The University of Newcastle upon Tyne

Department of Computing Science

VISUAL OBJECT-ORIENTED DEVELOPMENT OF PARALLEL APPLICATIONS

by

James Webber

Ph.D. Thesis

September 2000

Abstract

Developing software for parallel architectures is a notoriously difficult task, compounded further by the range of available parallel architectures. There has been little research effort invested in how to engineer parallel applications for more general problem domains than the traditional numerically intensive domain. This thesis addresses these issues. An object-oriented paradigm for the development of general-purpose parallel applications, with full lifecycle support, is proposed and investigated, and a visual programming language to support that paradigm is developed. This thesis presents experiences and results from experiments with this new model for parallel application development.

Acknowledgements and Dedications

Firstly I would first like to express my gratitude to my academic supervisor, Professor Pete Lee. Professor Lee has demonstrated a great deal of patience during his time as my supervisor, and offered innumerable valuable insights into my research topic. Without his guidance, I feel sure that my research would have been at a tremendous disadvantage. I owe Professor Lee a significant debt for his work, and I fear that it may be one I will never be able to repay sufficiently.

I would also like to thank the other two members of my thesis committee, Drs. Richard Snow and Paul Watson for their valuable input to my research work. Their opinions and constructive input has helped to focus my research, and increase its maturity. I would also like to thank the Department of Computing Science as a whole for providing a friendly environment and equipment for my research. In particular, I would like to express thanks to Dr. Lindsay Marshall for his advice and friendship.

To my dear friend and fellow Ph.D. student Savas Parastatidis, I would like to offer my most heartfelt thanks. Savas' own unswerving efforts in his own related work served as a terrific source of inspiration and encouragement. During the four years that we have known one another, Savas has offered welcome advice, and been a fine friend throughout.

To other assorted friends and flatmates: Anna Hillier, Rhodri Jones, Louise Barker, Garry Forster, David Ely, Jamie Leech, Philippe Leadbetter, Mark Joszt, and David Ingham, I can only offer my most sincere thanks for their continued and valued friendship.

In addition, I would like to thank my parents and family for their warm support and encouragement throughout my academic career.

And with all my love to Katherine Neasham.

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

Table of Contents

Chapter 1 Introduction 19

1.1 An Overview of Parallel Computing 19

1.2 Parallel Computing Drivers 22

1.3 Moore's Law and the Road to Parallel Computing 23

1.4 Affordable Parallel Platforms 26

1.5 Visual Programming Techniques 29

1.6 The Object-Oriented Paradigm 31

1.7 Research Goals 32

1.8 Summary 33

Chapter 2 Visual Parallel Programming Languages 35

2.1 Introduction 35

2.2 Visual Programming 35

2.2.1 Visual Development Environments 36

2.2.2 Visual Programming Languages and Techniques 37

2.2.3 Visual Parallel Programming 38

2.3 A Survey of Visual Parallel Programming Tools 39

2.3.1 HeNCE – Heterogeneous Network Computing Environment 39

2.3.2 CODE – Computationally Oriented Display Environment 49

2.3.3 VPE – Visual Programming Environment 62

2.3.4 ParADE –Parallel Application Development Environment 70

2.4 Related Work 79

2.5 Closing Remarks 79

Chapter 3 An Analysis of Visual Parallel Programming 81

3.1 Introduction 81

3.2 A Taxonomy of Visual Parallel Programming Languages 82

3.2.1 Syntactic Elements of Visual Parallel Programming Languages 83
3.2.2 Graph 84
3.2.3 Node 87
3.2.4 Arcs 89
3.2.5 Node Syntactic Elements 93
3.2.6 Node: Firing Rule 93
3.2.7 Node: Inputs and Outputs 95
3.2.8 Node: Program 96
3.2.9 Node: Output Token Production Rule 98
3.2.10 Support for Modes of Parallel Execution 99
3.3 Development Paradigm versus Execution Model 103
3.4 Requirements for Visual Parallel Programming Languages 107
3.5 Flow-Based Visual Parallel Programming 108
3.6 Object-Oriented Development 110
3.7 Summary 112
Chapter 4 A New Paradigm and Language for Visual Parallel Programming. 115
4.1 Introduction 115
4.2 Parallel Object-Flow: A New Paradigm for Parallel Application Development 116
4.2.1 The Parallel Object-Flow Development Paradigm 117
4.2.2 The Parallel Object-Flow Execution Model 119
4.2.3 Execution of a Parallel Object-Flow Application 122
4.3 The Vorlon Programming Language 123
4.3.1 Methods in the Vorlon Programming Language 125
4.3.2 The Vorlon Class Model 143
4.3.3 Built-in Types, Primitives, and Library, and Array Support 145
4.4 Summary 149

Chapter 5 Experimental Performance Results and Analysis 151
5.1 Introduction 151
5.2 Software Support - The NIP Parallel Run-Time System 152
5.3 Code Generation 155
5.4 Experimental Environment 165
5.5 Matrix Multiplication 165
5.5.1 Aims 166
5.5.2 Experimental Method 166
5.5.3 Matrix Multiplication Performance Results 184
5.5.4 Matrix Multiplication Conclusions 192
5.6 Parallel Compilation System 195
5.6.1 Aims 196
5.6.2 Experimental Environment 196
5.6.3 Experimental Method 196
5.6.4 Parallel Compilation System Performance Results 203
5.6.5 Parallel Compilation System Conclusions 207
5.7 Summary 208
Chapter 6 Reflections on the Vorlon Language 211
6.1 Introduction 211
6.2 Applying the Taxonomy Criteria to Vorlon 211
6.2.1 Graph 211
6.2.2 Nodes 212
6.2.3 Arcs 212
6.2.4 Node: Firing Rule 213
6.2.5 Node: Inputs and Outputs 213
6.2.6 Node: Program 213

6.2.7 Node: Output Token Production Rule 214
6.2.8 Support for Parallel Modes of Execution 214
6.2.9 Notable Omissions from the Vorlon Language 215
6.3 Software Engineering Issues from the Experiments 220
6.4 Syntactic and Semantic Improvements for Vorlon 222
6.4.1 Improvements for Current Language Syntax 222
6.4.2 Possible Future Additions to and Removals from Vorlon 224
6.5 Closing Remarks 227
Chapter 7 Conclusions and Further Thoughts 229
7.1 Overview 229
7.2 Reflections on Software Engineering Aspects 230
7.3 Vorlon Language Issues 231
7.4 Performance Results 231
7.5 Future Work 232
7.6 Closing Remarks 234
Appendix A Development of a Parallel Compilation System 235
A.1 Introduction 235
A.2 High Level Analysis 235
A.3 Application Design 237
A.4 Implementation 241
A.4.1 The Main Class 241
A.4.2 The Project Class 243
A.4.3 The Compiler Class 246
A.4.4 The SourceFile Class 248
A.4.5 The List and ListNode Classes 249
A.4.6 The String Class 253
A.5 Summary 261

References 263

Table of Figures

Figure 1-1 Moore's First Law 23

Figure 1-2 Moore's Second Law and the Origin of the "Software Stretch" 24

Figure 1-3 The Network of Workstations Architecture 26

Figure 1-4 A Typical Commodity SMP Architecture 27

Figure 1-5 A Possible SMP-NOW Architecture 28

Figure 2-1 The HeNCE Node Icons 42

Figure 2-2 HeNCE Matrix Multiplication Example 44

Figure 2-3 CODE Language Icons 51

Figure 2-4 Arc Topology Specifications Instantiating Data Parallel Activity 55

Figure 2-5 CODE Matrix Multiplication Main Graph 56

Figure 2-6 CODE Matrix Multiplication Sub-Graph 57

Figure 2-7 VPE Language Node Icons 65

Figure 2-8 VPE Matrix Multiplication 67

Figure 2-9 The ParADE Node Icons 72

Figure 2-10 ParADE Matrix Multiplication Example 76

Figure 3-1 Visual Programming Technology: A Roadmap 81

Figure 3-2 Main Syntactic Elements of Taxonomy 84

Figure 3-3 Layers of Abstraction in a Software System 103

Figure 3-4 The Orthogonal Abstraction Problem 106

Figure 3-5 Exploiting Natural Parallelism in an Algorithm 109

Figure 4-1 A Parallel Object-Flow Graph 118

Figure 4-2 Multiple Readers, Single Writer Method Invocation 120

Figure 4-3 Exploiting Parallelism in a Parallel Object-Flow Application 121

Figure 4-4 The Architecture of a Vorlon Application 123

Figure 4-5 Vorlon Arc 126

Figure 4-6 The Replicate and Merge Nodes 127
Figure 4-7 Computation Node 127
Figure 4-8 Computation Node: Graphical and Textual Views 129
Figure 4-9 Parallel Computation Node using Automatic Data Decomposition 130
Figure 4-10 Parallel Computation Node 131
Figure 4-11 Conditional Computation Node 132
Figure 4-12 Graph Start and Halt Nodes 133
Figure 4-13 Literal Source Node 134
Figure 4-14 Attribute Source Node 134
Figure 4-15 Object Interface Source Node 135
Figure 4-16 The Loop Node 135
Figure 4-17 Textual Representation of a Loop Node 137
Figure 4-18 The New Object Node 137
Figure 4-19 The Constant New Object Node 138
Figure 4-20 Method Call Node 138
Figure 4-21 Asynchrony in Textual Programming Environments 139
Figure 4-22 Task Parallelism without Explicit Asynchronicity 140
Figure 4-23 Parallel Method Call Node 142
Figure 4-24 Comment Icon 143
Figure 4-25 The Main Class 145
Figure 4-26 Vorlon Application Architecture at the Experimental Phase 147
Figure 4-27 Creating an Array of Objects 148
Figure 4-28 Setting an Array Element 148
Figure 4-29 Retrieving an Array Element 148
Figure 5-1 NIP Lazy Task Creation 154
Figure 5-2 Translation from Vorlon to Machine Code 155
Figure 5-3 An Example Class Diagram 157

Figure 5-4 Generation of NIP-Safe Types from the Vorlon Class Diagram 157

Figure 5-5 Main Method for Vehicle Simulator 159

Figure 5-6 Main Method for Vehicle Simulator (Tasklet View) 160

Figure 5-7 C++ Representation of Tasklet1 161

Figure 5-8 C++ Representation of Tasklet2 162

Figure 5-9 C++ Representation of the Main Program 164

Figure 5-10 High-Level Analysis of the Matrix Multiplication Problem 166

Figure 5-11 Composition Relation Between Matrix and Vector Types 166

Figure 5-12 Type Interfaces for the Matrix Multiplication Application 168

Figure 5-13 Matrix Multiplication Final Design 168

Figure 5-14 Vector Constructor 169

Figure 5-15 Vector Copy Constructor 170

Figure 5-16 Vector Get Method 171

Figure 5-17 Vector Set Method 171

Figure 5-18 Vector Size Method 172

Figure 5-19 Vector Multiply Method (Dot Product) 172

Figure 5-20 Vector Dot Product Loop Node Sub-Graph 173

Figure 5-21 Matrix Constructor 174

Figure 5-22 Matrix Constructor Loop Node Sub-Graph 175

Figure 5-23 Matrix Copy Constructor 176

Figure 5-24 Matrix Copy Constructor Loop Node Sub-Graph 177

Figure 5-25 Matrix Multiply Method 178

Figure 5-26 Matrix Multiply Parallel Computation Node Sub-Graph 179

Figure 5-27 Matrix-Vector Multiply Method 180

Figure 5-28 Matrix-Vector Multiply Loop Node Sub-Graph 181

Figure 5-29 Matrix Get Row Method 182

Figure 5-30 Matrix Get Column Method 182

Figure 5-31 Matrix Set Row Method 183

Figure 5-32 Matrix Set Column Method 183

Figure 5-33 Matrix Get Rows Method 183

Figure 5-34 Matrix Get Columns Method 183

Figure 5-35 Matrix Multiplication Main Method 184

Figure 5-36 Slowdown for Matrix Multiplication Application on a Four Processor SMP Platform 185

Figure 5-37 Slowdown for Matrix Multiplication Application on an Eight Node Computer Cluster 188

Figure 5-38 Slowdown for Matrix Multiplication Application on an Ad-Hoc Parallel Platform 191

Figure 5-39 Class Diagram for the Parallel Compilation System 197

Figure 5-40 Parallel Compilation System Main Method 198

Figure 5-41 Obtaining Out of Date Files for Compilation 199

Figure 5-42 Obtaining Out of Date Files for Compilation Sub-Graph Decomposition 200

Figure 5-43 The compile(…) Method from the Compiler Class 202

Figure 5-44 Speedup for Compilation System on a Four Processor SMP Platform 203

Figure 5-45 Speedup for Compilation System on an Eight Node Cluster Computer 205

Figure 5-46 Speedup for Compilation System on an Ad-Hoc Parallel Platform 206

Figure 6-1 Pipeline Reduces Potential for Parallelism 217

Figure 6-2 Race Conditions with ParADE Streaming 219

Figure 6-3 Possible Future Syntax for Conditional Execution Supporting Multiple Return Paths 223

Figure 6-4 Data Structure Decomposition to Facilitate Type-Driven Data-Parallelism 224

Figure 6-5 A Tree Data Structure 225

Figure 6-6 A Graph Data Structure 225

Figure 6-7 A List Data Structure 225

Figure 6-8 Reducing Visual Complexity by Removing Replicate Nodes..... 227

Figure A-1 Initial Class Diagram for the Parallel Compilation System..... 236

Figure A-2 Initial Design-Stage Class Diagram..... 237

Figure A-3 Final Application Design..... 240

Figure A-4 The main(…) Method..... 242

Figure A-5 getCompilerName(…) Method..... 243

Figure A-6 getCCFlags(…) Method..... 243

Figure A-7 getTarget(…) Method..... 243

Figure A-8 getOutOfDateFiles(…) Method..... 244

Figure A-9 getOutOfDateFiles(…) Method Sub-Graph..... 244

Figure A-10 The Default Constructor of the Project Type..... 245

Figure A-11 The compile(…) method of the Compiler Class..... 246

Figure A-12 The link(…) method of the Compiler Class..... 247

Figure A-13 SourceFile Class Constructor..... 248

Figure A-14 SourceFile Class Copy Constructor..... 248

Figure A-15 SourceFile Class getName(…) Method..... 249

Figure A-16 SourceFile Class getAge(…) Method..... 249

Figure A-17 List Class Default Constructor..... 249

Figure A-18 List Class addNode(…) Method..... 250

Figure A-19 List Class nodeAt(…) Method..... 251

Figure A-20 List Class nodeAt(…) Method Sub-Graph..... 252

Figure A-21 List Class noOfNodes() Method..... 252

Figure A-22 ListNode Default Constructor..... 253

Figure A-23 ListNode Copy Constructor..... 253

Figure A-24 ListNode Constructor..... 253

Figure A-25 String Class Default Constructor..... 254

Figure A-26 String Class Copy Constructor..... 254

Figure A-27 String Class Constructor 255

Figure A-28 String Class concat(String) Method 256

Figure A-29 String Class concat(char) Method 258

Figure A-30 String Class length(…) Method 259

Figure A-31 String Class c_str() Method 260

Chapter 1 Introduction

Parallel computation is a vast discipline with researchers and practitioners involved in a bewildering array of projects, unified by the common goal of achieving rapid computation. The vastness of the field and the many complex interrelations between its subdivisions means that research in parallelism often transcends several areas of work. This thesis is no exception.

It is certainly accurate to suggest that for a piece of research to be successful, it must not solely focus on one key area, but draw upon related research areas to gain depth and context. To this end, this chapter introduces the main focus of the research, visual parallel programming, and sets the context for the importance of this technique in terms of both technological and economic issues pertaining to today's and tomorrow's parallel computing world.

1.1 An Overview of Parallel Computing

As society evolves, it places an ever-increasing dependence on, and demands more power from, its computing systems. One way in which researchers are tackling the requirement for increased computing power is through the application of parallel processing. Parallel processing is the act of executing streams of instructions simultaneously with the intention that concurrent execution will decrease the amount of time taken to complete a computational task compared to the same program running on a single CPU. The ratio of decrease in computational time compared to a sequential application is known as speedup, and is fundamental to the entire discipline of parallel processing.

Though the performance benefits of parallel computation are potentially very attractive, parallel programming itself remains a notoriously difficult mode of computation to achieve. Simplistically, it seems reasonable to assume a linear relationship between the number of processors available to solve a problem, and the speedup which is attained. It is unfortunate that in practice such optimal speedups cannot be achieved. Amdahl's Law {Amdahl 1967} formalises the maximum speedup attainable as follows:

If N is the number of processors, s is the amount of time spent by a serial processor on serial parts of a program and p is the amount of time spent by a serial processor on parts of the program that can be done in parallel, speedup is given by the following:

$$Speedup = \frac{s + p}{s + \frac{p}{N}}$$

Speedup is measured as the ratio between the total time taken for the serial program to complete and the total time for the hybrid serial-parallel program to complete.

Amdahl's Law demonstrates the primary obstacle between a problem domain and the goal of a high-performance parallel application, in that certain parts of the problem may not be parallelism-amenable. Unfortunately, Amdahl's Law is not the only barrier; indeed it is only the very tip of the iceberg. Assuming that a reasonable proportion of an application is amenable to parallelism, which is not always the case, extracting and utilising that potential parallelism is often a non-trivial task.

In addition to Amdahl's Law, several other factors which complicate the task of developing parallel applications have been identified. Most significantly the tasks of co-ordinating the exchange of information between the multiple processors in a parallel machine, known as communication, and ensuring that communication occurs at the correct time, or synchronisation, are recognised as being key problem areas in the construction of parallel software. Failure to optimise the communication and synchronisation pattern between concurrent tasks will lead to poor application performance, and in the worst case render parallel applications slower than their serial equivalents.

It is not only the fact that communication and synchronisation patterns within an application are often difficult to understand, but the fact that the implementation of those patterns is often highly intricate and by necessity platform-dependent. To support the exploitation of parallel hardware, each platform provides the developer with a set of primitives which support communication and synchronisation abstractions specifically for that platform. Once an application is written for a specific platform, porting it to another platform offering different facilities may be non-trivial. Even if the application is to be ported between two broadly similar architectures, there still exists the problem of substituting one set of parallelism directives from the first architecture with those of the

second. To make matters worse, if the architectural differences are substantial, such as moving from a shared to a distributed memory platform, the entire application may have to be re-written, or in the worst case porting the application may be completely infeasible.

The problems of efficiency, portability and ease of implementation associated with programming a wide variety of parallel machines have not gone unaddressed by researchers within the parallelism community. The creation of source code which is both portable and provides efficient execution over a wide variety of parallel platforms remains an active research topic. Currently three solutions to the problem are popular {Allen 1998}:

- Automatically parallelising compilers for existing sequential languages;
- New parallel languages;
- Parallelism extensions to existing sequential languages, including parallelism libraries such as PVM/MPI.

Whilst each of the approaches provides some benefits, and ultimately provide some abstraction from the underlying parallel hardware, it is believed that even these methods do not sufficiently abstract the complexity inherent within a parallel computer, or do not exploit sufficient parallelism {Allen 1998}. The main drawback with using such methods is that the developer either has to be content with relatively simple forms of parallel activity such as those provided by parallel compilers, or explicitly program all parallel activity using language keywords or library calls. Using the compiler option may not necessarily yield an optimal parallel application, since such compilation techniques are only sufficiently advanced to extract simple data-parallel, or iterative activity. That is not to say that such tools are without merit, indeed in situations where loop or data-parallel activity is predominant, such tools would be able to achieve speedups with no further user intervention. If the problem domain contains more subtle forms of parallel activity, such as pipelined or task parallelism, then the compiler method is much less appropriate. Instead, it is likely that the developer will be charged with the identification and specification of parallelism, making use of language constructs as in the case of parallelism oriented languages, or making library calls in the case of a sequential language augmented by parallelism facilities.

The common difficulty in applying any of the current textual programming solutions to implement parallel applications lies in the fact that the linear form of the source code does not intuitively represent the non-linear parallel control-flow within an application. The significant burden of conceptualising parallel control-flow within the application is placed on the developer. Experience has demonstrated that the act of programming is itself intellectually demanding without the encumbrance of parallel control-flow and concurrency control mechanisms being added. Parallel applications development currently relies heavily upon the individual skills of the parallel programmer to manage the extra intellectual workload, and is fraught with potential pitfalls, making development slow and complicated. However, despite the inherent drawbacks to developing parallel applications, the demand for parallel software is set to increase rapidly in the coming decades, driven by increasingly performance dependent scientific and general-purpose computing applications.

1.2 Parallel Computing Drivers

Notwithstanding the difficulties associated with parallel computation, the need for very high performance computing facilities is already significant and set to increase. Members of the scientific and engineering communities have been able to absorb every performance increase that computing research has managed to win, for larger and ever more accurate numerically oriented applications. Interestingly, even desktop computer users have recently begun to require increasing hardware capabilities to run increasingly computationally intensive application and entertainment software. Whilst scientific and desktop users have historically formed distinct groups, both now have the common requirement for increased computational capacity, though they remain distinct from an economic viewpoint.

In the past the scientists and engineers have been able to increase the power of their computing systems by investing significant sums of money in specialised high-performance computing equipment. Conversely, desktop users have been supported by steady increases in the power of personal computers, and secured much reduced pricing due to market economics. Where the two groups differ is that scientific users are used to exploiting multiprocessor hardware to solve typically numerically-oriented problems with well known algorithms and requirements using parallel processing, whereas personal computer users have been using functionality-rich software on single processor systems to support a variety of tasks. Unlike the scientific community, developers of personal

computer applications have not yet embraced parallel computing technologies to facilitate a continued increase in software response and functionality.

Whilst the scientific community is implicitly prepared for the future from a technological viewpoint because parallel computing technology is already commonplace, the same cannot be said for the personal computing domain. Conversely, the personal computer users are already accustomed to utilising off-the-shelf hardware, an approach which remains uncommon in scientific computation. However, what is certain is that these two disjoint approaches must converge. Many scientific users can no longer afford to fund powerful proprietary architectures upon which to execute calculations, and by the same token no longer can the personal computer users be satisfied with a single commodity processing unit. This convergence of requirements by the two parties has set us along the path to parallel computation.

1.3 Moore's Law and the Road to Parallel Computing

Alongside the social drivers applying pressure to the computing community, economic factors have also forced a great deal of change. Perhaps the single most important economic factor currently impacting the discipline of computing is Moore's Law. In 1965, Gordon Moore made the observation that the number of transistors that could fit on a single silicon-based integrated circuit had doubled approximately every eighteen to twenty-four months {Moore 1965; Moore 1998}. Moore's First Law as it is now more commonly known, has been somewhat reinterpreted since it was first formulated and is now often cited as being the fact that computer hardware doubles in terms of performance every eighteen to twenty-four months. It has remained a remarkably accurate prediction, even some thirty years later as borne out by Figure 1-1 below:

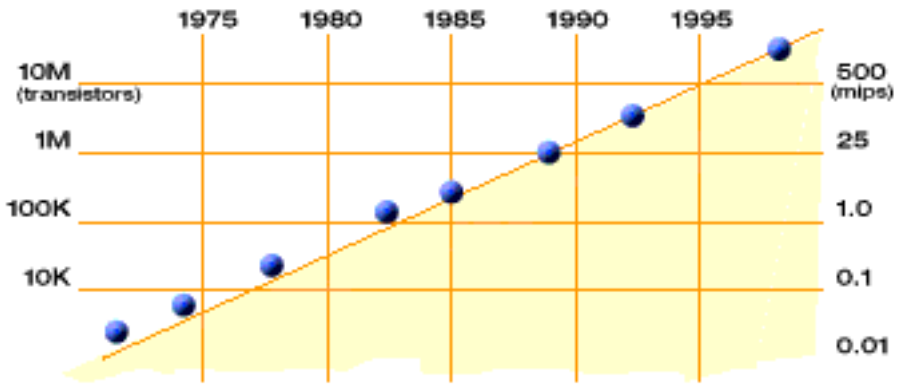


Figure 1-1 Moore's First Law

However, Moore himself has recently issued a warning as to the continued accuracy of his prediction {Moore 1997}. He notes that although there is no technical reason not to expect the First Law to hold, economic factors will eventually cause its breakdown as the capital outlay required to build each new generation of fabrication facilities increases. Moore predicts the breakdown in the First Law to occur at some point between 2010 and 2020, whereupon it will be no longer economically viable to develop and release next generation integrated circuits as frequently as the industry has come to expect. This observation has been termed Moore's Second Law {Moore 1997; Schaller 1997}, and its progress can be seen as the ogive in the figure below {Webber 1998}:

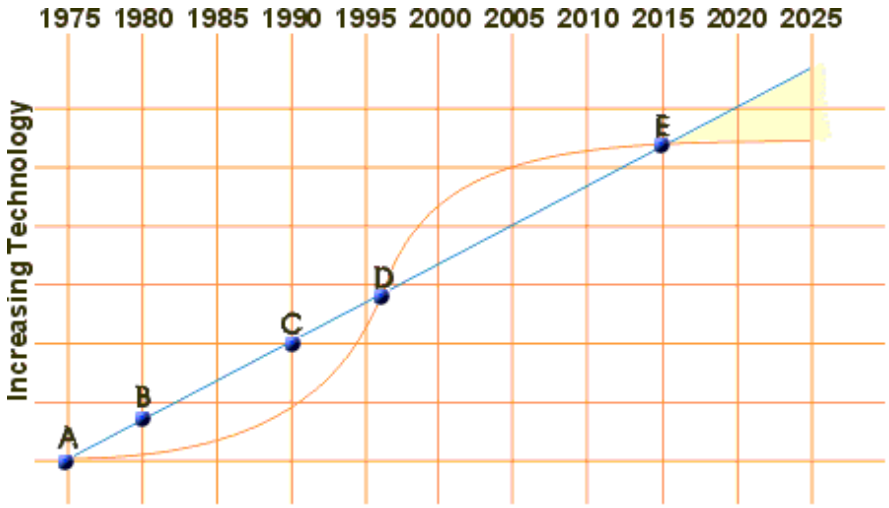


Figure 1-2 Moore's Second Law and the Origin of the "Software Stretch"

Plotted on a linear scale, as opposed to the logarithmic scale of Figure 1-1, Figure 1-2 shows the ogive-shaped curve of Moore's Second Law intersected by a second, linear relationship. The linear relation illustrates the user expectation that technology will simply improve over time, and is a far simpler relationship than Moore's: the user always requires that the functionality and response of computing systems improves – the next version of a software product should be both faster and more functionality rich. The relationship between Moore's Law and user expectation is punctuated by five points on the graph each marking an event in modern computing history:

A. Mainframe technologies are utilised in large corporations whilst minicomputers proliferate smaller companies. User expectation is low, and largely reconciled to a batch mode of operation.

B. Personal computers begin to appear and gain in popularity over the use of timeshared systems. Users begin to expect interactivity with their

computing systems. At this point, personal computer systems although interactive are woefully underpowered.

C. The first workstations with sufficient computational power to comfortably run graphical user interfaces and WIMP style applications appear. GUI-based computing becomes the normal mode, though GUI interaction is often unresponsive.

D. The raw computational power of computing hardware facilitates the production of highly functional and responsive software. For the time being, hardware power outstrips the computational requirements of all but the most demanding user group (the scientific and engineering users).

E. Moore's Second Law {Moore 1997} heralds the beginning of the "Software Stretch" {Webber 1998; Webber 2000}, where increases in hardware power become less frequent and profound. The functionality and response required by the users can no longer be supported solely in hardware. Software begins to bear the responsibility for meeting user requirements from a proportionally less powerful hardware base. In effect, the software is stretched between the falloff in computation power provided by uniprocessor hardware, and the relentless growth in expectation of the user base.

Computing has ridden the curve of Moore's First Law for some thirty years. Today, computing is in a time of plenty where, for the majority of users, the power of standard off-the-shelf hardware is more than sufficient. Nevertheless, as Moore's Second Law begins to encroach, other methods of providing increased computing power in addition to gains in raw processor speed are needed. For this, the computing community looks towards its current generation of high-performance computer users for guidance, and finds that utilising parallel processing techniques could provide the necessary performance increases as the single-processor approach begins to falter. However, the software engineering methodologies endorsed by the current parallel processing users have been low-level, and largely unsuitable for general-purpose parallel application development, and hardware costs involved have been prohibitively expensive. If the computing community is to survive the onslaught of Moore's Second Law, it will need to develop new technologies that bring the power of parallel computing into the domain of

mainstream software engineering practice, and crucially bring the cost of multiprocessor hardware further into the commodity price range.

1.4 Affordable Parallel Platforms

Access to parallel computing facilities has traditionally been the domain of universities and businesses with sufficient funding available to purchase expensive specialised parallel hardware. Even though computing hardware has increased in terms of power, whilst decreasing in terms of cost, specialist parallel hardware has remained enormously expensive. However, as technology has advanced, multiprocessor hardware has begun to move into the mainstream from two directions: downward from the expensive specialist parallel hardware, and upward from standard single-processor workstations {Sterling, Becker et al. 1998}. Perhaps most interestingly combinations of both technologies have been recently suggested.

The network of workstations, or NOW, approach to providing affordable parallelism was the first of the new affordable parallel technologies to make headway. Building upon the increasing computational abilities of single processor workstations, and the ubiquity of computer networks, it is a logical step forward to use networked workstations together to solve computationally intensive problems. In fact the architecture, as seen in **Figure 1-3** below, had already been successfully used in massively parallel computing systems. The NOW approach offers the advantage that workstations can be used to provide scaleable parallel computing facilities for computationally intensive tasks for the price of commodity hardware.

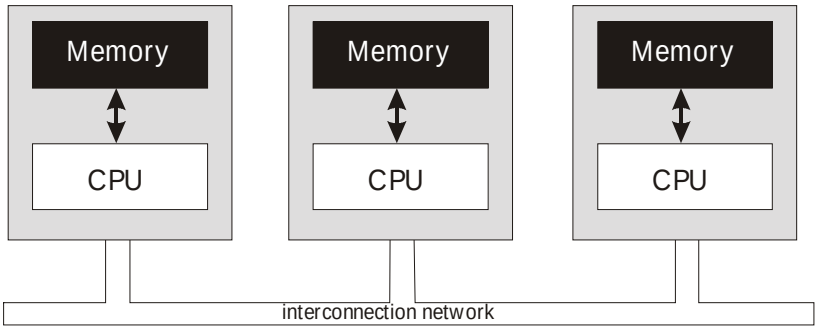


Figure 1-3 The Network of Workstations Architecture

The downside of a NOW, or for that matter any other distributed architecture, is the difficulty of the associated programming model. Each CPU in a NOW has its own local

memory which is not directly accessible to any of the other CPUs. Instead, when data is to be exchanged between concurrent processes, explicit message-passing must occur between sender and receiver. As messages travel over an interconnection network, data rates are slow and latency is high; because of this, NOWs are suited to classes of applications where the ratio between computation and communication is low. That is to say, the NOW architecture is well-suited to problems which have a naturally large *grain-size*.

A much more recent arrival to the field of affordable parallel computing is the shared memory multiprocessor, or SMP, which can be seen in Figure 1-4. SMPs began their migration from large, high-performance mainframes into low-cost server platforms, in response to the need for high-performance workgroup-level servers. As market forces push the cost of SMP hardware downward, proliferation of SMPs to the desktop is to be expected within the short term making them a viable proposition as a personal computer for executing parallel applications {Kennedy, Bender et al. 1997}.

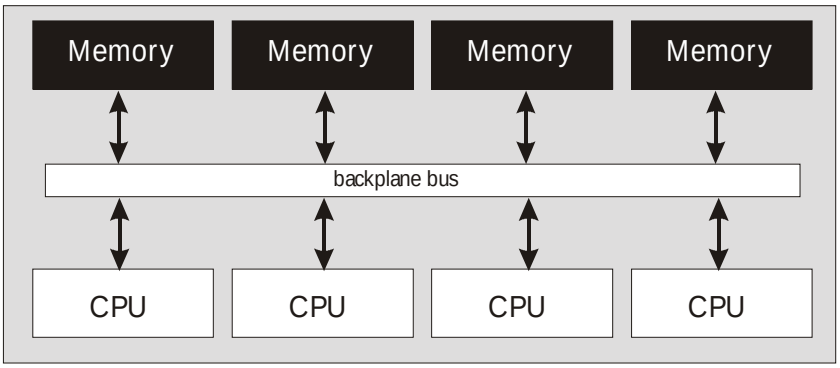


Figure 1-4 A Typical Commodity SMP Architecture

Though the SMP platform is generally limited to fewer processors than the NOW approach to parallel computing, SMPs benefit from a far simpler parallel programming model than the NOW architecture. Since all processors in an SMP share the same memory, moving data between concurrent processes is merely a matter of reading and writing between shared memory locations. Furthermore, using shared memory as a means of exchanging data between processes is not only simpler than the message-passing approach, but is also quicker because of the low latency of memory access. Using a low-latency, high-bandwidth backplane bus to support communication permits SMPs

to efficiently exploit much more finely grained parallelism than the NOW architecture permits, since the relative cost of communication within an SMP is lower. However, it is this backplane bus which limits the number of processors that an SMP can comfortably accommodate, and for problems requiring vast computational power, the SMP architecture alone may not suffice.

