is supplied whole so has a single arrowhead. The other differences in the datapaths are that Matrix B is supplied on a continuous datapath, indicated by the solid black arrowhead, so that the matrix is able to be read multiple times, and is not consumed by the MatMult actor. In contrast, the datapath supplying Matrix A to the MatMult actor is a discrete datapath, indicated by a normal (non-filled) arrowhead, meaning that the data on this path is consumed by the destination actor. Datapaths to a depth actor must be one of these two types, either supplied whole on a continuous datapath, or partitioned on a discrete datapath, in order to satisfy the firing rules, described in Chapter 3.

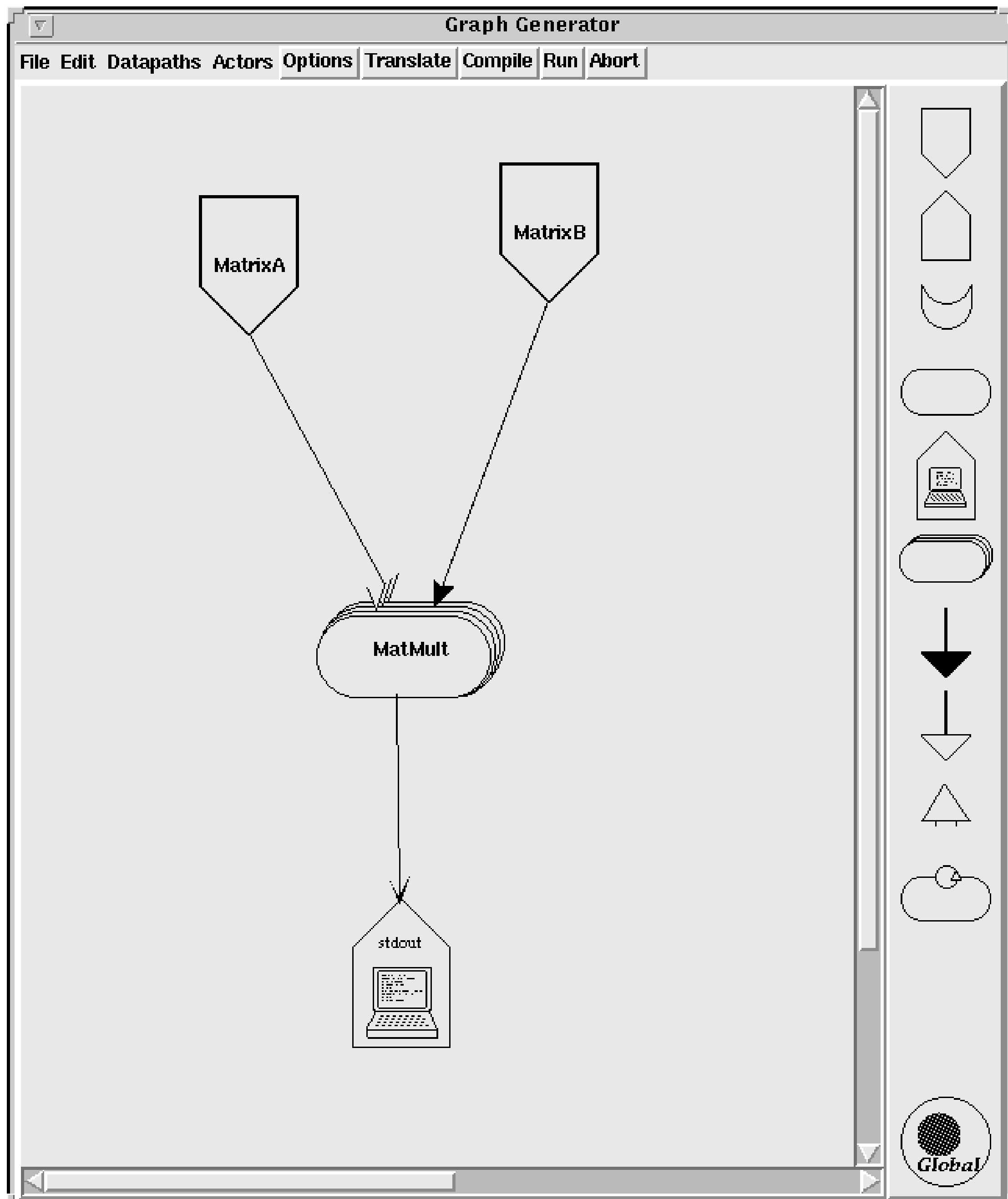


Figure 5-1 Matrix Multiplication Graph

The granularity of the program can be adjusted by changing the number of columns passed to any one depth actor instance. For example, with a matrix size of 10x10 elements, a partitioning using a single row of Matrix A would result in 10 processes being invoked, each calculating one column of the result matrix. A partitioning selecting the whole of matrix B, however, would lead to a single instance of the depth actor, and a sequential execution - parallel mechanisms removed and execution on a single processor. An intermediate granularity solution could use a partitioning of 2 segments of 5 columns each. The results below show the execution times and speedups for increasing matrix size over different column partitions, firstly for the Shared Memory implementation, and secondly for the Workstation Network.

5.4.3 Shared Memory Implementation

Tests were run for matrix multiplication using square matrices of size 100 to 400 floating-point elements, increasing in steps of 50. A 400-element array was the largest data structure supported by the tuple space of the Shared Memory implementation of Linda. The lower bound of 100 was chosen as it generated the smallest execution time that could reliably be measured. The following graph shows the speedup for 1, 2, 4 and 8 partitions - where speedup is measured relative to the single partition. More than 8 partitions were difficult to measure due to the limited number of processors (discussed below).

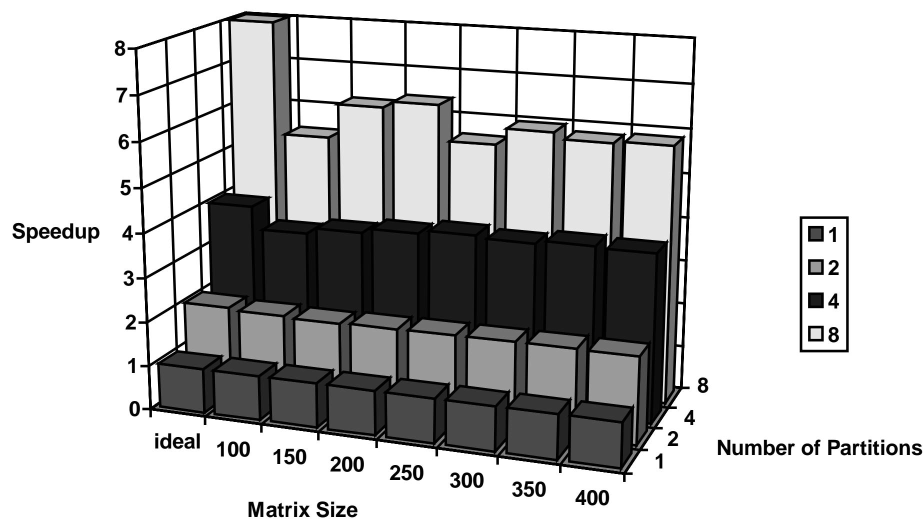


Figure 5-2 Matrix Multiplication Speedup (Shared Memory)

The shared memory implementation of matrix multiplication shows good speedup for partitions of two and four segments, achieving close to ideal speedup for all matrix sizes, but with some degradation for the 100-element matrix. This degradation is due to the relatively small computation time being affected by the parallelism overheads. As matrix size increases, and the computation increases proportionately, communication costs become less significant and speedup improves.

For a partition of 8 segments, speedup becomes less consistent, and drops significantly below the ideal. The inconsistency is partly due to the employment of nearer the maximum number of processors, increasing the effects of machine load from other users (which still existed despite attempts to minimise them). Where not enough free processors are available, or the load is high on some processors, the run-time system may schedule more than one actor to a single processor. The delays due to this sequential ordering of actor execution have an increasing effect as more processors are used. Grain-size issues also become increasingly significant, as doubling the number of partitions halves the size of the data block being processed by each actor. In addition, Amdahl's law plays a part, as, even in this highly parallel application, sequential sections such as the source, sink and (transparent) partitioning code limit the maximum amount of parallelism achievable.

Comparing the parallel execution times with a hand-written sequential program gave the results shown below.

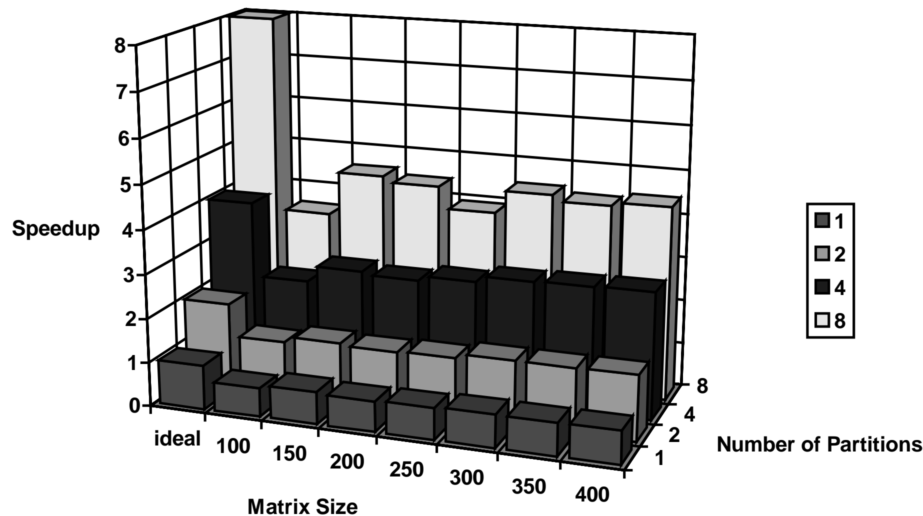


Figure 5-3 Speedup Relative to Pure Sequential (Shared Memory)

It can be seen that, for this example, speedup is reduced when comparing with the hand-written sequential program, but not by large amounts. For 8 partitions, speedup reaches just over 4 instead of around 5.5 on the previous graph, and 2.5 instead of 3.5 for 4 partitions. These observations indicate that, as expected, the parallel run-time system does introduce additional overheads to a sequentially run program, but these overheads are small enough for significant speedup to be achieved. Furthermore, the second speedup graph shows that while speedup is reduced, performance improvements are still scalable over the small sample of processors used.

5.4.4 Network Linda Implementation

The matrix multiplication experiments were repeated on the Workstation Network architecture. In this set of experiments, the matrix size ranged from 100 to 750; again, the lower limit was chosen to provide computations large enough to measure reliably. Although the Shared Memory implementation could only support up to 400-element arrays, it is useful to observe the trends exhibited by the Workstation Network at larger array sizes, as will be demonstrated below. Again, speedup was measured relative to the single partition, but only 1, 2 and 4 partitions were used. For this experiment, the 8 segment partition was abandoned, due to the unreliability of the workstation cluster - the Linda run time system was able to cope with dynamic changes in the network configuration, but the number of functional nodes fell below the 8 required to demonstrate this partition properly.

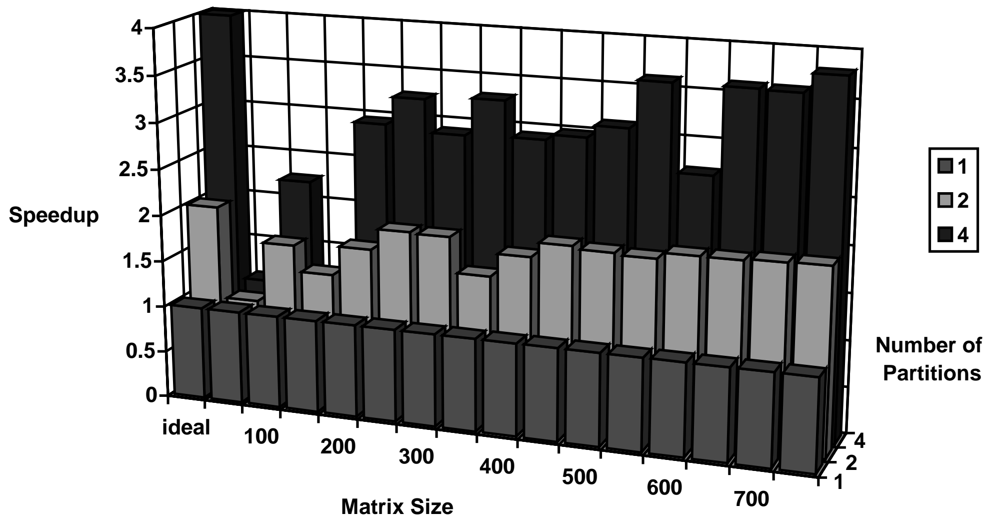


Figure 5-4 Matrix Multiplication Speedup (Network)

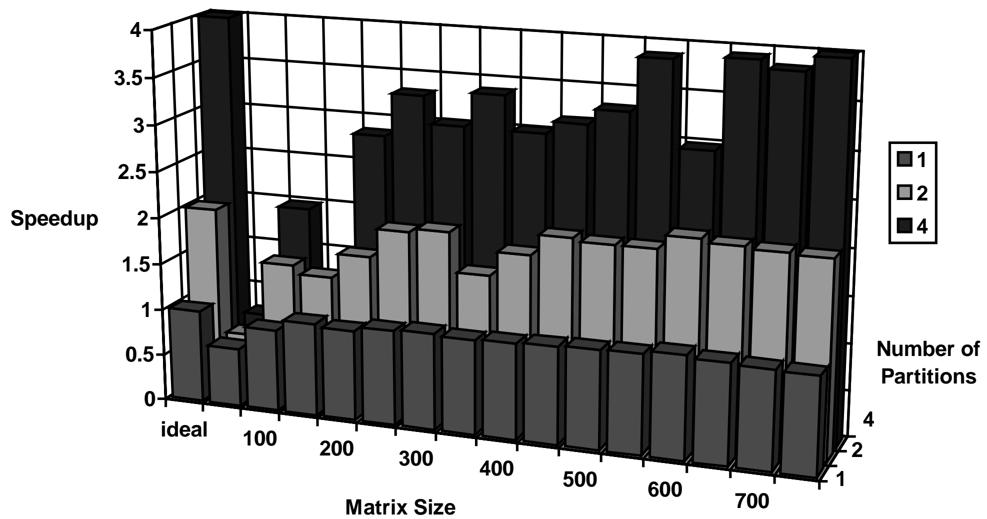


Figure 5-5 Speedup Relative to Pure Sequential (Network)

The Network implementation shows good speedup for both 2 and 4 segment partitions at large array sizes. As with previous experiments, smaller array sizes performed less well, due to the reduced computation/communication ratio, and less consistently, due to the interference effects from other users, as also experienced with the shared memory implementation. Performance began to stabilise at around the 200 element matrix size, and a general, but small improvement was observed within each partition as matrix size increased to 750.

5.4.5 Comparison of Network and Shared Memory Results

While the different restrictions of the two architectures mean that the experiments do not match completely in the range of matrix sizes and partitions investigated, there is sufficient overlap to make some interesting observations:

1. The shared memory implementation showed good speedup beginning at smaller array sizes than for that of the distributed memory system. Near optimal speedup for 2 and 4 segments, and a reasonable speedup for 8 segments was achieved as low as for a 100-element array.
2. In contrast, the Network implementation only achieves near optimal speedup for a 2 segment partition at a matrix sizes above 250, and speedup for a 4 segment partition shows a gradual increase from 250, approaching near optimal at 750.
3. The broad pattern of these results is as expected, as the lower overheads of communication and process management in the shared memory system allowed better speedup for the parallel execution of smaller computation units.

These observations support the initial aims of this experiment: that ParaDE provides easy-to-use graphical techniques which allow the abstract and flexible specification of data parallelism; that adjustment of the parallelism can be applied at the graphical level to vary grain-size, independently of the underlying architecture; and that this adjustment allows the user to easily target efficient performance for different parallel architectures. Appendix A presents the generated code for both architectures, illustrating the similarity of the two programs produced by ParaDE, and the context of the user and system generated sections of the code.

5.5 Experiment 2 - Multiple Matrix Calculation (Encore Multimax)

5.5.1 Aims

This experiment demonstrates the use of the actor folding technique (described in Chapter 4). Results will show how the grouping of a set of actors can improve the performance of a parallel program, by adjusting the grain-size of the program using a process-oriented method. In conjunction with the next experiment on the distributed memory system, this will again demonstrate how portability can be achieved, giving rise to efficient execution of the same parallel program on different architectures.

5.5.2 Description

The algorithm used for this experiment is a simple combination of matrix operations, producing the result ((A*B)*(C*D))+(E*F)+(G*H) where A, B, C, D, E, F, G and H are matrices. The graph for this program is shown in Figure 5-6.

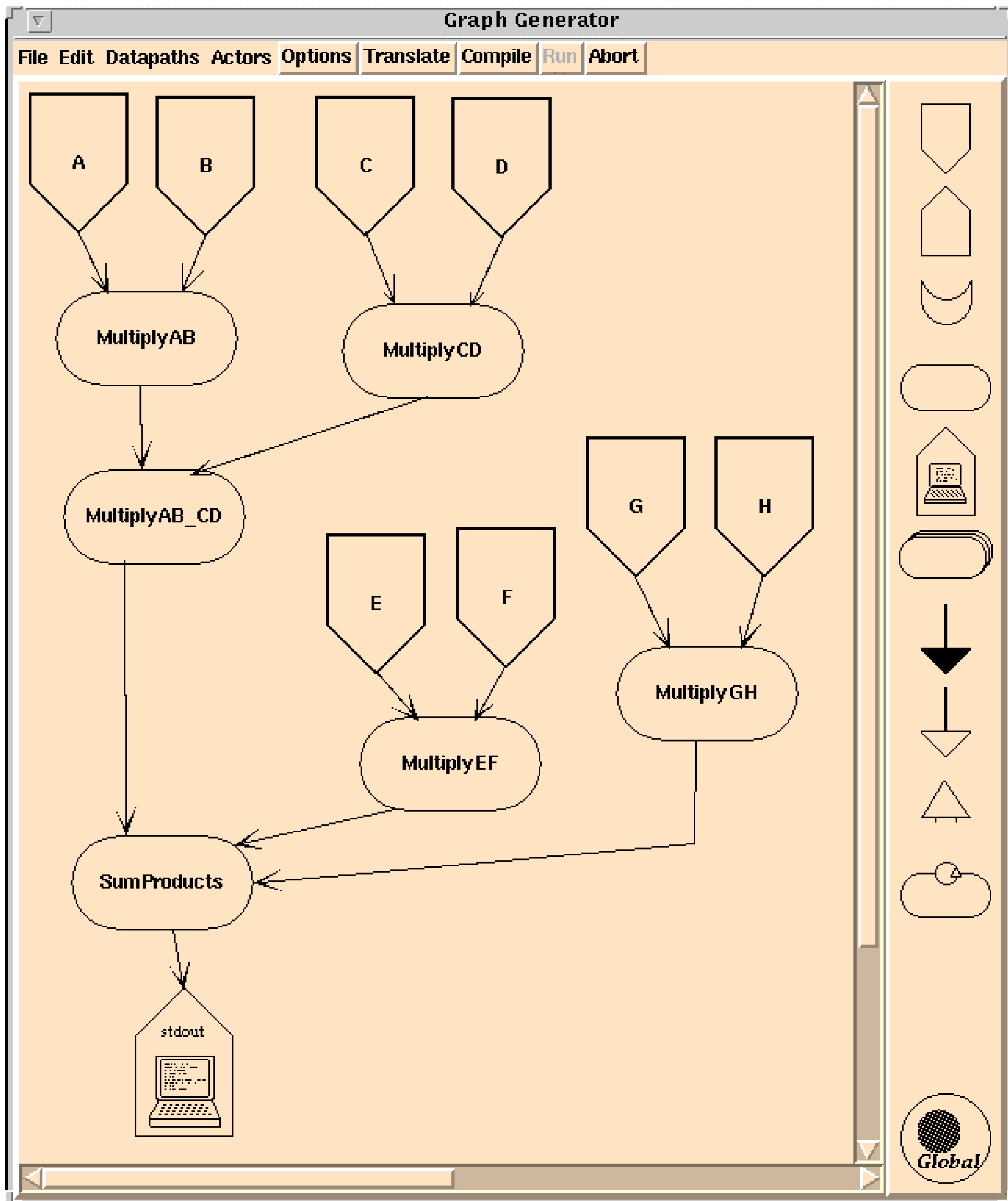


Figure 5-6 Matrix Calculation Program

Each general-purpose actor in the program graph represents a matrix operation - either multiplication of two matrices or addition of three for the SumProducts actor. The source actors merely initialise the matrix elements, and supply the matrix to the operations. A single stdout actor outputs the overall result.