As SMPs make their way onto desktops and into clusters, the possibility of utilising a third affordable parallel architecture class emerges, such as that seen in Figure 1-5. In this hybrid class, networks of SMPs are used to provide the computing platform. This third approach, though rare at the moment, promises the same advantages as the NOW approach, but with potentially far higher performance.

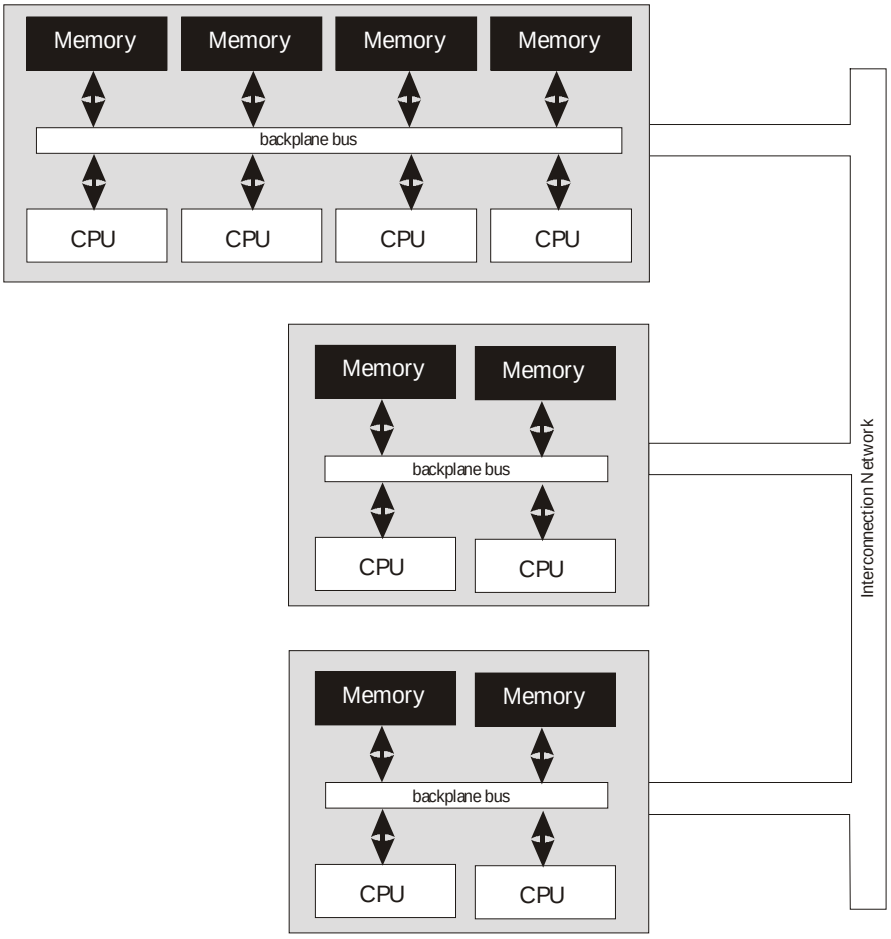


Figure 1-5 A Possible SMP-NOW Architecture

The hybrid architecture seen in Figure 1-5 is exciting in that it provides facilities for exploiting both medium-grained parallelism within individual (SMP) network nodes, and coarse-grained parallelism across network nodes. Unfortunately, the hybrid architecture

also inherits communication methods from both the standard NOW and SMP architectures, complicating programming as message-passing and shared-memory models of programming must be reconciled within a single application.

Though it is clear that there are some significant problems to be addressed before affordable parallel platforms will become mainstream computing devices, the promise of high performance computing at a significantly reduced cost compared to that of specialist machines is highly attractive. That is not to say that performance of the affordable and specialist architectures would be comparable, as that is unlikely, but the cost effectiveness of affordable parallelism is certain to be a priority of those investing in such equipment.

The effectiveness of the affordable parallel platform for parallel processing remains a function of the tool support provided to abstract away from the complexity and inefficiencies inherent within any of the three affordable architectures. There is no reason to think that affordable parallel platforms will succeed on their cost-effectiveness alone. If the true benefits of low-cost, high-performance computing are to be realised, suitable methods for developing software for such open platforms still need to be developed. The comparative level of comfort enjoyed by the software developers on proprietary parallel platforms, such as homogeneity and high-speed, low latency interconnection between processing nodes, is unlikely to be repeated by the affordable parallel platforms. Any applications development must acknowledge the complicated computing environment provided by affordable platforms, and mask their potential shortcomings.

1.5 Visual Programming Techniques

Since the advent of workstations with high-performance graphics capabilities, researchers have attempted to utilise visual techniques to improve application development. To date, a variety of graphics-based tools have been devised, from development environments for textual programming languages through to tools providing sophisticated graphical modelling facilities and pure visual programming languages.

The success of graphical application development tools and the potential held by graphics-enabled workstations has not gone unnoticed by researchers within the parallelism community. Language developers noted that the complicated control flow behaviour exhibited by parallel programs could be more eloquently expressed graphically rather than textually. This axiom continues to underpin the development of visual parallel

programming languages, and several visual language implementations based upon flow-like semantics have been developed as research prototypes. Typically, visual parallel programming languages consist of graphical objects interconnected by arcs which specify either control-flow or data dependencies between those objects. Where there are no interdependencies between objects, or a pipeline structure is present, parallel execution may occur. Thus, flow-based graphs provide a level of implicit parallelism which can be exploited without burdening the developer with the low-level programming detail.

As well as providing a means of abstracting low-level programming details, visual parallel programming languages also provide an architecture-independent programming model. Often, a language will be designed to exploit particular architecture features, or provide language constructs to optimise away potential inefficiencies. The abstract, visual nature of the languages helps to prevent them from being closely coupled to any one particular architecture, and thus aids software portability.

Nonetheless, current visual parallel programming languages do not provide the necessary level of abstraction required to build general purpose parallel software. Whilst they simplify the act of parallel programming, in that machine details can be effectively hidden from the developer, the programming models used by contemporary visual parallel programming languages support paradigms that are distinctly underpowered when considering the development of large-scale software projects. At best, the languages offer something akin to the procedural abstraction through visual black-box like mechanisms, whilst some do not even offer that level of abstraction, a topic which is explored in greater depth in Chapter 2.

If visual parallel programming is to make the leap from niche research area into mainstream software engineering practice, the programming paradigms in use must grow from being simple value-passing between processes into something more abstract and powerful. Examining modern sequential software engineering practice, it is clear that the object-oriented paradigm, amongst others, has been cited as being a good example of a high-level software paradigm. When the object-oriented paradigm is compared to the predominant procedural abstraction used in visual parallel programming, it is clear that the paradigms supported by visual programming languages are immature. If visual languages are truly to flourish, they must adopt more highly abstract paradigms which support the development of software, from inception through to release and beyond.

1.6 The Object-Oriented Paradigm

Whilst accepting that programming languages are of key importance to the process of software development, it must also be recognised that programming languages themselves are not a universal panacea. Indeed, when considering the lifecycle of any non-trivial software system, the actual implementation is but one of several important stages in the development of the application. What supports each of the stages is the underlying paradigm through which software is developed.

It is clear that today in software engineering the object-oriented paradigm has gained considerable favour amongst software practitioners. The object-oriented paradigm provides a single, coherent approach that underpins all stages of the software development process from analysis through to maintenance. Supported by a powerful and unified paradigm, modern (sequential) software has been able to increase in size and complexity and yet remains, to a large extent, well managed and maintainable.

Applying the same level of object-orientation to parallel application development has several important benefits. Object-orientation provides a powerful conceptual model with which a problem can be taken from inception through to release. All stages of the software lifecycle can be conceptualised in a completely machine-independent fashion, permitting the developer to concentrate upon getting the software correct, rather than having to be concerned about low level machine-specifics. In short, the abstraction provided by the object-paradigm manages the complexity of software, and thus allows highly ambitious, complex software projects to be managed through to completion. Furthermore, the object-paradigm provides a structured method of re-use through the inheritance mechanism. Re-use is particularly valuable when considering the increased cost of producing robust parallel code, and the potential to introduce bugs into an application when adding new routines – a problem which is exacerbated by the complicated nature of parallel code.

Object-orientation provides a further useful abstraction to the parallel application developer, in that objects themselves are a natural unit of parallelism. Traditionally, parallel programs have been written with a data-centric mindset, where the problem is viewed as a single data set which is partitioned, operated upon in parallel and recombined to recover a solution. Objects encapsulate this behaviour in a much more abstract fashion. As parallelism can be achieved through concurrent method invocation on several objects, as well as within an object's methods, the user is not forced into a low-

level parallelism-oriented view of a system, yet the potential for extracting parallel activity remains. Instead the decisions about the parallel activity are made at the design stage, where types are developed to provide problem domain abstractions and encapsulate parallel behaviour.

The object-oriented paradigm has been shown to work for serial application development, and offers some potentially attractive benefits for the parallel application developer. Although object-orientation constitutes a highly useful programming model, textual object-oriented programming does not offer assistance in managing the control-flow complexity of parallel applications. If object-orientation is to be successfully applied in producing high-performance parallel applications, programming environments which support the paradigm must also be able to abstract parallel activity. It is this hypothesis which suggests that an object-oriented visual parallel programming language could be an invaluable tool in producing software for the parallel machines of the future, and consequently is the main topic of this thesis.

1.7 Research Goals

The aim of the research is to investigate methods for the construction of parallel software and to develop new approaches for building general-purpose parallel applications, utilising parallel computing facilities composed from off-the-shelf computing and networking equipment. The objectives of the work undertaken were:

- to provide an architecture-independent programming model, and in particular enable the exploitation of parallelism from affordable parallel platforms,
- to exploit parallelism in as natural a fashion as possible, without forcing the developer into a data-centric mode of computation,
- to enable the development of general-purpose parallel applications,
- to support the non-specialist developer in producing high-performance parallel applications,
- to produce a programming language with the facilities to abstract complicated parallel activity and computation platforms using computer graphics.

1.8 Summary

This chapter has introduced some of the basic concepts and terminology of parallelism, and provided a justification for parallelism based upon Moore's Laws. The problems facing the software community because of the impact of Moore's Second law have been discussed and potentially valuable techniques for alleviating the "Software Stretch" have been identified. Furthermore, the problem of developing general-purpose applications has been described, and object-oriented development offered as a suitable paradigm through which parallel applications could be developed.

The following chapters each build upon and refine the basic concepts developed here. Chapter 2 presents a survey of visual parallel programming tools. It describes current research efforts in visual parallel programming, and discusses the range of abstractions available in contemporary visual parallel programming systems.

Continuing from the survey results of the second chapter, Chapter 3 presents a detailed analysis of visual parallel programming. The analysis consists of a discussion on execution and programming models, a taxonomy of key features of each of the languages surveyed, and proceeds to develop requirements for future languages.

Chapter 4 introduces the Parallel Object-Flow paradigm which is a novel methodology for object-oriented parallel programming, that seeks to address shortcomings in contemporary parallel programming methodologies by reinforcing them with object-orientation as a software development model. Additionally, it presents a new language for visual parallel programming, called Vorlon, which implements the Parallel Object-Flow paradigm. The syntax semantics of the Vorlon language are described alongside a discussion on the architecture of a Vorlon application.

In Chapter 5, the step-by-step development of two Vorlon applications presented. The development of each application is performed in stages from high-level analysis and design through to release, and performance characteristics of the final applications are also presented and analysed.

Evaluation and conclusions on the success of developing with Vorlon are given in Chapter 6. Problems with the language are discussed and possible solutions to those problems are offered. The overall validity of the Vorlon approach is debated with respect to the taxonomy criteria that previous visual parallel programming languages were subject

to, and a number of conclusions about software engineering aspects of Vorlon are drawn up.

Chapter 7 finalises the thesis. It considers current and likely future practice in parallel programming in light of the experience with Vorlon, and presents a number of suggestions for future work. Furthermore a discussion as to which aspects of visual parallel programming languages should be retained and which should be left out is offered before some closing remarks are presented.

Chapter 2 Visual Parallel Programming Languages

The concept of visual parallel programming languages, using computer graphics to develop parallel programs, is relatively young, stretching back no further than a decade. Even so, significant research effort has been invested in the development of graphical methods for parallel programming. However, there are other tools in contemporary computing science competing for the attribute "visual." In order to clarify the situation, it is necessary to dispel programming technologies often considered to be visually oriented (through clever brand management), and understand in detail the underlying philosophy of true visual application development. Once the core visual parallel programming issues have been separated, the task of identifying features within those technologies is simplified, and accelerates the process of building on those languages' strong points.

2.1 Introduction

This chapter begins by introducing the discipline of visual programming, covering both the fundamental principles of the field, and the impact of visual programming techniques on mainstream software engineering. The discussion then turns to the application of visual programming techniques to developing parallel applications and a survey of four earlier research projects in the field of visual parallel programming is presented. The survey presents, the development paradigm, execution model, and language syntax and semantics from each of the four surveyed languages, and an example application written in each of the languages. Drawing upon experience in developing with the surveyed languages, the chapter concludes by offering an informal critique of the languages as a whole, and identifies the need to formalise language characteristics in order to further extract useful concepts.

2.2 Visual Programming

Since computers became sufficiently powerful to support user interaction through graphical user interfaces (GUI), researchers have attempted to harness that same ease-of-use offered by GUIs to simplify the process of programming the machines themselves. To this end, a variety of graphics-based solutions have been developed, from enhanced GUI-based textual code editors through to languages with purely visual syntactic elements. The underlying principle of all of the graphics-based software engineering tools

is similar, regardless of the degree to which computer graphics has been applied: to simplify the act of building software.

Though modern (sequential) software continues to grow rapidly in terms of in size and complexity, the potential to simplify software construction should hold particular appeal to the developers of parallel applications whose software complexity far exceeds that of sequential software developers. It is hoped that if visual language techniques can truly deliver on their promise to simplify application development, parallel programmers may reap the rewards. However, it is by no means certain that current visual software engineering tools, including visual languages, are sufficiently well developed to deliver on that promise.

2.2.1 Visual Development Environments

More so than any point in the past, the adjective "visual" adorns a multitude of software engineering tools. In many cases the prefix does not refer to using a graphical syntax with which to program, but instead to the fact that software projects can be managed using graphical tools, and cosmetic details such as the appearance of an application's user interface can be constructed in a graphical environment.

Experience has demonstrated that graphics-enhanced software engineering tools provide significant productivity gains over their textual counterparts. The richness of information conveyed by good use of computer graphics is invaluable in giving the developer a better understanding of the construction of the application. Whilst it is true that there are few fundamental differences in methodology between the techniques used in a purely text-driven environment and the graphics-enabled versions, graphical tools are often simply easier to use. Even integrating several traditional tools behind a single consistent graphical interface, such as integrating a compiler and debugger with a source code editor, provides a far richer programming environment than the sum of the individual parts.

Though valuable, graphics-enhanced development environments do not themselves constitute a complete solution to the problems of producing high-quality software systems. Indeed, such environments are merely tools in the software engineer's toolkit, and whilst software engineering tools are certainly of a higher quality today than at any other time in the history of software, tools alone are only one aspect of a complete software engineering solution. What is actually required is a combination of a paradigm

through which to conceptualise a problem domain, and high-quality software engineering tools and techniques which may draw upon computer graphics for support.

Even with considerable support in terms of paradigms and software engineering tools, the fact remains that the level of abstraction offered by textual programming languages is low. Whilst modern textual languages support paradigms which are themselves highly abstract, actual source code remains at a low level of abstraction. Software engineering paradigms have evolved over the years, and now endorse highly abstract software models and programming languages, and are supported by sophisticated visually enhanced tools, and yet have remained at the same low-level of syntactic abstraction since their inception.

Realising that attaining a further level of abstraction at the source code level may prove to be valuable, researchers have begun to look past merely wrapping textual programming tools in graphical support, and begun to focus considerable effort on applying graphics to software development from the ground up. As the power of workstations has increased, the software community has begun to realise that the application of visual programming techniques may no longer be a pipe dream.

2.2.2 Visual Programming Languages and Techniques

Like graphics-enhanced textual programming environments, visual programming languages and their graphical environments also constitute one part of some larger development methodology. Unlike textual programming, visual programming offers abstraction not only through the underlying paradigm, but also at the source code level where previously, in textual programming there was little support.

The field of visual programming is underpinned by the axiom that in some cases, an iconic representation of program behaviour is better than a textual representation, though the extent to which this concept is applied varies between languages. At opposite ends of the spectrum, there exist languages whose syntax is purely visual, and languages which mix visual syntax with textual. Visual languages themselves may be further classified into general-purpose and application-specific forms, where the general-purpose languages tend to be of the hybrid style, whilst application-specific languages tend towards being more purely visual. The choice of one style of visual language over another thus depends upon the target application area and the availability of a suitable visual programming language.

The great advantage of visual programming is the additional level of abstraction that it offers. Visual programming is one way in which low-level primitives can be abstracted

away from the developer, which in turn frees the developer to concentrate upon higher concerns such as ensuring the correctness of the application rather than investing effort in low-level programming detail.

Unfortunately, though they brim with potential, visual programming techniques are still rather immature. There is a wide variance amongst visual languages which suggests that the field has not settled on any particular style, and opinion on their overall value to the field of software engineering remains diverse {Whitley 1997}.

Concerns over the migration from textual to visual programming are similar to that observed during the migration from assembly code programming to textual high-level languages. Developers are comfortable with current programming methods, and are often reluctant to experiment with what amounts to an unknown quantity, especially since current textual programming techniques appear to be delivering.

Having acknowledged that excellent results are being produced by software engineers world-wide, the higher level of abstraction offered by visual programming techniques may be considered superfluous to the current needs of developers. Though the complexity of the software may be high, the ability to manage complexity is well within the ability of text-based programming given a suitably simple execution platform such as a uniprocessor computer. However, when developing applications for parallel computing platforms, it is apparent that textual programming can be extremely difficult.

2.2.3 Visual Parallel Programming

The field of visual parallel programming seeks to address the problems traditionally associated with developing software for multiprocessor computers, namely the enormous complexity of building parallel applications. The rationale behind visual parallel programming is simple: parallel applications exhibit high levels of control-flow complexity; that complexity in turn can be managed through abstraction, and visual programming techniques provide a more abstract language syntax than textual programming.

The goal of visual parallel programming languages is to encapsulate low-level concurrency control primitives and abstract architectural details using a visual syntax, whilst ensuring that run-time efficiency of the parallel application is not seriously compromised. Once low-level concerns such as concurrency control and machine architecture are removed from the developer's workload, and efficiency issues addressed automatically by the language or compilation system, development effort can be focussed

ensuring that the application meets its specification. This in itself is a major step forward for the parallelism community who, until now, have spent a great deal of effort in dealing with precisely such issues and which certainly are not primary problem domain considerations. Free from the burden of low-level detail, developers should be able to target increasingly ambitious software projects which need the performance increases offered by a parallel mode of computation.

It is not therefore unreasonable to assume that applying visual programming techniques to parallel programming may yield programming techniques which support highly abstract, machine independent methods for developing general-purpose parallel applications. However, as with mainstream visual programming technology, visual parallel programming may not yet have reached that goal. Indeed, it is becoming increasingly clear that current visual parallel programming systems have a significant amount of ground to cover before general-purpose parallel application development becomes possible.

2.3 A Survey of Visual Parallel Programming Tools

This section introduces four visual parallel programming tools from recent research projects. For each tool this section presents: the authors abstract; a discussion of the programming paradigm used; and a review of the visual syntax. Furthermore, a simple parallelism-amenable problem, matrix multiplication, is completed in each of the languages and a discussion on the experience of developing with each is presented.

2.3.1 HeNCE – Heterogeneous Network Computing Environment

> HeNCE, Heterogeneous Network Computing Environment, is a graphical parallel programming environment. HeNCE provides an easy to use interface for creating, compiling, executing and debugging parallel programs which run under the PVM system. HeNCE programs can be run on a heterogeneous network of UNIX workstations.{Beguelin, Dongarra et al. 1994}

2.3.1.1. HeNCE Language and Paradigm

The HeNCE development paradigm takes the view that a parallel application is composed from a set of computational tasks related by a set of control-flow interdependencies. In HeNCE, both computational tasks and control-flow dependencies are represented graphically, where computation takes the form of nodes and control-flow dependencies take the form of arcs connecting nodes on a graph. In addition, HeNCE

provides a number of special node types, which in combination with the graph arcs, direct control-flow within an application.

Parallelism, essentially being a sophisticated form of program control-flow, is also managed graphically by HeNCE. Implicit parallelism is available wherever there exist no control-flow dependencies between computations, allowing any task-parallelism within an application to be exploited without explicit programmer intervention. Conversely, parallelism may be explicitly induced in HeNCE applications using the data-parallel and pipelining abstractions provided. The crux of the matter is that whether parallelism is implicit within the structure of an application, or explicitly invoked by the developer, no system-level features are exposed to the developer – all parallel activity is expressed in a highly abstract graphical form.

In addition to the graphical specification of parallel activity, HeNCE also provides a comfortable level of abstraction from the underlying parallel machine, in that the low-level concurrency issues such as locking and synchronisation are implicit within the structure of a HeNCE graph. Although nodes in a HeNCE program operate in a synchronous fashion, the overall execution of a HeNCE program is asynchronous, in that there is no global timing mechanism which governs execution – the only condition for the execution of a node is the completion of its predecessors. The consequence of this mode of operation is that execution can only be delayed by latent communication. The determinism of the program cannot be compromised by the unpredictable behaviour of the underlying network architecture. Thus, by applying graphical abstraction to the implementation of parallel applications, HeNCE affords developers the luxury of being able to concentrate on higher-level aspects of the program control-flow structure, rather than low-level locking and synchronisation issues.

Unlike the control-flow structure of a HeNCE application, computational tasks are in fact expressed textually. Within each of the computational nodes on a HeNCE graph, the developer embeds a call to an external program written in either C or Fortran.

Bindings between graphical and textual objects in HeNCE are managed by an entity known as the node program. The node program is a piece of textual source code written in a HeNCE-proprietary language, which forms an interface between the textual and graphical aspects of an application. Since HeNCE arcs provide only control-flow signals, the node program also bears the responsibility for obtaining any data which the node requires in order to carry out its computational requirements. This data is kept in a

globally accessible space, and in order for a node to use a piece of that data, its node program must be specified to obtain it.

At runtime, the mapping between the logical parallelism inherent within a HeNCE application, and the physical parallelism offered by the underlying network of workstations architecture is controlled by a mechanism known as the cost matrix, and allows HeNCE applications to take advantage of the underlying network-centric parallel architecture. The cost matrix is a static data structure which the developer specifies at compile-time, containing the relative costs of running each computation in the application on each of the available processors, or even architecture classes, in the underlying network multiprocessor. Using knowledge of the likely power and loading of nodes on the network, the developer can bias an application in favour of more powerful architecture types, or less heavily loaded hosts. A further burden managed by the HeNCE system is that for each of the architecture classes that appear in the cost matrix, HeNCE assumes the responsibility for compiling and distributing binaries of the user-specified textual computations to each workstation in the cluster.

2.3.1.2. The HeNCE Language Constructs

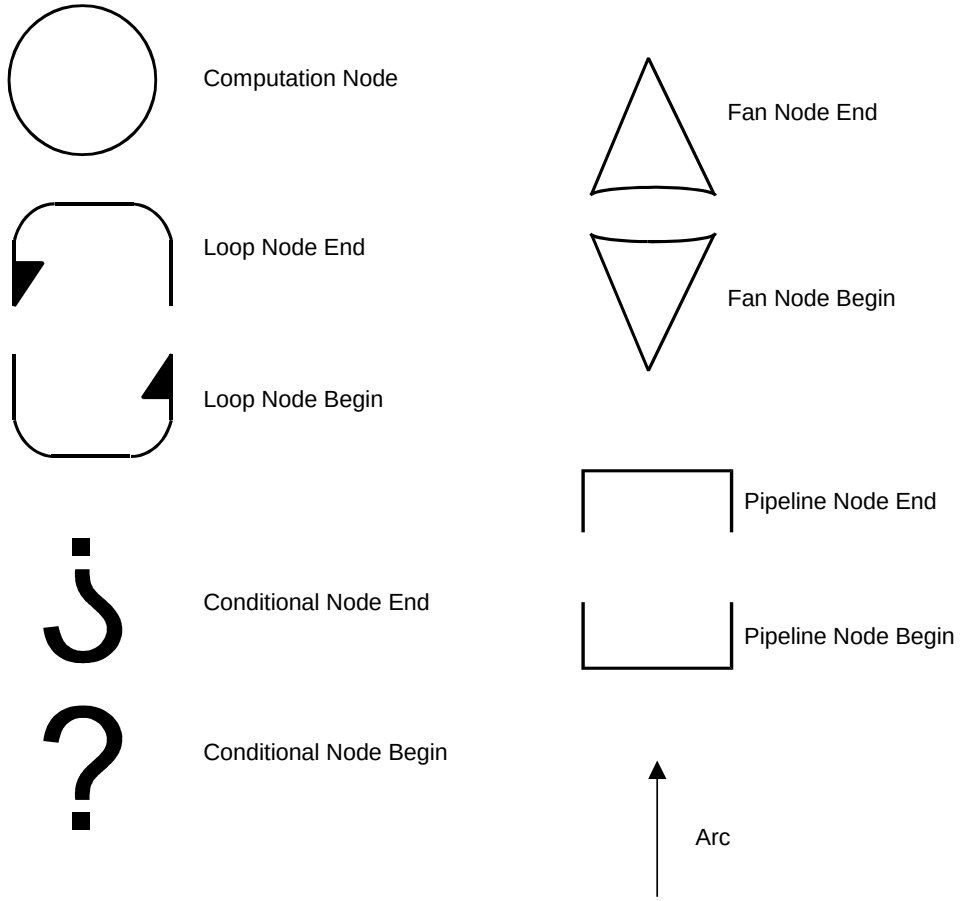


Figure 2-1 The HeNCE Node Icons

2.3.1.3. Arc

The arc in HeNCE describes a control-flow dependency between two nodes, where the node at the tail of an arc must have completed its execution before the node at the head of the arc can begin. Note that in HeNCE, graphs are read bottom-upwards which is why an arc's head is (somewhat counter intuitively) above its tail.

2.3.1.4. The Computation Node

The computation node is the fundamental building block of HeNCE programs. Each computation node contains a node program, which handles logistical aspects of the node's execution, and the name of an external procedure written in C or Fortran to execute when the node becomes active. The computation node is unique in HeNCE, since it is the only node type that is not specifically meant as a control-flow modifier.

2.3.1.5. The Loop Begin and Node End Nodes

The loop begin and end nodes define a section of iteration within a HeNCE graph, much like a `for` loop in the C programming language. The computations declared in the nodes between the loop begin and loop end nodes are repeated whilst the condition in the loop begin node specification holds true. It is perhaps noteworthy that HeNCE does not provide a parallel loop execution strategy such as that offered by the HPF programming language {Koelbel, Loveman et al. 1994}. Instead, HeNCE looping is used purely for iteration – all parallel activity is initiated through fan or pipeline nodes, or is implicit within the control-flow structure of the program graph itself.

2.3.1.6. The Conditional Begin and Conditional End Nodes

The conditional begin and conditional end nodes provide an if-then clause in the control flow of the graphs. If the condition specified within the specification of the conditional begin node evaluates to true, the graph elements contained within the begin-end node pair are executed, otherwise control flow moves directly to the successor(s) of the conditional end node.

2.3.1.7. The Fan Nodes

Pairs of fan nodes are used to delimit sections of data-parallel activity in a HeNCE graph. Copies of the nodes contained within fan node begin/end pairs are replicated and executed in parallel, with the intention that each set of parallel computations will manipulate a different set of input data. The overall partitioning strategy for the input data is determined by the node specification of a fan begin node, which may be determined either at compile-time or by runtime conditions.

2.3.1.8. The Pipeline Nodes

Pipeline nodes offer the second form of explicit parallel activity in the HeNCE language. A pipeline in HeNCE retains is common-use meaning, whereby data at each stage in the pipeline can be operated upon in parallel, whilst maintaining a particular sequential ordering. The number of steps, and thus the level of parallelism, in a pipeline is determined by the number of nodes in the area of the graph delimited by the pipeline begin and end nodes.

2.3.1.9. HeNCE Matrix Multiplication Example

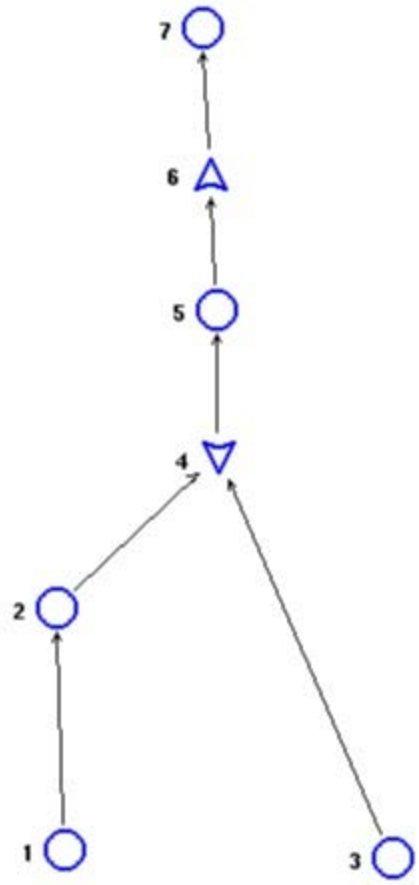


Figure 2-2 HeNCE Matrix Multiplication Example

The diagram in Figure 2-2 shows the HeNCE graph for a very simple, parallel, matrix multiplication algorithm, as follows:

- Two matrices are created.
- One of the matrices is split into a series of row vectors.
- Each row vector is multiplied with a copy of the second matrix in parallel.
- The final matrix is assembled from the results of the matrix-vector multiplication.

It should be noted that the algorithm employed is very crude. Firstly, the algorithm will fail if the matrices to be multiplied are not compatible. Secondly maximal parallelism (the parallel multiplication of individual matrix elements) has not been chosen for reasons of efficiency as the very fine granularity of multiplying two single matrix elements is difficult to exploit efficiently within the network of workstations architecture,

which is better suited to the exploitation of coarse-grained parallelism. With this in mind, the appearance of Figure 2-2 can be explained:

1. One of the matrices is created, and passed to a successor for splitting into its component vectors.
2. The same matrix, consisting of an array of floating point values is split into several smaller arrays, each representing one row in the original matrix.
3. A second matrix is created in parallel with the first and is made available to its successor node.
4. The fan out node distributes a copy of a whole matrix along with one unique row vector to each of its successor nodes.
5. The data arrives at the matrix-vector multiplication node for processing in parallel. Multiple copies are executed in parallel by dint of the fact that the node which carries out the operation is contained within a fan begin and fan end node pair. Each of the replicated nodes performs a matrix-vector multiplication and produces a single row vector along with a row identifier, which is passed to its successor.
6. The end fan out node collects the vectors produced by the matrix-vector multiplication node and passes them to its successor.
7. The vectors produced by the end fan out node are assembled into a matrix, using their associated row identifiers to ensure correct placement, and the result is output.

Whilst the graph clearly demonstrates the (parallel) structure of the program, it should be noted that the graph alone does constitute the entirety of the program. In addition to the construction of the graph, each of the nodes needs to be equipped with a node program, and the computations themselves need to be written.

2.3.1.10. HeNCE Observations

The originators of HeNCE suggest that it is non-exclusively geared to problems within the domain of computational science. HeNCE is intended to abstract away details

of underlying architectures, libraries and run-time systems thus alleviating the intellectual burden of handling low-level parallelism detail from the developer.

Insofar as its aims reach, HeNCE is a successful language in that it allows for problems which are currently solved using low-level parallel programming to be solved at a higher level of abstraction from the underlying parallel architecture. However, the goals set by the HeNCE research project were somewhat lacking in ambition by today's standards. That is not to say that HeNCE is a poor language, as it is not, but within the era from which HeNCE derives there was little to suggest that parallelism would be a tool for anything other than large, computational problems. As a result, HeNCE adopts the same low-level problem solving paradigm as any textual parallel-programming strategy from the same period, whereby problems are conceptualised as sequential processes, exchanging data from time to time, as typified by Hoare's CSP language {Hoare 1985}. Such an approach necessarily presents the developer with a cognitive model close to that of the computing system (distributed memory, and message-passing). Whilst it is true that HeNCE abstracts the lowest level of detail, the locking and message-passing primitives and so forth, the lack of a suitably highly abstract development paradigm to support HeNCE is its biggest downfall. Relying on a paradigm which itself in remarkably similar to the underlying parallel system seems almost self-defeating when the goal of HeNCE was to abstract that same system and manage implementation complexity. That being said, HeNCE is quite a suitable tool for expressing parallel applications whose structure is straightforward enough not to require highly abstract cognitive models. Indeed, where the communicating sequential processes view of a parallel application is sufficient to enable its construction, HeNCE is a valuable tool, allowing developers to concentrate on extracting as much parallelism as possible from the structure of the program, without the intellectual load of managing low-level detail.

On a positive note, HeNCE supports an important aspect of parallel programming, in that it separates control-flow and computation. Whilst in imperative sequential languages it has been accepted that computation and control flow can exist within the same body of source code and even using the same syntax, in a parallel application the control flow may exhibit far more complicated behaviour than its sequential counterpart. In a HeNCE program, the parallel behaviour of the application does not pollute the actual source code for computations. As control-flow and computation are somewhat orthogonal issues, the complexity of both can be managed individually.

Where parallel control-flow problems are concerned, HeNCE receives a mixed response. On the positive side, the ability to exploit parallelism implicit within the structure of the graphs is a great bonus, in that tasks which are mutually exclusive are free to execute in parallel without explicit intervention by the user. On the negative side, pipelined parallelism must be explicitly induced since HeNCE has a control-flow execution model, where under normal circumstances all a nodes predecessors must have completed before subsequent nodes may execute. The HeNCE pipeline nodes allow this model to be circumvented, permitting successor nodes to execute whilst their predecessors are still themselves computing, but only within the graph area delimited by pipeline begin/end node pairs. To the graph containing the section of pipelined activity, the image of a strict control-flow model is maintained. However, in other languages pipelined parallelism is claimed to be implicit within the structure of the application, and as such provides the facility at no cost to the developer. Data-parallel activity in HeNCE is once again explicitly induced, though here the penalty on the developer is not significant, since HeNCE's target computational problems (numerical analysis) often involve a great deal of data-parallelism, and it is often the case that data-parallelism is obvious for the developer to identify. Thus explicitly activating data-parallel activity in a HeNCE program is perhaps not an unreasonable burden for the developer, for whom the identification of data-parallelism is trivial, and for whom the implementation of data-parallel activity is simplified through visual abstractions.

Having said that HeNCE satisfies its goals comfortably, it does not satisfy them completely. HeNCE applications should, in theory, be portable across any architecture supporting the PVM {Geist, Beguelin et al. 1994} message-passing system, as HeNCE ultimately produces C/Fortran code containing PVM calls. In practice, things turn out to be a little more complicated. When a developer builds a HeNCE application, that application will run on the developer's own target architecture. However, just because a set of workstations supports the execution of PVM tasks, this does not mean that all PVM clusters are architecturally identical, and in fact the reality is often far from that. There is little chance that an application written with one PVM cluster in mind will execute with the same performance characteristic on another cluster. At the very least, a new cost matrix will have to be constructed for each PVM cluster that a HeNCE application is ported to, at worst applications may need considerable changes in order to run efficiently. However, the reduced portability of HeNCE applications is not a function of the language syntax itself. Theoretically, HeNCE programs, because of their

graphical construction, could be made portable across a range of parallel architectures. It is the fact that HeNCE targets the PVM run-time system as its target architecture which, ironically, reduces its portability.

Though the language is to be considered blameless for the fact that HeNCE applications are somewhat less portable than its name suggests, the language is to blame for introducing implementation work outside the scope of the problem domain. Moreover, the fact that each of the nodes in a HeNCE program requires a specification, which governs the control-flow logistics and bindings between text-level and graph-level objects, is detrimental. The addition of a third language component provides another route via which bugs could be introduced into programs, and although the language used to program each of the node specifications is simple, it is nevertheless an aspect outside the problem domain that a developer must contend with.

A further shortcoming of the HeNCE language is the lack of facilities for source code management. Though HeNCE may seem amenable to and could benefit from some graphical function-call mechanism or similar black-box mechanism, it has not been equipped with such {Browne, Hyder et al. 1995}. If a particular application consists of many nodes then the HeNCE graph representing that program may become very large, and ultimately become so dense with information that continuing development becomes difficult. It is perhaps an obvious deficiency that even well known control-flow mechanisms, such as the procedure call abstraction, are missing.

On a more positive note, HeNCE is supported by a useful post-mortem program animation facility as part of the development environment. The animation facility is useful in debugging and optimisation, providing an insight into both the activities of the program and the run-time system. From this, not only the high-level information about the program can be obtained, but also the low-level machine details are available, thus revealing both programming errors, and optimisations with respect to utilising the underlying hardware more effectively. The downside of this information is that the programmer must possess sufficient skill to make the optimisations required by the architecture, which itself may be non-trivial.

Though far from perfect, HeNCE provides a useful starting point into the field of visual parallel programming. Its virtues and weaknesses have been observed by contemporary projects, and armed with this knowledge other researchers have attempted to build languages which address some of HeNCE's shortcomings.

2.3.2 CODE – Computationally Oriented Display Environment

> The conceptual approach we have taken to providing a solution for the problem of programming MIMD parallel architectures is based upon raising the level of abstraction at which parallel structures are expressed and moving to a compositional approach to programming. The CODE 2.0 model of parallel programming permits parallel programs to be created by composing basic units of computation and defining relationships among them. It expresses the communication and synchronisation relationships of computation as abstract dependencies and runtime determined communications structures can be expressed.{Newton 1993}

2.3.2.1. CODE Paradigm

The CODE model of parallel computation views a parallel application as a set of potentially parallel computational tasks whose execution ordering is determined by their natural data dependencies. In CODE, the structure of an application is described visually in terms of a set of computational elements and the data dependencies that exist between those computational elements. Computations, though themselves expressed textually in the C programming language, are visualised as nodes on a graph and data dependencies as arcs interconnecting those computational nodes.

The CODE model of computation is interesting in that there exist no nodes which specifically deal with parallel control-flow aspects of an application. This has some profound implications for the way in which execution proceeds in CODE, in that the responsibility for routing data, and therefore overall control-flow in the application, rests with the Unit of Computation (UC) nodes (the basic computational element in CODE) and arcs. The way in which UC nodes manage such logistical aspects is through "node stanzas," which describe a node's interface in terms of number and type of input and output arcs. The node stanzas themselves can be made to co-operate with arc topology specifications, which are equivalent to node stanzas but specify arc behaviour, such that data from one node can be transmitted via an appropriate arc to an instance of another node.

The upshot of this mode of execution is that data-parallelism must be explicitly induced by the programmer, who must write a consistent set of node stanzas and arc topology specifications to manage the distribution and recombination of data across the set of nodes involved in the data-parallel activity. On a more positive note, other forms

of parallelism (pipelining and task) are implicit within the structure of the application and occur as and when the natural data dependencies of the application allow.

CODE retains a reasonably comfortable level of abstraction from the underlying parallel architecture with issues such timing and synchronisation being automatically managed. Additionally, CODE augments the dataflow programming paradigm by the inclusion of a subgraph-level shared memory abstraction. The shared memory mechanisms offered are valuable in that they support the construction of applications which are not easily modelled by value-passing. As with other low-level concurrency control mechanisms, access control to shared resources is handled automatically by CODE.

The problem of mapping such high-level abstractions into efficiently executing parallel programs has been central to the CODE model of computation. CODE tries to balance the developer's requirements of high levels of abstraction with the execution requirements of optimised machine code by using a proprietary translator. This incurs an overhead when compared to targeting a general-purpose run-time system such as PVM in that for each parallel architecture that CODE programs are to be executed upon, a new translator must be developed. However, the advantage of developing translators on a per-architecture basis is that features indigenous to particular architectures can be exploited to maximise the efficiency of program execution. Thus CODE allows programmers to enjoy the relative luxury of developing in an abstract, graphical environment, whilst, uniquely amongst the surveyed tools, offering the capability of producing platform-optimised executable programs.

2.3.2.2. The CODE Language Constructs

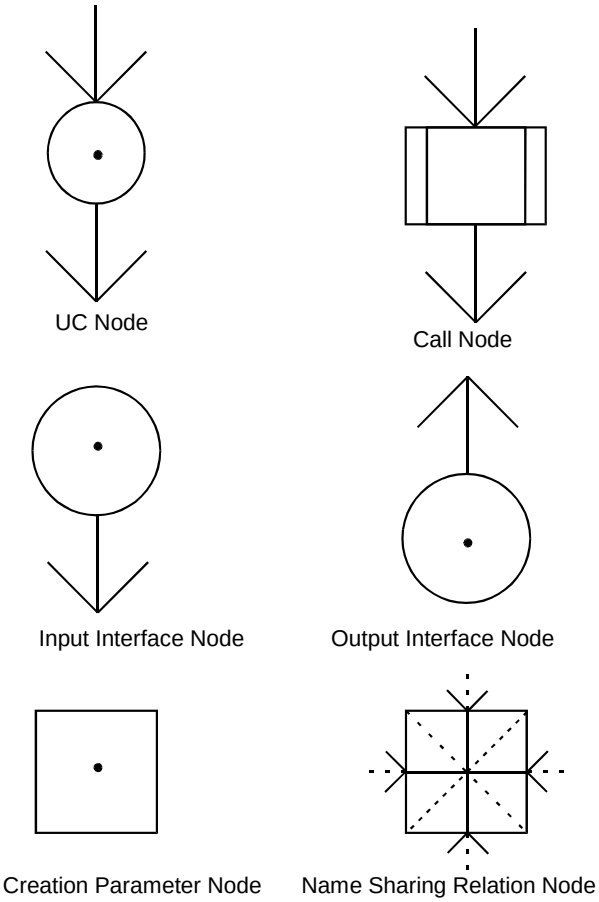


Figure 2-3 CODE Language Icons

2.3.2.3. UC (Unit of Computation) Node

The Unit of Computation, or simply UC, node is the fundamental building block within the graphical component of the CODE language. UC nodes represent sequential computations which may potentially execute in parallel, data dependencies permitting, promoting the exploitation of parallelism at the sub-routine level {Newton 1993}.

As is the case with other node types in CODE, each UC node within a graph is explicitly programmed with a set of node stanzas using a declarative style textual programming language. Here, the node stanzas are responsible for determining under which conditions the node's computation may execute (selecting which of the input ports containing data will invoke the node through the firing stanza), where the results of the computation will be sent, and binding graph-level constructs to local variables. The set of stanzas may also contain a computation written in a subset of the C programming language, or alternatively contain a procedure call to an external C program where the full expressiveness of the C language is required to implement the computation. It is

noteworthy that nodes are stateful, in that local variables defined in nodes retain their values between each activation of the node, with similar semantics to static variables in the C language. If required, the initial state of variables local to a node can be set before a graph is executed via the inclusion of an appropriate clause within that node's stanza.

2.3.2.4. Call Node

A typical CODE program may be composed of a number of intercommunicating graphs, arranged hierarchically. The call node is used as an interface between the graphs from which an application is composed. Graph calls made via a call node exhibit identical semantics to any other node in a CODE program. Thus no special measures need to be taken, even though activating a call node may cause the execution of an entire sub-program. The call node is analogous to a function call in sequential programming languages, and as such can be used in a similar fashion, including the ability to make recursive calls to the same graph.

Using call nodes and sub-graphing allows applications to be constructed in a structured fashion from components. Provided the interface to each of the components is known and adhered to, there is no reason why library components, and components from other developers or projects cannot be re-used with no change to the overall semantics of the execution model. Furthermore, a procedure-call abstraction mechanism can be used as a tool for managing software complexity and allowing top-down decomposition of a problem domain, which is valuable.

2.3.2.5. Input and Output Interface Nodes

Input and output interface nodes only appear when an application is composed from multiple CODE graphs. The function of input and output interface nodes is to provide a binding between arcs from a calling graph into the called sub-graph and back again. At the sub-graph level the interface nodes are visualised simply as local data sources and sinks. When a sub-graph is called through a call node, values pass from the input interface node into the main body of the graph where the computation occurs. Similarly, the output interface nodes provide a means of returning values from a sub-graph to outgoing arcs on the calling graph's call node.

The benefit provided by the use of input and output interface nodes is that components of an application can be developed independently, with the set of input and output nodes providing each component with its own environment. This in turn

decreases coupling between nodes and as such increases the scope for potential re-use of an application's code base.

2.3.2.6. Creation Parameter Node

The creation parameter node is a shared-memory abstraction whose purpose is to allow CODE graphs to be parameterised at their instantiation. The values held in the creation parameter nodes act as read-only variables within the scope of the graph within which they are declared. Though each creation parameter node must be bound to a dataflow arc from the calling graph, only the first value which arrives along that arc will be retained and shared by the rest of the graph. Any subsequent values arriving along that arc will be ignored by the creation parameter node, in keeping with its read-only semantic.

Within sub-graphs, creation parameter nodes do not obey the normal node-connected-by-arc syntax which characterises the normal semantic. Instead, the placement of a creation parameter node on a CODE graph implicitly connects each node in the graph to that creation parameter node. Whilst this contravenes the normal semantics of CODE graphs, this approach greatly improves graph readability, reducing the need for webs of what amounts to trivial arc interconnections on the graph.

2.3.2.7. Name Sharing Relation Node

The name sharing relation node provides a fully mutable shared variable facility. Unlike the creation parameter node, the name sharing relation node holds a value which may be read from or written to at any point during a graph's execution, applying the limitation that nodes must have declared their possible intentions as readers or writers (via the node stanza) before accessing the shared value. Concurrency issues pertaining to read, write and read-write access to name sharing relations are automatically resolved by CODE, freeing developers from the burden of explicitly programming concurrency control mechanisms.

Name sharing relation nodes, in common with other node types in CODE, require a set of stanzas that govern their run-time behaviour. In particular, the name sharing relation nodes contain the following stanzas, which differ from the standard UC node stanza set:

- A name for the node.
- The name and type of the shared variables.

- An (optional) activation computation which will activate before the node begins to share its shared variables. When instantiated, the name sharing relation nodes can perform useful computation which may affect the initial state of the shared variables which it contains.
- A set of external function signatures that may be called by the node's activation computation if any activation computation exists.
- A set of local variables to be used in the activation computation.

It is noteworthy that the functions defined with the node are available to the activation computation, but they themselves are not shared, for example in the same way that an object's methods would be visible to other objects in an object-oriented application. That is to say, the name sharing relation can operate upon the data that it exports, but no other nodes can coerce the name sharing relation into repeating those operations. Once complete, the name sharing relation becomes a standard shared value abstraction, which can be read and updated by other nodes in the graph.

2.3.2.8. Arcs

Arcs in the CODE language specify potential paths of communication between nodes along which values may pass. Each arc is essentially an unbounded FIFO queue, where the head of the queue is accessible to the receiver of the values and the tail accessible to the sender.

Whilst this would seem a suitably simple form for a node to take, arcs, in common with the other graphical CODE components, must be annotated. Specifically arcs are annotated with arc topology specifications, which map an output port from a source UC node to an input port on a destination UC node.

The reason why CODE requires arc topology rules for each arc is that they govern the run-time mapping between nodes and arcs, and thus any parallel activity within a graph. For example, using indices associated with each arc, values can be passed to multiple instantiations of UC nodes which can then operate on data in parallel which can be seen in Figure 2-4 below.

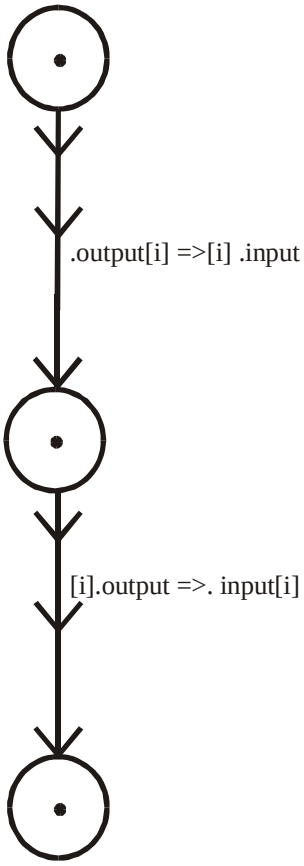


Figure 2-4 Arc Topology Specifications Instantiating Data Parallel Activity

In Figure 2-4, a CODE template for simple data-parallel activity is show. The UC nodes uppermost and lowermost are each instantiated once at run-time, whilst the middle UC node may be instantiated any number of times in parallel. The way that the CODE system knows when to instantiate a node multiple times is, as has been previously noted, through the arc topology specification. Here the first arc topology specification, `.output[i] => [i].input`, specifies that the uppermost node in fact has a choice of `[i]` arcs onto which it may deposit data, and that the middle node is therefore replicated `i` times. The second arc topology specification, between the middle and lowermost node, performs a similar operation except that in this particular case, `[i].output => .input[i]` maps arcs between multiple instances of the middle node with a single instance of the terminal node.

As is apparent, arc annotation is tightly coupled to the behaviour of both the sender and receiver. All must agree on the correct behaviour if the overall program is to function as expected, and presents a most opportune situation for the introduction of bugs into an application. As such, the developer must be cautious when constructing the

arc topology specification, or the behaviour of the final application may be somewhat different to that intended.

2.3.2.9. CODE Matrix Multiplication Example

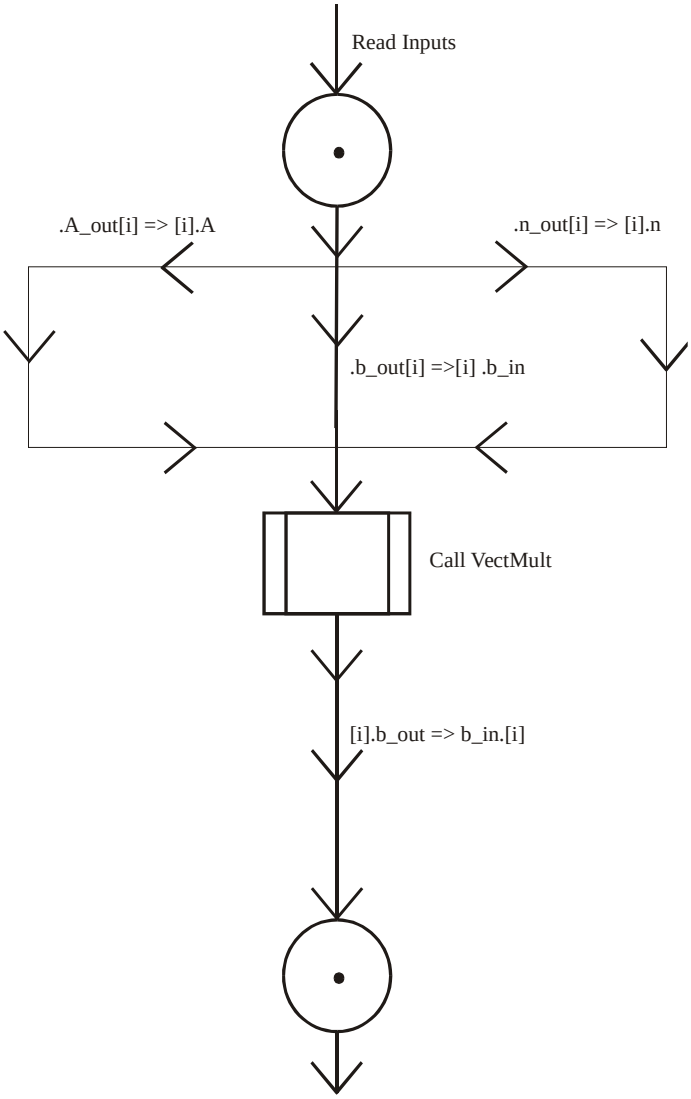


Figure 2-5 CODE Matrix Multiplication Main Graph

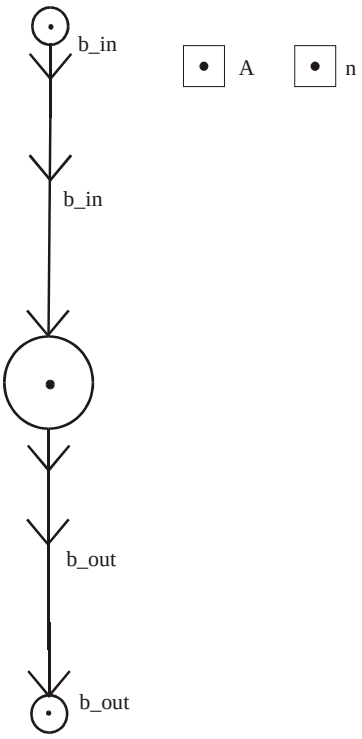


Figure 2-6 CODE Matrix Multiplication Sub-Graph

The two CODE graphs shown in Figure 2-5 and Figure 2-6 constitute a simple parallel matrix multiplication algorithm, an example similar to that offered by the originators of CODE {Newton 1993}. The algorithm used is the same as that used in the HeNCE example of Figure 2-2, where a matrix-vector multiplication is performed in parallel and the results merged to form the final solution. However, the CODE example exploits two features that HeNCE does not possess, namely hierarchical graph decomposition (procedure-call) and a shared memory abstraction.

It should also be noted that the CODE graphs are commented. The commenting is necessary to show the arc topology specifications, which are of fundamental importance to the execution of CODE programs. For those elements referred to by the arc topology specifications, it is inferred that there exists a corresponding element within the associated nodes. That is to say that for each binding there is the implication that there exist appropriate local variables exist for the graphical elements to bind to within each of the nodes.

The function of each of the graphs, and importantly the annotations, will now be described starting with the main graph of Figure 2-5.

1. The first UC node reads the inputs from the environment and sends the values onto three outgoing arcs. Each of these arcs carries one of: a matrix; an integer value for the current matrix column size; and a vector. The matrix and integer value for the matrix column size will be identical on each of their instantiated arcs, whereas each of the vectors may differ. The comments near each of the arcs show the actual bindings performed between the first UC node and the call node. The arcs marked `.A_out[i] => [i].A` and `.n_out[i] => [i].n` carry the matrix to be multiplied and its dimension to creation parameter nodes within the multiplication sub-graph. The indices present on the arc topology specifications show that the arcs may be instantiated more than once at run time, which infers that the call node is also instantiated more than once at run time. The remaining arc, `.b_out[i] => [i].b_in` carries a single vector to the call node in the same way.
2. The call node is used to instantiate a sub-graph and divert the flow of control from the parent into the newly created sub-graph. In this case, it is used to send a matrix, a vector, and an integer to a matrix-vector multiplication graph. Again arc topology specifications use indices, which indicates that the output from multiple instances of the sub-graph are to be gathered.
3. The output from the parallel matrix-vector multiplication is gathered to form the final result. Note that explicit identifiers for each of the vectors are not required as the subscript of each of the indices can be used to identify each of the incoming data elements.

Whilst normally the graph call node would be treated as a simple black-box, in the case of this example it is prudent to examine the behaviour of its sub-graph, which can be seen in Figure 2-6.

1. The creation parameter node marked `A` is bound to the arc marked `.A_out[i]` in the calling graph and is used in the sub-graph to hold a matrix. The creation parameter node marked `n` is bound to the arc marked `.n_out[i]` in the parent graph and used to hold the size of the matrix for the multiplication in the sub-graph. The

values held in both nodes are set when their graph is instantiated, and remain constant throughout the execution of the program.

2. The input interface node binds an arc from the calling graph to the local graph. Its output is identical to the input that it receives from the arc to which it is bound. The actual arc to which an instance of the sub-graph is bound depends on the value of the index of the arcs which are incident upon that sub-graph. That is to say that CODE allows for multiple instantiations of sub-graphs via the arc topology specifications of an application, in the same way that the UC nodes may be replicated.

3. A vector is received by the UC node from the input interface node, which in turn has received it from the arc in the calling graph marked `b[i]`, where `i` is the index for this particular instance of the sub-graph. The UC node also implicitly has access to the values stored in both creation parameter nodes as it exists within the same scope as those nodes. A simple matrix-vector multiplication is performed and the resulting vector is passed onto the successor of the UC node.

4. The output interface node binds the output from the sub-graph to an arc in the calling graph. The arc which is bound to a particular output interface node is dependent on the index of the arc, in an analogous way to the input interface node.

2.3.2.10. CODE Observations

The primary goals of the CODE language and execution model are to provide both an efficient and portable programming environment. Portability is attained through the use of a language which is abstract in appearance, and not closely coupled to any particular architecture class. Efficiency is achieved by providing translators which map the CODE language onto particular architecture types, thus enabling the facilities provided by each architecture to be exploited without presenting those features directly to the developer. To prove the efficiency case for CODE, its originators have developed an example using the Barnes-Hut particle simulation algorithm {Barnes and Hut 1986}. The CODE program for this simulation demonstrated an impressive speedup curve close to that achieved by a hand-crafted piece of C code (though both approaches failed to achieve a curve similar to that for ideal speedup {Newton 1993}).

Though the translation of CODE programs has indeed been shown to be efficient, the language itself presents some barriers to the straightforward development of parallel applications, because of its use of textual annotations which manage dataflow logistics. This is detrimental to the CODE language in three ways. Firstly, the structure of CODE graphs are not a true representation of actual program behaviour. Though it is recognised that representing the dynamic behaviour of a program on a static graph is a non-trivial problem, CODE graphs do not maximise the amount of potential run-time behaviour present in each graph, relying instead upon a proprietary textual language. Thus, it is impossible to predict accurately the behaviour of a CODE program by using the graph alone. This means that the advantages afforded by the separation of control-flow into a graphical language and computation into a textual language diminish, as the graphical language becomes increasingly dependent on the supporting textual component. Secondly, the introduction of a third language for writing node stanzas and arc topology specifications, in addition to the graphical language itself and the textual high-level language in which external procedures are created, is another potential entry point for bugs into an application. Thirdly,node programs and arc topology specifications do not constitute part of the problem domain under consideration and thus are an overhead on the use of CODE for constructing applications.

Bearing in mind that CODE is young and still being actively researched, some of these problems may yet be overcome. For instance, it may be possible to automate at least the arc topology specifications, as exemplified in ParADE, where graphical-textual bindings are resolved automatically {Allen 1998}. Furthermore, experience has demonstrated that whilst using CODE the more complex arc topology specifications tended to be used where data-parallelism was required. In this case, some form of pre-defined arc topology specification pattern or specialised node type such as HeNCE's fan-out node would provide significant benefits to the software developer.

Leaving syntax aside, the way in which CODE programs are made portable across a range of architectures is another aspect of CODE which distinguishes it from its contemporaries. The use of a translation layer for each architecture class that CODE programs are to be executed upon produces both benefits and drawbacks. On the beneficial side, the use of a specialist translator for each target architecture offers the possibility of tailoring CODE software to suit the architectural nuances of each machine class, and correspondingly increase application performance through exploiting those features. On the downside, the task of producing translation software for new

architectures limits the availability of CODE to specific architectures, unlike other systems whereby portable run-time systems have been targeted. In an attempt to alleviate such a burden, the CODE system is built in an object-oriented fashion, allowing for straightforward supplementation of existing translators, and indeed this feature is heavily praised in one thesis on CODE {Newton 1993}.

Although the originators of CODE make much of the fact that developing translators for a wide range of machine architectures is greatly eased by the CODE environment's object-oriented architecture, it is ironic that the CODE language itself does not support object-orientation. Using an object-oriented approach, it has been demonstrated that the existing translators can be supplemented, using useful concepts such as inheritance. In the same way, CODE source application could be re-used and extended rather than re-written. Parallel programming is far more difficult than sequential programming, yet it is sequential programs which receive support for re-use and maintainability. CODE's originators seem to suggest by this action that it is desirable to offer a rich development paradigm to the tool producers, but deny that paradigm to the users of that tool.

CODE does however provide a basic mechanism whereby source code can be managed via the sub-graphing facility. Source code can be kept in logical groups and partitioned to keep modules relatively simple, using a hierarchical graph decomposition strategy. With the aid of the call node facility CODE programs can be developed in a top-down fashion, using a hierarchical approach to manage complexity, and structure applications. Whilst this does lag behind the state of the art in software development, CODE is at least equal with contemporary systems in the field.

Whilst CODE has achieved some success and even managed to provide a portable parallel programming methodology, it is not without drawbacks, some of which it holds in common with the HeNCE system. For systems which avoid the drawbacks common to graphical flow-style languages, researchers from the original HeNCE project have progressed with a second visual parallel programming language based upon an entirely different parallel programming paradigm, which is discussed next.

2.3.3 VPE – Visual Programming Environment

> VPE is a visual parallel programming environment for message-passing parallel computing and is intended to provide a simple human interface to the process of creating message-passing programs. Programmers describe the process structure of a program by drawing a graph in which nodes represent processes and message flow on arcs between nodes. They then annotate these computation nodes with program text expressed in C or Fortran which contains simple message-passing calls. The VPE environment can then automatically compile, execute and animate the program. VPE is designed to be implemented on top of standard message-passing libraries such as PVM and MPI.{Newton and Dongarra 1994}

2.3.3.1. The VPE Paradigm

VPE is a hybrid textual-graphical language, intended to simplify the construction of message-passing programs built upon the standard PVM and MPI {Snir, Otto et al. 1996} message-passing libraries. As such, VPE splits an application into two components: the graphical aspect which represents the process and inter-process communication structure; and the textual part, written in C or Fortran, which forms the actual computations to be executed within potentially parallel processes.

VPE employs a simple top-down design strategy via a sub-graphing mechanism in the language, which supports structured, top-down programming techniques. VPE graphs show the structure of computations, or processes, and the communication that may occur between those computations. For each process (computational node) specified in a VPE graph, the programmer is presented with an environment within which to construct a sequential textual program. The exact environment presented depends on the arrangement of input and output ports attached to a computational node. However, because the structure of the overall program has been determined graphically, the text embedded within the computation nodes is simpler in comparison to the equivalent programs constructed by hand using the message-passing libraries. That is not to say that the complexity of the problem domain itself has somehow been reduced, nor that that the textual programming languages embedded within the computations are somehow made simpler under VPE. The advantage that VPE offers to the programmer at the textual level is that communication primitives are relatively simple, compared with the primitives offered by PVM and MPI. As the message-passing library calls are not used directly, VPE introduces the possibility of porting programs between the two message-

passing environments. Furthermore, as VPE calls map directly onto the underlying message-passing environment, very little overhead is added when using a VPE communications primitive compared to using a PVM or MPI primitive directly {Newton and Dongarra 1994}.

In addition to providing a top-down design strategy, VPE takes further advantage of the sub-graphing mechanism by permitting the execution of sub-graphs to be parameterised, and when appropriate, initialised before being executed, in a similar vein to CODE's shared variable abstraction introduced earlier. In VPE, graph parameterisation is a mechanism used to hold values that will be utilised by the actual computations within the same graph. Parameters are fixed before any computation begins and may be set either by an initialisation computation or by values received from a calling graph. Any of the graph parameters can be read from any of the computational nodes in the graph as if the parameters were declared locally within those computations, and as such provide a simple shared-memory abstraction. Computational nodes are also free to update as well as read the value held by the graph parameters, though any updates made by a node are only seen by that same node: communication via the shared memory abstraction is not permitted, save for initialising the graph values.

For those parameters not initialised by value arguments from a calling graph, an initialisation computation is used. The purpose of the initialisation computation is purely to set values for any graph parameters that are not to be received from a calling graph. It is noteworthy that initialisation computations may encapsulate any arbitrary application-specific functionality, but in the general case they are used merely for initialising graph parameters.

Apart from the shared memory mechanism made available during the initialisation of graphs, VPE utilises a message-passing paradigm. As mentioned above, there are no other facilities for communicating information through shared memory, as such mechanisms are at a higher level of abstraction than the VPE model encompasses. If a computational element is dependent upon data from another computational node, the data in question can only be transported via the explicit exchange of a message between those nodes.

Though admittedly low-level, the execution model supported by VPE naturally encourages parallel activity. Each computational node in a VPE program may potentially execute in parallel with any other. The only programming barrier to completely parallel

execution are blocking message-passing calls which enforce control-flow dependencies between computations.

As the VPE paradigm is purely message-passing based, there are no separate graphical components which control specific forms of parallelism. Task parallelism is the default mode of execution if possible, and pipelined parallelism is implicit within a program's structure. It is noteworthy that data parallelism is also available to the developer, though exploiting data-parallel activity must be explicitly specified (but not programmed since ultimately it is the message passing structure of the application which governs even data-parallel activity) using the VPE replication node.