The natural parallelism in this program is evident from the graph, with all the first level operations being data independent (MultiplyAB, MultiplyCD, MultiplyEF, MultiplyGH), then MultiplyAB_CD which is dependent on MultiplyAB and MultiplyCD, and sequential sections for SumProducts and stdout. This program, whilst simple in nature, offers a number of possibilities for grouping actors together, to enforce sequentiality and hence to increase grain-size on a group of otherwise independent operations. The following sets of experiments investigate the effects on execution time and speedup of a number of different actor groupings chosen by the

user. Matrix sizes of 5 to 200 elements square were used to see the effects of larger computation blocks on the execution time. Of course, this matrix size variation also increased the data size - with a corresponding increase in communication costs. An additional variant was therefore used to magnify the computation block size with no corresponding increase in the data size - a dummy loop was included in the inner calculation loop of the method code. This dummy loop used factors of 1, 5 and 10 to show the effects of increasing the computation to data ratio. Graphs showing the measured execution times and speedup are shown later in this section, but first, each folding strategy is described.

5.5.3 Folding Strategy: not-folded

This first test investigates the natural parallelism of the program, timing the execution of the original program, with no granularity adjustments. Figure 5-6 above shows the program graph. The theoretical maximum speedup of this fully parallel version of the program can be estimated by comparing the sum of the orders of all sections of the program, with the sum of the orders of the sections in the *longest path* of the parallel version. This can be explained more clearly by following through the calculation below.

Taking the source and sink actors to be linear computations - O(n), matrix summation to be proportional to the square of the matrix size - $O(n^2)$, and matrix multiplication to be cubic - $O(n^3)$, the total order of the sequential program is the sum of the order of all actors:

$$T_{sequential} = O(9n + 5n^3 + n^2)$$

If all parallelism is exploited, the longest computation path through the program comprises a single source and sink actor, two multiplication actors and the summation actor :

$$T_{parallel} = O(2n + 2n^3 + n^2)$$

The theoretical speedup is then the ratio of T_s to T_p, shown in Equation 1:

Equation 1

$$Speedup = \frac{T_s}{T_p} = \frac{O(9n + 5n^3 + n^2)}{O(2n + 2n^3 + n^2)} = \frac{O(9 + 5n^2 + n)}{O(2 + 2n^2 + n)}$$

For n=1, this equates to speedup=3, but this is not a useful figure, since a matrix of 1 element would not be a suitable candidate for data parallelism. Taking n=50, speedup=2.49, and in fact as n increases, the n^2 terms dominate as shown in Equation 2.

Equation 2

$$\underset{n \to \infty}{Speedup} = \frac{O(5n^2)}{O(2n^2)} = 2 \cdot 5$$

The maximum speedup, therefore, that can be expected from the results for the fully parallel version of the program is 2.5. The speedup graphs comparing the different folding strategies, shown later in

Figure 5-11, will demonstrate that the actual speedup achieved using ParaDE for this not-folded graph does in fact approach 2.5 for larger matrix sizes.

5.5.4 Folding Strategy: fold4 (Folding Source and Multiplication Actors)

Figure 5-7 shows the program graph for the matrix calculation, with the actors within each of the boxes grouped together into a single process. This combines the source and multiplication actors for the four groups AB, CD, EF and GH, removing the inter-process communication between the actors in the enclosing box, and concatenating the method code (subject to satisfying the original firing order).

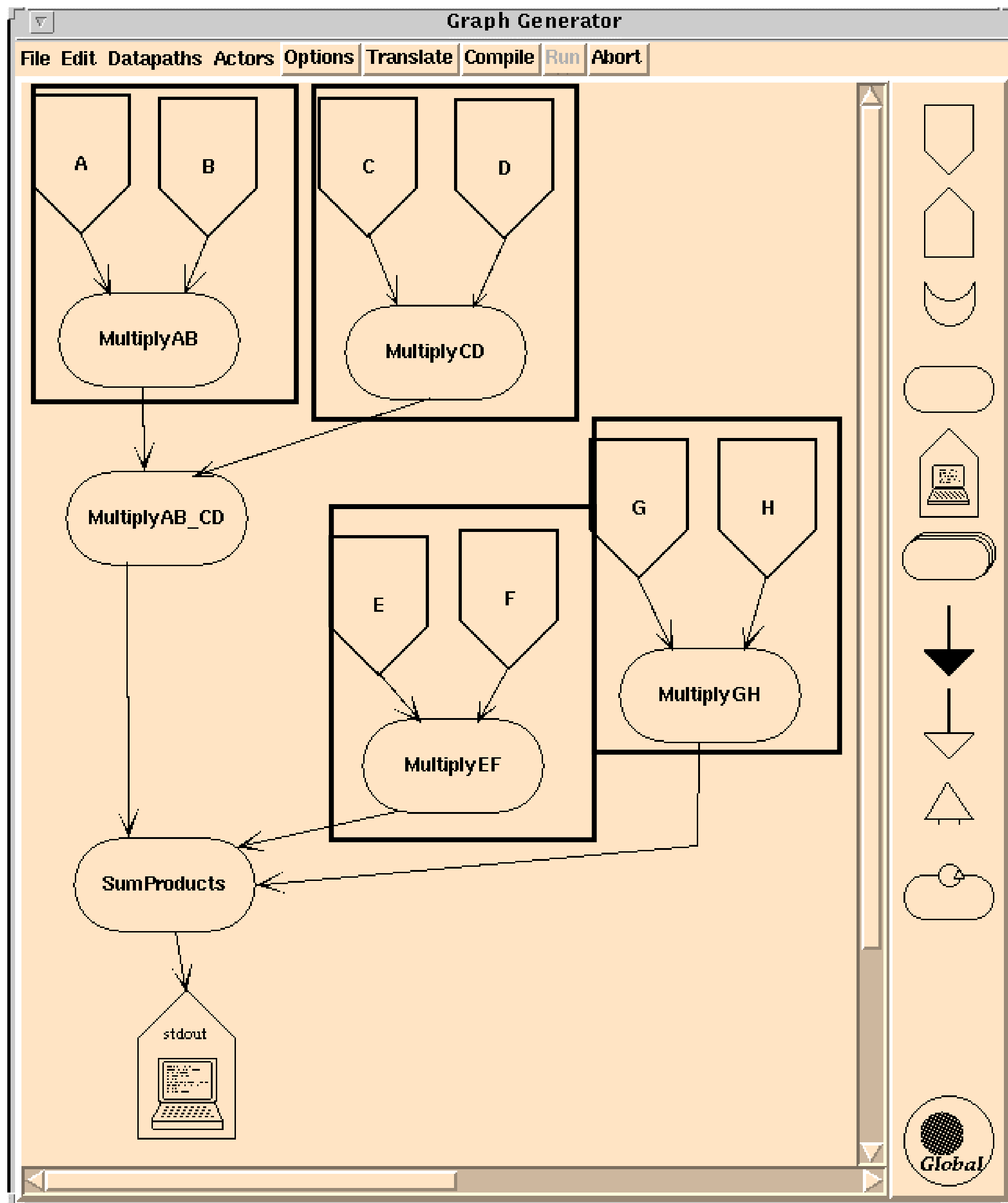


Figure 5-7 fold4 Folded Graph

Using a similar method to the section above for calculating theoretical maximum speedup, this configuration of the program can still achieve a maximum speedup of 2.5, as the folding of two (linear order) source actors into a group with a (cubic order) multiplication actor makes little difference to the calculation as n becomes large. Actual speedup measured reached over 2.4, as shown in

Figure 5-11.

5.5.5 Folding Strategy: fold5 (Folding Source and Multiplication Actors and Sequential Section)

Figure 5-8 shows the program graph for the matrix calculation as the previous folding strategy, but with the sequential section incorporating MultiplyAB_CD, SumProducts and stdout folded into a single process. As these actors required sequential execution due to the data dependencies, the additional effect of this extra grouping would be expected to be small - only the communication is optimised, there is no difference in the parallel structure of the program (the theoretical maximum speedup remains at 2.5). However, this test addresses an interesting problem - that of the communication costs between sequentially ordered actors. Measured speedup for this folding strategy was slightly below the maximum, but still reached over 2.3, illustrated below in

Figure 5-11.

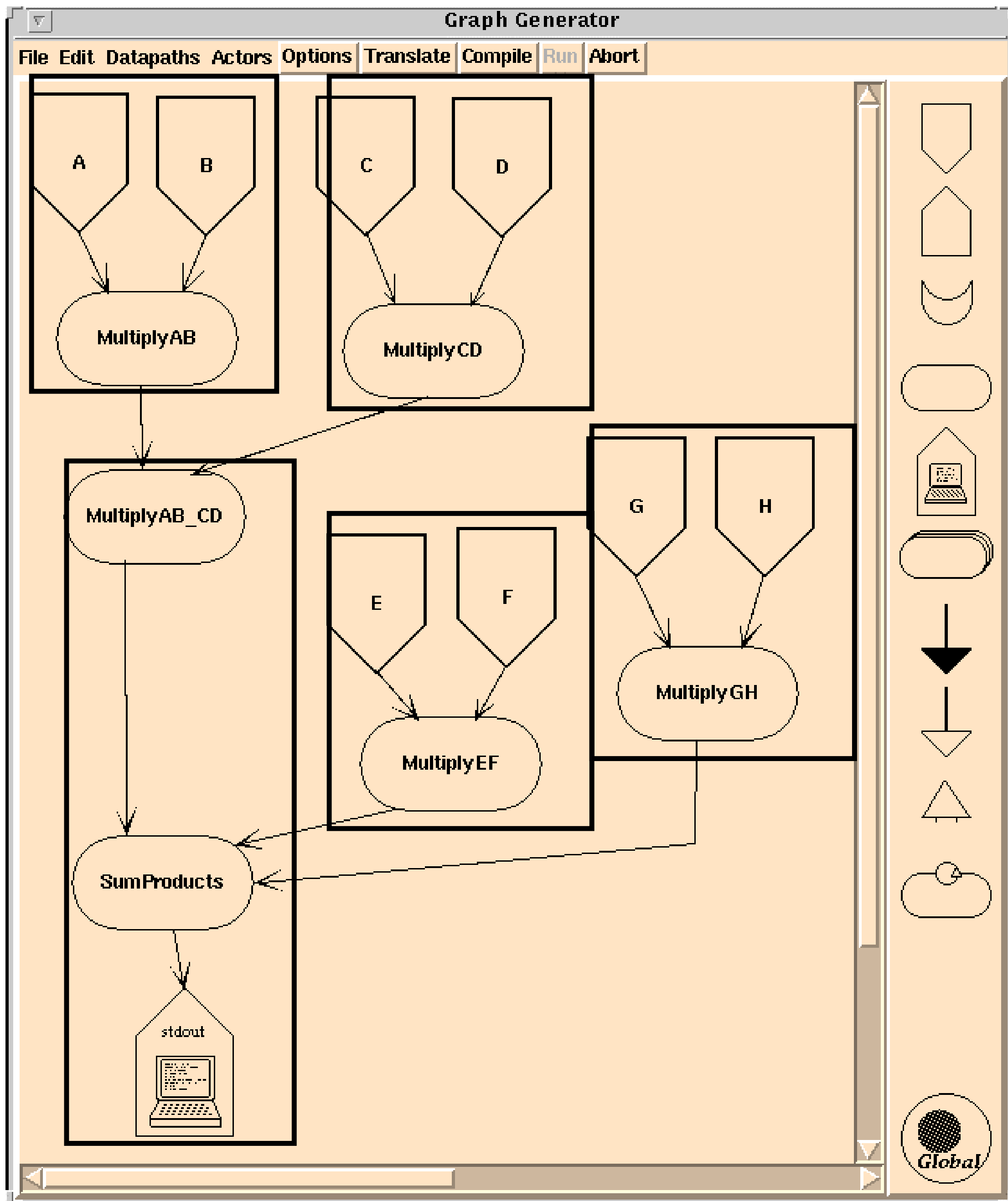


Figure 5-8 fold5 Folded Graph

5.5.6 Folding Strategy: fold2 (folding into two multiplication groups)

This folding strategy begins to limit the parallelism more extensively, grouping the source and multiplication actors for two multiplication calculations into a single actor. As can be seen in Figure 5-9, MultiplyAB and MultiplyCD are folded together, as are MultiplyEF and MultiplyGH, along with the corresponding source actors.

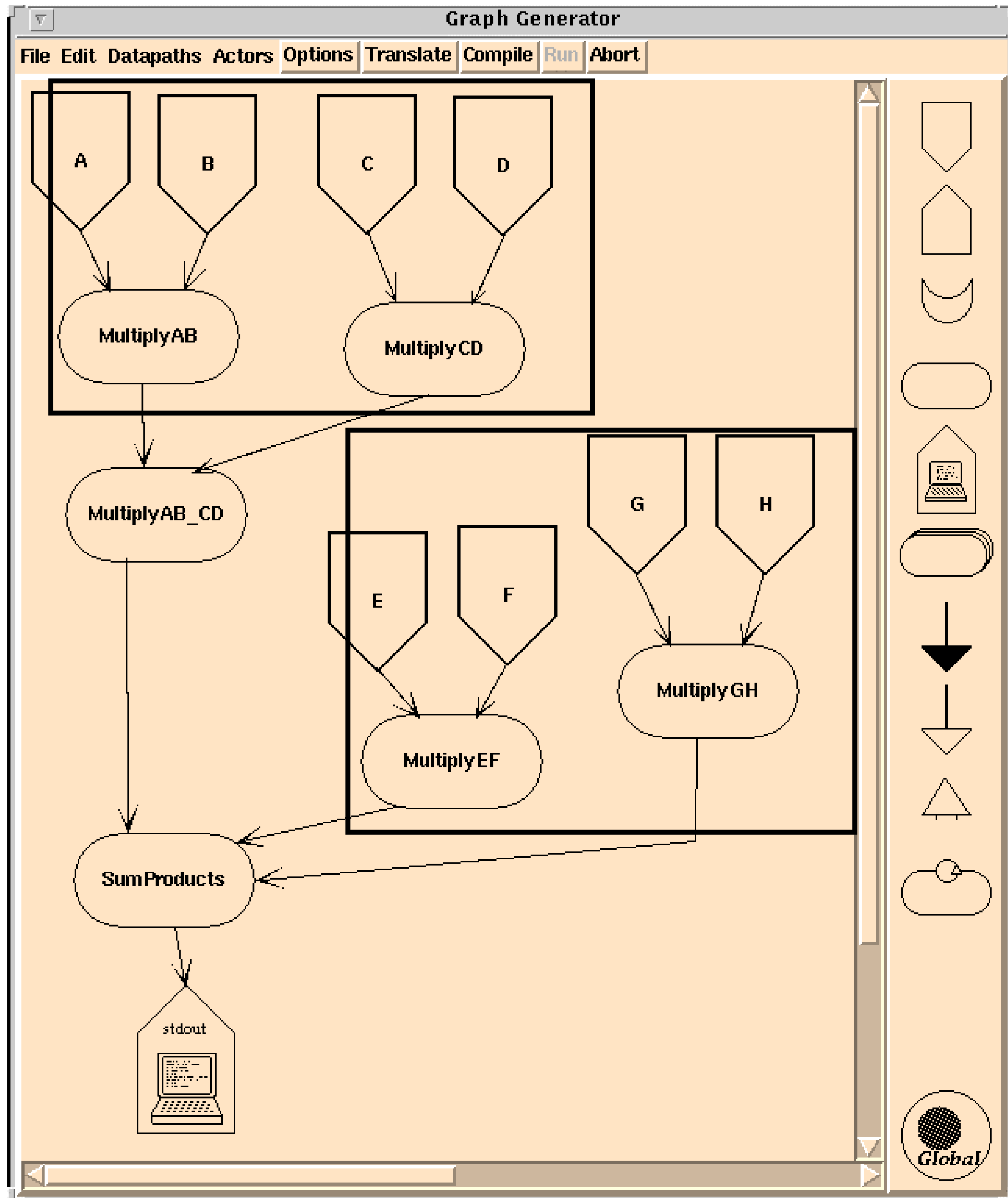


Figure 5-9 fold2 Folded Graph

The theoretical maximum speedup for this grouping is calculated below in Equation 3. In this configuration, three multiplication actors execute in sequence, leading to the reduced maximum speedup.

Equation 3

$$\underset{n \to \infty}{speedup} = \frac{O(9n + 5n^3 + n^2)}{O(5n + n^2 + 3n^3)} = \frac{O(5n^2)}{O(3n^2)} = 1 \cdot 67$$

The actual speedup measured for this parallel configuration, and shown in the speedup comparisons of

Figure 5-11, was in fact very close to this maximum of 1.67.

5.5.7 Folding Strategy: foldall (folding all actors)

This folding strategy uses the same graph program as all of the previous groupings, but folds all of the actors in the graph into a single actor. This means that the entire program executes sequentially, with parallel sections placed in series, and method code sections scheduled according to the original firing algorithm - to preserve the functionality and prevent deadlock. This folding strategy can be thought of as the control measure - timings for the sequential execution of the program, against which all the other strategies, with varying levels of parallelism, can be compared. Figure 5-6 at the start of this section can be used to describe this folding strategy, where all of the program is enclosed within the folding box.

5.5.8 Multimax Test Comparisons - Execution Times

The graphs below show the execution times on the shared memory architecture for each of the matrix sizes used in these folding strategies, from 5 to 200. For each matrix size, the execution time of each folding scheme is plotted, and a separate line is shown for each of the three dummy loop values, 1, 5 and 10. These graphs are shown to illustrate the trends in execution times across the different folding strategies as matrix size changes. More illustrative comparisons of performance can be seen in the subsequent speedup graphs.