As a consequence of VPE's approach to parallel application development, the programmer does not have to learn a textual language solely for programming the communication behaviour of the nodes. Such communication occurs using a binding in a language with which the developer is familiar, being either C or Fortran, which is significantly less open to accidental misuse than the approach whereby a third language element is introduced specifically for mapping textual and graphical level entities.

Furthermore, unlike its contemporaries VPE does not seek to address parallel control-flow issues graphically. The fact that VPE does not specify control-flow graphically is not a shortcoming, indeed it is not an aim of VPE to be able to do so. VPE is ultimately a visual interface to message-passing libraries rather than a fully-fledged parallel programming language in itself.

The VPE programming environment is extensive. Control of the underlying message-passing mechanism to which VPE is coupled is performed through a graphical console window within the development environment. VPE also automates one of the most common burdens in message-passing parallel programming in a heterogeneous environment, by ensuring that appropriate executable files are distributed around hosts within the networked multiprocessor. Without programmer intervention, source files are compiled for appropriate architectures and the resulting binaries placed on file systems which those hosts can read. Post-mortem style program animations are also available to the developer to aid debugging and optimisation.

2.3.3.2. VPE Node Icons

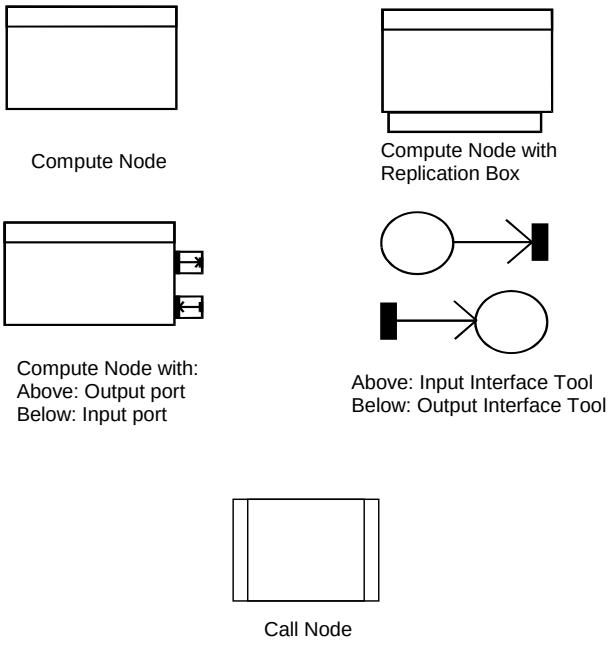


Figure 2-7 VPE Language Node Icons

2.3.3.3. Compute Node

The compute node is the basic building block in VPE. Each compute node is a template for the creation of a computational process, which the developer must express in either C or Fortran. To enable communications with other compute nodes, communication ports are attached to each compute node's edges. Indeed, a VPE program without ports can only be executed as a single serial process.

A compute node becomes active when its internal program becomes active. This may occur at the very beginning of a program's execution, or the compute node's execution may be suspended, awaiting the arrival of a message. The graphical structure of the program itself does not determine the order of execution, it merely describes possible communication paths which the internal programs of each compute node are free to use.

2.3.3.4. Replication Box

The replication box provides the mechanism for data-parallel execution within VPE. The function of the replication box is to execute a particular computation node in parallel with other instances of itself, but ideally with each working on disjoint sets of data. The amount of replication that occurs is governed by graph parameters which are set at run-time, thus permitting the program to adapt its behaviour according to the

metrics of the problem under consideration, and potentially modify the level of parallelism according to hardware run-time conditions.

The replication box alone does not fully address the problem of data-parallelism. Any messages exchanged with a replicated compute node must specify which of the replicated computations is being communicated with, whether they are sending data to, or receiving data from any of the replicated computations. To this end, the message-passing calls provided by VPE take an identifier of a replicated computation as one of their arguments in a manner not dissimilar to CODE's arc indices. Experience has shown that even with this extra burden, the message-passing calls used by VPE remain significantly simpler than using the message-passing libraries directly.

2.3.3.5. Port Nodes

VPE is unlike other systems where communication via graph level components must be explicitly programmed in a textual form. Instead, input and output nodes provide typed bindings between the variables used in the compute node programs and the communication structure expressed in the graphs. Within the textual computation such ports are accessed through function calls, whose usage is consistent with the language used to write the node programs, using appropriately typed arguments. For example, if an output port is configured to emit an array of integers, calls to that output port must supply an array of integers as an argument. The same is true for receive operations using input ports.

2.3.3.6. Call Node

VPE permits a hierarchical decomposition of programs through a procedure call-like mechanism. A call node is used when the programmer needs to call upon the functionality encapsulated within a separate VPE graph. The graph called may be another graph within the same application, a library component, or another developer's routine, though maintaining a consistent procedure-call semantic throughout.

2.3.3.7. Input and Output Nodes

When a VPE program is composed from more than one graph there must exist entry and exit points for all except the main graph. Input and output nodes supply that functionality by providing a binding between arcs in a sub-graph and arcs in calling graphs.

Within a sub-graph, there is no concept of coupling to any one calling graph. Instead, data is sourced from input nodes, which receive their data from input ports attached to the sub-graph's call node. Similarly data sinks in the form of output nodes send the values which they receive to the output ports on the sub-graph's call node. This reduces coupling between graphs and ultimately helps to improve the modularity of VPE source code.

2.3.3.8. VPE Matrix Multiplication Example

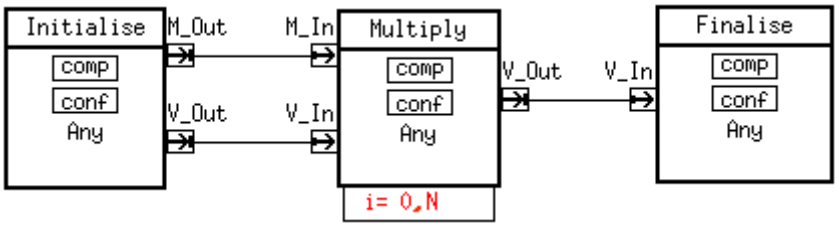


Figure 2-8 VPE Matrix Multiplication

The diagram in Figure 2-8 shows the VPE graph for the parallel matrix multiplication algorithm, whereby many matrix-vector multiplications are performed concurrently and the results combined to form the final result. In the diagram there are three potentially parallel processes, one of which may spawn concurrent copies of itself by means of the replication box annotation. The functionality of the graph is outlined below:

The graph initialisation computation is run and a graph parameter for the size of the matrices to be multiplied is set. Though this does not appear on the graph in the form of a compute node, it is worth recalling that VPE allows graph parameters to be initialised in this way. The graph parameters are themselves needed as they control the level of replication that occurs in the replicated compute node.

The initialise compute node creates two matrices, sizing them using the graph parameters. One of the matrices is split into vectors which are then transmitted via the output port `V_Out` to one of the replicated compute nodes identified by the index of the current row vector. For each outgoing vector, a copy of the matrix which is to be multiplied by the vector is sent to the output port `M_Out`. A single port could have been used to send both the matrix and the vector, but for the sake of clarity and ease of comparison this approach has not been adopted.

Upon receiving a matrix and a vector to multiply, an instance of the multiply node is invoked. The actual number of parallel instances invoked depends on the graph

parameter set for the number of rows in a matrix. The number of rows is read from the graph parameter that is set before the program is started. Each instantiated compute node outputs the results of its calculation to the output port `V_Out`, using the index of the replication to act as an identifier for the newly created vector.

The finalise node receives vectors through its input port `V_IN`. From the index of the replicated node which delivers values to `V_IN`, the finalise compute node places the vectors in the appropriate part of the matrix. When all vectors have been received and the final result assembled, the program outputs the result and terminates.

2.3.3.9. VPE Observations

The VPE graph, despite its small size, manages to exploit a great deal of parallelism. Data-parallel activity occurs due to the replication of the main part of the program, in the compute node marked `multiply`. Pipeline parallelism is also evident in that each of the compute nodes can be viewed as a stage in a pipeline, where each compute node works on the data sent to it by the previous node in the pipeline. In this example, it is entirely feasible that all parts of the program could be working on splitting, multiplying and re-assembling matrices in parallel, thus maximising parallel activity.

Though the VPE graph is compact and the program is highly parallel, there is a price to be paid. The price comes in the form of node programs that are of greater length and higher complexity than contemporary systems. Furthermore there are no facilities to support the development of these compute node programs, other than the simplified calls to the message-passing libraries, and thus had the problem under consideration been of significantly greater complexity it is uncertain whether VPE alone would have been a sufficiently powerful tool.

Unlike contemporary systems, VPE is not meant to be a visual parallel-programming language per se. Its aims from the outset have been somewhat different, in that VPE is designed purely to simplify explicit message-passing parallel programming with the standard PVM and MPI message-passing environments. As such, it is expected that the programmer will already be familiar with parallel programming and be able to construct parallel algorithms without assistance from VPE.

Where VPE excels is the fact that there are only two elements to the language, both of which are directly connected to the algorithm under consideration. Unlike contemporary systems where it is common that a third, proprietary language element is added to control the communication structure of the graph, the communication structure of a VPE program is entirely specified in terms of VPE graphs and standard high-level textual programming languages. Thus, in comparison to contemporary systems, VPE presents one less opportunity to introduce bugs into an application.

However, the fact that VPE is meant to be used as a front-end to the message-passing libraries and not as a language in its own right means that there is no clear distinction between computation and communication. In a VPE graph, possible communication routes are shown, though from the graph alone it is impossible to tell which nodes will execute under which circumstances. Whilst this is not a goal of VPE, its contemporaries are able to actually show sequencing and dependencies which may well be of value to the developer of a parallel application.

Run-time aspects such as load balancing and task creation strategies are not a consideration of VPE. However, VPE does allow computations to be mapped to particular hosts or architecture classes in the underlying virtual machine environment. Thus computations suited to particular architectures can be accommodated, or less heavily loaded machines can be utilised, in a similar fashion to the HeNCE cost matrix facility.

Furthermore, as VPE maps directly onto the standard message-passing systems, it can benefit directly from software that enhances those systems. Thus as the message-passing environments are improved, the performance of VPE programs will also improve.

In short, VPE's departure from the normal approach to building visual parallel programming tools has yielded a certain amount of promise. The paradigm used is close to the paradigm in use by many parallel programmers, and furthermore VPE is sufficiently low-level to not incur run-time overheads often associated with using higher-level visual parallel programming languages. As a tool for simplifying the implementation of today's parallel programming problems, VPE is a good effort. When considering the needs of future parallel application developers VPE may simply be too low-level.

2.3.4 ParADE –Parallel Application Development Environment

While the benefits of parallel computing – the performance increase due to simultaneous computations – are clear, achieving those 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.

The ParADE system provides a means 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. Furthermore, the ParADE language provides a graphical means for adjusting the granularity of parallel tasks to tune programs to the particular architecture upon which they execute. {Allen 1998}

2.3.4.1. The ParADE Paradigm

ParADE is a mixed textual-graphical parallel programming language based upon the dataflow execution model. The structure of a parallel application, including synchronisation and communication, is determined graphically, whilst computations are described textually in C.

The visual programming component in ParADE takes the form of directed, acyclic, top to bottom graphs where nodes, known as actors, represent computation or routing elements, and arcs represent data dependencies between nodes along which values may be passed. Actors become active when there is data on each of their input arcs. When activated, actors will either execute a user defined computation, or a predefined action depending on whether the actor is general or special purpose respectively.

In ParADE, general-purpose actors are the only components which must be programmed using a high-level textual programming language. The developer is not charged with maintaining interfaces between the graphical and textual programming components. Instead, tool support bears the responsibility for binding arcs onto local program variables within the actor source code. All the programmer must do is use those variables in a manner consistent with the high-level textual language with which the actors are programmed. That is to say that the programmer sees arcs merely as standard C programming language variables when writing the actor source code, omitting the extra interface code required by some of ParADE's contemporaries.

ParADE permits the exploitation of both parallelism implicit within the structure of its graphs, and parallelism explicitly programmed by the developer. Implicit parallelism is a strength of dataflow based languages such as ParADE, in that tasks which are mutually independent may execute in parallel by default. Furthermore, to some degree tasks that are not mutually independent may be able to execute in parallel through pipelined parallelism, which is also claimed to be implicit within the graph structure.

Data-parallelism is explicitly expressed by the programmer, using a general-purpose parallel activity actor, known as the depth actor. Unlike contemporary systems, ParADE's depth actor encapsulates the decomposition and recomposition of data which is operated upon in parallel. Furthermore, ParADE comes equipped with several common visual templates for data decomposition and recomposition, and the architecture of the ParADE tool support is such that it is possible to supplement the built-in templates with user-defined patterns.

Along with automated data distribution, ParADE also has a facility for controlling the grain size of a computation in order to tailor the characteristics of the application to the particular platform upon which it will be run. ParADE's predecessor, MeDaL {Harley 1993}, used a mechanism for grouping data to be sent along arcs as its mechanism for increasing granularity, though in practice the grouping of messages was found to be an unsatisfactory solution which was not pursued in ParADE. ParADE's unique graphical mechanism for altering the granularity of processes is known as actor folding. Actor folding is accomplished by selecting a group of actors to execute sequentially as a single task. The ParADE system itself automatically sequences the execution and optimises the communications of the folded actors without assistance from the developer.

As a design aid and code management facility, ParADE utilises hierarchical decomposition of programs through actor decomposition. Actor decomposition is similar to the use of sub-graphs in visual programming or procedure calls in textual high-level languages. However, ParADE does not seek to visually differentiate between an actor which has been decomposed and one which has not. In this way it seeks to simplify the black box approach to a level whereby a decomposed actor is identical even in appearance to a standard actor, which helps to reinforce the notion of the black-box semantic.

2.3.4.2. The ParADE Notation

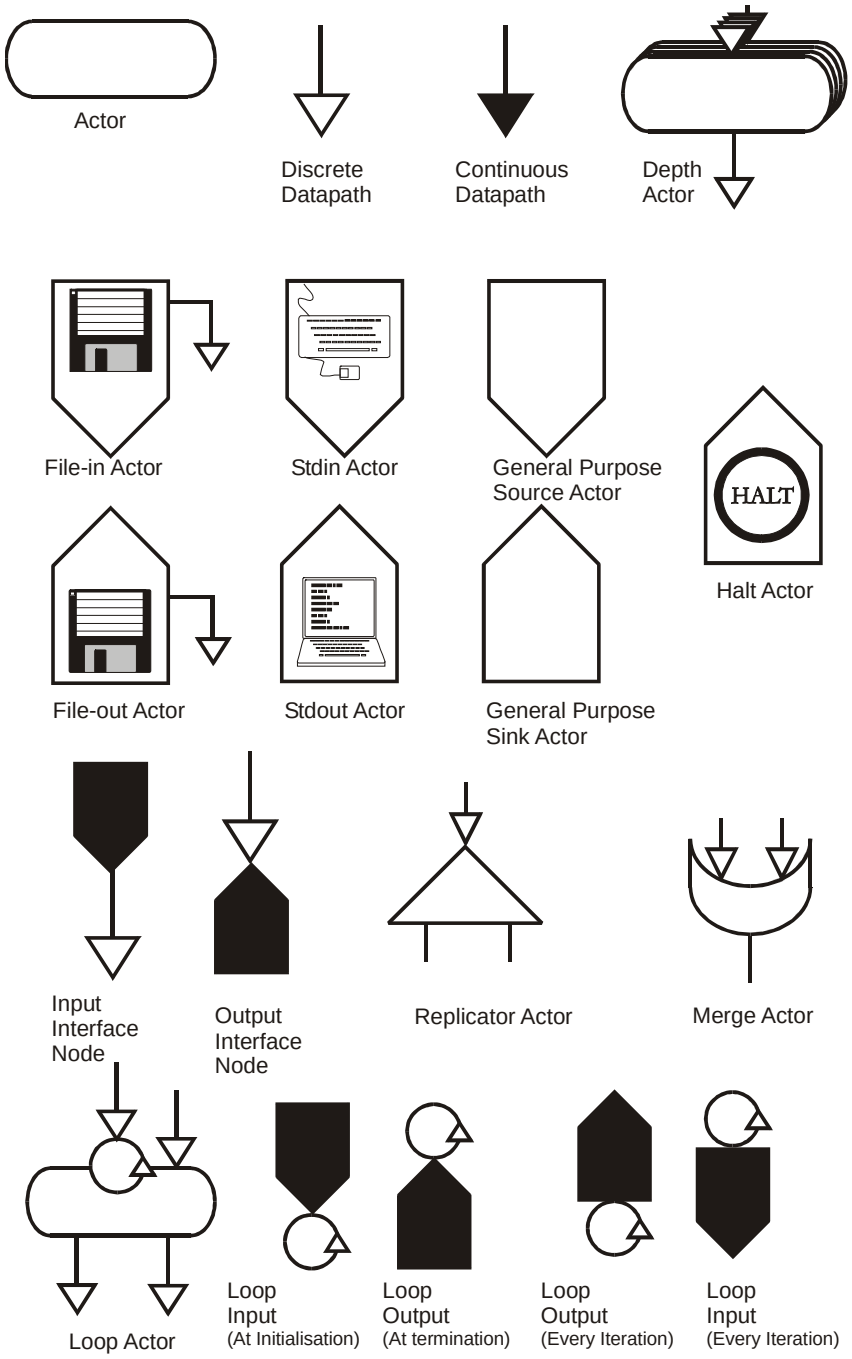


Figure 2-9 The ParADE Node Icons

2.3.4.3. General Purpose Actors

ParADE's general-purpose actor provides a harness within which the developer's textual source code resides. There is no default action specified for either the standard

actor or depth actor, and as such the programmer must always provide an action in C code for the actor to execute when it is activated.

Both the standard and depth actors are activated when there is data present on their input data arcs, which are incident upon their topmost edges. Upon activation the actor immediately consumes the data on the input arcs. If the consumer is a depth actor, the data may be partitioned according to the partition strategy specified by the programmer and the data operated upon in parallel by multiple instances of the actor code.

Operation of both standard and depth actors is synchronous in that each time the actor is activated there will be one set of input data consumed and one set of output data produced. Even the parallel activity that occurs within the execution of depth actor is abstracted away from the rest of the application in that for each set of input data, there exists a single set of output data. As the depth actor will be responsible for not only partitioning, but re-combining the data according to the specified template, a consistent set of semantics is maintained across both actor types.

2.3.4.4. Datapaths

ParADE provides two types of datapaths along which values can be passed between actors. The purpose of providing two types of datapath is for programming convenience and optimisation. The first of these datapaths, the discrete datapath, visualised as a white-headed arrow, is implemented as a classic FIFO queue. A producing actor will supply data, which then enters the datapath at its tail, and later leaves at the head when the receiving actor consumes the data. The second type of datapath, the continuous datapath, represented by a solid-headed arrow, is a similar structure to the discrete datapath, except that it has the additional property that under certain circumstances it will retain a copy of the last item of data in the queue. This retention of data occurs when the queue would otherwise be empty, and thus a continuous datapath will always offer data to actors once it has been used for the first time. The continuous datapath can therefore be used in programs where repeated transmission along datapaths would otherwise incur an overhead. Using a mixture of discrete and continuous datapaths it is possible to minimise communication overheads in a program, whilst maintaining the correct dependencies between actors.

2.3.4.5. Source and Sink Actors

The library of source and sink actors in the ParADE language provides facilities for ParADE programs to communicate with their environment. Source actors allow information from the environment to be used during the execution of the application, while sink actors permit the application to send messages out to the environment or to terminate the application and return control to the environment. The basic types of source and sink actors include the general purpose source and sink actors, the file I/O actors, the standard I/O actors, the graph interface nodes, and the halt actor.

The general-purpose source and sink actors provide a user-programmable means of instigating and terminating dataflow. They are somewhat similar to the general-purpose actors in that they are permitted to execute computations upon data. However, unlike the general-purpose actors, the general-purpose source and sink actors possess only a single arc.

The stdin actor allows the user to specify input to the program via the standard input stream, and is represented as a source actor containing a keyboard and mouse icon. Conversely the stdout actor permits the application to return values to the standard output and is represented by the monitor icon within a sink node.

Similarly, the file in and file out actors are depicted as being sources and sinks containing a floppy disk icon which is symbolic of their file-oriented operation. Additionally, the file actors have a second datapath emerging from their sides, which, when utilised, signifies some form of exceptional behaviour with respect to the files being used, such as lack of disk space, or the non-existence of a file.

The input interface and output interface nodes provide bindings between datapaths on the current graph and datapaths in a decomposed actor. For each datapath incident upon a decomposed actor, there exists an input interface node in the sub-graph. Correspondingly, for each datapath exiting from the sub-graph, there exists an output interface node which performs the binding between the sub-graph datapaths and the parent graph. Interface nodes do not alter the flow of data; their function is merely to provide an binding mechanism between graphs.

The only sink node that does not have a complementary sink is the halt node. The halt node's functionality is simply to provide a means by which program execution can be terminated. When activated, the halt node stops the execution of program, and potentially return values to the application's environment.

2.3.4.6. Merge Actor

The merge actor in ParADE takes as its input two or more datapaths and outputs the data from these paths onto one single datapath. The merge actor is useful for collating the output of several actors for further processing by other actors. The order in which the data items are placed on the output datapath is strictly chronological. If multiple items of data arrive at the input to the actor simultaneously, the order in which they are output is undefined.

2.3.4.7. Replicator Actor

The semantics of the replicator node are simple. When data is placed on the input datapaths copies are broadcast onto all of the going datapaths, though the actual placing of copies of data onto the outgoing datapaths may not occur strictly simultaneously.

2.3.4.8. Loop Actor

Unlike its predecessor MeDaL, ParADE language graphs are acyclic and thus iteration cannot be achieved through a feedback loop. Instead, ParADE provides a loop actor purely for iteration, whose semantics are consistent with the synchronous execution semantics of the other actor types.

However, unlike the other actors in the ParADE language, loop actors must be further decomposed to implement their behaviour. The loop actor itself thus acts merely as a black box at the level of the container graph to which data is provided and from which values are returned.

The actual iterated computation can only be seen at the level of the sub-graph where it is supported by four other node types which are specific to the decomposed loop actor. These nodes provide an input to the loop at the beginning and after each iteration and an output from the loop upon termination and an output after every iteration. These nodes are depicted in Figure 2-9 as the loop input and loop output nodes, where each type is uniquely identified by the position of the loop symbol and by the direction in which each points.

Within the decomposed loop actor, there must also exist at least one other actor. This actor encapsulates the computation that is to be iterated, and communicates with any of the data sources or sinks in the normal way. There is no special mechanism for coupling the iterated computation to the iteration mechanism, which helps to maintain a modular, de-coupled source code base.

2.3.4.9. ParADE Matrix Multiplication Example

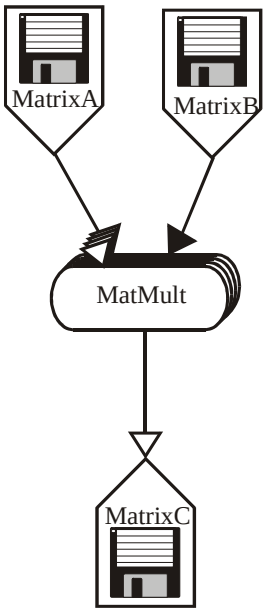


Figure 2-10 ParADE Matrix Multiplication Example

The diagram in Figure 2-10 depicts a ParADE graph for a parallel matrix multiplication problem, whereby multiple matrix-vector multiplications are concurrently executed.

1. The file-in actors read two matrices (`MatrixA` and `MatrixB`) in from disk files (potentially in parallel).
2. `MatrixB` is sent out along a continuous datapath to the depth actor marked `MatMult`. The reason that a continuous datapath is used, is because `MatrixB` is required throughout the operation of the depth actor `MatMult`. Clearly for each parallel task spawned by `MatMult`, a new copy of MatrixB could be sent along a standard discrete datapath, though using a continuous datapath offers an optimisation in terms of the amount of communication that occurs.
3. The data sent out from the file-in actor marked `MatrixA` is partitioned by the datapath connecting the file-in actor to the depth actor `MatMult`. This is a novel approach in that ParADE does not require the use of further actors to partition data before it can be

operated upon in parallel. Instead the data will be partitioned according to some graphical template specified for the datapath. Here, the decomposition template has been set to columns (not shown in Figure 2-10), and using that information, the ParADE system can arrange for individual column vectors to be extracted from the data, and sent to subsequent parallel computations.

4. The depth actor `MatMult` takes a copy of `MatrixB` and an element of the partitioned data from its predecessors, and for each data item on the datapaths executes a parallel matrix-vector multiplication. The results from these multiplications are automatically collated by the outgoing data arc according to the graphically defined template.

5. The final result is written to disk by the file-out actor marked `MatrixC`, and the program terminates.

2.3.4.10. ParADE Observations

ParADE aims to support the programmer in developing programs that are architecture independent, efficient and which exploit a wide-variety of parallel activity. Furthermore, ParADE is also intended to move parallel programming into the domain of the non-specialist programmer. In all of these programming-oriented aims, ParADE is generally successful. Through the use of a graphical co-ordination language and a common computation language (the C programming language), parallel programming becomes accessible to non-specialists, and furthermore the abstract computational model does not couple ParADE closely to any particular architecture or run-time system.

Whilst ParADE insulates the developer of an application from the complexities of parallel systems, this does not mean that the programmer can ignore parallel computation altogether. Though it is true that a great deal of parallelism may be implicit within the structure of an application, there are cases when the programmer will need to be aware of basic parallel-programming principles in order to exploit all of the available parallelism within an application. In particular, the programmer must be aware of when and how to apply data-parallelism in order to be able to utilise the depth actor and data partitioning facilities available within ParADE.

On the positive side, ParADE, unlike its contemporary systems, does not require the user to perform bindings between graph level objects and text level objects. The tool supporting the ParADE language automates this task, thus removing the potential for introducing inconsistencies between the co-ordination and computation components of an application. Furthermore as ParADE is data-centric, it provides a graphical mechanism for partitioning and combining data which circumvents the need to create actors for that purpose, thus removing this significant intellectual burden from the developer.

In addition to the graphical specification of data partitioning, ParADE also provides a graphical means of increasing the granularity of parallel tasks, through actor folding. The theory behind actor folding is simple, in that multiple fine-grained actors (where initialisation or communication costs outweigh the actual computation costs) can be combined to form a single medium or coarse-grained computation (where computation costs outweigh the costs of setting up the computation). Actor folding is a simple operation from the user's perspective. The developer must identify those actors which are to be grouped to form a more coarsely grained computation, and surround them with the actor folding box, merely by dragging the boundaries of the box to surround the actors. ParADE itself assumes the responsibility for combining the actors into a single computation, optimising communications, and ensuring that the computations are executed in the correct sequence.

Uniquely amongst visual parallel programming tools, ParADE extends the standard visual parallel programming approach by implementing a basic exception-handling schema. Although exception handling is limited to file operations, ParADE's graphical exception handling could be generalised to other node types. Whilst this would certainly result in graphs with a higher information density, it would also mean that the exception handling structure of a program could be represented in the same way as the non-exceptional behaviour. That being said, the responsibility for presenting different views of an application could be handled quite straightforwardly by a suitable development tool.

It is clear that the ParADE graph is much simpler than the graphs of the previous languages. It is perhaps obvious that ParADE graphs are simpler because ParADE does not require the developer to manually construct bindings between textual and graphical level variables, and encapsulates much more control-flow detail within the each of the

graphical components than its contemporaries. Furthermore, the richer set of graphical components that encapsulate common control-flow options mean that there are less control-flow issues to be handled at the textual level.

Although ParADE has impressive support for parallel programming, through its novel features and rich set of language constructs, it has very limited support for other phases in the software lifecycle. Support for designing programs is limited to a top-down approach using actor decomposition and there is no support for anything other than top-down design and granularity optimisation. Whilst the program visualisation and other post-mortem debugging tools could be added to the prototype ParADE system relatively easily, it is unclear how ParADE would be adapted to support problem domain analysis or more sophisticated design methods.

2.4 Related Work

In addition to the surveyed languages, there are a number of other visual programming tools and languages, each addressing particular aspects of programming parallel architectures. Support ranges from graphical distribution of source code in a heterogeneous environment, through novel visual approaches to program construction {Kramer-Fuhrmann and Brandes 1991}, reusability {Loques, Leite et al. 1998} and fault tolerance issues {Babaoglu, Alvisi et al. 1992}. Some are specifically aimed at a single class of parallel architecture, or are sufficiently similar to the surveyed languages not to warrant individual discussion {Dozsa, Kacsuk et al. 1997; Kacsuk, Cunha et al. 1997; Kacsuk, Dozsa et al. 1997; Kacksuk and Forrai 1999}. Since this research is primarily interested in the development of general-purpose parallel applications, surveys of these more specialist systems are not presented. However, the literature more than bears testament to the fact that there is a great deal of commonality across all current visual parallel programming techniques, and as such a full presentation of all work is neither trivial nor warranted.

2.5 Closing Remarks

This chapter has set the scene for further research and development work in the field of visual parallel programming. During the course of the survey, the work undertaken exposed strengths and weaknesses in both the language element and paradigm of each of the systems. This experience, coupled with experience of software engineering practice and that of developing text-based parallel applications has set in process trains of thought as to how improve visual parallel programming. However, a pre-requisite for turning

those thoughts into action is to formalise the relationships between the surveyed languages, and draw from that formalisation the threads of a new paradigm for building parallel applications. This is the topic of the next chapter.

Chapter 3 An Analysis of Visual Parallel Programming

Given that considerable intellectual effort has already been invested in the field of visual parallel programming, and several notations developed, the questions that need to be asked are whether there is scope for improving visual parallel programming techniques, and if so, where those improvements could be made. To answer those questions requires key aspects of each of the existing languages to be captured, generalised, and reasoned about such that key strengths and weaknesses can be identified. The answers will provide the basis for the next step in the evolution of visual parallel programming techniques.

3.1 Introduction

The previous chapter introduced and exemplified four prominent visual parallel programming languages. This chapter proceeds by developing a taxonomy of those languages, from which a framework for subsequent research efforts in the area was derived.

Once the framework from the taxonomy has been established, the chapter fittingly splits into several threads. Each thread either investigates specific aspects of the taxonomy, or combines issues identified by the taxonomy with other software engineering issues. The overall structure of the threads running through this chapter is based upon Figure 3-1 below.

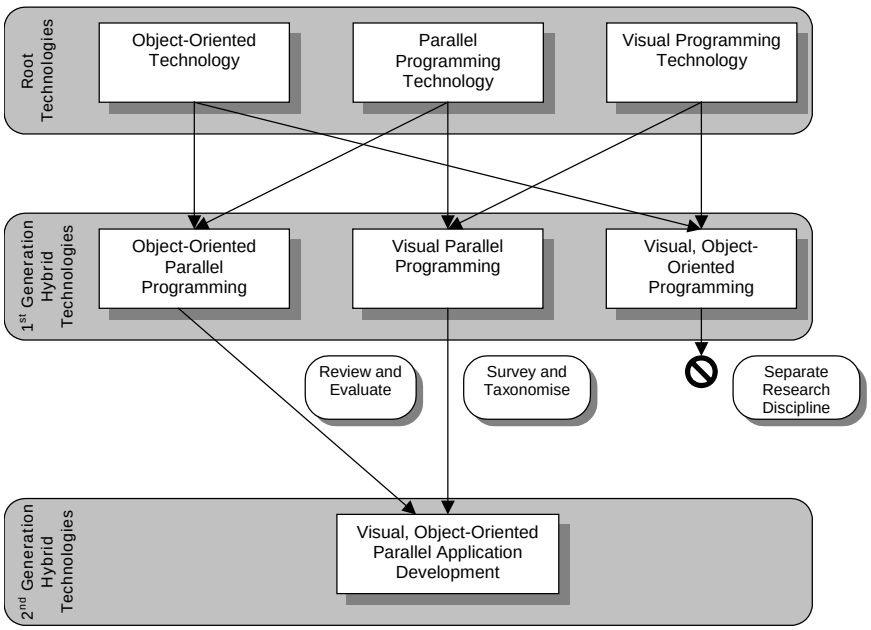


Figure 3-1 Visual Programming Technology: A Roadmap

Although the overall goal of this chapter is to reach a new paradigm for visual parallel programming, intermediate steps towards that goal, based on Figure 3-1, are also investigated. However, not all of the elements identified in Figure 3-1 are scrutinised to the same level. In particular, work on visual object-oriented programming such as that exemplified in {Burnett, Goldberg et al. 1995; Citrin, Doherty et al. 1995}, although valuable is not considered further since the goal here is supporting techniques for the development of parallel applications, and not on visual languages per se. Similarly work on object-oriented parallel programming has been dealt with in-depth by other researchers {Yonezawa and Tokoro 1987; Annot and Haan 1990; Treleaven 1990; Kale and Krishnan 1993; Wolf and Kramer-Fuhrman 1996}, and a full survey of the discipline is not presented although an overview of the salient features of the discipline is. The final of the threads from the middle layer of Figure 3-1 on visual parallel programming, has already been dealt with in-depth in Chapter 2. Instead, the taxonomy presented in Section 3.2 will recap the most pertinent points from the survey of visual parallel programming languages, whilst formulating a framework for their future development.