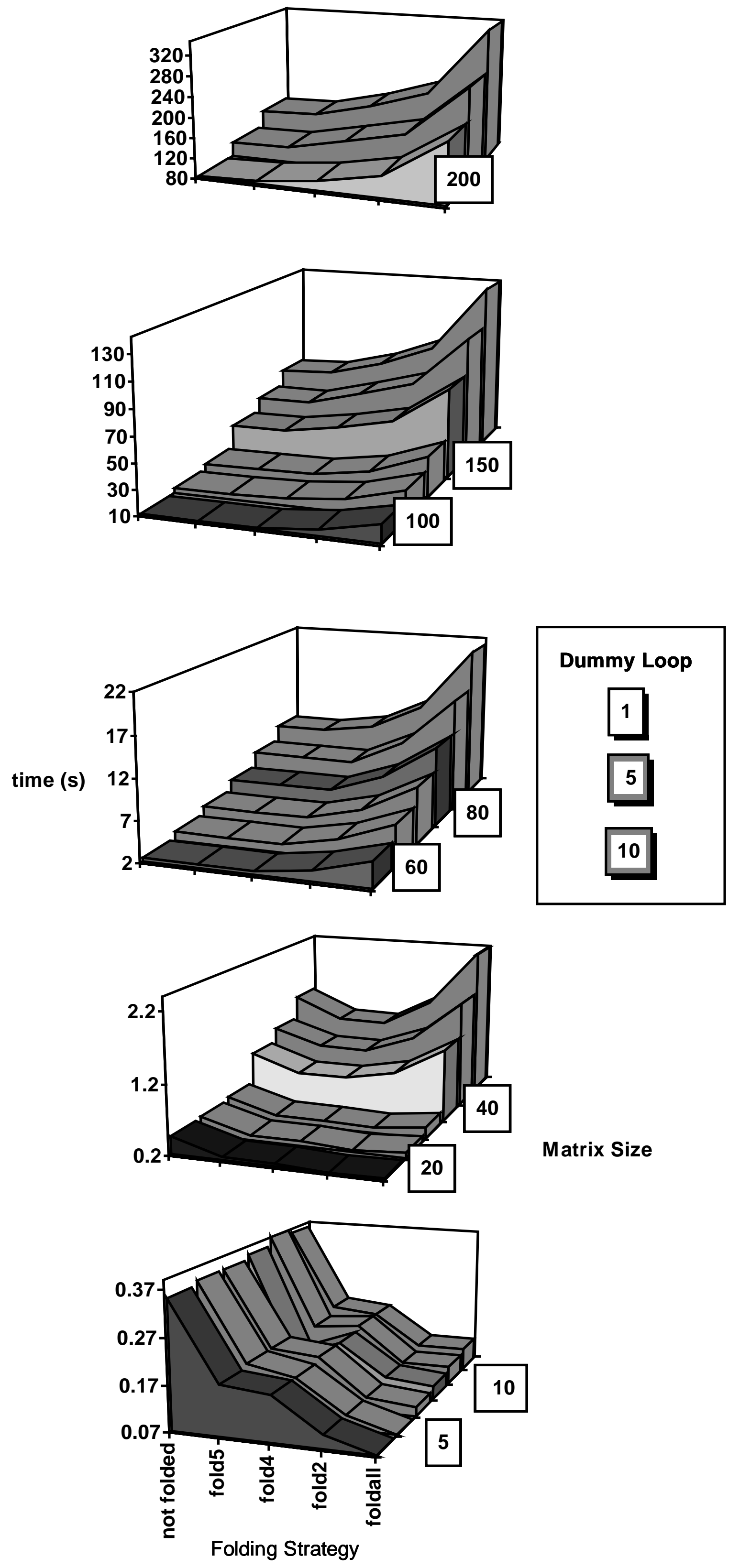


Figure 5-10 Execution Time Graphs (Shared Memory)

These graphs show some very interesting, but not unexpected, results for execution times over increasing matrix sizes for the set of different folding schemes. As will be examined in the next section on speedups, there does not appear to be any folding scheme which is *best* all of the time, that is which provides a consistent improvement in execution time over all the test data sets. In fact, each folding scheme can be seen to give the shortest execution time for one particular matrix size, or sub-range of matrix sizes. For example, the bottom graph of Figure 5-10 shows that, for a matrix size of 5, *foldall*, the completely folded (and therefore sequential) version performs best, and the more parallelism allowed in the program graph, the worse the execution time becomes. This is not surprising, as with a matrix size of 5, the matrix calculation is a small computation, and to compute the result in parallel requires more time in parallelism overheads than can be gained in simultaneous computation.

Progressing through the graphs as the matrix size and the corresponding computation size increases, the folding schemes with more parallelism become optimal. For example, for a matrix size of 40, the optimal folding scheme is from *fold4* or *fold5*, with the four folded sets of multiplication actors. The additional effect of folding the sequential section containing MultiplyAB_CD, SumProducts and stdout, is minimal, as can be seen by the indistinguishable difference in execution time between these two strategies. In contrast, the top graph shows the *not-folded* version - using the natural parallelism of the program - as performing optimally for a matrix size of 200. Again, this is understandable, as with the larger matrix sizes, the time to carry out the computation becomes far greater than the overheads of communication, and the maximum parallelism can be exploited.

These execution time graphs show clearly the wide variation of execution times for each of the folding strategies examined as matrix size increases. It would be expected that execution time increases with matrix size, and also that the exploitation of parallelism would become increasingly significant in improving efficiency as the matrices increase in size. However, the trends illustrated in these graphs pinpoint very accurately the circumstances in which one level of parallelism performs better than another, and, perhaps more significantly, where different levels of parallelism have a negative effect on efficiency. A discussion on the effects of the dummy loop is left until the speedup comparisons below.

The effects of the granularity adjustments shown above are now examined in more detail using speedup measures.

5.5.9 Test Comparisons - Speedup

The following graph shows speedup for the different folding schemes, relative to the execution time of the completely folded (sequential) version of the program. That version has all of the Linda communication constructs optimised out, and only sets up and runs a single process to execute the program. Therefore, the executable code resulting from the translation of this program graph is effectively free from any overheads caused by the parallel run time system (Linda), and can be considered as a reliable indication of a comparable sequentially written program (see Section 5.3).

This comparison for speedup equates to the comparison used in Experiment 1 - data partitioning for matrix multiplication - where speedup measured the execution time of a multiply-partitioned data structure to a singly-partitioned one. This approach meant

that additional processes were eliminated in the "sequential" version, and Linda communication constructs were therefore not needed.

Figure 5-11 shows the speedup for each of the dummy loop values (1, 5 and 10) over the range of matrix sizes 5 to 200. The purpose of this comparison is not only to observe the different speedup for folding schemes, but also to investigate the additional effect of varying the computation size while keeping the data size constant. Whilst the effects of the dummy loop are small, the trends caused by increasing computation size can be observed.

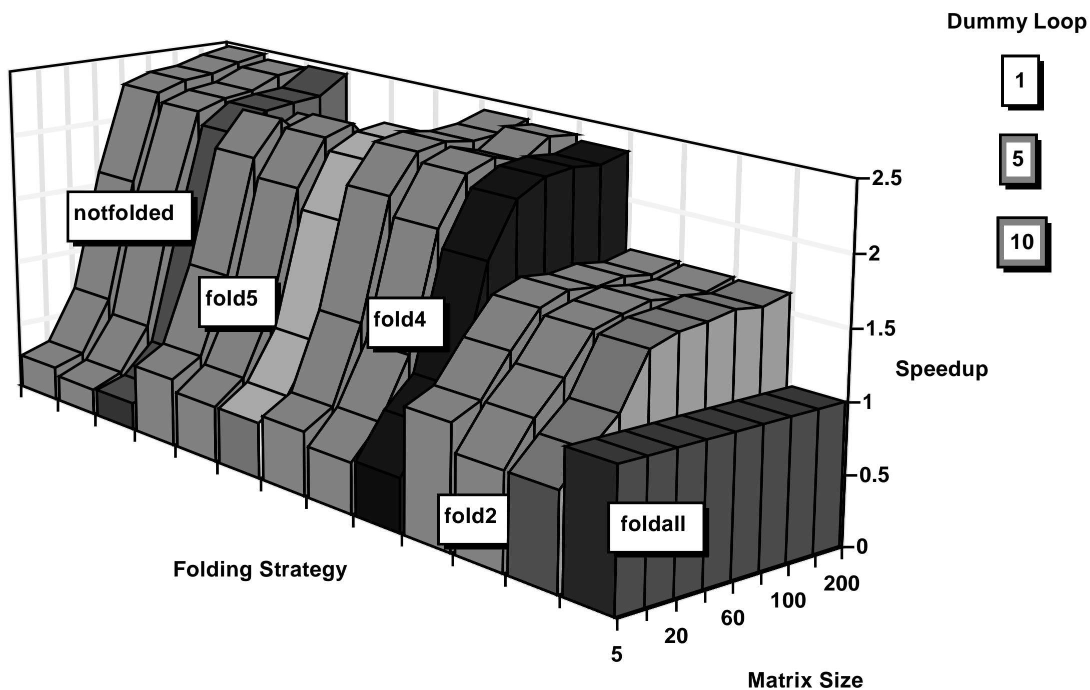


Figure 5-11 Speedup for Different Folding Strategies (Shared Memory)

Generally, across all values of the dummy loop, the graphs enforce the findings of the previous section comparing execution times. Many of these observations are obvious statements of parallel program behaviour, to be expected from any performance investigation of parallel programs. However, it should be borne in mind that the different versions of the program have been generated automatically, and that adjustment of the granularity of a program is achieved through simple graphical tools. This demonstrates that folding can be used to vary grain-size, with the implication that performance tuning can be carried out to target different architectures, but in a simple graphical environment. This re-targeting the program for a different architecture is a simple process and will be demonstrated in the next experiment. The observations on the speedup graph are now put forward for completeness, and to demonstrate that the speedups achieved are in line with handcoded parallel programs, and therefore that ParaDE does not degrade the achievable parallelism.

Speedup for small matrix sizes is poor, with the sequential version (*foldall*) giving the best performance. As matrix size increases, a small set of parallel actors, such as in

fold2 where 2 groups are used, begins to outperform the sequential version. This trend continues, with increasing parallelism providing good speedup as matrix size increases, but at smaller matrix sizes providing a very poor speedup. Speedup for all folding schemes tails off at some maximum as matrix size continues to increase. *Fold2*, with 2 parallel sections, reaches an asymptote of over 1.5, which is good, bearing in mind that the program has sequential sections which will limit the speedup to a maximum of 1.67 as described above.

To illustrate the effect of sequential sections, *fold5* emphasises the actor group encompassing MultiplyAB_CD, SumProducts and stdout by enforcing strict sequence, eliminating even (potentially beneficial) pipelining effects. This group becomes a limiting factor in the parallelism of the program as a whole, such that instead of the speedup merely tailing off as matrix size becomes large, it actually decreases once the maximum has been reached - just over 2.3 at around matrix size 100. In contrast, *fold4* which has an identical parallel structure except for the sequential section, reaches a plateau at this point, maintaining a speedup of over 2.4 (the theoretical maximum being 2.5) right through to matrix size 200.

The effect of changes in granularity due to the dummy loop become obvious when the intersection of the line representing one folding scheme with the line representing another is examined, that is, the point at which one folding scheme begins to outperform another. With no dummy loop (dummy loop = 1) *fold2*, with 2 parallel sections, only offers improved performance for a short period, between about matrix size 15 and 23. Before this point, the sequential version is more efficient, and afterwards the 4 parallel section version in *fold4* takes over. However, with a dummy loop of 10, the effect occurs earlier and covers a larger range, from approximately 10 to 23. This indicates that increasing the computation size without altering the data size (and hence communication size), improves the parallel behaviour such that performance increases are evident sooner. Interestingly, *fold2* only outperforms the sequential version at matrix size 10, even when the dummy loop is increased from 5 to 10, suggesting that a minimum limit has been reached for this folding scheme. However, the degradation of the speedup below this point occurs much less rapidly with the larger dummy loop size.

The same trend can be observed for all folding schemes shown in the graphs. For example, the point at which the *not-folded* (completely parallel) version and *fold2* (with 2 parallel sections) intersect falls back from matrix size 42 to 40 then 38 as the dummy loop increases. The conclusions that can be drawn from these results, therefore, are that, even for the relatively small sample of matrix sizes presented, the issue of granularity plays a very significant role in determining the efficiency of parallel execution. ParaDE presents a method for easily investigating and adjusting this granularity. Using conventional parallel programming methods, the programmer would have to estimate an appropriate granularity for the computation blocks. It would be difficult for the programmer to appreciate the effects of small changes in granularity, without re-executing and measuring following a significant rewrite of the code. ParaDE eliminates any code rewriting, allowing granularity adjustments by simple graphical user interface techniques. Furthermore, future developments of ParaDE could lead to this process being automated, that is, an appropriate granularity for a given architecture could be “guessed” by the design system (from heuristics supplied by the user or stored by ParaDE from previous program development).

ParaDE could then apply a best estimate folding solution, and subsequent automatic iteration could investigate additional folding strategies to fine-tune the performance.

It could be argued that the calculation of speedup relative to the completely folded version is not a *pure* measure of speedup, in that this version of the program could contain overheads of the parallel run time system. These overheads would increase the execution time of the sequential program, falsely improving the speedup figures. A popular alternative measure of speedup, as described in Section 5.3 would compare the parallel versions of the program, with a hand-written sequential program, compiled and run on the same architecture, but without any contact with the parallel compiler or run-time system.

The first version of speedup is preferred, not because it consistently gives better speedup figures, but because it demonstrates how ParaDE can produce scaleable results with little user effort. Any overheads inherent in the parallel run-time system can be overcome in the future either by optimisations of the run-time system itself, or by changing the translator to produce solutions for an alternative run-time system. However, the alternative speedup measures, using a *pure* sequential program are now given in Figure 5-12 for completeness.

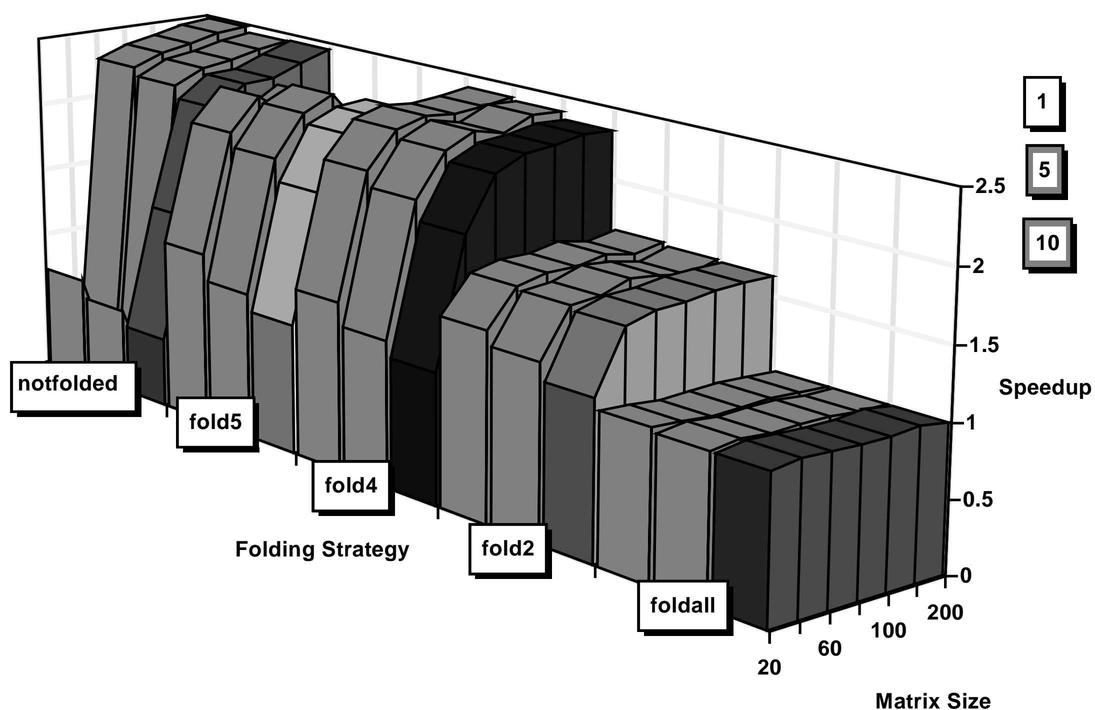


Figure 5-12 Speedup Relative to Pure Sequential (Shared Memory)

Figure 5-12 shows that, in fact, the speedup values are largely maintained at their previous values when a hand-written sequential program is used for the timing comparison. The completely folded version remains at a speedup of around 1, indicating that additional overheads from the parallel run-time system are negligible. This observation is confirmed by the other folding strategies, which still manage to

reach near the theoretical maximum speedup at matrix sizes of 40 or 60 and above. Speedup for matrix sizes below 20 were unable to be measured reliably.

5.6 Experiment 3 - Multiple Matrix Calculation (HP Network)

This experiment is identical to experiment 2, described in Section 5.5, but results are shown for implementation on the HP network of workstations.

5.6.1 Aims

This experiment demonstrates the use of the actor folding technique (described in Chapter 4) on an alternative architecture - the HP workstation network. Results will show how the grouping of a set of actors can improve the performance of a parallel program, by adjusting the grain-size of the program, using a process-oriented method.

5.6.2 Description

The algorithm used for this experiment is again the simple combination of matrix operations, producing the result ((A*B)*(C*D))+(E*F)+(G*H) where A, B, C, D, E, F, G and H are matrices. The graph and algorithm for this program are described in detail in Section 5.5.2. Sections 5.5.3 to 5.5.7 describe the individual folding schemes in detail.

5.6.3 Network Test Comparisons - Execution Times

The graphs below show the execution times for each of the matrix sizes used in these tests on the network architecture, from 5 to 200. For each graph, the execution time of each folding scheme is plotted, and a separate line is shown for each of the three dummy loop values, 1, 5 and 10.

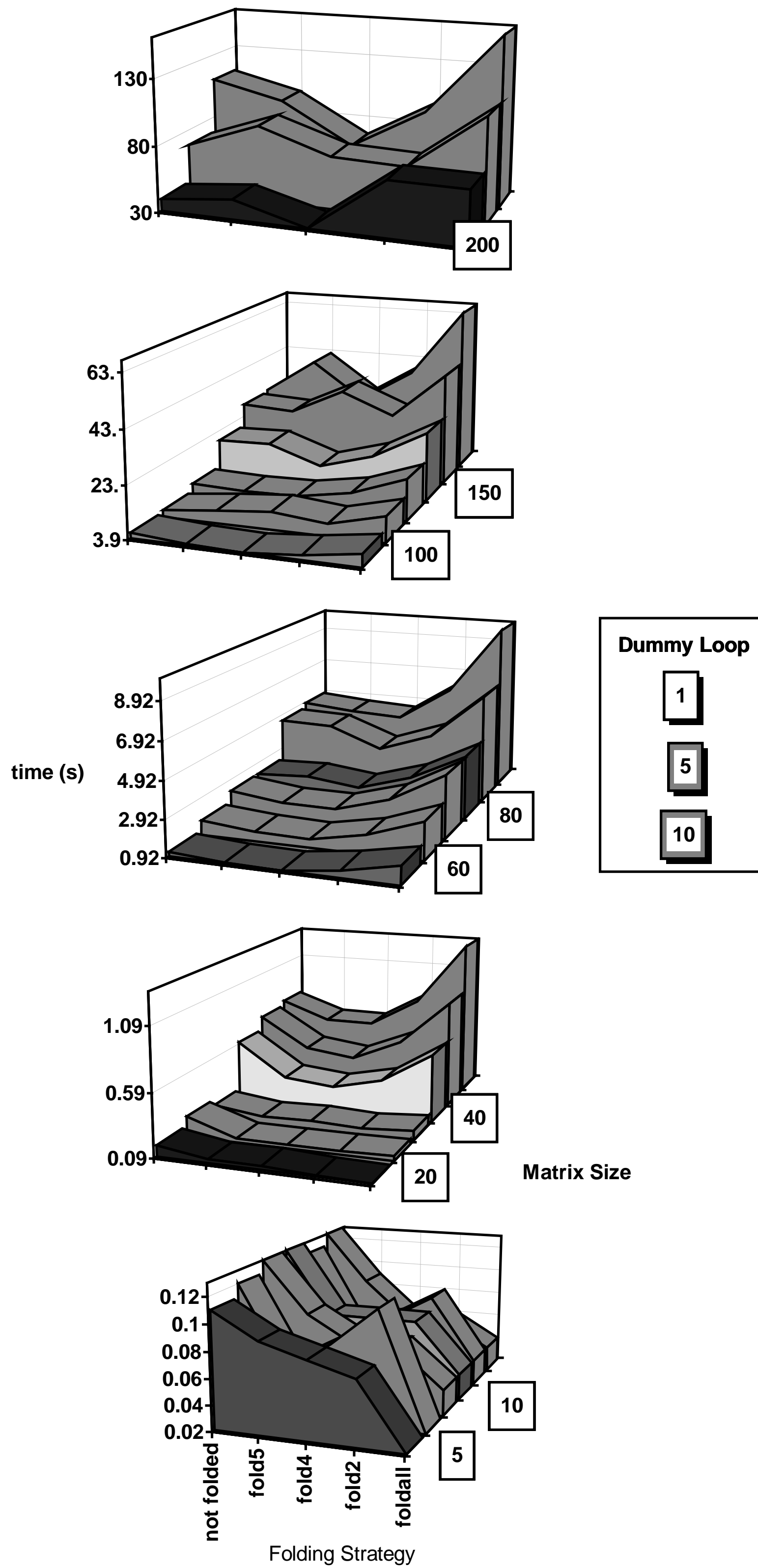


Figure 5-13 Execution Time Graphs (Network)

It can be seen that the execution times measured for the Network implementation in this experiment are somewhat less predictable and regular than those in the previous experiment. The reason for this is that the HP Network is subject to loads from other users at all times, leading to delays in setting up and running processes on these machines. The processes on which actors in this program graph are implemented often suffer suspensions or rescheduling due to other (external) processes competing for resources. Whilst attempts to reduce this effect were made, and multiple test results taken to eliminate extreme delays, the execution timings for this experiment still display some irregularity. However, taken as a whole, the results serve as a useful comparison to the Shared Memory implementation.

Although the trends in the execution time graphs for the HP Network implementation are less obvious than those of the Shared Memory implementation in Experiment 2, the same patterns can be seen emerging. The execution times for small matrices show a larger execution time for more parallel folding schemes, and a smaller execution time for less parallel folding schemes, with the best performance for the sequential version (all actors folded). In contrast, large matrices exhibit the opposite trend, with the worst performance (longest execution time) using the sequential folding scheme.

The difference between the results of the Network implementation and the previous Shared Memory implementation is in the matrix size at which different folding schemes become optimal. The Shared Memory experiment showed the *fold4* scheme achieving the fastest execution for matrices of around the 40 element size. Matrices larger than 40 elements performed better with fully parallel operation (*not-folded*). However, using the Network implementation, *fold5* continues to outperform the *not-folded* version as matrix size increases. This indicates that the fully parallel scheme (*not-folded*) fails to overcome the communication overheads for matrices of this range. Furthermore, the computations within a single actor are not of sufficiently large granularity to efficiently support this level of parallelism. These results, therefore, demonstrate exactly how the Network architecture differs from the Shared Memory architecture in its performance of this program example. The results also begin to show at what levels of parallelism this architecture is most efficient. This issue is now examined further using speedup measures.

5.6.4 Test Comparisons - Speedup

The following graph shows speedup on the network architecture for the different folding schemes, relative to the execution time of the completely folded, sequential version of the program.

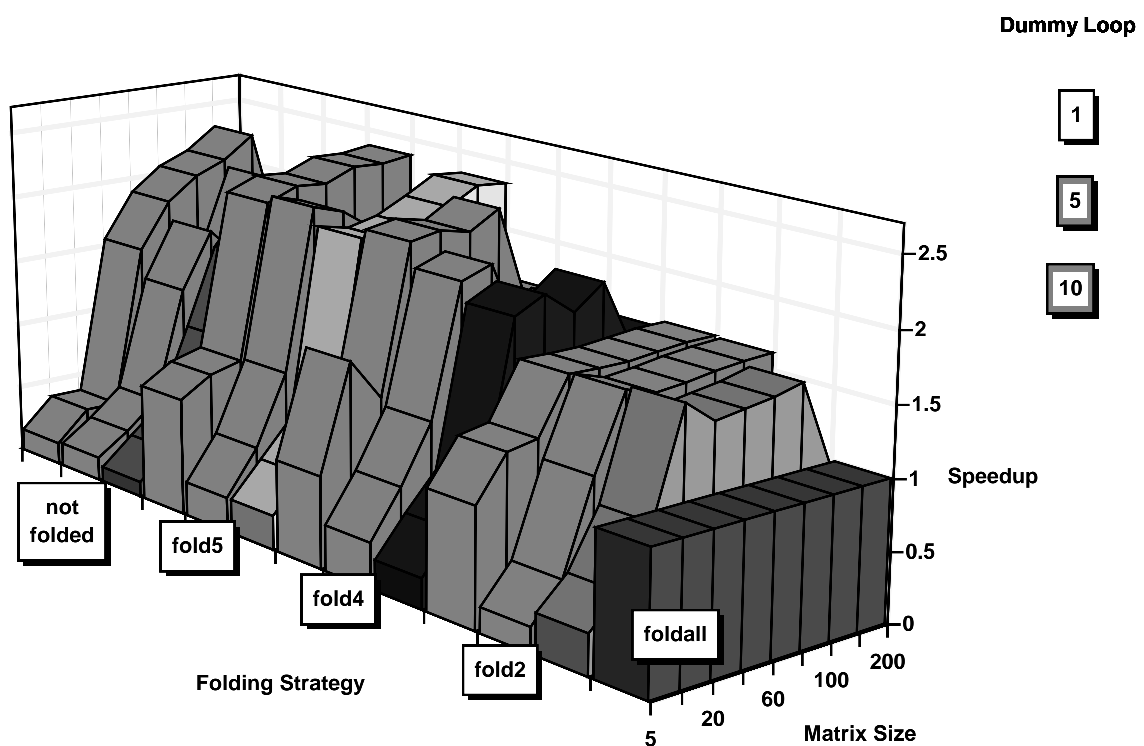


Figure 5-14 Speedup for Different Folding Strategies (Network)

Problems with irregularity of timings discussed above makes it difficult to read much into the speedup graph shown above. However, it can be seen that a similar pattern to that observed for the Shared Memory implementation is evident. As with the Shared Memory experiment, the following observations on parallel performance are not unexpected, but serve to demonstrate that ParaDE does not reduce the performance improvements, and the different parallel configurations are developed from the same user program, with minimal additional effort. Appendix B presents the generated code for three of the folding strategies on the Network architecture, to illustrate the adjustment in granularity by combining actor code using automatic code generation.

Speedup for small matrix sizes is poor, in fact speedup remains below one until the matrix size increases past 20 for no dummy loop (dummy=1). This compares to speedup rising above one for the Shared Memory experiment at around matrix size 15. These differences are relatively small, given the significant differences between the two architectures. However, looking at larger matrix sizes, it can be seen that while the shared memory architecture achieves near maximum speedup for all parallel configurations from about matrix size 100, the network architecture shows that for the more parallel configurations, speedup is still increasing at matrix size 200. It is expected that this trend would continue, with maximum speedup being attained for larger matrices. Of course, the maximum speedup values are based on the theoretical calculations given for the previous experiment as the graphs remain the same for both architectures. These theoretical maximums are 1.67 for *fold2* and 2.5 for other parallel

configurations. For the less parallel (*fold2*) configuration, maximum speedup is achieved more quickly - this is due to the communication overheads being less significant as more of the program is contained within one process. For limited parallelism therefore, the network architecture performs well at smaller matrix sizes, and the differences between the performance on the two architectures are smaller.

Increasing the dummy loop shows a reduction in the smallest matrix size to achieve above unit speedup, but this still occurs slightly later than in the Shared Memory experiment, for example at matrix size 40 instead of 20 or 20 instead of 10. These observations reinforce the findings of the previous section that parallelism can generally be exploited efficiently at a finer granularity for Shared Memory architectures.

It is difficult to pinpoint precise intersections between different folding schemes, due to the less regular results. However, it can be observed that, in general, folding schemes with limited parallelism show slightly better speedup at smaller matrix sizes, but peak at a lower level. Folding schemes which exploit more parallelism perform better at larger matrix sizes, reaching a higher level of speedup, as expected from the results in the previous section. The exception to this pattern is the fully parallel version, which fails to match the peak speedup of other folding schemes, due to the limiting factor of the communication costs for this architecture, discussed above. The

corresponding speedup graph measured against the pure sequential program is shown below.

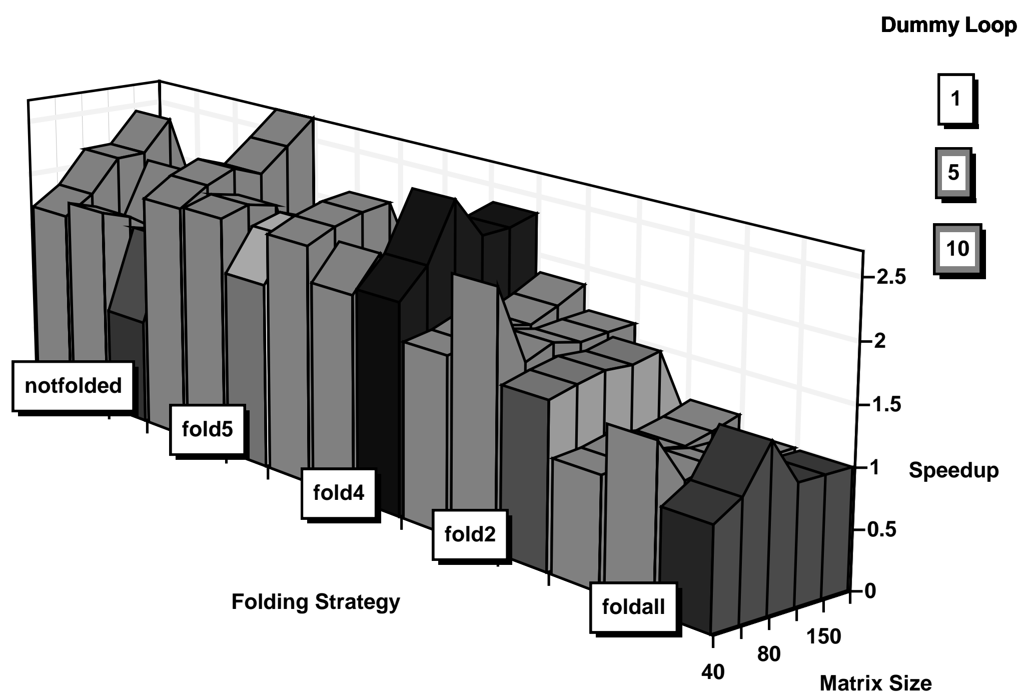


Figure 5-15 Speedup Relative to Pure Sequential (Network)

Values for matrices under 40 elements square were excluded as the results proved to be unreliable for such short execution times. Other parts of the graphs are also somewhat inconsistent, and irregular timings in both sequential and parallel programs can distort speedup significantly. However, the graph does show that, in general, speedups achieved are similar to the previous graph measuring speedup relative to the completely folded version of the program. As with the same matrix calculation experiment on the shared memory architecture, this indicates that the additional overheads introduced by the parallel run-time system are minimal, and that the speedups achieved are accurate representations of the performance improvement possible using ParaDE.

5.7 Experiment 4 - LU Decomposition Notation

5.7.1 Aims

This final experiment aims to show the effectiveness of the notation and design tools provided by ParaDE, in designing and implementing a more complex program than those used in previous experiments in this chapter. Performance results will be shown to indicate that overheads are acceptable, but the emphasis will remain on examining the usefulness of the design system.

5.7.2 Objectives of ParaDE

In order to fully understand the significance of the design system features explored in this experiment, it is useful to recap on the initial objectives of ParaDE. Chapter 2 discussed the motivations behind the work on graphical programming notations in general, and proposed the important themes on which this work is based. These themes can be summarised as:

- the graph should provide a clear behavioural description of the program, representing the design level,
- the textual description should serve as a refinement of the design,
- both data and control features should be incorporated, although data should be the primary factor,
- the computation details should be separated from the co-ordination details,
- in order to isolate the co-ordination, the firing rule must be predefined, or implicit in the graph structure.

The experiment in this chapter will show how ParaDE satisfies these aims, to provide a practical parallel programming environment for non-expert parallel software developers.

5.7.3 Description

The algorithm used for this experiment is a parallel version of the LU matrix decomposition problem. The aim of this problem is to derive two triangular matrices, a lower (L) and upper (U) matrix, from a source matrix (A), which produce the original matrix when multiplied together. The problem was chosen to demonstrate the complexity of data dependencies in common programming problems, and the way in

which the ParaDE notation can be used to design and implement algorithms with intricate data relationships. These complex data dependencies are shown in the relationships between elements of the lower and upper matrices, in that any single element requires the value of some of its neighbours and/or the values of the corresponding elements in the other matrix. The two formulae which make up the original algorithm from which this experiment was derived are given below :

$$L_{ps} = A_{ps} - \sum_{m-1}^{s-1} L_{pm} U_{ms}$$

Equation 4 Formula for L element

$$U_{pk} = \frac{A_{pk} - \sum_{r=1}^{p-1} L_{pr} U_{rs}}{L_{pp}}$$

Equation 5 Formula for U element

where X_{ab} is the element of the X matrix from row a, column b, and s is the stage of the algorithm (for a n*n matrix, s=1..log n, p=1..n, k=s+1..n).

Even from these simple equations, it can be seen that the U and L elements have complex data dependencies on their neighbours across both matrices and with their predecessors through each stage of the algorithm. To simplify the design process, a hierarchical approach was taken, using the decomposition facility provided for all actors. The following 3 sections describe the different levels of the design.

5.7.4 Design Level 1

Figure 5-16 shows the top-level design comprising of a source actor supplying the source matrix A, two other source actors supplying initialised values of the results matrices L and U, and stdout actors giving the final resulting values for L and U. The main body of the algorithm is then encompassed within the loop actor which iterates over the s stages of the algorithm. The datapath from the A matrix source actor is a continuous path, representing the static nature of the source matrix. The other two input datapaths from matrices L and U are discrete so that their value changes on each iteration of the loop, and are connected to the data-dependent symbol of the loop actor (described in Chapter 3). This means that the data on these datapaths form the input to each iteration of the loop, and their arrival signals the firing of the next iteration. In this way, the algorithm follows a set of discrete time steps, one for each stage of the algorithm, with the body of the loop actor performing the calculations at each stage on the current data. This style of graph would be common to many iterative programming problems, in that the first, top-level design would consist of an iteration operator - the loop - and this would be decomposed in a top-down manner to refine the design. ParaDE encourages such a top-down approach in a very practical way, where the graphs and sub-graphs actually form the resulting program. There is no additional transformation process from design to implementation, instead the decomposition and refinement of the graph leads the user, in an easy to understand manner, straight from top-level design through to detailed implementation.

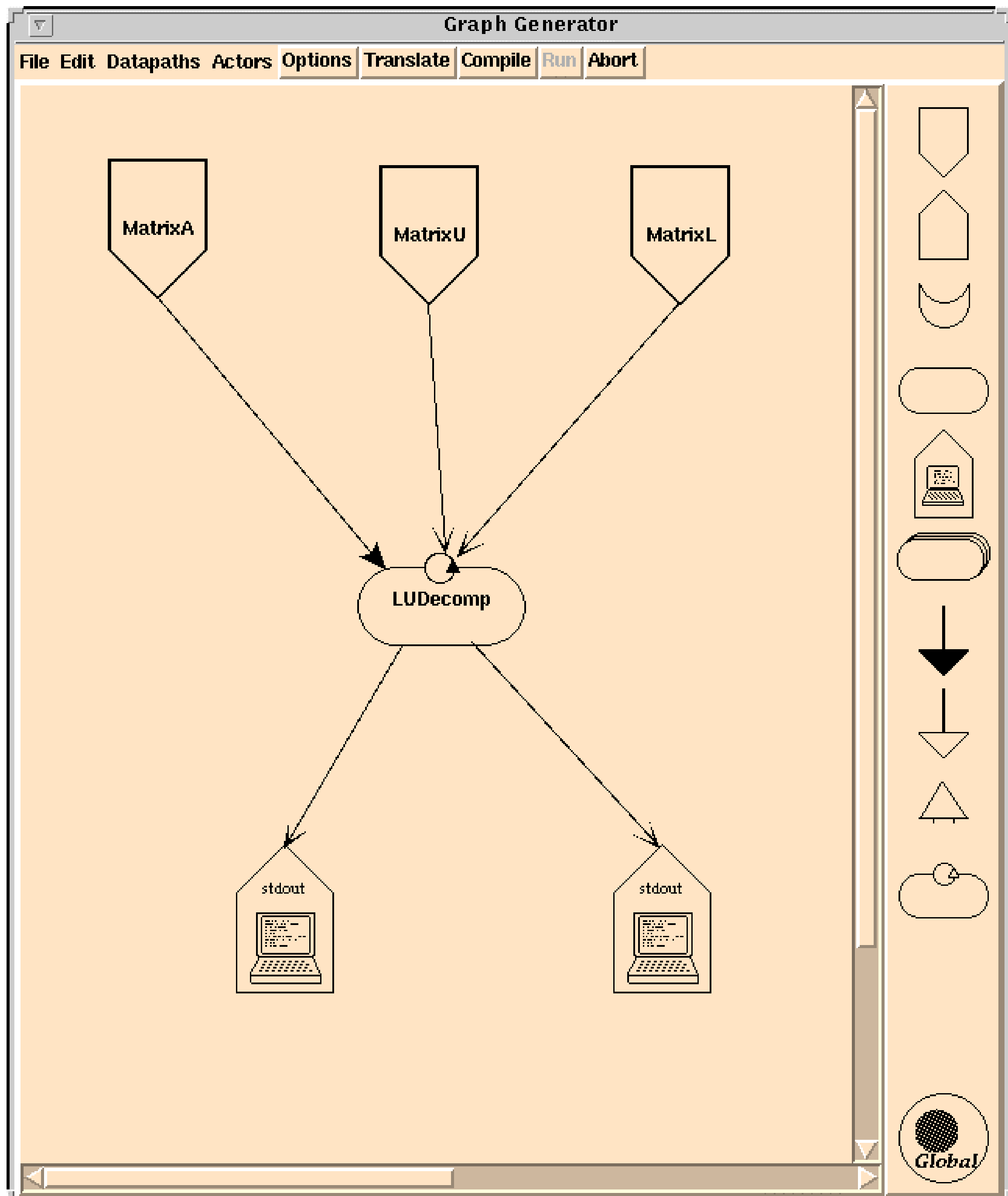


Figure 5-16 Top-Level LU Decomposition Graph

5.7.5 Design Level 2

In the second stage of development, the loop actor in Figure 5-16 is decomposed, with the connecting datapaths (in this case all of the datapaths) forming inputs and outputs to the subgraph, as shown in Figure 5-17.