Once each of the distinct threads has been presented, the chapter then draws aspects of each into a single coherent model for parallel application development, using the framework provided by the taxonomy as its basis. It is the development of this model and supporting language, which form the basis for the work presented in the remainder of this thesis.

3.2 A Taxonomy of Visual Parallel Programming Languages[1]

The purpose of this section is to explore the design space for visual parallel programming languages and to present a taxonomy that captures the important characteristics that underlie them. A number of existing languages, those examined in Chapter 2, will be used to exemplify the taxonomy, but the taxonomy will also be of use to future language designers for highlighting possibilities that might not otherwise be considered.

Interest here is in languages that take the form of a graphical mechanism for structuring and controlling parallel invocations of textually specified code written in a traditional (non-parallel) language such as C or FORTRAN. The languages are visual coordination languages, in the sense that the Linda system {Gelernter 1985; Carriero and

[1] This section adapted from a journal submission jointly authored with P.A. Lee.

Gelernter 1989} is a textual coordination language for parallel programming, targeted as producing software that will execute on commodity hardware. Languages that are targeted at special-purpose hardware systems (e.g. data flow machines {Gurd, Kirkham et al. 1985; Veen 1986}, graph-reduction machines) are not considered here.

Thus the taxonomy presented here concentrates on the parallel processing aspects of visual parallel programming languages. A key feature for this taxonomy to explore is the interaction between the visual features that a visual parallel programming language provides and the parallelism features required. The fact that some parallelism feature (e.g. synchronisation between parallel elements) could be obtained through the use of the textual programming language parts that a language utilises is not something to be considered. What is of interest is the way in which the visual features of the language help the implementation of the parallel activity desired by the user, and whether those features are appropriate. This interest is reflected by the use of the terms *explicit* and *implicit*, to represent respectively the situations where the user does and does not have to deal with a parallelism feature directly. In other words, *implicit* features are assumed to be helpful to the programmer.

One feature which does not form part of the taxonomy is tool support. Since each of the surveyed languages forms, more or less, a research prototype it is not expected that complete environments for software development are available. Therefore, one of the non-functional requirements of this taxonomy is to distil features pertaining to the visual language, and isolate them from features provided by tool support, since it is the language element, and not the tool support, which is of importance.

3.2.1 Syntactic Elements of Visual Parallel Programming Languages

Visually, a visual parallel programming language supports a graph structure in which the main programming constructs are:

- *Nodes*, that are linked together by;
- *Arcs* that transport *tokens* of differing types between nodes.

The main type of node is a computation node which is where the language user specifies the application-specific computation. Other nodes may be provided by a language to assist the management of the graph's structure. Nodes that are independent may be executed in parallel. As will be seen, nodes themselves have a number of syntactic elements from which they are composed.

Arcs interconnect nodes and provide an overall indication of the interdependencies between the nodes, amongst other things. Such inter-node dependencies are particularly important in a parallel processing environment as they determine the extent to which computation nodes may be executed in parallel.

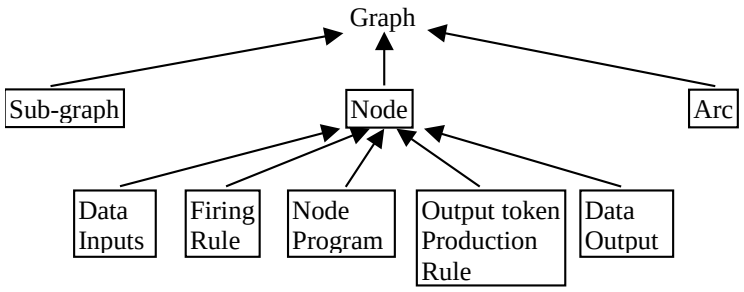


Figure 3-2 Main Syntactic Elements of Taxonomy

Figure 3-2 shows the main syntactic elements of the taxonomy. The arrows in Figure 3-2 indicate 'has-a' or compositional relationships rather than indicating the syntactic rules that dictate the legal ways in which graphs can be composed. In what follows, each of these syntactic elements will be examined in a top-down fashion with respect to:

- What options exist for the semantics of the syntactic element, and some discussion of those options;
- What sub-elements it is composed from (with those sub-elements themselves being examined subsequently); and
- Exemplification from contemporary languages.

3.2.2 Graph

	Composed of		
	Sub-Graphs	**Arcs**	**Nodes**

Table 3-1 Graph Composition

	Graph Semantics		
	Structure	**Node dependencies**	**Recursion**
Options	Directed Cyclic Directed Acyclic	Explicit via arcs Implicit	Permitted Not Permitted

Table 3-2 Options for Graph Semantics

3.2.2.1. Graph Discussion

Decomposition or modularisation of a complex system into more manageable (and reusable) components is a well-recognised requirement. This requirement is likely to be of particular importance in graphical notations which inevitably occupy more space than compact textual representations. Visual parallel programming languages can support this requirement by permitting one graph to be composed from other (sub-)graphs. Visual complexity can then be reduced by permitting a node on one graph to refer to a sub-graph constructed separately. This feature also permits graph-reuse (c.f. code reuse), a useful feature in itself for the usual reasons.

When sub-graphing is permitted the language will need to provide (visual) means for expressing the way in which connections represented on the parent graph are reflected onto connections in the sub-graph. These connections are usually represented by special types of connector nodes in the sub-graph and appropriate naming conventions to identify the cross-coupling.

The feature of sub-graphs raises the issue of whether recursion (at the graph level) is supported or not. A useful comparison may be drawn here between the issues of macros and functions in languages such as C. Functions may be recursive, requiring the dynamic generation of information at run-time for support. A recursive graph (i.e. a graph that "calls" itself) will also require dynamic support in the form of instantiation of a graph and its arcs at run-time. Non-recursive graphs (c.f. macros) can be completely instantiated at graph-compile time.

A statically generated system had the advantage that the complete parallel structure is known before execution commences, which may provide advantages for statically mapping the potential parallelism onto the underlying parallel platform. Dynamic generation has the advantage of supporting recursion, but the disadvantage is that dynamic run-time load balancing may be needed to support the evolving structure of the parallel application efficiently. Note however that run-time load balancing may be a requirement in any case if the load on the parallel platform may vary.

As mentioned earlier, the arcs in a graph show node interdependencies and hence impose order relationships on the execution of nodes. As will be discussed subsequently, the dependencies may be of the form of control-flow dependencies (e.g. node B must execute after node A) or data dependencies (e.g. node B needs data produced by node A). Since the graphs capture some of these dependency relationships, all of the languages

considered in this paper are directed, and may have a *cyclic* or *acyclic* structure. However, a language may or may not require explicit connectivity showing all dependencies. Where dependencies must appear explicitly, arcs join nodes which are dependent. If implicit dependencies are supported, any explicit dependencies created between nodes joined by arcs may be supplemented by implicit linking between nodes of certain types on a graph. While arcs show the possible links between nodes, it is assumed that these arcs carry *tokens* (of various types) that capture dynamic aspects of the behaviour of the application represented by the graph.

Explicit and implicit dependencies have advantages and disadvantages. On one hand the explicit method ensures that dependencies are visible on the graph and can be visualised at-a-glance, while implicit dependencies are "hidden" and cannot be discerned just from the graph's appearance. On the other hand, allowing implicit dependencies on a graph will minimise the need for arcs and therefore reduce that graph's visual complexity.

Cyclic graphs permit the expression of reusability, in an iterative sense. The nodes enclosed in a cycle essentially represent what would be a repetition loop in a sequential programming language, permitting (a) a section of the graph to be reused; and (b) feedback of results from one iteration to be fed back to the next. This is a useful programming paradigm to support even in parallel applications, and languages which are acyclic generally provide other means for expressing iterative repetitions (normally by using a special type of control-flow modifier node).

3.2.2.2. Graph Examples

	Graph Semantics			
	Sub-Graphs	**Structure**	**Node dependencies**	**Recursion**
HeNCE	Not permitted	Directed acyclic	Explicit & Implicit	Not permitted
CODE	Permitted	Directed cyclic	Explicit & Implicit	Permitted
ParaDE	Permitted	Directed acyclic	Explicit	Not permitted
VPE	Permitted	Directed cyclic	Explicit	Not permitted

Table 3-3 Graphs Semantics for Specific Languages

HeNCE graphs are directed and acyclic, and sub-graphs and recursion are not supported. While it was claimed [Browne95] that sub-graphing was just a support-tool issue, in fact other HeNCE features would make sub-graphing problematical for purposes of reuse (although not for managing visual complexity). The reason why HeNCE does not readily lend itself to supporting procedure-calls is due to the fact that data values within a HeNCE application are stored in a globally accessible namespace. Once a node has received sufficient control-flow stimulus from predecessor nodes, it

begins its execution and at that point accesses data elements from the global namespace according to the interface specified for that node. Since the interface explicitly names precise elements from the global namespace which are to be used during the execution of that node, there is no means of distinguishing actual and formal parameters, so in effect parameters are hard-coded into nodes at compile time. The addition of sub-graphs into the language does not alleviate this problem, since sub-graph nodes themselves, and their contents, must have their parameters similarly hard-coded.

CODE is the only language under consideration that offers mechanisms (discussed later) which provide a true procedure-call abstraction, and as a consequence that supports recursion. Furthermore, unlike the other flow-based languages under consideration, CODE also permits implicit dependency relations to occur between nodes in a sub-graph. In short, what this means is that a nodes within the confines of a sub-graph may be involved in a dependency relation with other nodes, without arcs being present to signify the existence of that relation. The approach is clearly meant to improve the visual clarity, and thus scalability, of the notation whereby obvious dependencies, in CODE's case being shared memory abstractions, do not require the explicit addition of arcs to each node in a sub-graph.

Given that there are no features for procedure call, or for cyclic graphs to provide feedback in either HeNCE or ParADE, other means for supporting such activity were found. In particular, both HeNCE and ParADE supported the notion of special-purpose iterative graph nodes which could circumvent the normal graph semantics, within a strictly delimited area of the graph, in order to provide iterative behaviour. In the case of HeNCE, areas of a graph which are to be iterated, are delimited by special loop-begin and loop-end nodes, whilst in the case of ParADE, a special sub-graphing mechanism (the loop actor) is used which is capable of repeating the contents of a whole sub-graph.

3.2.3 Node

Nodes in a graph support the specification of the functionality of the parallel application. While a visual parallel programming language may provide a variety of different nodes, those that support user-defined computations are clearly the fundamental building blocks from which a parallel solution is built.

	Node Semantics		
	Type	**Purpose**	**State**
Options	Computational Graph management	Special General	Can be retained Cannot be retained

Table 3-4 Node Semantics

3.2.3.1. Node Discussion

It is convenient to distinguish two types of node: *computational nodes*, that permit a user to define computational behaviour; and *graph management nodes*, that are concerned with controlling aspects of the graph such as:

- Identify particular structures within a graph (e.g. such as node replication or parallelism); and
- Providing behaviour which would not be obtainable by using the general-purpose nodes (e.g. they may have different firing rules).

Nodes may provide special-purpose (i.e. predefined) behaviour or be general-purpose (user-programmable), and may use a variety of shapes for their representation, the differences in which are not important here.

All of the languages considered in this paper use a sequential programming language as the basis for the computational node. All permit the use of C or FORTRAN (or both). The CODE system provides a C-like language for node programming, although permits callout from a node to a C function.

The state issue for a node is a fundamental part of the computational model provided to the user. A node may be able to retain state between executions of that node or may not. The ability to retain state is useful in that it can avoid the overhead of re-obtaining identical input data needed for every execution (e.g. a node repeatedly executing with a varying input and a fixed input). In a cyclic graph the ability to retain state can be used to reduce the visual complexity of the graph in that a feedback arc is not needed. In acyclic graphs state retention can implement the feedback path which otherwise is not available by using arcs.

When executed, as dictated by the firing rule and tokens on incoming arcs, a node obtains input data from some other part of the parallel computation, processes that data using the normal features of the sequential language, generates data to be consumed elsewhere, and produces tokens on any outgoing arcs. Discussion of the constituent parts of a node (below) elaborates on how this behaviour can be achieved.

3.2.4 Arcs

Arcs are a fundamental building block of visual parallel programming languages and are not decomposable. As discussed earlier, graphs in the surveyed languages are directed; arcs therefore capture interdependency relationships between nodes.

	Arc Semantics					
	Direction	**Tokens Carried**	**Structure**	**Capacity**	**Consumption**	**Connect-ion**
Options	Supply Demand	Control Data value Reference	Single item Container	Single item Infinite	When used Explicit	1 : 1 N : M

Table 3-5 Arc Semantics

3.2.4.1. Arc Discussion

Though arcs are conceptually simple, there are in fact numerous subtle nuances which are worthy of further investigation. Several of the semantic options for arcs dictate the model of execution for the nodes in a graph, although detailed discussion of these models is best left to the section on node firing rules below.

The direction characteristic of an arc captures whether an arc is used as a supply route from a source node to a destination node that follow it, or as a demand route from the destination node back to the source node.

Arcs can carry tokens of various types: control information, data values, references to data values, or some combination of these. Arcs that carry just control information indicate an execution ordering relationship between two nodes. In effect an arc carries a signal indicating completion from the source node to the destination for supply arcs, or a signal which requests execution with demand arcs. The input data required by the receiving node must then be obtained through some other mechanism; this is discussed in the section below on node data inputs.

Arcs that carry data values are naturally used to carry data that has to be shared between nodes. Languages using supply data arcs are based on the dataflow model of parallel computation {Treleaven, Brownbridge et al. 1982} where the execution of a node is determined primarily by the availability of data on its input arcs. Thus data arcs can indicate execution order relationships as well as holding the data a node requires, and therefore carry control information implicitly. (This topic is returned to in the discussion of firing rules below.) Demand data arcs would be used by a destination node to signal the preceding node to commence generation of the required data.

Although none of the surveyed languages permit arcs to carry references, conceptually there is no reason preventing this type of token. References (or "pointers") would permit nodes to share data, thus avoiding the need to partition and copy data items around. Traditionally, references have not been supported on data arcs for two main reasons. Firstly, as languages have been targeted at parallel architectures that encompass distributed memory there would be some implementation difficulties in implementing the abstraction of sharing. A language that targeted just shared memory multiprocessor architectures would not suffer from this implementation problem. Secondly, if data is shared, concurrency-control mechanisms, such as those found in the shared memory abstractions of the CODE language, are required to ensure parallel accesses to that data are properly synchronised.

Data-arcs can be typed in that the language associates a data type with an arc and hence can provide traditional "compile-time" error detection on node interconnections.

A further issue with data arcs is whether they carry individual data values, or whether they support the transport of "containers" of values. Since most languages are concerned with FORTRAN-like programming languages and numerical applications, arrays are the only containers supported although one might imagine that support for C `structs` (i.e. records) could be appropriate when the C language was the textual programming language used in the language. Clearly many other container types exist (sets, bags, lists, etc.) but as these will depend on user-level programming conventions, it would be difficult for a language to be able to automatically transform and transport such containers. The limitation in practice of containers being only arrays is not therefore surprising.

Since "supply" arcs connect independent sources and destinations (whether carrying data or control), it may be possible for a source to "race ahead" of the consumer. The capacity of an arc is therefore an issue to be considered. If an arc can only hold a single item then additional synchronisation is implicit in the language using such an arc since a producer cannot generate a new item on an arc before the previous has been consumed. "Demand" arcs effectively carry only a single item and hence keep source and destination nodes in step. If arcs have an infinite capacity (limited by practical considerations of course) then the language potentially provides a more relaxed execution model with more parallelism opportunities. Note, however, a language that uses control arcs for node synchronisations coupled with some sharing mechanism for common data is likely to be

difficult to program if those arcs have a capacity for more than one completion signal, since there would be a difficulty in keeping the multiple incarnations in order as has been seen in the tagging issues encountered by the Manchester Dataflow Machine researchers {Gurd, Kirkham et al. 1985}.

Arcs normally provide 1:1 connections between nodes in a graph. However, there may be *N*-to-*M* relationships required between the nodes. For arcs carrying control tokens, a 1 to *N* arc effectively fires *N* nodes in parallel (e.g. to obtain data parallelism) while *N* to 1 arcs effectively provide a synchronisation point for *N* parallel nodes. For arcs carrying data tokens, a 1 to *N* arc may be used to provide the same data to *N* nodes for data parallelism, while an *N* to 1 arc would indicate the recombination of a number of separate data streams into one. Such behaviour could be provided by permitting *N*-to-*M* arcs. (Clearly, such arcs would be problematical for data arcs carrying different types of data.)

An alternative to *N*-to-*M* arcs which may be adopted in languages is to combine 1:1 arcs with special node types to split and merge arc contents, or to require the use of a general-purpose node which can be explicitly programmed with replicate or merge-like facilities.

Initially, one might question whether arcs other than 1:1 were justifiable, given that nodes can be used to achieve splitting and merging. Requiring the user to explicitly program a node to do splitting/merging, while straightforward, is perhaps not providing the right level of (visual) language support to the parallel programmer, particularly for the situation of arcs carrying data values. Here one would prefer the language to take the burden of managing the data flows; the use of special-purpose replicate/merge nodes goes some way to achieving this, and then the only argument in favour of *N*-to-*M* arcs is if they simplify the visual complexity of the graph as compared to that containing extra nodes.

Considering further the issue of managing the data together with the ability for an arc to carry containers such as arrays, other possibilities emerge. When applying parallel processing to arrays of data, it is common to split up those arrays into smaller units which can then be processed in parallel before being recombined back into the output data structure. Of course all of this data partitioning can be achieved in a general-purpose node, but that is an additional requirement on the programmer and adds additional complexity (and possible errors) to the code they have to generate. For example, splitting

a matrix into rows (or columns) usually requires additional code to permit the rows to be explicitly identified for subsequent recombination into a result matrix. In effect what is required is a 1:N arc with partitioning capabilities (splitting a structure up for parallel processing) and an N:1 arc with recombination capabilities.

In the languages under consideration, only the CODE language actually takes an approach which relies on 1:N and N:1 arcs to achieve this behaviour. Since CODE arcs are programmed with the sources and destinations of any data passing through them, it is quite feasible for either the sources or destinations to be the same node for many instances of the arc. For both the HeNCE and ParADE notations, arcs are somewhat simpler, and it is the existence of special purpose nodes which instantiate data decomposition and re-composition. The advantage of using special purpose nodes to distribute data as opposed to smart arcs, is that the nodes can be given helpful behaviours which support the user in data distribution and recombination.

3.2.4.2. Arc Examples

The CODE system with its dynamically instantiated arcs suffers from the fact that not all possible execution paths can be seen in its program graphs, but does allow a program to alter its structure (to the limited extent of adding or choosing not to add more or less of the same arcs specified in a program graph) at run time.

HeNCE arcs carry control only. Clearly, a language such as Hence that provides only control-information arcs has to provide some other means for the communication of data between interdependent nodes.

In all of the surveyed languages, both single values and arrays of values were permitted to travel between nodes. However, where the languages differed was in the support provided for decomposition of such structures to feed into parallel processes. Perhaps the best example came from the ParADE language, where an arc can be equipped with a decomposition template (from a variety supplied with the language) which automatically decomposes and distributes data from a structure to a number of parallel processes before recombining the results of those processes back into a single structure. The level of support provided by the other languages was significantly less than that provided by ParADE. Although it is possible to decompose, parallel process, and recompose data in the other languages, it is left to the developer to implement such schemes.

The CODE language in common with the ParADE language allows decomposition to be specified at the meta (i.e. graphical) level, though unlike the ParADE system there is no graphical support for doing such. Instead, arcs in the CODE system are given an index as they are instantiated at runtime, which may be used to identify units decomposed data to facilitate re-composition.

3.2.5 Node Syntactic Elements

In order to understand fully the behaviour of a node in a language, the following issues to be considered include:

- When a node is executed (firing rules);
- The data inputs - how a node obtains data from other parts of the application;
- The (textual) program within a node and how that interacts with the visual features;
- The data outputs that a node generates for subsequent processing; and
- The rule for the production of output tokens from a node.

	Composed of				
	Firing Rule	**Input Data**	**Node Program**	**Output Token Production Rule**	**Output Data**

Table 3-6 Node Composition

3.2.6 Node: Firing Rule

The firing rule specifies the conditions under which the execution of a node is commenced. In turn this may affect the synchronisation that will be needed between the parallel nodes.

3.2.6.1. Firing Rule Semantics

	Node: Firing Rule Semantics		
Options	No rule	Fixed	User-programmable

Table 3-7 Node Firing Rule Semantics Options

3.2.6.2. Firing Rule Discussion

As has been discussed, the arcs in a visual parallel programming language are directed and provide dependency information in a supply or demand direction. Thus one set of

semantics for firing rules is naturally based upon the arcs that are incident (i.e. directed towards) to the node and their contents. However, some nodes may have no firing rule, and instead commence their execution when the graph of which they are a part is "executed". One set of nodes for which this would be the appropriate behaviour is those which do not have any incident input arcs. Nodes which represent the starting point(s) for the execution of the application represented by a graph using "supply" arcs (or the end node for a graph using "demand" arcs), or those that provide an interface between a filing system and data to be processed by the application, are cases in point.

For nodes that have incident input arcs, a "no-rule" firing rule is less appropriate. A language that combined a no-firing-rule scheme with arcs carrying control information would not make any sense. A language that uses data arcs with the no-firing-rule is providing little more than a graphical interface to part of a message passing system for parallel programming (VPE is such a language), and would still requires the programmer to deal explicitly with synchronisation aspects (e.g. through message read operations). Thus most visual parallel programming languages have arc-based firing rules which simplify what the parallel programmer has to provide.

The options for firing a node based on arc contents, coupled with the arc semantics discussed earlier, give rise to node executions that follow a common classification {Treleaven, Brownbridge et al. 1982} of *data-driven*, *control-driven* or *demand-driven* firing rules. When incident arcs have:

- "Supply" + "data value" semantics, arc-based firing corresponds to the data-driven semantics, where a node can execute when its inputs are available;
- "Supply" + "control" semantics, arc-based firing naturally corresponds to the control-driven situation, indicating that a preceding node has completed its execution
- "Demand" semantics, arc-based firing would cause a node to fire when outputs from that node are needed. Demand-driven semantics give rise to a lazy form of execution.

In the most general case, a node may have multiple incoming arcs which have to be considered in the firing rule. Of particular importance is the rule that applies to computational nodes, since the programmer has to understand the rule in order to use

the system. The rule may be *fixed* by the language, the simple possibilities being that the node fires when *all* or *any* of the arcs contain values. (Other special-purpose nodes may break the fixed rules in order to provide different semantics.) Flexibility would be provided in a *user-programmable* scheme which would permit the programmer to specify the exact firing rules based on logical combinations of arc contents. While this flexibility appears on the surface to be desirable, in practice it means that the overall behaviour of the graph can only be discerned by examining all of the firing rules in detail. A language with fixed firing rules may therefore be easier to comprehend only from the graph level.

If a computational node can fire when only some of its incoming arcs contain values then the language has to provide some means for the program in the node to differentiate between the various cases that could arise - for example, to determine which data arcs do or do not contain valid values. This is clearly more complex than the "fire when all present" case.

The ability for a node to fire when only some inputs are present provides one form of conditional execution within a node in a graph. With the "fire when all present" rule the only means by which conditional execution in a graph can be achieved is through the conditional production of tokens on output arcs. This is discussed in the output token production rule section below.

3.2.7 Node: Inputs and Outputs

Interest here concerns the model that indicates how a node obtains data from other parts of the parallel computation, and how obtaining that data is supported by the language. Issues concerning the mapping of input data to variables in the (sequential) programming language used in the computational nodes is addressed in the next section (Node: Program).

3.2.7.1. Data Input Semantics

	Node: Data Input Semantics		
Options	**Copy**	**Shared**	
		Implicit	Explicit

Table 3-8 Node Input Semantics Options

3.2.7.2. Node: Inputs and Outputs Discussion

For parallel programming, the key issue concerning non-local data is the issue of copy versus shared - that is, whether a node receives copies of data values that it is to process, or whether the data is accessed from some shared area. Data copying suffers the

overhead of that copying, mitigated by the fact that a node can then access that data locally and without restriction. Shared data can avoid the copying overhead but may require locking overheads to be imposed to ensure orderly use of the shared space. Ideally, locking will be implicitly provided by the language in order to avoid the burden of requiring the programmer to deal explicitly with the locking issues. Of course, the parallel platform may itself impose further overheads on these semantics - for instance, to implement the abstraction of data sharing on a distributed memory machine.

For arcs that carry data, the most natural semantics are those of copying the data. For arcs that carry references, the sharing semantics follow. For arcs that just carry control information, the data inputs are an orthogonal issue that has to be supported somehow in the language, such as the in the node interfaces in HeNCE, supporting either copy or sharing semantics.

3.2.8 Node: Program

The node program is where the programmer specifies the computation that a particular computational node undertakes. If the language supports a standard sequential programming language, programs written can only deal with data through variables as supported in those languages. How such variables get mapped onto the data inputs that are provided to the computational node, and where any such mappings are specified are issues to be addressed here.

3.2.8.1. Node Program Semantics

	Node: Program			
Options	**Language**		**Variable mapping**	
	Proprietary	Standard	Implicit	Explicit

Table 3-9 Node Program Options

3.2.8.2. Node Program Discussion

Clearly a visual parallel programming language can provide a proprietary programming language with features that fit into its graphical framework, such as dealing with data inputs. Generally, such languages are likely to be sequential, as the graph-level is where the parallelism of the application is meant to be captured. More commonly, languages support traditional sequential languages such as C and FORTRAN, because the purpose of the language is to build upon programmer's strengths in those languages while implicitly handling the parallelism features (to a greater or lesser extent).

Programming the computational nodes, particularly in standard languages, will require the use of variables, and an issue therefore to be considered is how the variables in the node program are mapped onto the data inputs and outputs. Ideally, a language will provide implicit mappings from the data inputs (e.g. those arriving on arcs) to the variables in the node program. Alternatively the language could permit the mappings to be explicitly specified using some form of (proprietary) language such as the HeNCE scheme whereby local variables are mapped to a global namespace in the node's interface. Similar issues arise for data that a node has to generate for its data outputs.

In either scheme, textual and graphical elements of the application must be closely coupled. Conditional execution of nodes based upon arrival of tokens necessitates the construction of node programs which understand the node's interface, which is undesirable.

The issue concerning variable mappings also concerns what in traditional programming are called formal and actual parameters. Programming languages allow a procedure to have formal parameters that are mapped to actual parameters at run-time. This permits a procedure to be reused from a number of places. The same issue arises in languages - to permit a computational node to be programmed in a general manner and then reused, in a parallel application.

3.2.8.3. Node Program Examples

In HeNCE there is no mechanism for defining formal parameters for a node. Instead, the actual parameters are hard-coded into the node interface, and from those hard-coded parameters the appropriate variables are accessed from the global namespace.

In CODE, ParADE, and in some sense VPE too, there exist mechanisms for mapping actual onto formal parameters. In CODE, the formal parameters are presented by the number and type of data-flow arcs and the actual parameters by data values flowing along those arcs. A similar scheme is seen in ParADE, though ParADE automates the mapping between graph level parameters and those in the node program where CODE does not.

VPE provides formal parameters in the sense that its port nodes decouple intercommunicating processes. The code residing in a VPE compute node has an environment consisting of a textual interface to each of the port nodes attached to the compute node, and so in some sense has the notion of formal parameters. Actual

parameters in VPE are data values exchanged during the message-passing phase of a compute node's program.

3.2.9 Node: Output Token Production Rule

Given that the arcs between nodes indicate some kind of dependencies between those nodes, the options for the alternatives by which one node can control the generation of tokens on its outgoing arcs need to be considered.

3.2.9.1. Output Production Rule Semantics

	Node: Outputs						
Options	Arc Coverage			Explicit Send/Implicit Send		Streamed Output	
	All	Some	None	Implicit	Explicit	Yes	No

Table 3-10 Node Output Options

3.2.9.2. Output Production Rule Discussion

It is assumed that a computational node may be permitted to have multiple outgoing arcs, since a restriction on there only being one arc may be a limitation in a parallel application. However, some nodes may have no outgoing arcs, where such nodes act as sinks for data (e.g. storing results into a data file, or acting to terminate the execution), and other nodes may be permitted to output on N from M possible output arcs.

The language has to provide some means for indicating when output tokens can be generated. This could be explicit, via some special instruction used in the node program (such as an explicit send in a piece of VPE textual code). This could be implicit by the system taking action when a node program's execution has completed. Note that the former would permit a node to generate a stream of output tokens (e.g. to provide a pipeline structure) while the latter would not. Again, special-purpose nodes can provide different semantics in order to achieve the streaming behaviour.

3.2.9.3. Output Production Rule Examples

In the CODE language, routing rules specified as part of the node's compliment of stanzas, determine under which conditions the output arcs from the node will become active. As such, CODE allows for some arcs (i.e. *N:M*) to be activated. Sending is implicit in CODE since arcs simply appear as variables in the node program, and variable access is then equivalent to moving data onto an arc. Streaming from any one arc is permitted since a computation may contain a loop which continuously updates the variable that represents the output arc and thus continuously cause output on that arc.

HeNCE adopts an all arc output rule whereby control-flow signals from a predecessor node will reach all successors and instigate their activity. The sending of output tokens is implicit in that the completion of the node program causes the tokens to be produced, and since the node program can, by definition, complete only once, streaming is not permitted at the level of an individual node.

ParADE's actor node permits output on a subset of its output arcs for conditional execution. Sending is implicit in ParADE since arcs are bound to textual variables and any of those variables which are updated during the course of the execution of the node program are automatically sent along their output node when the node program finishes. Although streaming is supported by ParADE graphs, only the loop actor can instigate it since it possesses a per-iteration output semantic.

VPE also permits output along some arcs. Arcs in VPE represent possible routes of message-passing, not dependencies per-se and as such can be used as and when messages are to be passed in whatever combination is required at that time. Sending is explicit via message-passing calls in the textual node program (as is receiving). Streams are permitted through asynchronous sends.

3.2.10 Support for Modes of Parallel Execution

A language must, by definition, provide means for expressing parallel activity. Some of these, such as pipelining, have already been encountered in the discussion whilst others, such as task parallelism, have been alluded to. Each of the languages under examination provide a mechanisms or patterns of mechanisms which support parallel activity. The functionality of such mechanisms is the concern of this section.

3.2.10.1. Parallel Execution Mechanisms

Behaviour	Parade	Hence	Code	VPE
Data parallel	Depth Actor	Fan-in/out	Node stanza	Replicate Box
Task Parallel	Yes	Yes	Yes	Yes
Pipeline	Loop	Pipeline begin/end nodes	Repeated output generation from a Node	Using asynchronous sends in node program
Iteration	Loop actor	Loop begin/end	Cyclic graphs	No graph-level iteration
Conditional	Conditional output generation from actor.	Conditional begin/end nodes	Node stanza	No graph-level conditional construct.
Recursion	No	No	Yes	Yes

Table 3-11 Parallel Execution Mechanism Options

3.2.10.2. Parallel Execution Mechanisms Discussion

Whether the means of extracting parallel activity are in the form of patterns, and thus supporting implicitly parallel activity like the textual UFO language {Sargeant 1993}, or those means are provided through explicit language features, has an effect on the way the developer builds a parallel application. In the languages under scrutiny here, mechanisms for explicitly instigating parallelism and patterns for implicitly exploiting parallelism are present.

Given that it is preferable for parallel activity to be implicit within the structure of an application, rather than to have the developer explicitly identifying parallelism, the main issue for the languages is to what degree they encourage the construction of naturally parallel applications (and indirectly the suitability of the underlying execution models), and what degree of support they provide when they require the developer to explicitly invoke parallel computation.

In general, the languages only support task parallelism implicitly, whereby computational elements which are not directly interconnected may execute independently. This is rather an odd paradox, since the data-centric execution models supported tend to influence the developer into building data-parallel applications, whose construction necessitates explicit programming with special languages constructs. Where special language constructs for data parallelism are provided, the provision of features to deploy those constructs is valuable. In common with data-parallelism, pipelining is also

explicitly programmed, though features to aid deployment are of less importance since execution is more of a known quantity at compile-time.