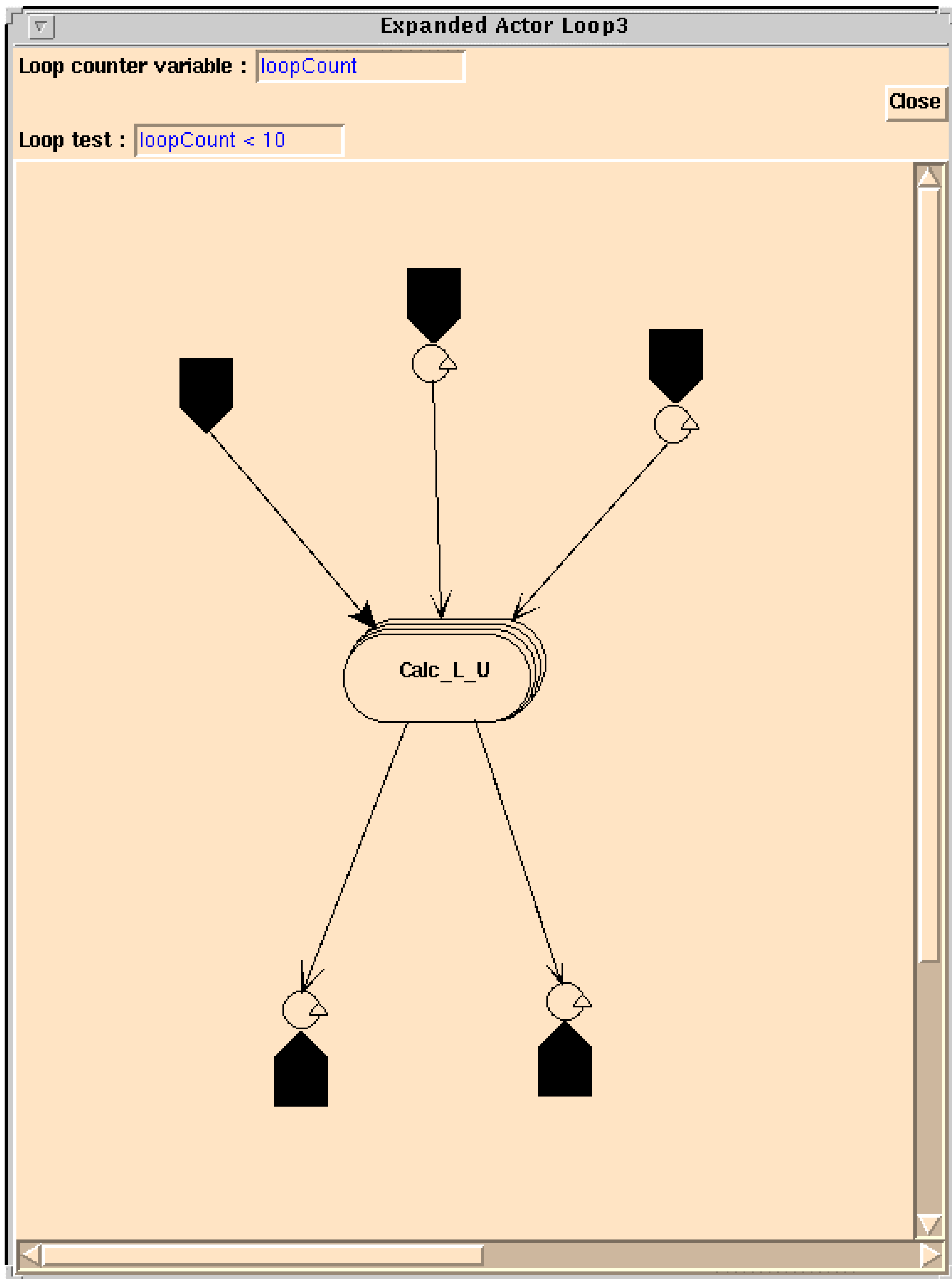


Figure 5-17 Second-Level LU Decomposition Graph

A single actor is used to define the main body of the subgraph, and while this may seem an unnecessary use of the hierarchical properties available in ParaDE, in fact the simple nature of each level of the graph demonstrates that the hierarchical property encourages a simple top-down design. No level is overly complex, and the design of the program can be easily understood.

This second level, with the single depth actor, represents the parallel calculation of the L and U elements for one stage (iteration) of the algorithm. No detail is given at this level of the nature of the parallelism, except that the datapaths from L and A would be shown as partitioned (in this prototype version, datapath labels and additional split arrowhead symbols to indicate partitioning are not implemented). A normal arrowhead would indicate that the U matrix is used in its entirety, while split arrowheads would show that the L and A matrices are split up and distributed over the instances of the depth actor.

5.7.6 Design Level 3

The final level of the LU decomposition design graph is shown in Figure 5-18. This is a slightly more complex graph, but still only contains two actors, maintaining the principle of a hierarchy of simple graphs. The first actor performs the calculation of the L elements, and the second actor performs a parallel calculation of the U elements using a depth actor. The choice of a depth actor for U calculations but not for L calculations reflects the algorithm which dictates that the U elements for a particular row are dependent on elements from the column of L calculated in previous stages. In fact, only one extra L calculation is required at each stage to calculate a new row of U elements. At the level above however, another depth actor represents the calculation of L and U elements, such that L elements are calculated in parallel, and for each of these L elements, one row of U elements can be calculated. Whilst this is difficult to express in words, the formulae in Equation 4 and Equation 5 show the calculation for a single L and U element, and the dependencies shown in these equations are directly translated to the more understandable format of the parallel graph notation.

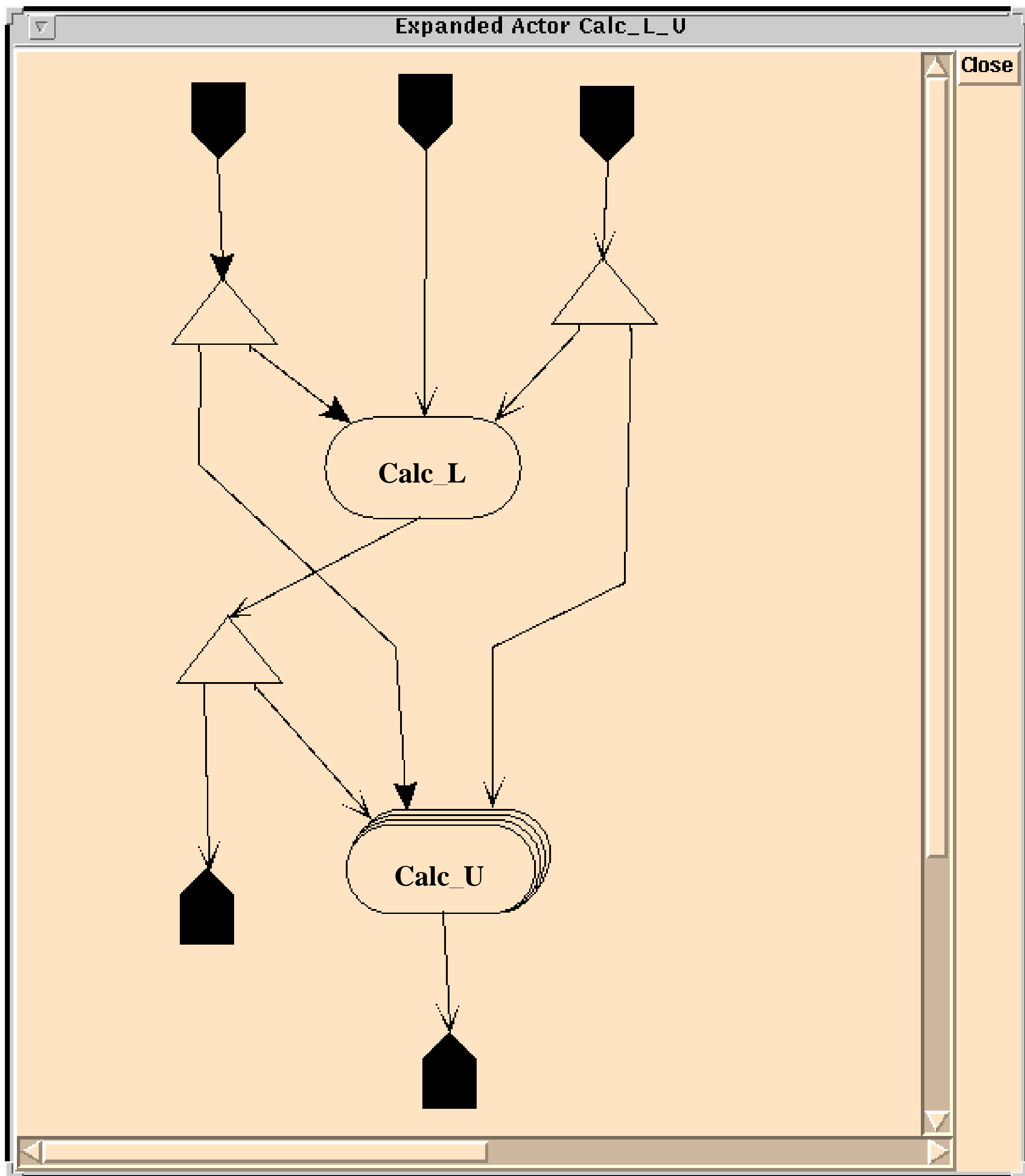


Figure 5-18 LU Decomposition Level 3

The datapaths connecting the two actors show how the complex data relationships can be captured with relative ease, using three particular features: the replicator, data partitioning and discrete/continuous datapaths. Ease of use of the notation and design system was one of the key objectives stated initially, and these three features are now discussed with reference to the example program, to show applicability of the notation.

1. The replicator data junction, represented by the triangular node as described in Chapter 3, provides a means of copying the data from a single source datapath to multiple destination datapaths - in Figure 5-18 a replicator is used to supply data from a single matrix, such as the A matrix, to different actors performing different

calculations. In conjunction with the data partitioning (discussed below), the replicator can also provide different representations of the same data to the multiple destinations. This feature restricts the need for unnecessary duplicated datapaths, and offers a compact means of describing a complicated division of data.

2. The data partitioning facility is key to this example, and many others relying on data parallelism. Capturing the nature of the division of data succinctly but allowing flexibility underpins a successful implementation of a data-parallel algorithm. Using data partitioning associated with each datapath allows the specification of multiple partitions of the same data, without over-complicating the design. In this example, the data from Matrix A is partitioned at level 2 into rows, but at level three, the datapath feeding into the U calculation is further partitioned into single elements, whilst the datapath feeding into the L calculations remains partitioned into rows. Similarly, the U matrix is partitioned into columns for the U calculations, but left whole for the L calculation.

3. Discrete and continuous datapaths are both used in this example, representing the dynamic and static data respectively. The continuous datapath type is used for the source data from Matrix A, which remains unchanged throughout the algorithm, and supplies data from the original matrix at different points in the calculation. In contrast, the discrete datapath type is used for the data which is updated periodically during the algorithm, such as at each stage in the loop.

5.7.7 Implementation of LU Decomposition

So far, the LU example graph has been shown as a hierarchical set of simple sub-graphs. All that is required to complete the implementation of the program is to specify the interfaces between the actors and insert the method code for the final-level "leaf" actors. The interfaces are specified using the basic datapath form, and the more advanced data partitioning facility, both described earlier in Chapters 3 and 4. The interesting feature in this example is the use of multiple data partitioning as described above. The method code is very simply an adaptation of the equations given in Equation 4 and Equation 5, and is shown below in Figure 5-19 and Figure 5-20.

Method Code : Calc_L

Input variables

array<int> dp5
Size 10

array<int> dp1
Size 10

array<int> <int> dp7
Size 10, 10

Output variables

array<int> dp9
Size 10

Local Variables

Name	i	j	sum	
Type	int	int	int	

```
sum=0;
for (i=1;i<loopCount-1;i++)
  sum=sum+ (dp1[parts][i] * dp7[i][loopCount]);
dp9[parts][loopCount]=dp5[parts][loopCount]-sum;
```

Figure 5-19 Method Code for L Calculation

Method Code : Calc_U

Input variables

array<int> dp6
Size 1

array<int> dp8
Size 10

array<int> dp10
Size 10

Output variables

array<int> dp4
Size 1

Local Variables

Name	i	j	sum	
Type	int	int	int	

```
sum=0;
for (i=1;i<iteration6-1;i++)
  sum = sum + (dp10[parts][i] * dp8[parts][i]);
dp4[iteration6][parts] = (dp6[iteration6][parts] - sum)/dp10[part
s][iteration6];
```

Figure 5-20 Method Code for U Calculation

5.7.8 Performance Results

Performance has been investigated in great depth in previous experiments using the matrix multiplication and multiple matrix calculation examples. The main purpose of the LU decomposition example is not to investigate performance, but to demonstrate the use of the ParaDE notation in developing more complex programs. However, a notation which is useful in design but which doesn't lead to acceptable performance has limited application. To demonstrate that good performance is still achievable with complex designs, therefore, a limited set of performance tests were carried out.

The LU decomposition was executed with matrices ranging from 60 elements to 140 and with the data partitioned into 1, 2, 4 and 8 segments to observe the effect of increasing parallelism. Because of the experimental nature of the code generation tool, and the complexity of the example used, only results for the Encore Multimax platform were collated. It is expected that performance on the HP network would follow similar patterns, subject to the different architecture effects explored in earlier experiments. Figure 5-21 shows the performance of this program, plotting the speedup relative to the single partition (no parallelism) against matrix size for each partition.

It can be seen that while 2 and 4 segments improve performance (reduced execution time), increasing the parallelism to 8 segments leads to an increased execution time, hence a degradation of performance.

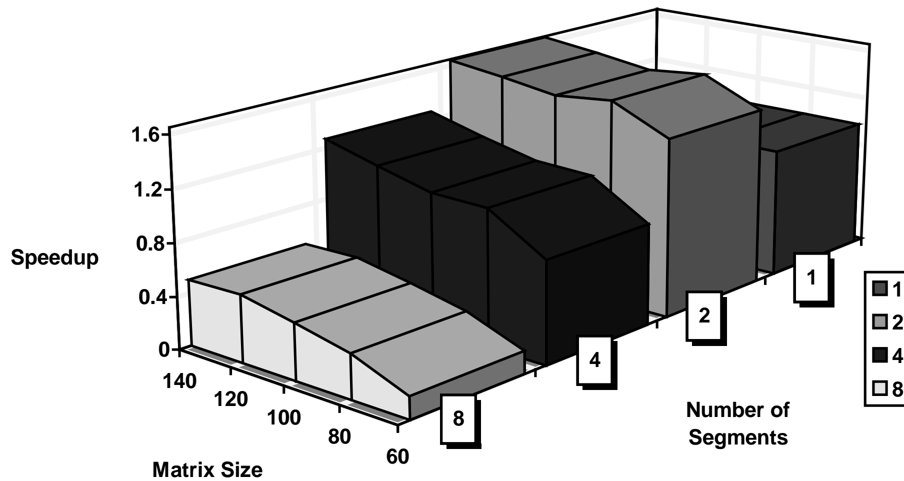


Figure 5-21 Speedup for LU Decomposition

The speedup graph shows that partitioning with 2 segments achieves a good speedup, of over 1.6. In fact this is likely to be near-optimal, as the limit of 2 could not be achieved with this algorithm, due to the necessary sequential sections which limit parallelism. A 4 segment partitioning still offers speedup, but of only around 1.3 at maximum matrix size. This is lower than that achieved by the 2 segment partition, so it can be seen that this algorithm is not suitable for this level of parallelism. This finding is reinforced by the results for the 8 segment partition, which demonstrate a considerable slow-down in performance.

All of the plotted graphs show an increasing speedup with matrix size - this is to be expected, as the larger matrix sizes offer an increased proportion of computation effort relative to the communication effort. This communication effort could have been expected to be much higher, due to the complex data relationships in this example, and the corresponding data exchange between actors to manipulate the correct input and output data. However, the results demonstrate that the partitioning technique reduces the communication required, by specifying only the necessary data to be distributed between actors, and the amount of redundant data communication is minimised.

5.7.9 Observations on Experiment 4

The performance results shown above indicate that even for a relatively complex design graph, with a high level of data communication, efficient parallel solutions can be achieved. The partitioning facility offers the means to specify data communications succinctly and precisely, and allows the user to investigate at what level of parallelism a given program achieves speedup.

The development of the LU decomposition graph described in this section demonstrates that the hierarchical graph feature encourages a top-down design methodology, specifying a small set of actors at each level, and the interfaces between them. This leads to a straightforward implementation at the method code level, and a design which is easy to understand and maintain.

5.8 Results Summary

This chapter presented a set of experiments demonstrating the usefulness and performance of the graphical programming language proposed in this thesis, and the encompassing parallel programming environment, ParaDE. These experiments were presented in relation to the motivations and objectives summarised at the start of the chapter, and described in detail in Sections 5.4 to 5.7. The environment in which the experiments took place was also described, and the factors considered important in the measurement of performance of parallel programs.

Experiment 1 in Section 5.4 presented a matrix multiplication program, used to demonstrate the data partitioning facility for adjusting grain-size and optimising performance. The performance results from this experiment showed that the same user program could be used on two different parallel architectures. Good performance was shown on both architectures by the speedup values obtained, without any change in the program written by the user. Additionally, the results demonstrated how the different architectures reached optimum performance at different granularities.

Experiments 2 and 3 in Sections 5.5 and 5.6 demonstrated the alternative granularity adjustment facility, using actor folding. The folding technique was applied to the multiple matrix calculation program on both architectures, and the results compared. The findings from all three experiments matched the expectations, that finer-grain parallelism could be exploited efficiently on Shared Memory architectures, whereas good performance on the network architecture was restricted to larger-grain implementations. These results were shown to be valid within a certain range, specific to the algorithm used for the example program and the architecture on which it was implemented. Outside these limits, increasing parallelism was inefficient due to the

disproportionate communication overheads. ParaDE was shown to provide an easy way of pinpointing the level of parallelism which could be most effectively exploited on the given architecture.

Experiment 4 investigated a different aspect of the graphical programming environment, judging the effectiveness of the notation and its encompassing environment in developing more complicated programs. An LU matrix decomposition algorithm was used as an example, demonstrating how a program with complex data relationships could be represented in the notation, and how a program solution could be developed using the features provided in ParaDE. In particular, a hierarchical design methodology, making use of the actor decomposition, was used to develop a concise solution. Additionally, multiple data partitioning was featured, to capture the complex data dependencies. A working solution was presented, along with performance results to demonstrate that efficiency could be maintained with more complex designs.

The experiments presented in this chapter addressed the three aims outlined initially:

1. the ease of use and suitability of ParaDE in developing parallel software,
2. the performance of the parallel software developed using ParaDE,
3. the portability of the software generated by ParaDE for different parallel platforms.

The LU decomposition problem was designed and implemented using ParaDE, demonstrating how complicated data dependencies and multi-level parallelism could be expressed in the ParaDE notation. Despite the complexity of the problem, a simple, hierarchical design was presented which captured all of the data relationships and parallelism in a straightforward manner.

Performance of the parallel programs developed in ParaDE exceeded all expectations, producing speedup values very close to the theoretical maximums calculated. Both data partitioning and actor folding were investigated as methods which, whilst allowing the specification of parallelism in a flexible and algorithm-oriented manner, supported the adjustment of granularity to optimise performance.