In addition to task, pipeline, and data parallelism, parallel languages have previously supported parallel activity from logically distinct loop iterations (such as in HPF), and have exploited parallelism from recursive operations. Support for iterative style parallel activity is lacking in these languages since data parallel mechanisms are used to achieve the same affect. Furthermore, it has been reasoned by Allen that iteration should be a purely sequential construct in a visual parallel programming language, and that it is a parallel construct in textual languages only because of the limitations imposed by textual syntax {Allen 1998}. Parallelism based upon individual instantiations of a recursive problem is permitted in some of the languages (those that support formal and actual parameters and calls between graphs), though exploitation of such parallelism is left to the developer, much as it would have been in a textual language.

3.2.10.3. Parallel Execution Mechanisms Examples

The CODE language provides an explicit form of instantiating data-parallel activity through the use of "arc topology specifications" which allow a node to communicate with a run-time determined number of successors or predecessors, and to identify which nodes are communicating via the index of the arc through which data is being passed. Being text-based, the developer has little support in deploying data-parallelism and furthermore must bear the burden of cross-referencing graph- and text-level objects. Conversely, task parallelism is implicit within the structure of an application and requires no explicit developer input, other than the requirement that developers express solutions without introducing unnecessary dependencies into the graphs. Pipelines (and iterative behaviour) are achieved through a combination of intricate node interfaces and feedback loops which can be used to trigger the multiple execution of a node. Recursion is supported through the graph call node which can be used to call the current graph. Since computational nodes execute in parallel with other computational nodes, recursion is naturally parallel in CODE.

Where CODE uses combinations of textual meta-programming and graphics, HeNCE uses only graphical components to initiate parallel activity. Areas of data-parallel activity are delimited by special purpose nodes which may take values at run-time to determine the level of replication for the contained computations. Pipelined parallelism is also delimited by special purpose nodes, but there is no need to specify any further

control metrics, since they are implicit within the structure of the pipeline. Task parallelism is, once again, implicit within the structure of the application. Iterative parallelism is not supported, though there is a graphical mechanism for supporting sequential iterative behaviour. As HeNCE does not support sub-graphing (let alone formal and actual parameters) recursion is not possible.

ParADE takes a similar approach to HeNCE in that parallelism is considered purely graphically. Data parallelism is achieved through the depth actor, which also takes responsibility for decomposing data into pieces for each parallel instance of the actor to work on, and for recompiling the results from those actors into a single result structure. The process of manipulating the data for the parallel actors is specified graphically, whereby the user selects from a number of templates when constructing the application, which are then applied to the partitioning of data at run-time. Whilst this is conceptually no different to the approach taken by the HeNCE and CODE data parallel mechanisms, ParADE's approach significantly simplifies implementation, and thus eases the developer's job. Pipeline parallel activity is supported through the loop actor (as is straightforward iteration), though it is known that pipelining in current versions of ParADE leads to non-determinacy. Task parallelism is implicit within the structure of an application, though recursion is not supported since ParADE does not provide a mechanism whereby a named graph can be called.

Since it supports an explicit message passing, as opposed to flow-based, approach, VPE is somewhat different to the other languages under consideration here. Data parallelism is supported in a manner which resembles elements from both CODE and ParADE. In VPE, a specific graphical notation is used to denote the potential replication of a computation (like ParADE's depth actor), yet communication with instances of that computation are identified through indices (like CODE's arc topology specifications). The result is that at the graphical level, the developer can specify a potentially parallel computation, but then must invoke that parallel computation with appropriate instructions at the textual level. However, unlike other languages both task and pipeline parallelism, are implicit within the structure of the application. Iteration is not supported at the graph level, though it is permissible within the textual node program. Similarly recursion is not supported at the graph level (and thus cannot be exploited in parallel), though sub-graphing is.

3.3 Development Paradigm versus Execution Model

In software development, abstraction is the key element in managing the complexity of both problem domain and implementation. In addition to the syntactic abstractions outlined in the taxonomy, a common strategy to leverage the benefits provided by abstraction, is to utilise a development paradigm whose semantics are close to that of the problem domain. That is, to use a development method whose features and vocabulary either suit the problem domain at hand, or which can be extended to suit that problem domain. Although there may ultimately be constraints from the underlying hardware architecture (such as number of processors or interconnection speed), the development paradigm remains an expression of the problem, and not of machine-level activity. The development paradigm interfaces to an execution model, which provides an abstraction of the underlying computing system through programming language syntax and semantics. The challenge in developing paradigms for software development lies in the fact that the development paradigms and execution models must work in unison to support the application developer and the application. Each must provide an appropriate abstraction: the execution model to the development paradigm, and the development paradigm to the developer, as can be seen in Figure 3-3 below.

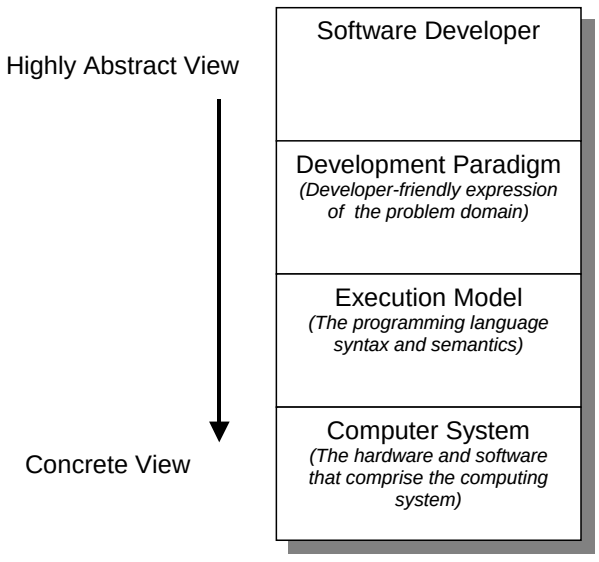


Figure 3-3 Layers of Abstraction in a Software System

In the absence of a layered model of abstraction, software developers may be unnecessarily exposed to inappropriate detail of a computing system. It is self-evident that if either of the development paradigm or execution model is at an inappropriate level of abstraction, the software system as a whole will suffer. This was seen in the

current generation of visual parallel programming languages where machine-level execution models were allowed to permeate through the layers of abstraction to the development paradigm.

Reflecting upon the taxonomy, it is apparent that there is no distinction made between execution model and development paradigm in any of the languages surveyed. When using any of those languages, the developer's view of the software system (as presented by the development paradigm) and the manner in which processing proceeds (as dictated by the execution model), are similar in each case. Each of the development paradigms adopted by the languages resemble execution models normally associated with lower-level level parallel activity. The programming abstraction utilised consists of values passing between (parallel) procedure calls, which in effect gives a semantic not dissimilar to processes and inter-process communication. Although each of the languages abstracts low-level architectural detail, in that locking and synchronisation primitives are implicitly handled, the fact remains that the user is exposed to behavioural patterns which have permeated from the computing system level to the development paradigm level.

The fact that system aspects, like processes, are exposed to the developer cannot but lead the developer into a system- rather than problem domain-centric mode of development. The result is that developers are left ill-equipped to deal with potentially complex problem domains, since their only vocabulary for conceptualising problems is biased towards computational rather than cognitive activity. The misuse of low-level execution models as high-level development paradigms increases the gap between problem-domain and computerised solution. If a low-level execution model is the only paradigm through which the problem domain can be conceptualised, the developer is forced to bridge a large gap between problem domain and implementation of a solution. This results in software whose construction, debugging, porting, and maintenance will all be hindered due to source-code complexity. In short, the problem with visual parallel programming languages is that whilst they support an abstract mode of implementation (through a graphical syntax) they do not offer sufficiently powerful paradigms through which problems may be conceptualised – their programming models and development paradigms are simply too similar.

The problem of providing suitable development paradigms and execution models is not restricted to visual parallel programming. For example, in a typical sequential-software project, it is likely that a methodology such as object-orientation will underpin

development. The object-oriented paradigm provides a suitably abstract development paradigm (through interrelated and intercommunicating objects), though software implementation is based upon textual annotations. Since the implementation of object-oriented software is intricate, the execution model must be simple enough such that overall complexity can be managed. This is reflected in the fact that most object-oriented software is built for a sequential rather than parallel execution.

When building parallel application with textual object-oriented languages, the developer must bear the intellectual effort of understanding multiple parallel flows of control, from an inherently sequential-looking piece of textual code. Coupled with the fact that parallel applications demand longer development and testing periods, a pattern emerges that not only are parallel applications difficult to build, but are also expensive.

Contrasting the problems faced by both the object-oriented developer and the visual parallel programmer when working on similar software products, it is clear that they face two rather orthogonal problems. The visual parallel programmer struggles to represent the problem domain since the available vocabulary is limited to that of parallel execution models. The object-oriented developer has no such problem-domain difficulties since the development paradigm provides abstractions which are extensible. In a parallel programming environment, the object-oriented developer will struggle with implementation issues since the textual style of the programming language does not lend itself particularly well to managing parallel activity, whilst the visual parallel programmer will not. The essence of the problem is therefore that visual parallel programming has a very low level of abstraction in its development paradigms, whilst offering quite suitable methods for implementation, whereas object-oriented (textual) programming may offer highly abstract execution models, but provide the user with little in the way of a helpful means of implementing (parallel) code. This problem is summarised in Figure 3-4 below.

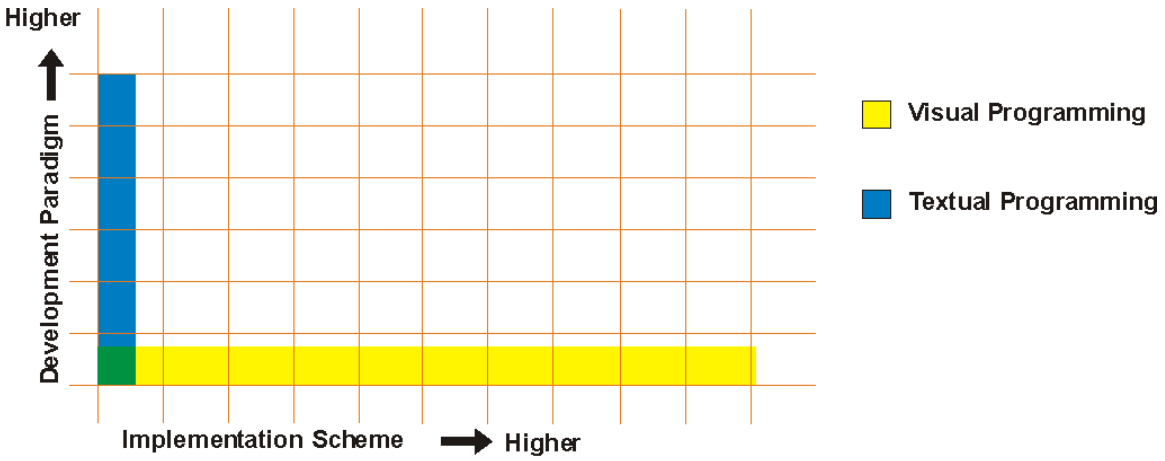


Figure 3-4 The Orthogonal Abstraction Problem

The Orthogonal Abstraction Problem {Webber 1998; Webber 2000} shown in Figure 3-4 indicates the primary difficulty that lies between the parallel programming community and the mainstream of software engineering. If the overall level of abstraction provided by a particular implementation style is given by the area covered on the plane, and the horizontal and vertical dimensions give the "strength" of the execution model (the power of the source code syntax and semantics) and the development paradigm (the power of the software methodology as a whole) respectively, then a pattern of abstractions emerges. The textual programming languages in common use have tended to provide highly abstract models of computation, through which problems can be conceptualised and managed, with object-orientation providing the richest and most extensible set of abstractions of any methodology to-date. Although implementation may be based on intricate textual syntax, the actual implementation complexity, if free of parallel activity, is manageable. The reciprocal problem arises when considering visual parallel programming languages which successfully abstract the complexities of the underlying (parallel) system, and offer a development paradigm which is very well suited to parallel programming. However, visual parallel programming languages do not offer a sufficiently abstract execution model to warrant their uptake in more general software engineering, since their execution models reflect those of parallel machines, and not of typical problem domains (whilst the converse is true for typical textual programming). Visual parallel programming to-date is simply underpowered to the extent where its use is not feasible for general-purpose application development, and likewise standard textual programming techniques are underpowered for developing parallel applications.

3.4 Requirements for Visual Parallel Programming Languages

Given that the previous discussion suggests that contemporary visual parallel programming languages are not well suited to developing general-purpose parallel applications, and that there is a clear and increasing need for such applications {Webber 1998}, there is a distinct requirement to capture exactly what is required of languages to be considered successful. This section introduces a number of requirements for future visual parallel programming methodologies.

- *Support Abstract Means of Implementation.* Utilising abstract programming language syntax (such as a visual syntax) reduces the implementation complexity of parallel software. Since contemporary visual parallel programming languages have succeeded in encapsulating the details of parallel architectures, future languages should continue likewise. In short, languages should be visual.

- *Support Abstract Development Paradigms.* To facilitate the construction of general classes of applications, support for conceptualising and managing the complexity of arbitrary problem domains is required. The paradigm through which problem domains are conceptualised should support detailed analysis and both high-level and unit-level design, as is supported by object-orientation.

- *Maximise Potential Parallelism.* Whilst the developer should not generally be concerned with explicitly instigating parallel activity, the amount of potential parallelism available within the source code should be maximised. As a consequence, the low-level data-oriented view taken by contemporary visual parallel programming tools is to be avoided, and higher-level models of parallel activity should be embraced. The implication of this is that the language should implicitly enable the identification of as much potential parallelism as possible, and it is the responsibility of some other mechanism to ensure that such (logical) parallelism is mapped efficiently onto the available (physical) hardware.

- *Encourage Re-Use.* Though visual parallel programming simplifies implementation, source code for parallel applications remains intricate. To reduce the potential for introducing bugs into

applications through the re-implementation of code, visual parallel programming languages should encourage the re-use of existing, tested code. However, future visual parallel programming languages should not adopt the black-box reuse mechanism favoured by the surveyed languages. Thought the procedural abstraction underpinning most of the contemporary efforts provides a basic mechanism for re-using a library of procedures, it is clear that software engineering best practice has moved on. Furthermore, modern software paradigms such as object-orientation allow re-use much earlier in the software lifecycle through, for example, re-use of types at design-time, which cannot be supported by existing means.

- *Use Appropriate Implementation Methods.* Throughout the visual languages community there is the implicit assumption that visual is best. For representing parallel activity, a visual notation has certain strengths which cannot be matched by a textual language. However, there are times in the development of an application where algorithms are sequential, and would be better specified textually rather than visually. In such cases, the developer should be allowed to use a textual representation for implementation purposes, and that code should fit into the overall framework provided by the visual element of the language. In effect, the visual component of the language *may* be used to coordinate the activity of textual computations, in a similar fashion to the textual LINDA coordination language {Carriero and Gelernter 1989}.

3.5 Flow-Based Visual Parallel Programming

Although this thesis has identified a number of problems with contemporary visual parallel programming languages as a solution to the problem of building general-purpose parallel applications, it is not the case that existing approaches are entirely without merit. Indeed, it has been argued that aspects of current visual parallel programming technology are in fact extremely useful.

Given the track record of visual parallel programming, which has predominantly been flow-based, it is a logical step to examine the benefits afforded by flows. Flow-based visual programming is based around the notion that flows of information between

computational elements, either in the form of data or control-flow signals, can be used to govern the overall execution path of an application. For example, in a dataflow language such as CODE {Newton 1993}, it is the arrival of data at a computational element which triggers the activation of that element. In HeNCE {Browne, Hyder et al. 1995}, it is the arrival of control-flow signals from prior computational elements which acts as the stimulus for invoking subsequent elements. Regardless of whether control- or data-flows are used, the commonality between the two is that both implicitly manage timing and synchronisation issues within an application, which is indeed a very desirable property for a parallel language to exhibit.

In addition to the fact that flow-based computing is a natural method of scheduling the interaction between concurrent tasks, it also provides the benefit that identifying potential parallelism from the overall communication/synchronisation structure of an application is straightforward. For example, tasks that are mutually independent with respect to either data or control-flow are free to execute in parallel, and are straightforward to identify as they do not exhibit any direct interrelationships, as can be seen in Figure 3-5.

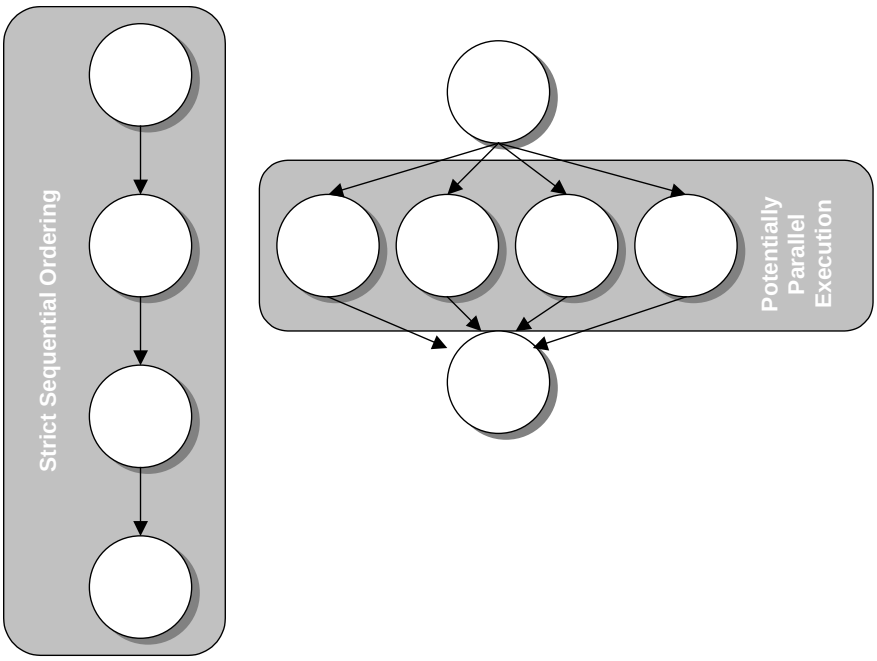


Figure 3-5 Exploiting Natural Parallelism in an Algorithm

Flow-based visual parallel programming in its current forms is by no means a comprehensive solution to the problems inherent within developing general-purpose parallel applications. Building applications with contemporary visual parallel programming languages remains difficult. Contemporary systems offer little more than a

means of transporting values between concurrent processes as a development paradigm, which is distinctly underpowered. Undoubtedly the value-passing approach is not without merit, though its overall worth is nevertheless limited when considering problems outside of traditional numerical applications – a fact which is acknowledged by the language developers {Newton 1993; Beguelin, Dongarra et al. 1994}.

Though visual parallel programming is in its infancy, the fact remains that flows are a very natural way of representing dependencies in a parallel application. This thesis conjectures that flows are fundamental, and that a new paradigm for visual parallel programming should adopt a flow-based approach as its central tenet.

3.6 Object-Oriented Development

Object-oriented development is one of the most powerful and widely adopted software paradigms in use today. The power of the object-oriented paradigm is that it supports the developer with a rich framework for tacking problem domains, and an extensible toolset for designing solutions and implementing applications.

As a concept, object-orientation is as simple as it is elegant, viewing a software system as a group of intercommunicating objects which offer certain services in order to complete computational tasks. What is more, the objects within an object-oriented software system may be of particular types to suit the problem domain at hand and it is quite usual for those types to be interrelated. Herein lies the primary benefit of the object-oriented approach:

> … in managing complexity and hence improving productivity in the software development process {Winbald, Edwards et al. 1990}.

In essence, object-orientation allows a problem domain to be conceptualised in its own terms, which are then propagated into the design and implementation of software product. In addition, the object-oriented methodology encourages the construction of highly modular designs and software which provide benefits in terms of reuse and maintenance.

Though the opinions of software engineering researchers and software practitioners are coherent in their belief that object-orientation is an important and powerful software paradigm {Coad and Yourdon 1991; Coad and Yourdon 1991; Jacobson, Christerson et al. 1992; Coad and Nicola 1993; Booch 1994; Booch 1996}, its uptake in the field of parallel application development has been slow. Whilst it may seem surprising that

benefits of applying object-orientation has not penetrated further into the field of parallel programming, reviewing literature reveals that the overwhelming majority of applications written are numerical in nature, and low in terms of overall functionality. As such the consensus seems to be that advanced software engineering techniques are unnecessary in a field which still maintains from its very name to be a purely implementation-oriented discipline.

Clearly parallel computing and object-oriented development have met beforehand. Indeed, there have been some notable attempts to merge object-oriented technology with parallel computing technologies {Yonezawa and Tokoro 1987; Annot and Haan 1990; Treleaven 1990; Kale and Krishnan 1993; Geist, Gropp et al. 1996}, current parallel programmers remain seemingly steadfast in their opinion that the application of object-oriented technology is superfluous to their needs, and this is reflected in the tool support that has attained widespread acceptance within the community (PVM, MPI, and so forth). It may simply be the case that the problems tackled by parallel programmers are very well understood, and that applying advanced software development methodologies would simply be overkill. Furthermore, the perceived overheads of developing with an object-oriented language over a purely procedural language are often considered unacceptable, in environments where every CPU cycle is deemed precious[2].

Although the parallel programming mainstream may have rejected object-orientation, it does not mean that the technology has nothing to offer the field. On the contrary, as parallel computing technologies move further into the realm of general-purpose computing and the cost of parallel computing platforms at the commodity end of the market continues to decrease, it is no longer unthinkable that future practice will become quite different to current. However, the kind of applications that the general-purpose computing market requires differs greatly from the style of current parallel applications. A typical productivity or entertainment application for a commodity parallel system will be large, complex, and functionality-rich, and it is precisely here that object-orientation has the potential to provide significant benefits.

Whilst object-orientation wholly endorses the notion of abstraction for managing complexity, the common (textual) object-oriented programming languages are deficient

[2] Ironically the lack of proper engineering of code may cost more cycles in the long run, since software may be crudely built using low-level tools and not be sufficiently sophisticated to take full advantage of the computing system as a whole. Concentrating on low-level details may simply detract from the big picture.

in terms of facilities for managing parallel activity. In terms of the orthogonal abstraction problem of Figure 3-4, they are lacking in terms of a suitable execution model. If the current popular object-oriented programming languages such as C++ {Stroustrup 1991; Stroustrup 1997} and Java {Flanagan 1997; Flanagan 1997; Morrison 1997; Microsoft 1998} are considered, it is clear that the concurrency features of those languages often seem to be secondary issues. Generally the developer is presented with models which fit uncomfortably with objects, such as threading, and is left with the burden of effectively managing objects and threads – proverbial chalk and cheese. This is not surprising, in that most object-oriented languages were not designed with parallelism in mind. What is surprising is that even those languages which were built to address parallelism {Yonezawa and Tokoro 1987; Annot and Haan 1990; Kale and Krishnan 1993; Bik and Gannon 1997} still place the burden of explicitly managing parallel control-flows upon the developer. Furthermore, the mixture of control-flow code and computational code means that the source code base may become highly intricate and difficult to maintain and debug. As any practitioner in parallel programming will concede, maintaining a clear view of program behaviour from textual source code requires significant intellectual effort. This is all in stark contrast to the visual flow-based languages which support parallel activity with comparative ease, through a separation of control and computation aspects, and choosing the most appropriate type of syntax, visual or textual, for each.

Despite the fact that textual object-oriented languages have not been widely adopted by the parallel programming community, the concepts that underpin object-oriented technology are sound. Moreover, language developers are beginning to take an interest in at least making concurrency control mechanisms a standard part of their languages {Flanagan 1997; Flanagan 1997; Morrison 1997; Microsoft 1998}. Whilst this is a far cry from fully-fledged parallelism aware languages, it is some measure of the recognition of the growing importance of concurrent execution within applications. Given that the object-oriented paradigm is valuable, what remains is to deduce the best way of combining object-oriented and visual philosophies within the context of parallel application development to harness the key strengths of both, and this is the subject of the next chapter.

3.7 Summary

This chapter has provided a taxonomy for visual parallel programming languages, and the results of that taxonomy were used to identify weaknesses in the current generation

of visual parallel programming languages. In particular the problems associated with closely coupled development paradigms and execution models were highlighted, and from the identification of that problem, a set of requirements for future visual parallel programming languages was derived.

Provisions to satisfy the identified requirements were examined, and in particular flow-based programming and object-orientation were introduced as a combined solution. In subsequent chapters a paradigm based upon visual, object-oriented, and flow-based programming is presented, and a language supporting that paradigm is developed and evaluated.

Chapter 4 A New Paradigm and Language for Visual Parallel Programming.

Visual parallel programming is a technology still in its infancy and it is clear that current systems are somewhat lacking in support for the software engineering lifecycle. As general-purpose applications begin to demand increases in hardware performance, it is unlikely that the current generation of visual parallel programming languages will provide a reasonable solution to building such functionality-rich applications.

In order to meet the need for faster and more functional software products, visual parallel programming must be endowed with better abstractions in terms of cognitive and execution models. Paradigms must be developed and languages to support those paradigms must be built and tested to explore solutions to the problem of engineering parallel software.

4.1 Introduction

The single most important fact that previous chapters have revealed is that the current generation of visual parallel programming languages fails to support the developer throughout the software engineering process. It is true that current languages fare a little better in terms of helping the actual implementation of parallel applications, but nonetheless if there is no support for software engineering then, simply put, software cannot be *engineered*.

In order to identify engineering that could be applicable to the discipline of visual parallel programming, it is necessary to look to mainstream software engineering for inspiration. In the mainstream, it is clear that the object-oriented paradigm has risen to prominence above all others since it alone provides a sufficiently rich and extensible set of abstractions to allow arbitrary problem domains to be managed.

This chapter begins by presenting a new paradigm for parallel application development based upon the combination of object-orientation together with the more useful aspects of current visual parallel programming technology. This paradigm, *Parallel Object-Flow*, is then used as a basis for the development of a new visual language to enable the development (by visual means) of general-purpose (object-oriented), high performance (parallel) applications. The syntax and semantics of the language are

introduced in this chapter as are its development paradigm and executions models, covering both implementation issues and software engineering aspects.

4.2 Parallel Object-Flow: A New Paradigm for Parallel Application Development

The creation of a new paradigm for application development is non-trivial. However, prior work suggests the following as a basis:

- The application of object-orientation will provide a familiar cognitive model and development paradigm;
- The application of visual programming notations can be used to abstract details of parallel computing systems;
- Flow-based languages support the straightforward identification and exploitation of parallelism, and provide the additional benefit of implicitly synchronising concurrent tasks.

It is these three notions that form the essence of the Parallel Object-Flow paradigm. This paradigm maintains the view of a parallel application as being composed from a set of intercommunicating objects, where each object is a potentially parallel entity. Furthermore objects encapsulate methods which may exhibit parallel activity, where such parallel activity is obtained by constructing those methods in a visual, flow-based language. The use of object-orientation empowers Parallel Object-Flow with a powerful development paradigm which provides facilities for the management of complexity throughout the software lifecycle, whilst the flow aspects of the paradigm enable high-performance through an inherently parallel model of computation.

The way in which a type's methods are specified in a Parallel Object-Flow application differs significantly from the imperative style which traditional object-oriented languages have adopted, in order to provide a syntax which both abstracts underlying parallel systems and which is amenable to parallelism. To achieve this, the most successful aspects of the current generation of visual parallel programming languages (flows between computational elements to implicitly support parallel activity, and visual syntax to abstract architectural complexity) are used to build larger abstractions (classes). Specifically, classes are composed of attributes and methods, but those methods are implemented using visual flow-based notations, not textual source code.

The fact that distinct paradigms for the development (object-orientation) and execution model (flow-based graphs) are available is of importance, since they solve the orthogonal abstraction problem introduced previously. Although approaches based upon a mixture of objects and flows, such as the object-flow execution model of Prograph language {Cox, Giles et al. 1995} have been investigated within the serial programming arena, no similar approach has as yet been applied to parallel computing. The premise of the work described in this chapter is that the application of a hybrid of flow and object paradigms to parallel computing is novel, and will yield benefits in terms of both software engineering and parallelism.

4.2.1 The Parallel Object-Flow Development Paradigm

The fundamental abstraction of the Parallel Object-Flow development paradigm is the *class*. During the analysis and high-level design stages of the software lifecycle, the developer is concerned with identifying classes and capturing class interrelationships from the problem domain. In that sense, Parallel Object-Flow is no different to the object-oriented methodology. At the implementation phase, Parallel Object-Flow applications diverge from standard object-oriented languages. The primary differences lie in the facts that

- Parallel Object-Flow applications are constructed from an intercommunicating set of *autonomous* objects, and
- each method in a class is specified by a graph.

When constructing a Parallel Object-Flow application, creating such graphs to represent the computational activity of the program becomes the developer's main workload. Graphs themselves consist of nodes and arcs, where nodes represent some form of computational element and arcs describe dependencies between nodes. In addition arcs constitute routes via which parameters may pass from one node to another, rather like the dataflow languages CODE and ParADE. However, the parameters passed between nodes in Parallel Object-Flow graphs are unlike parameters in other visual parallel programming systems, in that they are in fact simply pointers to objects (called object handles) stored somewhere within the computing system. The actual detail of where objects are stored is of no direct concern to the programmer who deals only with object handles within a physically shared memory or perhaps in a logically shared memory {Keleher, Cox et al. 1994; Protic, Tomasevic et al. 1998; Watson and

Parastatidis 1999; Watson and Parastatidis 1999}, or handles to null objects (which are indicative of a temporal, rather than data, dependency).

Nodes (computational elements) represent either computation local to the current method, or method calls on other objects. Nodes act as an interface through which parameters (object handles) are passed to textual subroutines (local computation), or to other graphs within the application (method call).

A typical Parallel Object-Flow method graph will resemble that of Figure 4-1 below.

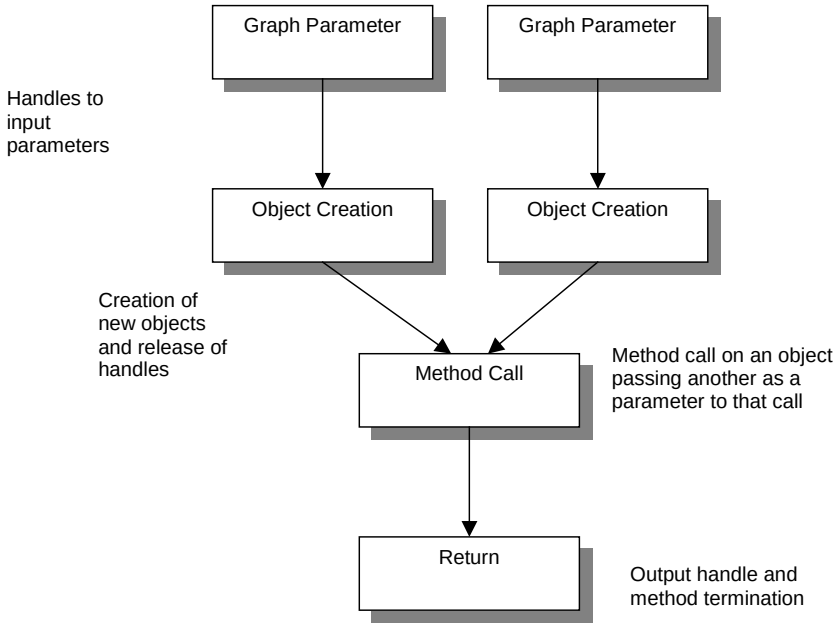


Figure 4-1 A Parallel Object-Flow Graph

Although the overall style of the graph in Figure 4-1 resembles that of contemporary visual dataflow languages, there are a number of differences.

1. Object handles rather than copies of values flow between nodes.
2. Since object handles travel around Parallel Object-Flow graphs, multiple nodes could attempt to invoke methods on the same object concurrently[3]. Any language based upon the Parallel Object-Flow paradigm should ensure that concurrency control mechanisms are provided to prevent undesired concurrent accesses.

[3] In the dataflow paradigm concurrency control is not an issue since the semantic is one of copies of values passing between computational elements. This is potentially costly, since it is generally not more efficient to pass a copy of an object than to pass a handle to that object, especially on a parallel platform where an interconnection network may be involved in the transfer.

3. The arrival of the complete set of incoming handles triggers the execution of a node, in the style of classic dataflow computation. For the sake of elegance and simplicity no other firing rules are permitted. This mode of execution ensures that what the developer specifies on the graph happens at run-time – there are no situations where the firing of a node is unclear.

4. For each set of input handles received by a node, there is only one output handle produced. There is no possibility for a single node to generate a "stream" of output object handles. A clean "single shot" model of computation at the graph level is thus provided.

5. Handles in-transit to a node are queued until the destination node is able to consume them. Once handles are consumed by a node, they are removed from the application as whole. If a particular object is to remain accessible from a graph, it must remain referenced elsewhere in the application.