Re-targeting the programs developed in ParaDE to different parallel architectures was demonstrated, and the granularity adjustment features were shown to allow the optimisation of performance for a particular architecture without any re-writing of the user code or program graph. It was shown how ParaDE programs could be re-generated after simple graphical granularity adjustments, and the parallel structure information extracted from the graph. An iterative process, applying different folding strategies or data partitionings, allowed execution profiles to be produced quickly and effectively, rapidly converging on an efficient solution.

To extract execution information for different granularity adjustments from programs in conventional parallel programming environments would be both time consuming and error prone, as the code would have to be re-written each time. ParaDE allows execution profiles to be extracted without changing the original user's code, thus giving the opportunity to explore granularity adjustment. In future developments of ParaDE, this granularity adjustment could be performed automatically, based on a *best guess* first attempt using heuristics developed for different architectures, and automatic iteration to achieve optimal performance for the selected architecture.

6. Conclusions

This final chapter draws together the work presented in this thesis, summarising the research undertaken and discussing the impact of the results achieved. In order to fully explore the scope of these conclusions, this chapter begins with an overview of the various aspects covered, before specific results and achievements are addressed in Section 6.2. The possibilities for future work are investigated in Section 6.3 and this thesis concludes with some closing remarks in Section 6.4.

6.1 Overview

The opening statements of this thesis described how the computational demands for certain applications were exceeding the performance of single processor computers, and discussed how the advent of parallel computing offered the potential for meeting these demands. However, the difficulty of programming parallel architectures was introduced as major stumbling block for the widespread acceptance of parallel programming, in particular the complexities of co-ordinating the multiple processors to exchange data, and the impact of communication and synchronisation on performance. Chapter 2 described in more detail the nature of parallelism and the techniques required for parallel programming. The differences in exploiting parallelism on different architectures was discussed, and the current methods used in writing parallel programs were investigated.

A key point was raised in the discussion of different approaches to parallel programming, and that was the lack of good tools for developing parallel software, and the low-level nature of the mechanisms used to exploit parallelism. This was contrasted with the increasing sophistication and abstraction level of sequential programming methods, with the role of abstraction highlighted as an important issue in achieving widespread use of parallelism across the wide variety of parallel architectures available. Some abstract approaches were introduced, notably dataflow and graphical programming, as potential influences on the direction and progress of parallel programming, leading to the description of a graphical dataflow language, MeDaL.

Chapter 3 was dedicated to the investigation of design notations for parallel programming, based on an evaluation of the author's practical experience with the MeDaL approach. The motivations of using hybrid graphical/textual languages such as MeDaL were discussed, with examples of different systems in current use. A critical evaluation of the motivations and current practices led to the proposal of a new design notation, drawing on the work of MeDaL, but making some significant changes to the visual syntax, hierarchical structure and data persistence features. In addition, to address the problems of MeDaL, a number of key new objectives were introduced:

1. adoption of an architecture-independent run-time system,
2. provision of graphical abstractions of actor and datapath interfaces,
3. creation of a new loop actor to cater for iterative algorithms,
4. introduction of features for automatic data partitioning across the depth actor,
5. development of an encompassing design system, called ParaDE.

The design system, ParaDE, was described as an environment, centred around graphical tools and techniques, for supporting the flexible specification of parallelism and the structured development of parallel programs in user-friendly conditions.

Chapter 4 centred on the exploration of performance issues in ParaDE, outlining the requirements for efficient performance on parallel architectures, and looking at the performance features and achievements of other graphical programming environments. The tools developed for ParaDE to address performance issues were described, including:

1. automatic code generation,
2. actor folding,
3. data partitioning.

Code generation was presented as a method of transparently creating parallel code by extracting the information about parallel structure from the program graph developed in ParaDE. The graphical techniques incorporated in the notation and the encompassing design environment provided abstractions of the mechanisms required to implement parallelism, and the code generation tool performed a translation of these abstractions into actual parallel code for the Linda programming model. Whilst this Linda model was presented as conceptually a good implementation model for ParaDE, the choice of Linda was identified as neither necessary nor significant. It was indicated that the code generation tool could be adapted to produce code for other programming models, as the ParaDE notation and features provided an abstract specification of parallelism which was not tied to any model or architecture.

Actor folding was the first of two performance adjustment techniques proposed and investigated for varying grain-size. It was demonstrated how a simple graphical technique could be used to select groups of actors to be combined into a single actor, optimising out the communication between the folded actors, and increasing the grain-size of the resulting program. A re-generation of the program from the ParaDE graph produced a new executable parallel program with adjusted grain-size, without any requirement for the user to re-write any code or alter the program graph. This approach was contrasted with traditional methods which required a significant programmer effort to adjust the grain-size of a program.

Data partitioning was the second granularity-adjustment tool, allowing the specification of data distribution across copies of the depth actor in a simple graphical manner. The subsequent grouping of the original data partition was shown to provide another technique for increasing grain-size from the original algorithm-specific data specification. Again, re-generation of the parallel program from the ParaDE graph implemented the granularity adjustment with minimal programmer effort.

6.2 Results

Chapter 5 presented a set of experiments for investigating the success of the ParaDE system. A number of key areas were identified for exploration:

1. the performance of parallel programs generated from a ParaDE program graph,

2. the portability of software developed using ParaDE across different parallel platforms,

3. the ease of use and suitability of ParaDE in developing parallel software.

Four main experiments were described to investigate these points.

The first experiment presented a matrix multiplication program, measuring the performance of generated solutions on two different parallel architectures. Data partitioning was used to vary the parallelism to investigate the effects on performance of adjusting grain-size in a data-parallel program. This specification of data partitions and the subsequent grouping of data blocks to increase grain-size was achieved using simple graphical techniques, without any requirement on the user to rewrite method code or alter the program graph. Speedup measurements on the two parallel architectures showed that good performance could be achieved on both machines, but at different granularity. Speedups of 1.95 for 2 partitions, 3.83 for 4 partitions and 6.36 for 8 partitions were achieved for matrices of around 200 x 200 elements on the shared memory architecture. Similar speedups were achieved on the Network architecture, but at matrix sizes of around 700 x 700 elements. The data partitioning and grouping facility allowed the simple and rapid adjustment of grain-size such that the most appropriate granularity for each architecture could be determined easily, and performance could be optimised.

Experiments 2 and 3 demonstrated the use of the actor folding technique on each of the two parallel architectures with a multiple matrix calculation problem. A set of folding strategies was presented to investigate granularity adjustment in a program exhibiting task-parallelism. Each folding strategy grouped together a set of actors for combining method codes and optimising communication. Again, this folding technique was a simple graphical mechanism obviating the need to manually alter the program developed by the user. Performance was measured for each of the folding strategies, and on both of the architectures, to explore how grain-size adjustment using actor folding could target optimum performance for a given architecture. Results showed how performance varied for each folding strategy across the range of matrix sizes. Using small matrices, folding of all actors was required to achieve speedup, but as matrix size increased, folding strategies with increasing parallelism became efficient. It was demonstrated how the folding technique did not detract from performance, with speedups of over 1.6 for a folding strategy with theoretical speedup of 1.67, and speedups of over 2.4 for strategies with theoretical speedup of 2.5. Again, it was possible to identify suitable matrix size and folding strategies to achieve efficient performance on each of the parallel architectures with minimal programmer effort, merely an iterative process of graphical selection of a group of actors for folding and a re-translation of the graph.

The final experiment addressed the suitability of the notation and the ease of use of the design environment in ParaDE for developing a more complex parallel program. A matrix LU decomposition problem was presented which exhibited complex data dependencies and multi-layer parallelism. The design procedure was demonstrated, developing the program graph in a hierarchical, top-down manner, and using many of the ParaDE notation's features for encapsulating parallelism and control structures. In particular, different forms of data partitioning were used on different datapaths to allow the specification of the data requirements for each actor. Permitting this

flexibility in data specification encourages efficient use of data, minimising unnecessary movement of that data. In addition, focusing on the algorithmic data requirements in the program design, and ignoring granularity and performance at the design stage, leads to a structured, well-defined program which can later be manipulated for performance tuning using grouping techniques without affecting the design. Performance figures were extracted to show that speedup was still achieved with such complex problems.

The experiments presented in Chapter 5 both demonstrated the key features of the design system ParaDE and provided practical evidence of how these features addressed the initial objectives: performance, portability, and ease of use.

Performance was shown to be very good, with ParaDE adding minimal additional overheads to the programs produced. Results were shown to be scalable, limited only by the parallel architecture itself and Amdahl's law.

Portability was a key issue, and the granularity adjustment techniques demonstrated how simple abstract methods could be used to vary the grain-size to target optimum performance on the target architecture, without needing to re-write any of the user code or change the program graph.

Ease of use was demonstrated by showing how a program with multi-layer parallelism and complex data dependencies could be specified in ParaDE. The use of the hierarchical properties, control structure features such as the loop actor, and efficient use of data by accurate data partitioning specification led to a program solution which was concise, clear and readable, specifying purely the algorithmic characteristics. Performance related adjustments were ignored at the design stage, avoiding the confusion between design and performance-oriented implementation seen in many traditional parallel programming methods.

6.3 Future Work

ParaDE is a prototype development, and some of the features demonstrated in this thesis do not have the full implementation of a complete design system. An obvious starting point for future work would be the completion of the design system features. This aside, there are still many areas of work touched on by the ParaDE development which could lead to interesting and significant research. A number of these areas are now described.

ParaDE has featured a number of facilities which are significant in developing parallel programs. Some of these facilities question the whole design style of traditional parallel programming methods and the specific techniques used in parallel languages. Although the experiments presented in Chapter 5 have attempted to examine all of the aspects of ParaDE, the prototype nature of the design system and the time limitations on this work have prevented widespread use of ParaDE and its features in developing a variety of parallel programs. More comprehensive use of ParaDE for real parallel programming problems would allow a more in-depth evaluation of the visual syntax of the notation, the graphical abstractions of parallel mechanisms, and the facilities for granularity adjustment and code generation.

Code generation is a key feature of the system, and currently adopts Linda as an intermediate parallel programming model. As was stated earlier, the choice of Linda is

neither necessary nor significant, and other parallel models such as PVM, which are also widely adopted on a variety of parallel architectures, may offer additional benefits in performance. As the ParaDE notation and design environment are abstract, and completely isolated from the underlying programming model, moving to a different parallel programming model would only require changes to the translation tool.

Granularity adjustment is a significant factor in the success of ParaDE's portability. The data partitioning and actor folding techniques proved very successful in targeting the appropriate granularity for a given parallel architecture. However, the iterative process of adjusting granularity and re-generating code, whilst a significant improvement in both time and effort over traditional methods of adjusting performance, can still be a repetitive and tedious procedure. It is likely that some of this process can be automated, with ParaDE choosing a best-fit granularity, and the user applying minimal fine-tuning. This granularity choice could be made in a number of ways:

1. Heuristics could be stored in ParaDE, identifying the granularity size most appropriate for a given architecture, and programmed into the system from measurements of a variety of parallel architectures.

2. ParaDE could itself develop heuristics, based on experience gained from the targeting of programs to architectures. Granularity measures could be stored by the system, and updated and improved every time a program is developed.

3. The iteration of granularity adjustment and re-generation could be automated, with the system applying different granularity adjustments and repeating the process until the performance measurements are optimum.

These options could be applied separately or in conjunction. All three approaches could be combined starting with heuristics built into the system, these could then be adapted and improved by the system through their use in developing programs, then a final stage could apply an automatic iteration to fine-tune the granularity.

The grouping of data blocks using the data partitioning facility is one of the granularity adjustment features, and has been proved successful. However, the initial data partitioning specification stage, where input and output data templates are selected and aligned, is limited in the type and number of applications it can be used for in its prototype state. Only a handful of basic data patterns are implemented such as a matrix column or row. Further development could implement the whole range of data reference patterns identified in other research [21], and investigate the use of these data patterns in developing parallel programs in ParaDE. An investigation of this area might identify differences between the data patterns recognised in current FortranD programs [21], and those which would be used when adopting the top-down, problem-oriented approach encouraged using ParaDE. Wider use of ParaDE using these data patterns might also clarify and support the hypothesis that misuse of looping structures for implementing performance optimisation in traditional parallel programming methods confuses the design with performance issues and promotes architecture dependence.

Actor folding, the second granularity adjustment feature, could also be developed further, eliminating some of the hit-and-miss aspects of choosing an appropriate granularity. Using heuristics, or other methods outlined above, a suitable granularity

for the target architecture could be determined, and this could be applied automatically to the actor folding facility. Actors which exhibit too fine a granularity could be automatically selected for folding with neighbouring actors, and some of this selection process could be applied statically, even before the first code generation is performed.

6.4 Closing Remarks

Parallel programming is fraught with difficulties such as the complexity of programming the co-ordination of parallel computations and the architecture-dependence of traditional programming methods. The variety of parallel architectures available has led to widely differing approaches to parallel programming, and the field of parallel software has become fragmented. This thesis has shown how, by drawing on a number of novel approaches to programming such as dataflow and graphical programming, it is possible to find a representation of parallel programming mechanisms which is sufficiently abstract to be independent of the target architecture. Combining this with graphical user interface techniques and automatic code generation, parallel programming becomes within the reach of sequential language programmers, and ceases to be a very specialised and complex field. The design system ParaDE demonstrates not only abstraction of low-level detail and portability of program solutions, but also encourages a different approach to program design and development. Hierarchy is used to support a top-down design methodology, and facilities for data partitioning enforce a solely problem-oriented domain for developing parallel programs. Performance issues are clearly separated from design issues, and performance rightly becomes a final-stage adjustment process. This adjustment is supported by simple, abstract, graphical mechanisms which allow re-targeting of a program to a different architecture with minimal programmer effort.

The use of abstract approaches for parallel programming such as graphs and dataflow have been attempted before, and some success has been achieved. However, a common downfall of other approaches has been the lack of coherence and consistency in using abstraction. Often, specialist low-level mechanisms have remained, or been introduced, requiring the knowledge and understanding of new language features, and leading to the lack of comprehension of the overall design procedure. While ParaDE clearly introduces a new notation, the syntax and semantics of the notation is intended to be intuitive, and is applied at a design level. Existing programming skills are exploited at lower levels of refinement of the design, and the design process is structured and problem-oriented.

A common criticism of abstraction is the loss of efficiency caused by the introduction of an additional layer of software. The success of ParaDE as a viable design system hinges on its ability to produce programs that perform well over different architectures. The use of code generation and granularity adjustment have shown that this is indeed achievable, and performance figures have demonstrated that ParaDE outperforms its counterparts, producing efficient solutions on widely different architectures.

The techniques investigated in this research, and demonstrated in ParaDE, have proved successful both in performance and usability in the experiments detailed in this thesis. Abstraction, combined with efficient automation and a coherent graphical

design methodology, has been shown to achieve the ideal of architecture independence without sacrificing performance. The architecture model, whether it be shared or distributed memory, becomes transparent using these techniques, and it is hoped that this will help move towards the aim of a common parallel programming approach. The ParaDE system has been demonstrated successfully on two parallel architectures and if applied with similar success to other sectors of the fragmented parallel software field, ParaDE could lead the way with a unifying methodology and a revolution in parallel programming.

Bibliography

1. Amdahl, G.: "The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *AFIPS*, 1967.
2. Harley, J.W.: *Dataflow Development of Medium-Grained Parallel Software*, PhD Thesis, University of Newcastle upon Tyne, September 1993.
3. Gajski, D.D., et al.: "A Second Opinion on Data Flow Machines and Languages", *IEEE Computer*, vol. 15, pp. 58-70, February 1982.
4. Almasi, G.S., Gottlieb, A.J.: *Highly Parallel Computing*, Second ed., Benjamin/Cummings, 1994.
5. Carriero, N., Gelernter, D.: "How to write parallel programs: A guide to the perplexed", *ACM Computing Surveys*, vol. 21, pp. 323-357, September 1989.
6. Chandy, M., et al.: "Integrated Support for Task and Data Parallelism", *International Journal of Supercomputer Applications*, August 1993.
7. Hassen, S.B., Bal, H.: "Integrating Task and Data Parallelism Using Shared Objects", *Technical Report*, Vrije Universiteit, Amsterdam, 1995.
8. Chakrabarti, S., Demmel, J., Yelick, K.: "Modelling the Benefits of Mixed Data and Task Parallelism", *Technical Report*, University of Tennessee, 1995.
9. Flynn, M.J.: "Some Computer Organisations and Their Effectiveness", *IEEE Transactions on Computers*, vol. 21, pp. 948-960, 1972.
10. Kuck, D.J.: *The Structure of Computers and Computations*, John-Wiley, 1978.
11. Treleaven, P.C., Brownbridge, D.R., Hopkins, R.P.: "Data Driven and Demand Driven Computer Architecture", *ACM Computing Surveys*, vol. 14, p. 93-143, March 1982.
12. Perrott, R.H., Hoare, C.A.R.: *Operating Systems Techniques*, 1972.
13. Dijkstra, E.W.: "Cooperating Sequential Processes", *Programming Languages*, Academic Press, New York, 1968.
14. Hoare, C.A.R.: "Monitors: an operating system structuring concept", *Communications of the ACM*, vol. 10, pp. 549-57, 1974.
15. Sunderam, V.S.: "PVM: A Framework for Parallel Distributed Computing", *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315-339, December 1990.
16. Gelernter, D.: "Generative Communication in Linda", *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 80-112, 1985.
17. Kennedy, K., Kremer, U.: "Automatic Data Alignment and Distribution for Loosely Synchronous Problems in an Interactive Programming Environment", *Technical Report*, Center for Research on Parallel Computations, Rice University, April 1991.
18. Agarwal, A., Kranz, D., Natarajan, V.: "Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared Memory Multiprocesors", *ICPP*, 1993.

19. Kremer, U., et al.: “Automatic Data Layout for Distributed-Memory Machines in the D Programming Environment”, *Technical Report*, Center for Research on Parallel Computations, Rice University, February 1993.

20. Wholey, S.: “Automatic Data Mapping for Distributed-Memory Parallel Computers”, *International Conference on Supercomputing*, ACM Press, Washington D.C., 1992.

21. Crooks, P.: *An Automatic Program Translator for Distributed Memory MIMD Machines*, PhD Thesis, Queens University, Belfast, 1995.

22. Fox, G.E.A.: “Fortran D Language Specification”, *Technical Report,* Rice University, April 1991.

23. Gelernter, D., Carriero, N.: “Coordination Languages and their Significance”, *Communications of the ACM*, vol. 35, no. 2, pp. 97-107, February 1992.

24. Hall, M.W., Kennedy, K., McKinley K.S.: “Interprocedural Transformations for Parallel Code Generation”, *Supercomputing 91*, IEEE, Albuquerque, 1991.

25. Hoare, C.A.R.: “Communicating Sequential Processes”, *Communications of the ACM*, vol. 21, pp. 666-677, August 1978.

26. Gehani, N.: *Ada Concurrent Programming*, Prentice Hall, 1984.

27. Inmos, L.: *Occam Programming Manual*, 1985.

28. Loveman, D.: “High Performance Fortran”, *IEEE Parallel and Distributed Technology*, vol. 1, pp. 25-42, February 1993.

29. Gurd, J.R., Snelling, D.F.: “Manchester Data-Flow: A Progress Report”, *International Conference on Supercomputing*, ACM Press, Washington D.C., 1992.

30. Mohr, E., Kranz, D.A., Halstead, R.H.J.: “Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs”, *IEEE Transactions on Parallel and Distributed Systems*, 1990.