6. Where no handles to an object exist on any graph within an application, that object is "garbage" and can be removed by a garbage collecting mechanism.

7. Since the Parallel Object-Flow paradigm supports the use of a mixture of textual and visual implementation styles, the automatic integration of graph-level and objects in textual subroutines must be fully supported.

4.2.2 The Parallel Object-Flow Execution Model

Where the goal of a development paradigm is to present the application developer with a rich cognitive model within which to express solutions, the execution model supporting the application at run-time is not of immediate concern to the developer. In essence the execution model is the intermediate step between the semantics of the programming language and the semantics of the underlying computing system (run-time, operating system, and hardware).

Given that applications will be composed from objects, the most pertinent place to begin the discussion of the Parallel Object-Flow execution model is with an individual

object. At the execution model level, the requirements for an object in the Parallel Object-Flow paradigm are twofold:

- The internal data structures of an object must not become corrupted through potentially concurrent access;
- Objects must not constitute a performance bottleneck.

These requirements clearly are not mutually compatible, and in order to accommodate both into the execution model, some compromise is necessary. In the Parallel Object-Flow paradigm, a multiple readers – single writer protocol derived from the work on Actors {Agha and Hewitt 1987} and Active Objects {Lavender and Schmidt 1996} is used to mediate access to objects. Using this protocol, an object may simultaneously service multiple methods which do not update the state of that object (reader), whilst serialising any requests for service which may update the state of the object (writer, which may also be a reader). An example of such activity can be seen in Figure 4-2 below.

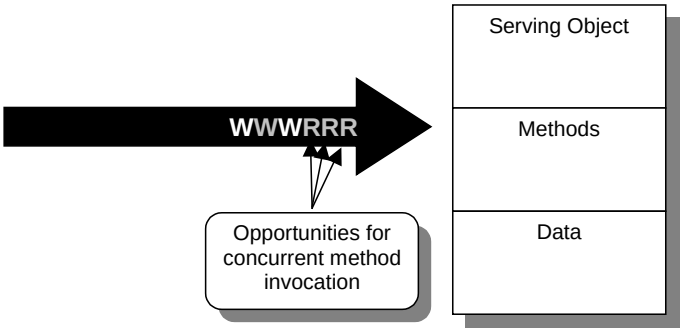


Figure 4-2 Multiple Readers, Single Writer Method Invocation

Figure 4-2 demonstrates a hypothetical situation where the object is requested to service a number of methods. In this example, it receives requests for three read-only methods which it can handle in parallel, and three write methods which the object serialises after completing the read-only methods to ensure consistency.

In addition to the multiple readers / single writers protocol, the Parallel Object-Flow paradigm supports parallel activity across multiple objects. Since an application will typically comprise a number of objects, it is clear that multiple objects could potentially execute (multiple) methods at any one time. The Parallel Object-Flow paradigm allows methods themselves to be parallel routines, specified in a visual, parallelism-amenable syntax. The execution model for a Parallel Object-Flow application is therefore

inherently parallel, where parallel method code, may call multiple (parallel) methods on multiple objects. This is illustrated in Figure 4-3.

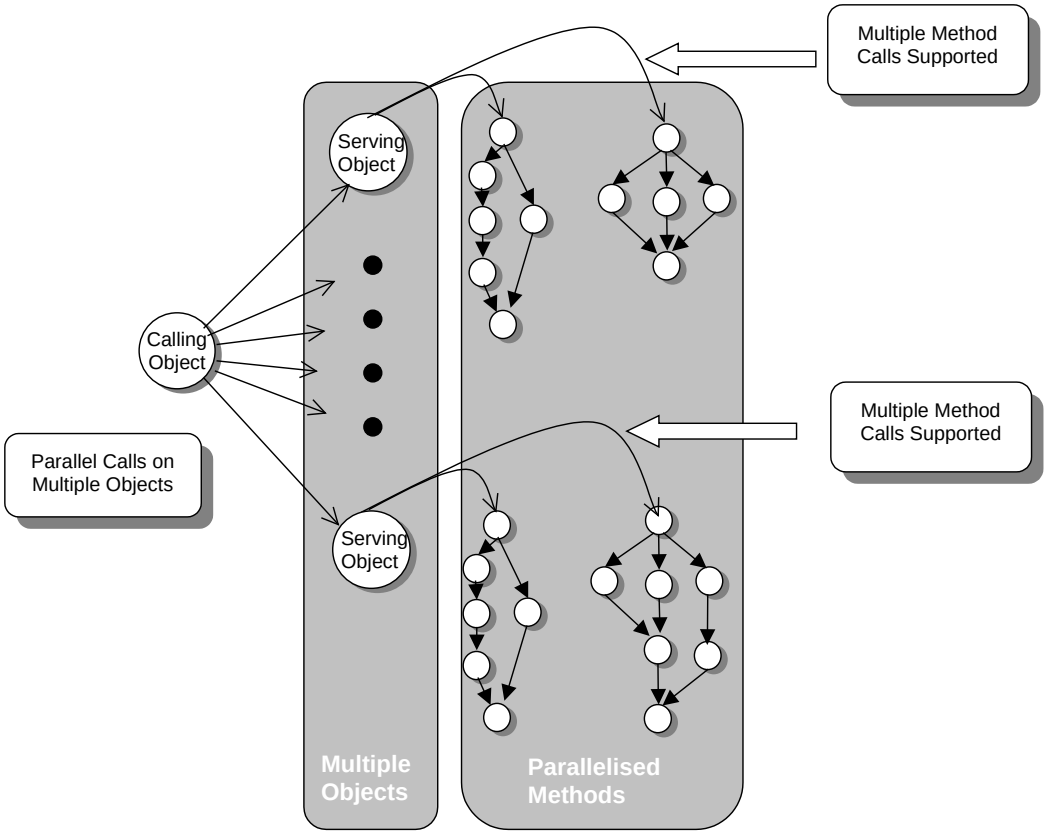


Figure 4-3 Exploiting Parallelism in a Parallel Object-Flow Application

For the current generation of visual parallel programming languages, exploiting potentially high levels of parallelism shown in Figure 4-3 could be seen as troublesome. Where most current languages have adopted a rigid mapping of logical to physical parallelism (what appears as a parallel task on a graph will be parallel at run-time), the Parallel Object-Flow paradigm adopts a more subtle approach in that logically parallel tasks may not necessarily be executed concurrently at run-time. In fact, tasks will only be executed concurrently if there are sufficient free computational resources available, in a scheme similar to that proposed in {Mohr, Kranz et al. 1991}. This lazy scheme inhibits swamping of hardware with too many parallel tasks. This is an important point, that the exploitation of available parallel hardware is not directly derived from the application, but instead is more sensibly managed by the parallel run-time support system itself, and the language element merely points out *opportunities* for parallelism (and may even be able to

perform optimisations)[4]. This is beneficial since it decouples the programming and execution of applications, resulting in increased portability and less need for performance debugging between hardware platforms.

4.2.3 Execution of a Parallel Object-Flow Application

Given that there are several aspects to the Parallel Object-Flow paradigm, it is helpful to piece these aspects together to give an overview of how the execution of a Parallel Object-Flow application might proceed. This section takes a high-level view of both the development paradigm and execution model previously introduced, and shows how those components interact in order to support application execution.

When initially executing an application, a designated method graph begins its execution. This method is the entry point to the application and is equivalent to the `main` function in the C/C++/Java languages. Within the `main` function objects may be created and methods called upon those objects. When methods are called, flow of control effectively spreads into other method graphs giving rise to parallelism. The graphs themselves implicitly handle communication and synchronisation issues as previously mentioned. The location of the objects created by an application is unimportant to the software developer, and is managed by the run-time system in such a way that locality of objects in a multiprocessor environment can be exploited to improve application performance. The overall level of parallel activity is also managed by the run-time system where the logical parallelism found in the application may be constrained to suit the physical resources of the target parallel architecture.

Simplification of constructing parallel applications is not the only benefit afforded by the Parallel Object-Flow paradigm. In fact the benefits are threefold:

1. Parallelism (and thus the potential for application speedup) is *implicitly* obtained, and exploitation of that potential parallelism is not explicitly invoked;
2. System complexity can be managed through the application of visual programming techniques;

[4] Though current visual parallel programming languages abstract low-level details like locking and so forth, they do not provide mechanisms for automatically managing physical hardware. Instead, the developer considers the logical to physical mapping of parallelism during the development process.

3. Problem domain complexity can be managed through object-orientation.

Given that the three major problem areas for developing parallel applications (management of problem domain, implementation complexity, and parallelism management) are each addressed by the Parallel Object-Flow paradigm, there is room for optimism in this approach. What remains for this thesis is the design of a language supporting the paradigm, to test whether such a language can be used not only to build general-purpose parallel applications, but also show potential for yielding speedup.

4.3 The Vorlon Programming Language

This section introduces the Vorlon programming language, which has formed the majority of the practical work for this thesis. Vorlon is the primary vehicle for evaluating the Parallel Object-Flow as a methodology for developing high-performance parallel applications.

Although Vorlon is intended to be a complete system for the development of parallel applications, here only the language element is presented. Issues pertaining to code translation and run-time support are left for later chapters. The context within which the work is focussed is highlighted in Figure 4-4 below, which shows the architecture of a Vorlon application.

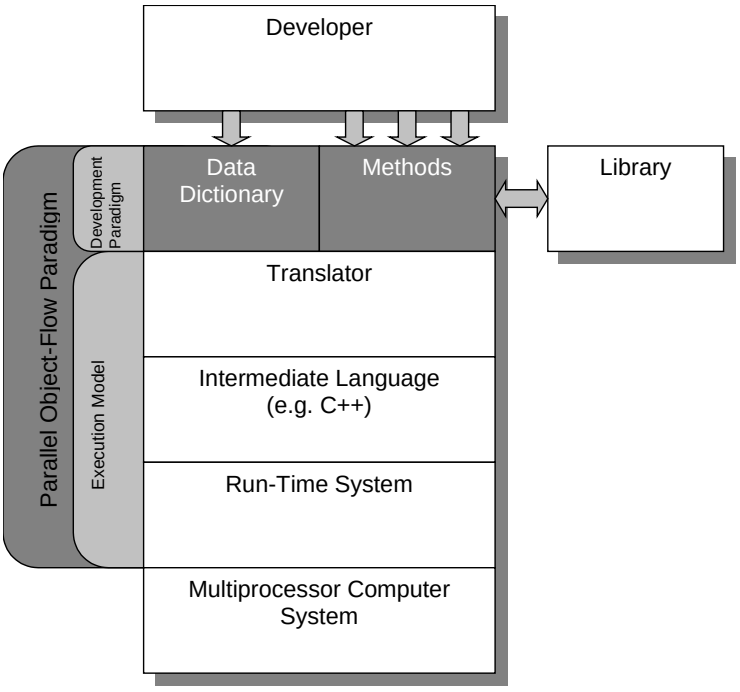


Figure 4-4 The Architecture of a Vorlon Application

Figure 4-4 shows the five levels in the architecture of a Vorlon application. At the bottom resides the actual hardware upon which the application is executed. As is apparent from the number of layers in between the hardware and the developer, there is a high level of abstraction inherent in the application architecture. It is important to understand that the layers are not meant as a source of potential inefficiencies, but as a framework under which software can be developed. The executable code produced in this architecture is no further away from the computer system than it would be if a "manual" programming method such as PVM had been used, but the developer is granted a higher-level view of the application.

The run-time system, one level above the hardware, bears the responsibility for taking compiled source code from the intermediate language level, and managing its execution. The run-time system must provide facilities for managing parallel execution of an application, such as supporting mechanisms for communication and synchronisation between concurrent computations, and implementing appropriate concurrency controls.

At the centre of the architecture is the intermediate language layer which is used to interface Vorlon graphs with run-time system services. An intermediate language is simply used to enable Vorlon to target a range of existing run-time systems via normal compilation and linking techniques.

The translation layer of the Vorlon application architecture is responsible for converting Vorlon graphs to textual source code suitable for compilation via traditional compilation methods. It is intended that the translation layer will be flexible in terms of the intermediate language ultimately produced such that no options, in terms of run-time system or machine architecture, are ruled out. In this respect, the translation layer is considered a plug-in component, in a similar fashion to the translation model adopted by the CODE system {Newton 1993}.

The uppermost layer of Figure 4-4 is the language element of the architecture, the level at which the user develops software. It consists of both a data dictionary {Layzell and Loucopoulos 1989} where the issues of modelling the problem domain through classes and class-interrelationships are tackled, and a set of methods belonging to those classes, which encapsulate computational activity in Vorlon. Both the data dictionary and methods are specified graphically. Specifically, the data dictionary is constructed using a UML-like notation {Fowler and Scott 1997; Booch, Rumbaugh et al. 1999} where classes and interrelationships are specified visually to aid developer comprehension; methods

belonging to those classes are specified in a graphical, flow-based notation in accordance with the Parallel Object-Flow paradigm.

As approaches for the object-oriented analysis and design of systems are already well known and not a research topic explored by this thesis, the discussion on constructing the data dictionary is left until the end of this section. Since the novel work undertaken has been in the area of expressing methods graphically, it is the syntax and semantics of Vorlon which is examined next.

4.3.1 Methods in the Vorlon Programming Language

This section introduces the syntax and semantics of the Vorlon programming language. The syntactic elements are presented in a bottom-up style with the simpler elements, and those familiar from previous visual parallel programming work presented first and the less familiar and more complex elements presented subsequently.

To construct software which is insulated from the complexities of the underlying parallel system, the Vorlon programming language borrows from proven technology in visual parallel programming and offers a visual flow-based syntax as stipulated by the Parallel Object-Flow paradigm.

For each method of each of the problem domain classes, a Vorlon method graph exists which describes that computation. A Vorlon method graph is an acyclic, top-to-bottom graph consisting of nodes which represent some form of computation, and arcs which describe dependencies between those computations. The arrival of handles to objects (including the possibility of handles to null objects) along arcs on a graph triggers a computational node's execution. In Vorlon, the arrival of such handles signifies the completion of previous computation(s) and thus signals readiness for successor computations to proceed, just as in the HeNCE computational model, which yields a clear graph-execution semantic.

In contrast to previous work in visual parallel programming, Vorlon arcs do not carry data values between computational nodes but in general carry handles to objects (i.e. Parallel Object Flow) which may be used to invoke methods on referenced objects. As an optimisation, for primitive types (coincidentally the same set of types offered by the C++ language) values are transmitted such that both pass-by-reference and pass-by-value semantics are supported. Pass-by-value for objects is realised by using the type's copy constructor and passing a handle to that newly created copy. As Vorlon arcs carry handles to typed objects, the language benefits from being strongly typed. As type

information is known at compile-time, unlike previous objects-plus-flows methods {Cox, Giles et al. 1995}, it is possible for a Vorlon compiler to perform static type-checking of applications.

Having established the overall flavour of a Vorlon method graph, it is now germane to review each of the syntactic elements available to the developer. The following sections introduce each piece of graph syntax, its semantics and common use.

4.3.1.1. Arc

Arcs are the "glue" with which the computational elements of Vorlon applications are linked together. Arcs are used to describe dependencies between nodes, where such dependencies may be either temporal (one node must await the completion of a prior node) or data dependence (where one node requires the results of another node).

The arc has straightforward semantics. Arcs are typed in that only handles to objects of a particular type, or subtype, are allowed to flow through any arc. Attempting to pass a handle of an incorrect type along an arc will result in a compile-time error. Arcs are unidirectional, and to accommodate the acyclic, non-stream nature of Vorlon have a capacity for only one item. The visual representation of Vorlon arcs can be seen in Figure 4-5.

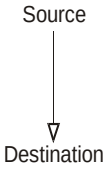


Figure 4-5 Vorlon Arc

4.3.1.2. Replicate and Merge Nodes

The replicate and merge nodes are inspired by features of the MeDaL and ParADE languages where replication and merging were used to distribute and gather data passed to different parts of an application. In Vorlon, the replication facility is used to make copies of object handles to transmit to multiple nodes in a graph where each of those nodes requires access to the same object. The merge node is used where conditions occur in graphs (see conditional node discussed later) to maintain consistent firing rules even where some paths in a graph may not be used. The replicate and merge node syntax is presented in Figure 4-6.

Figure 4-6 The Replicate and Merge Nodes

The replicate node accepts a handle to an object at its uppermost edge and emits one handle to the same object along every arc at its lowermost edge. It is important to understand that the node does not copy the object itself, but only the handle to that object. Copies of actual objects must be created using a copy constructor, as discussed below.

The merge node accepts inputs from any *one* of its input arcs and echoes that handle to the output arc. All input arcs must be of the same type for this node to work, and a compile time error will occur if this is not the case. The merge node is the only node in the Vorlon language which does not obey the strict all-arcs firing rule, though without this node conditional execution at the graph level is not possible.

4.3.1.3. Computation Node

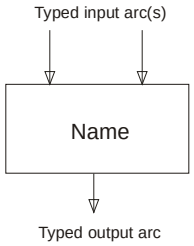


Figure 4-7 Computation Node

The computational node, shown in Figure 4-7, is represented as a rectangular icon with a descriptive name at its centre. Its purpose is to encapsulate a single computational task (written textually) or group together a number of such tasks (described graphically), and is semantically similar to the procedure call mechanism in the C programming language, allowing for multiple arguments (input arcs) and a single return parameter (output arc). The choice of a single output arc is defensible since it maintains a familiar function call semantic (though it has been noted that additional parallelism could be invoked by allowing multiple return paths).

The encapsulated computation can be expressed either:

1. Textually using a subset of the C++ programming language {Stroustrup 1997}; or
2. Graphically, in which case the node acts in a similar fashion to a decomposed actor in ParADE, to manage visual complexity.

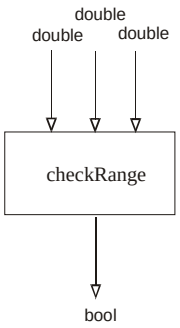
It is permitted for nodes containing graphs to nest arbitrarily, though sub-graph recursion is not supported (recursion instead being achieved via method calls as discussed later).

If a computation is expressed in C++ rather than using Vorlon graphs, any code within a computational node will be executed sequentially, even if there is potential for parallel activity. Since parallelism is potentially restricted by programming computational nodes textually, a developer must therefore be aware that the best way to interact with objects is via the graphical mechanisms (to be described subsequently). A piece of textual code should be used only where it is apparent that there is strict sequential ordering of operations, or a piece of computation is trivial enough not to warrant the support of a graphical syntax (such as the manipulation of primitive values or where it is obvious that there is no parallelism to be extracted from a piece of code).

Like the ParADE system, the computation node automatically handles conversion between graph-level arcs and appropriately typed objects in the textual host language. When viewing application structure at the graphical level, the developer is presented with a set of typed arcs connected to the computation node. When the developer chooses to view a decomposed node, either the encapsulated graph or a piece of source code providing a textual representation of the node's environment will be presented. The provision of a textual representation for a textually-programmed node can be seen in Figure 4-8 where the automatically generated (C++) code is highlighted[5], and the developer's own code can be seen nested within that.

[5] Note that there may be other code associated with the node to provide the necessary locking and synchronisation for the node program, though the developer would never be presented with such information.

Graphical



Textual

```
//////////////////////////////////////
// Check Range Computation Node
//////////////////////////////////////
// Inputs: double, double, double
// Output: bool
//////////////////////////////////////

bool checkRange(double d1, double d2, double d3)
{
  if(d1 <= d2 && d2 <=d3)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Figure 4-8 Computation Node: Graphical and Textual Views

Execution of a node obeys the same strict firing rule as other Vorlon nodes, whereby only when items are present on all input arcs will the node program be activated. At some point later in time, the node program will finish and a single handle is emitted along the single output arc (returning `void` from a textual program has the effect of producing a handle to a null object on the arc).

4.3.1.4. Parallel Computation Node

The parallel computation node exhibits a useful behavioural pattern when used in conjunction with container-like data structures. Its semantics derive from the data-parallel mechanisms of other visual parallel programming languages, whereby it is able to spawn multiple concurrent instances of its node program, and indeed its look-and-feel are adapted from the depth actor in the ParADE language. However, when the situation arises where each input arc carries a handle to a suitable data structure such as a List, the parallel computation node will automatically perform decomposition and re-composition of the elements contained within the list. The visual syntax of this operation can be seen in Figure 4-9 below, alongside the decomposition pattern for this mode of operation.

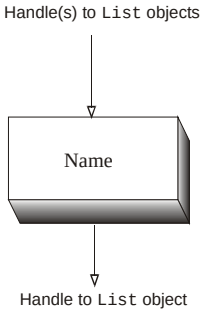


Figure 4-9 Parallel Computation Node using Automatic Data Decomposition

In Figure 4-9, each element of the list will automatically be extracted and provided as an argument to each parallel invocation of the node's program. This scheme is similar to Prograph {Cox, Giles et al. 1995}, where is has been argued that such a scheme is quite adequate for the purposes of handling parallelism in an object-oriented application. Furthermore, the notion of parallelism being driven by typing is consistent with the object-oriented philosophy of Vorlon where the developer concentrates on high-level software engineering issues and tool support takes care of the parallelism. Whilst it is clear that linear structures such as lists and arrays can easily be dealt with, it may also be possible to automatically decompose arbitrary data structures for parallel processing in this way, though such research is beyond the scope of this thesis.

Though the ability to decompose a data structure automatically is desirable, there are some conditions that the developer must adhere to:

1. The type of the objects held in the data structure must be compatible with the types expected as individual arguments to instances of the parallel computation node's program.
2. The number of elements contained in each data structure must be identical, otherwise the firing condition for some instances of the node program will be satisfied whilst for others it will not.

If the Vorlon parallel computation node is compared to the depth actor in ParADE, the Vorlon version is less flexible in the patterns of decomposition that it supports. However, it is thought that the inclusion of the more advanced partitioning schemes seen in ParADE is unlikely to yield any additional benefits to the Vorlon language since

developers are building object-oriented and not dataflow applications, and as such task- and not data-parallelism is the norm. If a situation arises where more intricate partitioning strategies are required, the developer must manage their implementation.

In its other form, the parallel computation node provides the simplest explicit form of parallelism in the Vorlon programming language. The iconic representation of the parallel computation node can be seen in Figure 4-10 below.

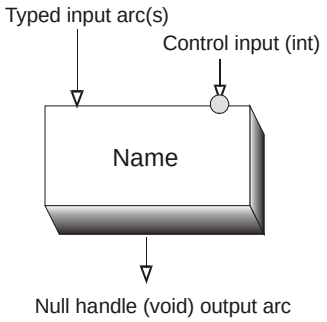


Figure 4-10 Parallel Computation Node

The parallel computation node's only other significant difference from the standard computation node is the presence of a control input which governs the level of logical replication of the node's program or sub-graph at run-time, and the imposition of a null handle to act as a timing signal when the node has finished on the output arc. The control input expects a single integer value to be present on its incident arc which is used to specify the logical extent of the replication of parallel instances of the node program at run-time. Inside the parallel computation node, each of the arcs incident on the node are presented to the programmer in a manner consistent with the standard computation node, including the control input, so that each instance of the node program can be identified from within the node program.

4.3.1.5. Conditional Computation Node

The conditional computation node behaves in an almost identical fashion to the standard computational node with the exception that the execution of its node program (but *not* the activation of the node itself) can be made conditional on the items carried by a subset of its input arcs. The visual representation of the conditional computation node can be seen in Figure 4-11.

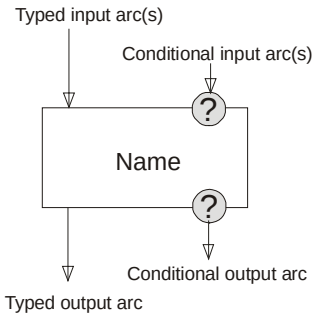


Figure 4-11 Conditional Computation Node

The semantics of the conditional computation node are straightforward. Items arrive via the arcs at the topmost edge of the icon, with the conditional input(s) denoted by the grey circle containing the question mark symbol. Once all input arcs contain handles, the node will activate (though not necessarily execute its node program), and its first operation will be to evaluate the condition (written in standard C++ and yielding a Boolean result) using the items supplied to the conditional input area, to determine whether or not the node program will be run. If the program runs, all input parameters from both the normal input arcs and the conditional arcs are supplied to the node program (regardless of whether it is an actual C++ computation or another graph). The node program then executes and at some point later returns a single item via its typed output arc.

If the condition specified by the node's condition statement is not satisfied then the node program will not be executed. Immediately a null object handle will be sent along the conditional output arc to indicate non-execution. This is important since the non-execution of a conditional node's program is equally as important to subsequent nodes in the graph as the execution would be. That is to say it is not permitted in Vorlon for a node to not signal its completion even if it is deemed to have performed no useful work.

4.3.1.6. Graph Start and Halt Nodes

A typical Vorlon application will be composed from many graphs (and sub-graphs). To manage graphs and their interactions, a visual mechanism is required to decouple them – in effect to provide actual and formal parameters such that called graphs can be isolated from their calling graphs. To satisfy this requirement, the Vorlon programming language borrows technology originally developed for the MeDaL {Harley 1993} system, and later implemented by the ParADE {Allen 1998} language, in the form of graph start and halt nodes. The visual syntax for those nodes can be seen in Figure 4-12 below.

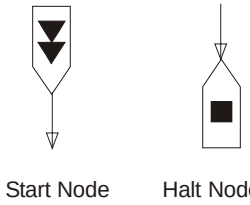


Figure 4-12 Graph Start and Halt Nodes

The purpose of a start node is to provide a graph icon to act as a formal parameter. It isolates an incoming arc from a calling graph, or in the case of an application's root graph to isolate arguments from the environment (to act as command line arguments). Similarly, the purpose of the halt node is to provide a binding between the output arc on the called graph and its associated arc on the calling graph. To be consistent with the computational node, where there may be any number of input arcs yet only a single output arc, there can be any number of start nodes in a graph, but only one halt node.

In addition to its role as a parameter-passing mechanism, the halt node is also equipped with certain synchronisation characteristics. It is clear the activation of a halt node is in itself a synchronisation, since it indicates the completion of a graph. However, the halt node is also able to accommodate any number of arcs carrying null handles (of type `void`), and in the cases where the return type of the graph is not `void`, up to one other arc carrying a handle to any non-null object. In each case the semantics of the halt node are that the graph will complete when all incident arcs are carrying object handles, though in the first case a null handle will be returned to the graph's caller whilst in the second the non-null handle will be returned. This semantic ensures that all nodes in a graph must have completed before a return will occur, and thus avoiding potential race conditions.

4.3.1.7. Literal Source

To facilitate the introduction of literal values for primitive types into an application, the literal source node was developed, whose purpose is simply to accept a literal value at compile time and provide a handle to an object holding that value at run time. The syntactic element for a literal source can be seen in Figure 4-13 below.

Figure 4-13 Literal Source Node

4.3.1.8. Attribute Source

Since Vorlon nodes may only operate upon objects to which they possess handles, a mechanism which supports access to an object's attributes (its data members) must be provided. Unlike contemporary object-oriented programming languages where attributes are implicitly available within the scope of any methods, nodes in the Vorlon language must be supplied with handles to any attributes which are required within the scope of a graph. The mechanism which provides handles to attributes is known as the Attribute Source, which grants access to an object's attributes to methods in that object's type.

The iconic representation of the Attribute Source can be seen in Figure 4-14 below.

Figure 4-14 Attribute Source Node

Attribute source nodes emit an object handle to part of the current object's state as soon as the graph to which they belong is activated. That is to say that the attribute source provides a mechanism by which a method can access the private or protected data members of its type. Precisely which of the data members is sourced by a particular attribute source is determined by the developer at compile-time. Once the parameters have been selected, there is no way to alter which attribute will be sourced at run-time. As such the developer must take care to ensure that any attributes which may be required during the execution of a graph has a corresponding Attribute Source.

Providing a visual mechanism through which object state is accessed is certainly unusual in object-oriented languages, where attributes are normally implicitly available throughout the execution of all methods. In the case of the Vorlon programming language, it is actually beneficial. Attributes will be referenced in only the scopes in which they are needed, and as such will need to be locked only for those scopes. This allows for

a fine granularity of locking involving parts of an object's state, which in turn promotes parallelism.

4.3.1.9. Object Interface Source Node

In addition to the fact that a method in the current object may require access to the attributes of the current object, there may be cases when a method may need to call another method on the same object. Since methods can only be invoked upon objects whose handles appear in the current graph, Vorlon requires a syntactic mechanism for introducing handles to the current object's interface into a method graph (equivalent to the `this` pointer in C++/Java). This is the function of the Object Interface Source node, which can be seen in Figure 4-15.

Figure 4-15 Object Interface Source Node

When activated, the Object Interface Source node provides a single handle to the current object on its output arc. That handle may then be used in a consistent fashion with handles to any other objects in the method graph, with the exception that both public and private methods in the local object may be invoked. Using the Object Interface Source in conjunction with the Attribute Source allows all of the facilities of the referenced object to be exploited in a consistent fashion.

4.3.1.10. Loop Node

The loop node is intended to provide a graph-level control-flow abstraction mechanism for iteration, and can be seen in Figure 4-16 below.

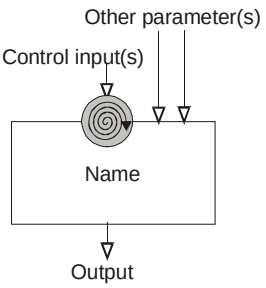


Figure 4-16 The Loop Node

The look and feel of the loop node has been adopted from the loop actor in the ParADE system, although its semantics have been changed to remove the ill-defined behaviour of ParADE's loop actor which possessed both iterative and parallel aspects. In particular, the Vorlon loop node removes the per-iteration loop input and loop output that characterised the ParADE version. Removal of the per-iteration input and output renders the loop node suitable for iteration only, and not as a mechanism for invoking streamed activity, since Vorlon does not support streaming (or pipelines for that matter) for reasons discussed in Chapter 7.

Having established that the loop node is a purely iterative construct, the syntax for loop node usage can now be introduced. It should be noted that the loop node has two areas upon which arcs carrying object handles can be incident. Any arcs incident upon the uppermost grey circle are taken as being variables used within the loop control statement (discussed next). These variables are consumed according to the normal firing rules, once per node activation, and determine the conditions under which iteration is allowed to proceed. The loop control statement is expressed textually in a subset of the C++ programming language. Arcs incident on the uppermost edge of the loop node, though not incident on the grey circle, provide handles for use inside the loop in much the same fashion as arcs incident upon the uppermost edge of a computation node provide handles for use within that particular sub-graph or computation. Handles transmitted along such arcs are made available to each iteration of the loop.

The output arc provides a means of returning a handle to the environment once all iterations of the loop have completed. Whilst it may at first seem strange that a loop can return an object, as opposed to merely iterating over objects, it makes more sense when considering the loop node as a specialisation of the computational node which repeats the body of the function until a certain condition is met. Once the loop condition has been met, the node can then release a handle onto its output arc. This handle can be any handle within the scope of the node (its parameters), or can be a null handle, as depicted in Figure 4-17.