31. Allen, R.: “Exploiting Multiple Granularities of Parallelism in a Compiler”, *COMPCON 90*. IEEE Computer Society Press, San Francisco, 1990.

32. Alverson, G., et al.: “Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor”, *International Conference on Supercomputing*, Washington D. C., 1992.

33. Agarwal, A.E.A.: “The MIT Alewife Machine : Architecture and Performance”, *ISCA 95*, 1995.

34. Ruddock, D.E., Dasarathy, B.: “Multithreading Programs: Guidelines for DCE Applications”, *IEEE Software*, vol. 13, pp. 80-90, January 1995.

35. Ackerman, W.B.: “Data Flow Languages”, *IEEE Computer*, vol. 15, February 1982.

36. Bohm, A.P.W., et al.: *SISAL Reference Manual*, Lawrence Livermore National Laboratory, Colorado State University, 1989.

37. Cann, D.: “Retire Fortran? A Debate Rekindled”, *Communications of the ACM*, vol. 35, no. 8, pp. 81-89, August 1992.

38. Davis, A.L., Keller, R.M.: “Data Flow Program Graphs”, *IEEE Computer*, vol. 15, February 1982.

39. Gurd, J.R., Treleaven, P.C.: “A Highly Parallel Computer Architecture”, *Technical Report*, University of Manchester, 1976.
40. Kodama, Y.E.A.: “A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation”, *InfoJapan*, 1990.
41. Ruggiero, C.A., Sargent, J.: “Control of Parallelism in the Manchester Dataflow Computer”, *Lecture Notes in Computer Science*, vol. 274, pp. 1-15, 1987.
42. Bohm, A.P.W., Teo, Y.M.: “Resource Management in a Multi-Ring Dataflow Machine”, *CONPAR*, 1988.
43. Foley, J.F.: “Manchester Dataflow Machine: Benchmark Test Evaluation Report”, Technical Report, Manchester University, 1989.
44. Watt, D.A.: *Programming Language Concepts and Paradigms*, Prentice Hall, 1990.
45. Jagannathan, R.: “Coarse-Grain Dataflow Programming of Conventional Parallel Computers”, *Technical Report*, SRI International, 1995.
46. Ashcroft, E.A.: “Dataflow and Eduction: Data-driven and demand-driven distributed computation”, *Current Trends in Concurrency*, Springer Verlag, 1986.
47. Scientific Computing Associates: *C-Linda User’s Guide and Reference Manual*, Scientific Computing Associates, 1992.
48. Ichikawa, T., Hirakawa, M.: “Iconic Programming : Where to Go?”, *IEEE Software*, vol. 7, pp. 63-68, November 1990.
49. Beguelin, A., et al.: “Graphical Development Tools for Network Based Concurrent Supercomputing”, *Supercomputing 91*, IEEE, Albuquerque, 1991.
50. Beguelin, A., et al.: *HeNCE: A Users’ Guide Version 2.0*, University of Tennessee,1994.
51. Browne, J.C., Azam, M., Sobek, S.: “CODE: A Unified Approach to Parallel Programming”, *IEEE Software*, vol. 6, pp. 10-20, July 1989.
52. Browne, J.C., Newton, P.: “The CODE 2.0 Graphical Parallel Programming Language”, *ICS92*, ACM Press, Washington D.C., 1992.
53. Browne, J.C., et al.: “Visual Programming and Debugging for Parallel Computing”, *Parallel and Distributed Technology*, Vol. 3, no. 1, p. 75-83, 1995.
54. Gibbons, A., Ryter, W.: *Efficient Parallel Algorithms*, Cambridge University Press, 1988.
55. Browne, J.C., et al.: “Visual Programming and Parallel Computing”, *Technical Report*, University of Tennessee at Knoxville, 1994.
56. Newton, P., Dongarra, J.: “Overview of VPE: A Visual Environment for Message-Passing Parallel Programming”, *Technical Report*, University of Tennessee at Knoxville, 1994.
57. Ousterhout, *Tcl and the Tk toolkit*, 1995.

Appendix A: Listings for Data Partitioning Example

This appendix presents two copies of the automatically generated code for the matrix multiplication program, demonstrating data partitioning on different architectures. The first copy is generated for the Encore Multimax (shared memory) architecture, the second for a network of workstations. User code is identified by comments in the code, the remaining code is generated by ParaDE.

```
/*#####################################################*\
#                                                     #
#           ENCORE LINDA                              #
#    matrix multiplication (automatically generated)  #
#    file work/tcl/tests/matmult5/linda.l             #
#    Robert S. Allen  Tue May 21 10:08:25 BST 1996    #
#                                                     #
\*#####################################################*/
/*  #define GLOBALS  */
#include "global.h"
#include "linda.h"
#include <stdio.h>
int more=1;
char buf[64];
char buf2[128];

int actor0(numOfSegments)
int numOfSegments;
{
int i ;
int j ;
int x ;
int dp1_counter;
int dp1[100][100];
int tempdp1[100][100];
int temp1;
LINDA_BLOCK DP1;
int *pdp1 = &(dp1[0][0]);
int *ptempdp1 = &(tempdp1[0][0]);
  DP1.data=(long *)pdp1;
  DP1.size= 100 * 100;
  dp1_counter=0;
  dp1[0][0] = EMPTY;
  start_timer(); /*4*/
    /**** Start of User's Code ***/
    x=1;
    for (i=0;i<100;i++) {
      for (j=0;j<100;j++) {
        if (i==j) {
          dp1[i][j] = 1;
        } else {
          dp1[i][j] = 0;
```

```
          }
        }
      }
      /**** End of User's Code ***/
    timer_split("actor0");
    print_times();
    if (dp1[0][0] != EMPTY) {
      out("dp1", dp1_counter ,   DP1); }
    dp1_counter++;
}

int actor1(numOfSegments)
int numOfSegments;
{
int i ;
int j ;
int x ;
int dp0_counter;
int dp0[100][100];
int segments;
LINDA_BLOCK DP0;
int *pdp0 = &(dp0[0][0]);
    DP0.data=(long *)pdp0;
    DP0.size=100 * 100 / numOfSegments;
    dp0_counter=0;
    dp0[0][0] = EMPTY;
    start_timer(); /*4*/
      /**** Start of User's Code ***/
      x=1;
      for(i=0;i<100;i++) {
        for(j=0;j<100;j++) {
          dp0[i][j]=x;
          x++;
        }
      }
      /**** End of User's Code ***/
    timer_split("actor1");
    print_times();
    if (dp0[0][0] != EMPTY) {
    for(segments =0; segments < numOfSegments; segments++)
{
      out("dp0", dp0_counter, segments, DP0);
      pdp0=pdp0+100 * 100 / numOfSegments;
      DP0.data=(long *)pdp0;
    }
    dp0_counter++;
    }
    dp0_counter++;
}

int actor2(iteration, numOfSegments)
int iteration;
int numOfSegments;
{
```

```
int i ;
int j ;
int sum ;
int dp2_counter;
int dp2[100][100];
LINDA_BLOCK DP2;
int *pdp2 = &(dp2[0][0]);
int which;
int input_counter;
int parts;
int dp0[100][100];
LINDA_BLOCK DP0;
int *pdp0 = &(dp0[0][0]);
int dp1[100][100];
int temp1;
int tempdp1[100][100];
LINDA_BLOCK DP1;
int *pdp1 = &(dp1[0][0]);
  input_counter=0;
  dp2_counter=0;
  DP2.data=(long *)pdp2;
  DP2.size=100 * 100 / numOfSegments;
  DP0.data=(long *)pdp0;
  DP0.size=100 * 100 / numOfSegments;
  DP1.data=(long *)pdp1;
  DP1.size=100 * 100;
  while (more) {
  dp2[0][0] = EMPTY;
   rd ("dp1", ? &temp1,     ? &DP1);
  in("dp0",input_counter, ? &which, ? &DP0);
  input_counter++;
  start_timer();
  for(parts=0;parts < (100 / numOfSegments); parts++) {
    /**** Start of User's Code ***/
    for(i=0;i<100;i++) {
    sum=0;
      for(j=0;j<100;j++) {
        sum=sum+dp1[i][j]*dp0[parts][j];
      }
      dp2[parts][i]=sum;
    }
    /**** End of User's Code ***/
  }
  timer_split("actor2");
  print_times();
    out("dp2",dp2_counter++, which, DP2);
  dp2_counter++;
  }
}

int Stdout3(numOfSegments)
int numOfSegments;
{
long printLoop;
```

```
long printLoop2;
int input_counter;
int dp2[100][100];
int temp2;
int tempdp2[100][100];
int iterations;
int segments;
LINDA_BLOCK DP2;
int *pdp2 = &(dp2[0][0]);
  input_counter=0;
  DP2.data=(long *)pdp2;
  DP2.size=100 * 100 / numOfSegments;
  while (more) {
  for(segments = 0; segments < numOfSegments; segments++)
{
    in("dp2",input_counter,segments, ? &DP2);
  pdp2=pdp2 + 100 * 100 / numOfSegments;
  DP2.data=(long *)pdp2;
 }
  input_counter++;
  start_timer(); /*4*/
 printf("Result from Stdout 3 is : \n");
    for(printLoop=0;printLoop< 100;  printLoop++) {
      for(printLoop2=0;printLoop2< 100; printLoop2++) {
        printf("%d ",dp2[printLoop2][printLoop]);
      }
      printf("\n");
    }
  printf("\nFinished\n");
  timer_split("actor3");
  print_times();
  out("stop");
  input_counter++;
  }
}

lmain(argc,argv)
int argc;
char *argv[];
{
int iterations;
int i;
int x;
int iterations1;
  iterations1 = atoi(argv[1]);
  eval("Stdout3",  Stdout3(iterations1));
  for(i=0;i<iterations1;i++)
    eval("actor2",  actor2(i, iterations1));
  eval("actor1",  actor1(iterations1));
  start_timer();
  eval("actor0",  actor0(iterations1));
  in("stop");
  timer_split("main");
  print_times();
```

```
  lexit();
}
```

```
/*#######################################################*\
#                                                       #
#           NETWORK LINDA                               #
#    matrix multiplication (automatically generated)   #
#    file work/tcl/tests/matmult5/linda.cl              #
#    Robert S. Allen  Tue May 21 09:18:59 BST 1996     #
#                                                       #
\*#######################################################*/
/*  #define GLOBALS  */
#include "global.h"
#include <stdio.h>
int more=1;
char buf[64];
char buf2[128];

int actor0(numOfSegments)
int numOfSegments;
{
int i ;
int j ;
int x ;
int dp1_counter;
int dp1[100][100];
  dp1_counter=0;
  dp1[0][0] = EMPTY;
  start_timer(); /*4*/
    /**** Start of User's Code ***/
    x=1;
    for (i=0;i<100;i++) {
      for (j=0;j<100;j++) {
        if (i==j) {
          dp1[i][j] = 1;
        } else {
          dp1[i][j] = 0;
        }
      }
    }
    /**** End of User's Code ***/
  timer_split("actor0");
  print_times();
  if (dp1[0][0] != EMPTY) {
    out("dp1", dp1_counter ,   dp1); }
  dp1_counter++;
}

int actor1(numOfSegments)
int numOfSegments;
{
int i ;
int j ;
int x ;
int dp0_counter;
int dp0[100][100];
int segments;
```

```
  dp0_counter=0;
  dp0[0][0] = EMPTY;
  start_timer(); /*4*/
    /**** Start of User's Code ***/
    x=1;
    for(i=0;i<100;i++) {
      for(j=0;j<100;j++) {
        dp0[i][j]=x;
        x++;
      }
    }
    /**** End of User's Code ***/
  timer_split("actor1");
  print_times();
  if (dp0[0][0] != EMPTY) {
  for(segments =0; segments < numOfSegments; segments++)
{
    out("dp0",           dp0_counter,           segments,
dp0[segments*100/numOfSegments]:100 *100/numOfSegments);
  }
  dp0_counter++;
  }
  dp0_counter++;
}

int actor2(iteration, numOfSegments)
int iteration;
int numOfSegments;
{
int i ;
int j ;
int sum ;
int dp2_counter;
int dp2[100][100];
int which;
int input_counter;
int parts;
int dp0[100][100];
int dp1[100][100];
  input_counter=0;
  dp2_counter=0;
  while (more) {
  dp2[0][0] = EMPTY;
   rd ("dp1", ? int,     ?dp1);
  in("dp0",input_counter, ?which, ?dp0[0]:);
  input_counter++;
  start_timer();
  for(parts=0;parts < (100 / numOfSegments); parts++) {
    /**** Start of User's Code ***/
    for(i=0;i<100;i++) {
    sum=0;
      for(j=0;j<100;j++) {
        sum=sum+dp1[i][j]*dp0[parts][j];
      }
```

```
      dp2[parts][i]=sum;
    }
    /**** End of User's Code ***/
  }
  timer_split("actor2");
  print_times();
    out("dp2",dp2_counter++,       which,       dp2[0]:100
*100/numOfSegments);
  dp2_counter++;
  }
}

int Stdout3(numOfSegments)
int numOfSegments;
{
long printLoop;
long printLoop2;
int input_counter;
int dp2[100][100];
int temp2;
int tempdp2[100][100];
int iterations;
int segments;
  input_counter=0;
  while (more) {
  for(segments = 0; segments < numOfSegments; segments++)
{
  in("dp2",input_counter,segments,         ?dp2[segments*
100/numOfSegments]:);
 }
  input_counter++;
  start_timer(); /*4*/
  printf("Result from Stdout 3 is : \n");
    for(printLoop=0;printLoop< 100;  printLoop++) {
      for(printLoop2=0;printLoop2< 100; printLoop2++) {
        printf("%d ",dp2[printLoop2][printLoop]);
      }
      printf("\n");
    }
  printf("\nFinished\n");
  fflush();
  timer_split("actor3");
  print_times();
  out("stop");
  input_counter++;
  }
}

real_main(argc,argv)
int argc;
char *argv[];
{
int iterations;
int i;
```

```
int x;
int iterations1;
  iterations1 = atoi(argv[1]);
  eval("Stdout3",  Stdout3(iterations1));
  for(i=0;i<iterations1;i++)
    eval("actor2",  actor2(i, iterations1));
  eval("actor1",  actor1(iterations1));
  start_timer();
  eval("actor0",  actor0(iterations1));
  in("stop");
  timer_split("main");
  print_times();
}
```

Appendix B: Listings for Actor Folding Example

This appendix presents three versions of the code generated for the multiple matrix calculation program. All versions are generated for the network architecture, the aim here is to illustrate the effect of folding strategies. From the same program graph and user code, code is regenerated for the folding strategies referred to as notfolded, fold4, and foldall which are described in Chapter 5.

```
/*#####################################################*\
#                                                       #
#           NETWORK LINDA                               #
#    matrix calculation (automatically generated)       #
#    file work/tcl/tests/matcalc5/linda.cl              #
#    Robert S. Allen  Sun Nov  3 20:00:25 GMT 1996      #
#                                                       #
\*#####################################################*/
#include "linda.h"
#include <stdio.h>
#define MATRIXSIZE 5
#define MAXDUMMY 10
int dummy;
int more=1;
char buf[64];
char buf2[128];

int actor0()
{
int a ;
int b ;
int dp0_counter;
int dp0[MATRIXSIZE][MATRIXSIZE];
dp0_counter=0;
    /**** Start of User's Code ***/
    for(a=0;a<MATRIXSIZE;a++)
      for(b=0;b<MATRIXSIZE;b++)
        dp0[a][b]=1;
    /**** End of User's Code ***/
out("dp0",dp0_counter,      dp0);
}

int actor1()
{
int c ;
int d ;
int dp1_counter;
int dp1[MATRIXSIZE][MATRIXSIZE];
dp1_counter=0;
    /**** Start of User's Code ***/
    for(c=0;c<MATRIXSIZE;c++)
      for(d=0;d<MATRIXSIZE;d++)
        dp1[c][d]=1;
    /**** End of User's Code ***/
out("dp1",dp1_counter,      dp1);
```

```
}

int actor2()
{
int e ;
int f ;
int dp2_counter;
int dp2[MATRIXSIZE][MATRIXSIZE];
dp2_counter=0;
    /**** Start of User's Code ***/
    for(e=0;e<MATRIXSIZE;e++)
      for(f=0;f<MATRIXSIZE;f++)
        dp2[e][f]=f+1;
    /**** End of User's Code ***/
out("dp2",dp2_counter,      dp2);
}

int actor3()
{
int g ;
int h ;
int dp3_counter;
int dp3[MATRIXSIZE][MATRIXSIZE];
dp3_counter=0;
    /**** Start of User's Code ***/
    for(g=0;g<MATRIXSIZE;g++)
      for(h=0;h<MATRIXSIZE;h++)
        dp3[g][h]=h+1;
    /**** End of User's Code ***/
out("dp3",dp3_counter,      dp3);
}

int actor4()
{
int i ;
int j ;
int dp4_counter;
int dp4[MATRIXSIZE][MATRIXSIZE];
dp4_counter=0;
    /**** Start of User's Code ***/
    for(i=0;i<MATRIXSIZE;i++)
      for(j=0;j<MATRIXSIZE;j++)
        dp4[i][j]=1;
    /**** End of User's Code ***/
out("dp4",dp4_counter,      dp4);
}

int actor5()
{
int k ;
int l ;
int dp5_counter;
int dp5[MATRIXSIZE][MATRIXSIZE];
dp5_counter=0;
```

```
    /**** Start of User's Code ***/
    for(k=0;k<MATRIXSIZE;k++)
      for(l=0;l<MATRIXSIZE;l++)
        dp5[k][l]=1;
    /**** End of User's Code ***/
out("dp5",dp5_counter,      dp5);
}

int actor6()
{
int m ;
int n ;
int dp6_counter;
int dp6[MATRIXSIZE][MATRIXSIZE];
dp6_counter=0;
    /**** Start of User's Code ***/
    for(m=0;m<MATRIXSIZE;m++)
      for(n=0;n<MATRIXSIZE;n++)
        dp6[m][n]=1;
    /**** End of User's Code ***/
out("dp6",dp6_counter,      dp6);
}

int actor7()
{
int o ;
int p ;
int dp7_counter;
int dp7[MATRIXSIZE][MATRIXSIZE];
dp7_counter=0;
    /**** Start of User's Code ***/
    for(o=0;o<MATRIXSIZE;o++)
      for(p=0;p<MATRIXSIZE;p++)
        dp7[o][p]=1;
    /**** End of User's Code ***/
out("dp7",dp7_counter,      dp7);
}

int actor8()
{
int q ;
int r ;
int s ;
int sum ;
int dp8_counter;
int dp8[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int dp0[MATRIXSIZE][MATRIXSIZE];
int dp1[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
  dp8_counter=0;
while (more) {
   in ("dp0", input_counter,     ? dp0);
   in ("dp1", input_counter,     ? dp1);
```

```
     /**** Start of User's Code ***/
     for(q=0;q<MATRIXSIZE;q++)
       for(r=0;r<MATRIXSIZE;r++)
       {
         sum=0;
         for(s=0;s<MATRIXSIZE;s++)
           {
           sum=sum+dp0[q][s]*dp1[s][r];
           for(dummy=0;dummy<MAXDUMMY;dummy++);
           }
         dp8[q][r]=sum;
       }
     /**** End of User's Code ***/
out("dp8",dp8_counter,      dp8);
  }
}

int actor10()
{
int t ;
int u ;
int v ;
int sum1 ;
int dp9_counter;
int dp9[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int dp2[MATRIXSIZE][MATRIXSIZE];
int dp3[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
  dp9_counter=0;
while (more) {
   in ("dp2", input_counter,     ? dp2);
   in ("dp3", input_counter,     ? dp3);
    /**** Start of User's Code ***/
    for(t=0;t<MATRIXSIZE;t++)
      for(u=0;u<MATRIXSIZE;u++)
      {
        sum1=0;
        for(v=0;v<MATRIXSIZE;v++)
          {
          sum1=sum1+dp2[t][v]*dp3[v][u];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
          }
        dp9[t][u]=sum1;
      }
    /**** End of User's Code ***/
out("dp9",dp9_counter,      dp9);
  }
}

int actor9()
{
int v1 ;
int w ;
```

```
int x ;
int sum3 ;
int dp11_counter;
int dp11[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int dp4[MATRIXSIZE][MATRIXSIZE];
int dp5[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
  dp11_counter=0;
while (more) {
   in ("dp4", input_counter,     ? dp4);
   in ("dp5", input_counter,     ? dp5);
    /**** Start of User's Code ***/
    for(v1=0;v1<MATRIXSIZE;v1++)
      for(w=0;w<MATRIXSIZE;w++)
      {
        sum3=0;
        for(x=0;x<MATRIXSIZE;x++)
        {
          sum3=sum3+dp4[v1][x]*dp5[x][w];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
        dp11[v1][w]=sum3;
      }
    /**** End of User's Code ***/
out("dp11",dp11_counter,      dp11);
  }
}

int actor11()
{
int m1 ;
int n1 ;
int o1 ;
int sum4 ;
int dp12_counter;
int dp12[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int dp6[MATRIXSIZE][MATRIXSIZE];
int dp7[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
  dp12_counter=0;
while (more) {
   in ("dp6", input_counter,     ? dp6);
   in ("dp7", input_counter,     ? dp7);
    /**** Start of User's Code ***/
    for(m1=0;m1<MATRIXSIZE;m1++)
      for(n1=0;n1<MATRIXSIZE;n1++)
      {
        sum4=0;
        for(o1=0;o1<MATRIXSIZE;o1++)
        {
          sum4=sum4+dp6[m1][o1]*dp7[o1][n1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
```

```
        }
        dp12[m1][n1]=sum4;
      }
    /**** End of User’s Code ***/
out("dp12",dp12_counter,      dp12);
  }
}

int actor12()
{
int p1 ;
int q1 ;
int r1 ;
int sum5 ;
int dp10_counter;
int dp10[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int dp8[MATRIXSIZE][MATRIXSIZE];
int dp9[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
  dp10_counter=0;
while (more) {
   in ("dp8", input_counter,     ? dp8);
   in ("dp9", input_counter,     ? dp9);
    /**** Start of User’s Code ***/
    for(p1=0;p1<MATRIXSIZE;p1++)
      for(q1=0;q1<MATRIXSIZE;q1++)
      {
        sum5=0;
        for(r1=0;r1<MATRIXSIZE;r1++)
        {
          sum5=sum5+dp8[p1][r1]*dp9[r1][q1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
        dp10[p1][q1]=sum5;
      }
    /**** End of User’s Code ***/
out("dp10",dp10_counter,      dp10);
  }
}

int actor13()
{
int y1 ;
int z1 ;
int dp13_counter;
int dp13[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int dp10[MATRIXSIZE][MATRIXSIZE];
int dp11[MATRIXSIZE][MATRIXSIZE];
int dp12[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
  dp13_counter=0;
while (more) {
```

```
   in ("dp10", input_counter,     ? dp10);
   in ("dp11", input_counter,     ? dp11);
   in ("dp12", input_counter,     ? dp12);
    /**** Start of User’s Code ***/
    for(y1=0;y1<MATRIXSIZE;y1++)
      for(z1=0;z1<MATRIXSIZE;z1++)
        {

dp13[y1][z1]=dp12[y1][z1]+dp11[y1][z1]+dp10[y1][z1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
    /**** End of User’s Code ***/
out("dp13",dp13_counter,      dp13);
  }
}

int Stdout14()
{
int input_counter;
int dp13[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
while (more) {
   in ("dp13", input_counter,     ? dp13);
printf("%d %d %d %d”,
dp13[0][0],dp13[0][1],dp13[0][2],dp13[0][3]);
  printf("\nFinished\n");
out("stop");
  }
}

real_main(argc,argv)
int argc;
char *argv[];
{
int i;
int j;
int x;
  eval("Stdout14",  Stdout14());
  eval("actor13",  actor13());
  eval("actor12",  actor12());
  eval("actor11",  actor11());
  eval("actor9",  actor9());
  eval("actor10",  actor10());
  eval("actor8",  actor8());
  eval("actor7",  actor7());
  start_timer();
  eval("actor6",  actor6());
  eval("actor5",  actor5());
  eval("actor4",  actor4());
  eval("actor3",  actor3());
  eval("actor2",  actor2());
  eval("actor1",  actor1());
  eval("actor0",  actor0());
  in("stop");
```

```
 timer_split("main");
 print_times();
 lhalt();
}
```

```
/*########################################################*\
#                                                          #
#           NETWORK LINDA                                  #
#    matrix calculation  (automatically generated)        #
#    file work/tcl/tests/matcalc5/linda.cl                 #
#    Robert S. Allen  Sun Nov  3 18:52:44 GMT 1996         #
#                                                          #
\*########################################################*/
/*  #define GLOBALS  */
#include "linda.h"
#include <stdio.h>
#define MATRIXSIZE 5
#define MAXDUMMY 10
int dummy=0;
int more=1;
char buf[64];
char buf2[128];

int actor0()
{
int a ;
int b ;
int dp0_counter;
int dp0[MATRIXSIZE][MATRIXSIZE];
int c ;
int d ;
int dp1_counter;
int dp1[MATRIXSIZE][MATRIXSIZE];
int q ;
int r ;
int s ;
int sum ;
int dp8_counter;
int dp8[MATRIXSIZE][MATRIXSIZE];
int input_counter;
  dp8_counter=0;
/**** Start of User's Code ***/
    for(a=0;a<MATRIXSIZE;a++)
      for(b=0;b<MATRIXSIZE;b++)
        dp0[a][b]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(c=0;c<MATRIXSIZE;c++)
      for(d=0;d<MATRIXSIZE;d++)
        dp1[c][d]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(q=0;q<MATRIXSIZE;q++)
      for(r=0;r<MATRIXSIZE;r++)
      {
        sum=0;
        for(s=0;s<MATRIXSIZE;s++)
          {
```

```
          sum=sum+dp0[q][s]*dp1[s][r];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
          }
        dp8[q][r]=sum;
      }
    /**** End of User's Code ***/
out("dp8",dp8_counter,      DPOUT8);
}

int actor2()
{
int e ;
int f ;
int dp2_counter;
int dp2[MATRIXSIZE][MATRIXSIZE];
int g ;
int h ;
int dp3_counter;
int dp3[MATRIXSIZE][MATRIXSIZE];
int t ;
int u ;
int v ;
int sum1 ;
int dp9_counter;
int dp9[MATRIXSIZE][MATRIXSIZE];
int input_counter;
  input_counter=0;
  dp9_counter=0;
while (more) {
    /**** Start of User's Code ***/
    for(e=0;e<MATRIXSIZE;e++)
      for(f=0;f<MATRIXSIZE;f++)
        dp2[e][f]=f+1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(g=0;g<MATRIXSIZE;g++)
      for(h=0;h<MATRIXSIZE;h++)
        dp3[g][h]=h+1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(t=0;t<MATRIXSIZE;t++)
      for(u=0;u<MATRIXSIZE;u++)
      {
        sum1=0;
        for(v=0;v<MATRIXSIZE;v++)
          {
          sum1=sum1+dp2[t][v]*dp3[v][u];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
          }
        dp9[t][u]=sum1;
      }
    /**** End of User's Code ***/
out("dp9",dp9_counter,      DPOUT9);
  }
```

```
}

int actor4()
{
int i ;
int j ;
int dp4_counter;
int dp4[MATRIXSIZE][MATRIXSIZE];
int k ;
int l ;
int dp5_counter;
int dp5[MATRIXSIZE][MATRIXSIZE];
int v1 ;
int w ;
int x ;
int sum3 ;
int dp11_counter;
int dp11[MATRIXSIZE][MATRIXSIZE];
int input_counter;
  input_counter=0;
  dp11_counter=0;
while (more) {
    /**** Start of User's Code ***/
    for(i=0;i<MATRIXSIZE;i++)
      for(j=0;j<MATRIXSIZE;j++)
        dp4[i][j]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(k=0;k<MATRIXSIZE;k++)
      for(l=0;l<MATRIXSIZE;l++)
        dp5[k][l]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(v1=0;v1<MATRIXSIZE;v1++)
      for(w=0;w<MATRIXSIZE;w++)
      {
        sum3=0;
        for(x=0;x<MATRIXSIZE;x++)
        {
          sum3=sum3+dp4[v1][x]*dp5[x][w];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
        dp11[v1][w]=sum3;
      }
    /**** End of User's Code ***/
out("dp11",dp11_counter,      DPOUT11);
  }
}

int actor6()
{
int m ;
int n ;
int dp6_counter;
```

```
int dp6[MATRIXSIZE][MATRIXSIZE];
int o ;
int p ;
int dp7_counter;
int dp7[MATRIXSIZE][MATRIXSIZE];
int m1 ;
int n1 ;
int o1 ;
int sum4 ;
int dp12_counter;
int dp12[MATRIXSIZE][MATRIXSIZE];
int input_counter;
  input_counter=0;
  dp12_counter=0;
while (more) {
    /**** Start of User's Code ***/
    for(m=0;m<MATRIXSIZE;m++)
      for(n=0;n<MATRIXSIZE;n++)
        dp6[m][n]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(o=0;o<MATRIXSIZE;o++)
      for(p=0;p<MATRIXSIZE;p++)
        dp7[o][p]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(m1=0;m1<MATRIXSIZE;m1++)
      for(n1=0;n1<MATRIXSIZE;n1++)
      {
        sum4=0;
        for(o1=0;o1<MATRIXSIZE;o1++)
        {
          sum4=sum4+dp6[m1][o1]*dp7[o1][n1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
        dp12[m1][n1]=sum4;
      }
    /**** End of User's Code ***/
out("dp12",dp12_counter,      DPOUT12);
  }
}

int actor12()
{
int p1 ;
int q1 ;
int r1 ;
int sum5 ;
int dp10_counter;
int dp10[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int dp8[MATRIXSIZE][MATRIXSIZE];
int dp9[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
```

```
  dp10_counter=0;
while (more) {
   in ("dp8", input_counter,     ? &DPIN8);
   in ("dp9", input_counter,     ? &DPIN9);
    /**** Start of User's Code ***/
    for(p1=0;p1<MATRIXSIZE;p1++)
      for(q1=0;q1<MATRIXSIZE;q1++)
      {
        sum5=0;
        for(r1=0;r1<MATRIXSIZE;r1++)
        {
          sum5=sum5+dp8[p1][r1]*dp9[r1][q1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
        dp10[p1][q1]=sum5;
      }
    /**** End of User's Code ***/
out("dp10",dp10_counter,      DPOUT10);
  }
}

int actor13()
{
int y1 ;
int z1 ;
int dp13_counter;
int dp13[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int dp10[MATRIXSIZE][MATRIXSIZE];
int dp11[MATRIXSIZE][MATRIXSIZE];
int dp12[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
  dp13_counter=0;
while (more) {
   in ("dp10", input_counter,     ? &DPIN10);
   in ("dp11", input_counter,     ? &DPIN11);
   in ("dp12", input_counter,     ? &DPIN12);
    /**** Start of User's Code ***/
    for(y1=0;y1<MATRIXSIZE;y1++)
      for(z1=0;z1<MATRIXSIZE;z1++)
        {

dp13[y1][z1]=dp12[y1][z1]+dp11[y1][z1]+dp10[y1][z1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
    /**** End of User's Code ***/
out("dp13",dp13_counter,      DPOUT13);
  }
}

int Stdout14()
{
long printLoop;
long printLoop2;
```

```
int input_counter;
int dp13[MATRIXSIZE][MATRIXSIZE];
  input_counter=0;
while (more) {
   in ("dp13", input_counter,     ? &DPIN13);
printf("%d              %d              %d              %d
",dp13[0][0],dp13[0][1],dp13[0][2],dp13[0][3]);
  printf("\nFinished\n");
  out("stop");
  }
}

real_main(argc,argv)
int argc;
char *argv[];
{
int i;
int j;
int x;
  eval("Stdout14",  Stdout14());
  eval("actor13",  actor13());
  eval("actor12",  actor12());
start_timer();
  eval("actor6",  actor6());
eval("actor4",  actor4());
eval("actor2",  actor2());
eval("actor0",  actor0());
  in("stop");
  timer_split("main");
  print_times();
  lhalt();
}
```

```
/*#######################################################*\
#                                                       #
#           NETWORK LINDA                               #
#    matrix calculation  (automatically generated)      #
#    file work/tcl/tests/matcalc5/linda.cl              #
#    Robert S. Allen  Sun Nov  3 10:45:43 GMT 1996      #
#                                                       #
\*#######################################################*/
/*  #define GLOBALS  */
#include "linda.h"
#include <stdio.h>
#define MATRIXSIZE 5
#define MAXDUMMY 10
int dummy=0;
int more=1;
char buf[64];
char buf2[128];

int actor0()
{
int a ;
int b ;
int dp0_counter;
int dp0[MATRIXSIZE][MATRIXSIZE];
int c ;
int d ;
int dp1_counter;
int dp1[MATRIXSIZE][MATRIXSIZE];
int q ;
int r ;
int s ;
int sum ;
int dp8_counter;
int dp8[MATRIXSIZE][MATRIXSIZE];
int input_counter;
int e ;
int f ;
int dp4_counter;
int dp4[MATRIXSIZE][MATRIXSIZE];
int g ;
int h ;
int dp5_counter;
int dp5[MATRIXSIZE][MATRIXSIZE];
int v1 ;
int w ;
int x ;
int sum3 ;
int dp11_counter;
int dp11[MATRIXSIZE][MATRIXSIZE];
int i ;
int j ;
int dp2_counter;
int dp2[MATRIXSIZE][MATRIXSIZE];
```

```
int k ;
int l ;
int dp3_counter;
int dp3[MATRIXSIZE][MATRIXSIZE];
int t ;
int u ;
int v ;
int sum1 ;
int dp9_counter;
int dp9[MATRIXSIZE][MATRIXSIZE];
int p1 ;
int q1 ;
int r1 ;
int sum5 ;
int dp10_counter;
int dp10[MATRIXSIZE][MATRIXSIZE];
int m ;
int n ;
int dp6_counter;
int dp6[MATRIXSIZE][MATRIXSIZE];
int o ;
int p ;
int dp7_counter;
int dp7[MATRIXSIZE][MATRIXSIZE];
int m1 ;
int n1 ;
int o1 ;
int sum4 ;
int dp12_counter;
int dp12[MATRIXSIZE][MATRIXSIZE];
int y1 ;
int z1 ;
int dp13_counter;
int dp13[MATRIXSIZE][MATRIXSIZE];
long printLoop;
long printLoop2;
start_timer(); /*4*/
    /**** Start of User's Code ***/
    for(a=0;a<MATRIXSIZE;a++)
      for(b=0;b<MATRIXSIZE;b++)
        dp0[a][b]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(c=0;c<MATRIXSIZE;c++)
      for(d=0;d<MATRIXSIZE;d++)
        dp1[c][d]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(q=0;q<MATRIXSIZE;q++)
      for(r=0;r<MATRIXSIZE;r++)
      {
        sum=0;
        for(s=0;s<MATRIXSIZE;s++)
          {
```

```
      sum=sum+dp0[q][s]*dp1[s][r];
      for(dummy=0;dummy<MAXDUMMY;dummy++);
      }
    dp8[q][r]=sum;
  }
/**** End of User's Code ***/
/**** Start of User's Code ***/
for(e=0;e<MATRIXSIZE;e++)
  for(f=0;f<MATRIXSIZE;f++)
    dp4[e][f]=f+1;
/**** End of User's Code ***/
/**** Start of User's Code ***/
for(g=0;g<MATRIXSIZE;g++)
  for(h=0;h<MATRIXSIZE;h++)
    dp5[g][h]=h+1;
/**** End of User's Code ***/
/**** Start of User's Code ***/
for(v1=0;v1<MATRIXSIZE;v1++)
  for(w=0;w<MATRIXSIZE;w++)
  {
    sum3=0;
    for(x=0;x<MATRIXSIZE;x++)
    {
      sum3=sum3+dp4[v1][x]*dp5[x][w];
      for(dummy=0;dummy<MAXDUMMY;dummy++);
    }
    dp11[v1][w]=sum3;
  }
/**** End of User's Code ***/
/**** Start of User's Code ***/
for(i=0;i<MATRIXSIZE;i++)
  for(j=0;j<MATRIXSIZE;j++)
    dp2[i][j]=1;
/**** End of User's Code ***/
/**** Start of User's Code ***/
for(k=0;k<MATRIXSIZE;k++)
  for(l=0;l<MATRIXSIZE;l++)
    dp3[k][l]=1;
/**** End of User's Code ***/
/**** Start of User's Code ***/
for(t=0;t<MATRIXSIZE;t++)
  for(u=0;u<MATRIXSIZE;u++)
  {
    sum1=0;
    for(v=0;v<MATRIXSIZE;v++)
      {
      sum1=sum1+dp2[t][v]*dp3[v][u];
      for(dummy=0;dummy<MAXDUMMY;dummy++);
      }
    dp9[t][u]=sum1;
  }
/**** End of User's Code ***/
/**** Start of User's Code ***/
for(p1=0;p1<MATRIXSIZE;p1++)
```

```
      for(q1=0;q1<MATRIXSIZE;q1++)
      {
        sum5=0;
        for(r1=0;r1<MATRIXSIZE;r1++)
        {
          sum5=sum5+dp8[p1][r1]*dp9[r1][q1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
        dp10[p1][q1]=sum5;
      }
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(m=0;m<MATRIXSIZE;m++)
      for(n=0;n<MATRIXSIZE;n++)
        dp6[m][n]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(o=0;o<MATRIXSIZE;o++)
      for(p=0;p<MATRIXSIZE;p++)
        dp7[o][p]=1;
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(m1=0;m1<MATRIXSIZE;m1++)
      for(n1=0;n1<MATRIXSIZE;n1++)
      {
        sum4=0;
        for(o1=0;o1<MATRIXSIZE;o1++)
        {
          sum4=sum4+dp6[m1][o1]*dp7[o1][n1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
        dp12[m1][n1]=sum4;
      }
    /**** End of User's Code ***/
    /**** Start of User's Code ***/
    for(y1=0;y1<MATRIXSIZE;y1++)
      for(z1=0;z1<MATRIXSIZE;z1++)
        {

dp13[y1][z1]=dp12[y1][z1]+dp11[y1][z1]+dp10[y1][z1];
          for(dummy=0;dummy<MAXDUMMY;dummy++);
        }
printf("%d            %d            %d            %d
",dp13[0][0],dp13[0][1],dp13[0][2],dp13[0][3]);
  printf("\nFinished\n");
  out("stop");
}

real_main(argc,argv)
int argc;
char *argv[];
{
int i;
int j;
```

```
int x;
start_timer();
eval("actor0",  actor0());
  in("stop");
  timer_split("main");
  print_times();
  lhalt();
}
```