```
Foo bar(Bar b) //node interface
{
  for(…) //loop condition (user specified)
  {
    //body of function (user code) here
  }
  //return statement
  return b; //returns a handle from current scope
}
```

Figure 4-17 Textual Representation of a Loop Node

Since each instantiated Vorlon graph can be used only once in the execution of an application, during the execution of a loop each iteration is presented with a fresh *copy* of the loop's sub-graph. This semantic can be seen as being somewhat akin to a macro expansion semantic, in that each iteration effectively creates and uses a new copy of the sub-graph defined within the loop node.

4.3.1.11. New Object Node and Constant New Object Node

The new object node is the means by which objects are instantiated and introduced into a Vorlon graph. The semantics of a new object node are relatively simple. Handles representing arguments to a constructor call for a new object flow into the node via its input arcs, causing the construction of a new object (the location of which is of no concern to the developer). Once the new object has been constructed, a handle to that object flows out of the node along its single output arc.

The visual syntax of the new object node is shown in Figure 4-18 below. The new object node is oval in shape, in common with other object interaction nodes, to distinguish it from control-flow and computational elements of a Vorlon graph. The type of object to be created appears in textual form towards the bottom of the icon along with any parameters required to instantiate the object (in effect the signature of its constructor). The mapping between input arcs and parameters is performed left to right, such that the leftmost arc maps onto the leftmost parameter and so forth.

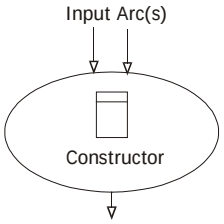


Figure 4-18 The New Object Node

It should be noted that the new object node does not support functional polymorphism with respect to selecting the most appropriate of any available

constructors based upon the handles received as its input arguments. The new object node obeys the strict firing rule that all other nodes obey, in that it will only become active when all of its input arcs are carrying handles. If other constructors are to be invoked on an object, then their invocation will have to occur within other new object nodes, where the number and type of arguments required for construction is in agreement with the number and type of input arcs to the new object node itself.

In addition to the new object node, the Vorlon language also supports a simple variant called the Constant New Object Node. The constant version of the node shares similar syntax and semantics as the standard version of the new object node, but produces handles to immutable objects. That is to say, the constant new object node produces objects whose state cannot be updated, and therefore upon which only methods that do not update state may be invoked (methods that are declared as `const`). This may in turn have implications for locking optimisations, in addition to the software engineering benefits provided (some objects simply don't change their state, and a way of modelling this is useful). The icon for the constant new object node is shown in Figure 4-19.

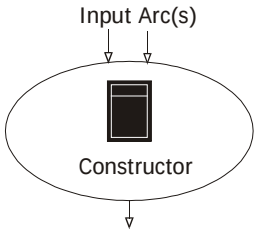


Figure 4-19 The Constant New Object Node

4.3.1.12. Method Call Node

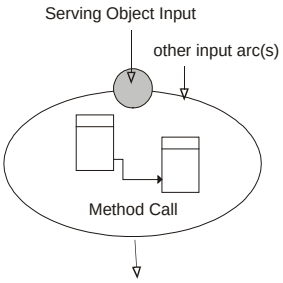


Figure 4-20 Method Call Node

The method call node as seen in Figure 4-20, provides the primary means in Vorlon by which parallelism is expressed, since it is the activation of multiple method call nodes

which propagate control-flow between graphs. Whilst it may seem strange that a method call node is the primary abstraction for parallel execution in Vorlon, consider the traditional approach to building concurrent applications, using some imperative textual language. Since textual code is written in a single dimension and effectively executes one statement at a time synchronously, where one statement completes before another starts, asynchrony was developed as a mechanism through which the normal control-flow rules could be circumvented. With the aid of asynchrony, it became possible to execute multiple one-dimensional flows of control within an application, as illustrated in Figure 4-21 below.

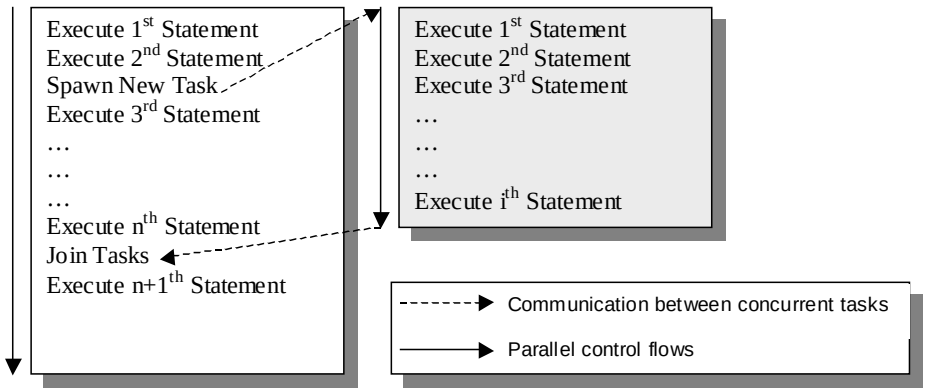


Figure 4-21 Asynchrony in Textual Programming Environments

Figure 4-21 shows a typical textual parallel program, where an initial computation spawns a second computation, both of which then proceed concurrently until the spawned computation has completed. Upon completion, the spawned computation waits until the main computation is ready to allow its spawned task to re-join.

In the Vorlon programming language, using asynchronous mechanisms to invoke concurrent methods does not fit comfortably with the programming model. Unlike imperative textual programming languages where asynchronicity is used to circumvent the sequential execution of statements, Vorlon method graphs have the ability to route flows of control around nodes, and thus permit further potentially parallel computations to execute without employing asynchronous mechanisms. Figure 4-22 demonstrates a situation similar to that exemplified in Figure 4-21, whereby two distinct parallel flow of control are present in the example code. However, Figure 4-22 makes no use of explicit asynchrony to achieve this situation, instead using routing to spread control-flow to parts of the graph that can be active concurrently.

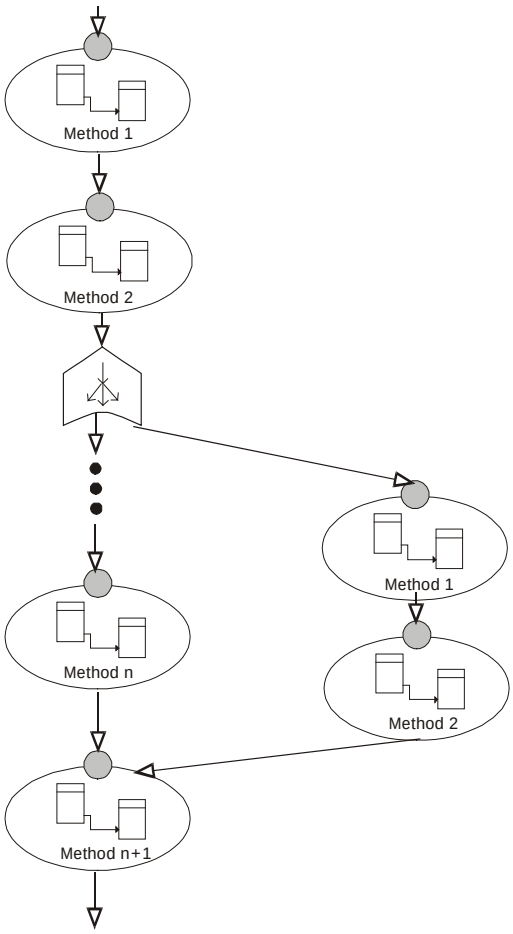


Figure 4-22 Task Parallelism without Explicit Asynchronicity

Having understood the importance of the Method Call node pertaining to parallel activity, it is now germane to investigate the mechanics of the node. Arcs incident upon the method call node may be attached at one of three possible points. The simplest to explain is the output arc. The output arc of a method call node is attached at the lowest point of the node's lower edge. Since Vorlon methods are permitted to return only a single object handle, a single output arc must be present at the base of a method call node which enables the return of a result, with the possibility that result may be a null object handle. Upon completion of a method, a handle is transmitted to subsequent nodes via this arc.

Arcs carrying input handles to the method call node are a little more intricate in that the positioning of those arcs is of importance to the functioning of the method call node. This importance stems from the fact that object handles flowing into the method call node fall into one of two categories. An object handle can:

1. Refer to the object upon which a method is to be invoked; or

2. Refer to parameters for that method invocation.

This distinction is of such significance that it warrants acknowledgement in the visual syntax of the language. It was decided that the single handle pertaining to the serving object should attach to the method call node differently from the potentially more numerous parameter object handles. As such, the arc carrying the handle to the object which is to service the method call is attached to the grey circle, or serving object input, uppermost on the method call node's icon, and is the only arc permitted to be incident upon that area of the icon. Other arcs providing parameters to the method call must only be attached to the upper side of the node's icon. Any attempts to attach multiple arcs to the server object input will result in a compile-time error. Actual parameters are mapped onto formal parameters with a simple left-to-right ordering of arcs.

Visual syntax aside, the method call node has the same semantic as a method call in C++ or Java – similar to that of the computational node introduced earlier. Method call nodes invoke a method on a specified object, pass any parameters to that method call as required, and await the completion of that method before returning control to their caller. In Vorlon, this involves passing the flow of control to the graph representing the method, and awaiting completion of that method graph before control is returned to the caller. In addition, the method call node can be used to access public attributes from an object by specifying the name of that attribute and no other parameters.

There remains one aspect of the method call node semantics which warrants further explanation. In general, when a method is invoked through a method call node, handles to objects are transmitted as parameters. When parameter is an instance of a built-in primitive type, it is not a handle which is sent, but a copy of its *value* (c.f. Java). If there exists a requirement for such types to be transmitted by reference, all that is required is for the developer to wrap a primitive value within a user-defined type, or within the built-in `Object` type provided by Vorlon (discussed below). Once a user defined type wrapper has been written for a particular primitive type, the user-defined type can be used to implement pass-by-reference semantics for primitive types, allowing both pass-by-value and pass-by-reference semantics, benefiting from both approaches. A pass-by-value semantic can also be achieved for non-primitive types by making copies of an object via the type's copy constructor and passing a handle to that newly created object.

4.3.1.13. Parallel Method Call Node

The parallel method call node behaves in an almost identical manner to the standard method call node, but with the exception that it implicitly supports "data-parallel" [6] activity, whereas the standard method call does not. The icon for the parallel method call node can be seen in Figure 4-23 below.

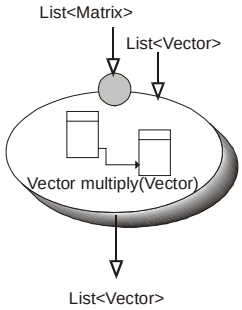


Figure 4-23 Parallel Method Call Node

Although in many respects the parallel version of the method call node is similar to the original method call node, its intended usage is somewhat different. Whilst it is clear that parallelism can be achieved using multiple instances of the method call node together with the replicate node, this is a limited data-parallel pattern where the level of parallelism invoked is determined at compile-time. Since the level of data-parallelism available in a particular circumstance is often unknown until run-time, using several instances of the standard method call node is an unsatisfactory solution. Instead it will usually be the case that method call nodes will be used to invoke task-parallel activity, where distinct methods are invoked concurrently through multiple method call nodes. True data-parallel activity will be exploited through the parallel method call nodes.

Where a standard method call node will consume a single set of input object handles, and execute a method before returning a single object handle, the parallel method call node is equipped to decompose linear data structures (such as a `List`) in the same way as the parallel computation node (Section 4.3.1.4). The size and type of any incident data structures must be both mutually compatible in terms of size, and compatible with the method call's parameter list in terms of type, lest a compile-time error occur. If handles

[6] The term "data parallel" is used here is a cognitive aid to described a replicated computation operating on different inputs. Since Vorlon is not a dataflow language, parallel method call or a similar term would be a better interpretation.

to single objects, as opposed to structures of objects, are presented on any arcs, then those handles will be made available to all instances of the method call.

4.3.1.14. Comments

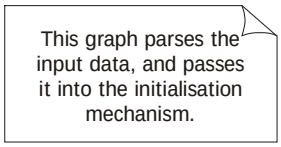


Figure 4-24 Comment Icon

During the construction of software, extra information on particularly interesting or intricate aspects of source code will require additional documentation than the source code (Vorlon graph) itself provides. Acknowledging this need, the Vorlon programming language provides a mechanism adapted from UML {Fowler and Scott 1997}, seen in Figure 4-24 above, via which comments can be added to the application source code.

4.3.2 The Vorlon Class Model

The Vorlon class model is the means by which the data dictionary, or repository of types used within an application, is described. The class model supports the analysis and high-level design phases of the software lifecycle where the problem domain is decomposed in terms of types, interrelations between those types, and the methods that those types support.

The modelling of classes and interrelationships was never an area of research which was to be covered by this thesis, particularly since aspects of the Unified Modelling Language, or UML {Fowler and Scott 1997; Booch, Rumbaugh et al. 1999} were thought adequate for the modelling of classes and interrelationships. It was therefore proposed that UML could be used as a suitable notation for object-oriented analysis and design in Vorlon. UML is composed from a comprehensive set of modelling methods to capture the behaviour of an object-oriented system at several different levels, but it is the UML class model which is of particular interest with respect to the Vorlon programming language. Whilst several other notable object-modelling languages exist {Coad and Yourdon 1991; Coad and Yourdon 1991; Bruegge, Blyth et al. 1992; Booch 1994}, UML has sought to extract the salient features of these and condense them into a single unified methodology. It is precisely this refinement which has lead to the belief that the UML is

most suitable of the available methods as a notation for representing the Vorlon data dictionary.

Of particular interest in the context of the work presented in this thesis, the UML class model provides a graphical method of expressing the interfaces and interrelations between the types that exist within a particular problem domain. The UML class model presents the developer with a set of icons representing types, and allows those icons to be connected via various relations including:

1. Inheritance – to permit the specialisation of types to manage particular problem domain requirements, and promote source code re-use.
2. Composition – to allow new types to be formed by the aggregation of objects from existing types.
3. Association – to allow objects of one type to maintain references to other objects.
4. Uses – publishes interface details of one type to a second type, permitting the second type to manipulate objects of the first type.

The UML class model was never intended as a language for programming, but as a means of modelling object-oriented systems. As such, raw UML is unsuitable for use within the context of a programming language. Specifically UML's class model lacks the ability to specify a starting point for an application. To remedy this, inspiration was drawn from object-oriented programming languages such as C++ and Java, where a single static method called `main` is used as an entry point into an application. In C++, the main method is a function at global scope, whilst in Java the main method is given static scoping within a particular class in an application. The Vorlon approach takes aspects of both the C++ and Java approaches in that it both enforces class membership for methods, yet acknowledges the semantic importance of the main method as a separate entity. The result is the Vorlon `Main` class which can be seen in Figure 4-25. The semantics of the `Main` class icon are that it has only a single method called `main` which executes immediately, and once only, upon application startup.

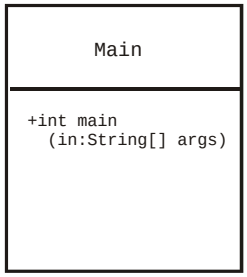


Figure 4-25 The Main Class

`Main` class is also *abstract*, which prevents objects of type `Main` from being created. The fact that the `Main` class is abstract implies that it cannot form part of a two-way relationship with other classes; whilst it is permitted for the `Main` class to use, associate or be composed from objects of other classes, the reciprocal is not true. In addition, the `Main` class, to use Java terminology, is said to be *final* which implies that it cannot be sub-classed by other types through the inheritance relationship. It is not permitted for the inheritance relationship to be attached to a `Main` class, nor is it permitted for another class to instigate an interrelation with the `Main` class.

It is clear that UML has a great deal of expressive power. Coupled with the notion of a `Main` class, the notation becomes not only suitable for modelling object-oriented systems, but with the Vorlon methods graphs and suitable library support, also as a basis for programming them.

4.3.3 Built-in Types, Primitives, and Library, and Array Support

It has become commonplace to consider a modern programming language not only in terms of its keywords and its syntax, but also in terms of any built-in types and library support. For example, the C++ programming language definition now contains not only rules governing syntax and semantics, but also contains a number of built-in primitive types and the definitions of the C++ standard library which must accompany any vendor's C++ implementation {Stroustrup 1997}. It is thus reasonable to assume that the Vorlon programming language should also provide a degree of developer support in the form of robust and re-useable library components and a number of useful built-in types. The provision of a set of built-in types is relatively straightforward. Since Vorlon targets the C++ as its intermediate language, it is simple to allow instances of C++ primitive types to appear in Vorlon programs.

In addition to these primitive types, Vorlon also provides three additional types: `Object`, `List`, and `String`. The `List` type has already been discussed as one

means by which parallel computation nodes and parallel method call nodes can automatically decompose regular data structures into individual elements for parallel processing. `List` is a parametric type (influenced by C++ templates) allowing a `List` to contain objects of any type.

`Object` is another parametric type used as a generic wrapper class for primitive types. If the execution semantics of a Vorlon graph are recalled, usually only non-primitive types are passed by reference, whereas primitive types are passed by value (as an optimisation). The `Object` type can be used to wrap primitives when pass-by-reference, as opposed to pass-by-value, semantics are needed.

The final non-primitive type which Vorlon supports is `String`. The main purpose of the `String` type is to represent sequences of characters in an object-oriented fashion. To that end, the `String` type supports a variety of operations for application use (substring operations and the like).

The construction of high-quality libraries for general-purpose programming relies not only on the availability of stable versions of the language in which to develop the libraries, but also on experience drawn from using the language to determine precisely what should and should not form part of its standard library. In addition, the sheer size of a typical standard library for a modern programming language means that considerable resources are required for their construction. Since the Vorlon programming language is intended as a vehicle primarily for validating the Parallel Object-Flow paradigm, no such library support was developed.

Notwithstanding the fact that a standard library for Vorlon was not available, the requirement for some form of support to handle issues such as input and output remained. To satisfy this requirement, the experimental version of the Vorlon programming language implemented a scheme of Vorlon-safe wrappers whereby the library support of the intermediate language component could be exploited by a Vorlon application. Thus the architecture of Vorlon applications was modified to reflect experimental needs, as can be seen in Figure 4-26.

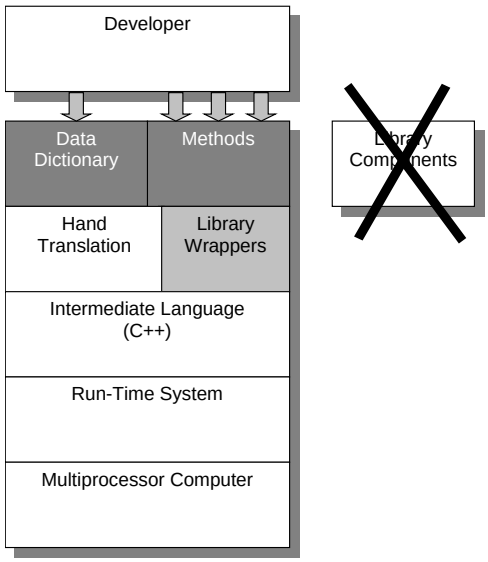


Figure 4-26 Vorlon Application Architecture at the Experimental Phase

Figure 4-26 shows the approach taken in creating an architecture for the experimental version of Vorlon. This architecture is unlike the set of well-defined interfaces presented in Figure 4-4, where the translator level provided the language component of the architecture with complete isolation from the intermediate language which is ultimately compiled and executed. Instead, the method component of the Vorlon language environment is presented with a set of hand-generated wrappers that encapsulate the functionality of the libraries supplied with the targeted intermediate language (C++). The wrappers implement the necessary concurrency controls around the components supplied by the intermediate language library, so that they can be called safely from Vorlon language components, though this is certainly not an optimal solution. In particular, the fact that non-native libraries were not written with parallelism in mind implies that their performance will not approach that of a set of native Vorlon language libraries. This has the implication that although a large body of functionality becomes available to the Vorlon developer, each time part of that functionality is utilised the performance of the application may be compromised.

The final aspect of the Vorlon language to be introduced is its support for arrays. In Vorlon, arrays are considered to be objects in their own right, as in Java, though in Vorlon arrays are more akin to optimisations for multiple instantiation since their functionality is low (similar to C/C++ arrays). Since Vorlon arrays support methods, albeit a limited set, they are accessed through method called nodes.

Creation of and access to arrays is facilitated by the New Object Node and Method Call Node respectively, where the node name is used to indicate either creation of an array or array subscripting. Examples of array creation and access can be seen in Figure 4-27, Figure 4-28, and Figure 4-29.

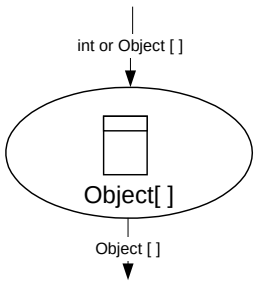


Figure 4-27 Creating an Array of Objects

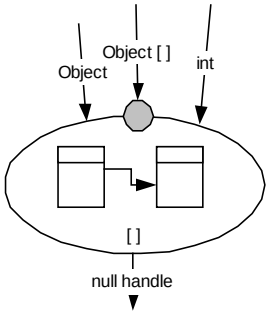


Figure 4-28 Setting an Array Element

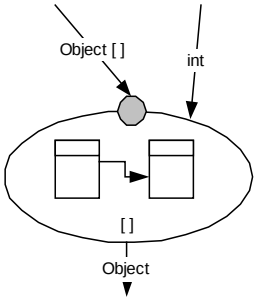


Figure 4-29 Retrieving an Array Element

Figure 4-27 shows the familiar New Object Node as introduced in Section 4.3.1.11, except in this example, the type of the object the be created (`Object`) is suffixed with square brackets (`[ ]`) indicating that it is an array which is to be created and not a single instance. Only one input parameter is allowed, either an integer or another array whose contents will be copied, and this parameter is used to set the size of the array and where the input is another array of objects to fill the new array. The implication of allowing only a single parameter is that only the default constructor of a type can be called when an array of objects is to be created, and that instances of types which do not support a default constructor cannot be held in an array (in common with C++ and Java).

Figure 4-28 shows the syntax for setting an element of an array. In this case, the array which is to be operated upon arrives via the serving object input (uppermost grey circle). A handle to the object which is to be inserted into the array arrives along an arc, as does the index of the desired array element. Once all arcs are carrying handles, the node overwrites the handle currently held in the indexed array element with the new handle, and emits a null handle along the output arc to signify the completion of the node's execution.

Figure 4-29 shows the opposite action to Figure 4-28, where an element is retrieved from an array. In this case, a handle to the array to be operated upon is presented at the serving object input and the index of the required handle is also transmitted to the node.

When these two handles arrive, the node executes and sends a *copy* of the *handle* at the specified location onto the output arc.

It should be noted that the array mechanism in Vorlon is only suitable for reference types, and not for primitives (c.f. Java). If arrays of primitive types are required, then each primitive must be wrapped in an instance of Object as described previously in this section. Furthermore it is important to remember that arrays in Vorlon are not range checked, and if an array operation causes an out of bounds error then it is likely that a run-time failure will occur.

4.4 Summary

This chapter has introduced the Parallel Object-Flow paradigm. The notion of a flow-based semantic has been introduced whereby object handles flow between graph nodes representing method calls and computation, alongside a set of other requirements, such as concurrency control and communication mechanisms, needed to support execution. In addition the aspects of the Parallel Object-Flow paradigm which address both system (visual language and execution model) and problem domain complexity (object-orientation) were discussed. The notion of building parallel software visually from objects was proposed as a major improvement over traditional approaches to parallel programming.

This chapter also introduced the Vorlon language for visual, object-oriented parallel programming. The architecture of a Vorlon-based application was described, with particular attention being paid to the way in which methods are constructed, and the parallel-by-default execution model which methods in the Vorlon language exhibit. The way in which parallelism is expressed not only through concurrent method calls on objects, but also through the traditional means of extracting parallelism from a flow-based graphs, was also explained.

The next chapter presents an investigation of the development of two applications using the Vorlon programming language, with an aim to evaluate the language both in terms of software engineering and performance.

Chapter 5 Experimental Performance Results and Analysis

Given that a methodology and tools - the Parallel Object-Flow paradigm and the Vorlon programming language - have been developed on paper, the evaluation of that technology must follow. This chapter presents a set of experimental test-beds and procedures designed to test the performance capabilities of application developed with the Vorlon language, and analyses the performance of two applications developed with Vorlon. This chapter does not cover language and other higher-level software issues, which are discussed in-depth in the next chapter.

5.1 Introduction

This chapter investigates the Vorlon programming language within a formal experimental framework. As with formal experimentation in the pure sciences, this chapter adopts the approach of introducing aims, experimental apparatus, methodology, and results before presenting conclusions. The parallelism community has often fallen into the trap of providing benchmarks which are specifically designed to demonstrate a system in its prime. It the aim of this experiment to check *whether* the prototype Vorlon programming language is a useful tool, and not to *prove* that it is.

In order to validate the Vorlon language and the Parallel Object-Flow paradigm as a means by which high-performance parallel applications can be constructed, of necessity requires experimentation with software systems. To that end, the main thread of this chapter charts the course of two software experiments, a matrix multiplication application and a Vorlon compilation system, from inception through to release. The matrix multiplication experiment allows for direct comparison between Vorlon and its predecessors within the problem domain favoured by previous programming methods. The production of a (parallel) compilation system for a language has traditionally been the starting point for that language to become self sufficient, and indeed writing such a tool in the Vorlon language itself is a robust test of the language and paradigm, as well as an opportunity to investigate a more object-oriented system than problems in numerical analysis.

After describing the construction, experimentation, and evaluation of each of the software projects individually, the chapter closes by drawing together the experiences of developing Vorlon applications. However, before experimentation can begin, the systems used to run the experiments must be introduced. To that end, the discussion diverges at

this point to tackle issues of run-time support and mappings between that run-time support and the Vorlon language itself.

5.2 Software Support - The NIP Parallel Run-Time System

The NIP (**N**ewcastle **I**mplicit **P**arallelism) parallel run-time system {Watson and Parastatidis 1999} is an experimental system whose goals are to abstract the complexity of a parallel computer, and present the abstraction of an idealised parallel computer to a compilation system[7]. The NIP system does not reveal the details of the underlying architecture, but instead portrays that hardware as an abstract machine with large memory and a similarly large number of processors, regardless of the physical configuration of hardware (shared memory, distributed memory, or a mixture of both). Thus NIP is responsible for the efficient exploitation of the underlying hardware, since it is encapsulated away from the compilation system and developer. This is a major advance compared to previous tools, such as PVM and MPI, where the developer had to bear the responsibility of managing computational resources, and ensuring that application-level parallelism was mapped correctly to physical resources.

However, it is important to understand that the normal "rules" for parallelism apply even though optimisations are supported. If the parallelism that the user identifies is relatively coarse grained, then the NIP run-time system will have far more chance of successfully optimising its exploitation than with fine grained parallelism. Still, the NIP philosophy rejects the notion of explicit logical to physical mappings of parallelism, and instead conjectures that it is the run-time system which is ideally positioned to exploit hardware resources efficiently, and not the developer – characteristics which were assumed in the Vorlon architecture, as discussed in Chapter 4. NIP provides two important mechanisms to support this notion in the form of a lazy task creation scheme and a distributed shared memory abstraction.

NIP's lazy task creation scheme {Watson and Parastatidis 1999} provides the illusion of an arbitrary number of processors. It is based around a unit of parallelism known as the "Tasklet" {Watson and Parastatidis 1999} which is a logical unit of work which may be executed in parallel, should sufficient computational resources exist. If resources are available then a parallel task will be extracted from the Tasklet, and executed on an

[7] Originally, NIP was designed to support functional-plus-objects languages such as UFO {Sargeant 1993}, though it was also found to be suitable to support declarative-style language semantics, such as those exhibited by Vorlon.

available processor. If sufficient resources are not available then the work will be executed serially. What distinguishes NIP from contemporary lazy task creations schemes is the fact that the NIP API provides a number of Tasklets which implement common parallel patterns (iteration, recursion, etc.) on behalf of the user, without requiring the instantiation of multiple Tasklets, and thus optimising run-time overheads. The API also provides an excellent target for syntactic elements from the Vorlon language to map onto.

The general behaviour of NIP lazy task creation can be seen in Figure 5-1, where two snapshots of potential activity are presented. In the earlier of the two snapshots, at time T_1, two processors are shown busy processing parallel tasks. Some time later, at time T_2, one of the processors has become idle after finishing all of the tasks in its task queue. The NIP load balancing mechanism will then steal available work from a non-empty task queue in order to keep all processors busy. Under a normal lazy task execution scheme one potentially parallel task could be moved from one queue to another. Since one Tasklet can potentially describe a number of potential parallel tasks, it is possible for its load balancing mechanism to steal whole parts of the work described by that Tasklet. This optimisation reduces the need for many potentially parallel tasks in situations such as loop unrolling where traditionally the overhead of creating potentially parallel tasks for each unrolled iteration has had to be paid. The upshot of this mode of execution is that the NIP user can assume that swamping of hardware will not occur, and that execution of all potentially parallel tasks is sensibly managed. Furthermore, since the NIP user deals only with identifying potentially parallel tasks and not executing them, the user not need to be concerned with the actual processor configuration of the underlying system[8].

[8] The closest that contemporary visual parallel programming has come to achieving a model like this, is the "cost matrix" of the HeNCE language, but that influences the load balancing mechanism and not the task creation strategy.

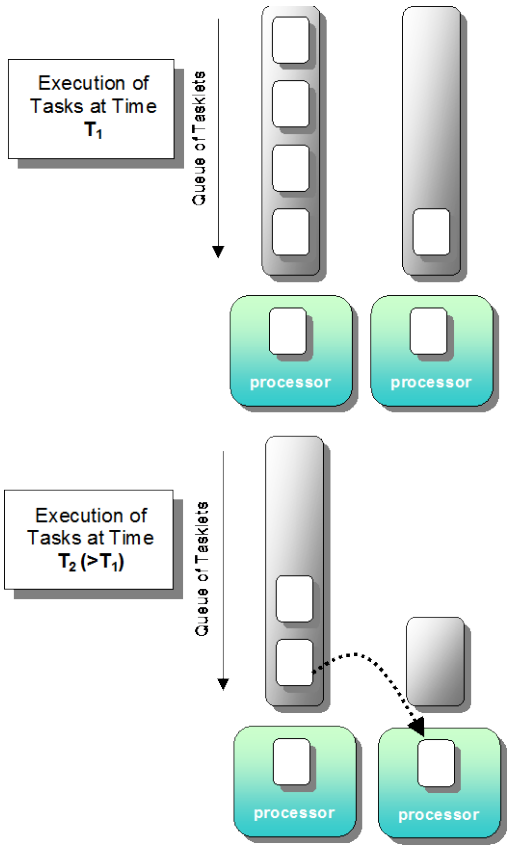


Figure 5-1 NIP Lazy Task Creation

NIP's distributed shared memory (DSM) abstraction provides the communications mechanism through which objects in an application may interact[9]. The NIP DSM provides the illusion of a logically shared object-based memory across a number of computational nodes, and provides a means of locking and accessing objects stored in that memory space. Additionally the NIP system provides a level of insulation from the non-uniform memory access (NUMA) nature of its shared memory abstraction by the inclusion of caching schemes.

Importantly the NIP DSM model supports a similar access semantic to Vorlon's relaxed active object model. In the NIP DSM, serving objects may be read or write locked. Where objects are read locked, multiple (cached) copies of the object in question are allowed to be operated upon in parallel, one copy per processing node executing a method call on the serving object. When an object is write locked, all read proxies are

[9] In addition to providing the shared memory abstraction, NIP also allows explicit message-passing to occur, though the mechanism has not been exploited in this thesis, it may be used in future work (discussed later).

invalidated and only a single method on a single object is allowed to proceed until the write lock is released.

Having introduced the necessary features of the NIP model of computation, the next section describes the approach taken to mapping Vorlon onto NIP, and sets the scene for experimental work.

5.3 Code Generation

From the Vorlon software architecture introduced in the previous chapter it was seen that an intermediate language is used to provide a target for Vorlon to map onto. Since NIP provides a C++ interface, and Vorlon has adopted elements from the C++ language for its textual and library components, it was clear that Vorlon graphs should lead to the production of C++ suitable for linking with the NIP libraries. The steps taken to execute a Vorlon application can be seen in Figure 5-2 [10].

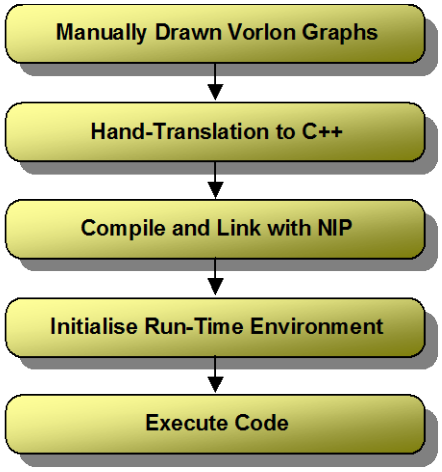


Figure 5-2 Translation from Vorlon to Machine Code

Though conceptually simple, there are some issues that arise from Figure 5-2 which are influential enough to warrant discussion here. Since parallel languages, and not tool support, was the primary focus of this research there was no tools support developed for the drawing of Vorlon graphs. Instead graphs were drawn with standard chart-drawing packages. The implications of manually drawing graphs are twofold:

The quality of the graphs may not be as high as if generated with tools support since no automatic syntax checking is available;

[10] Note that the compilation / linking layer is itself tackled as a Vorlon application in later sections, though for prototyping traditional compilation methods were used.

There is no way of automatically extracting compilable source code from graphs.

The first of these problems can be mostly protected against by proof-reading. The second is more difficult and requires that rigorous rules are followed in the manual conversion from Vorlon graph to C++ source code. It is important to note that the translation process, although executed manually, was done mechanistically as an automatic translation system would have. No expert knowledge, that is making use of information which would not be available to an automatic translator, was applied during the translation process in order to ensure a fair test. Thus, although the following text make use of phrases like "the compilation system" and "automatic" it should be borne in mind that these operations were human-executed (and not completely infallible).

Having understood the manual nature of code production, there are a number of other aspects which must be addressed in the translation of Vorlon graphs into NIP compatible C++. Of particular importance are:

- Creation of NIP compatible (i.e. thread-safe) classes from those types identified in the problem domain;
- Identification of potentially parallel tasks from Vorlon graphs from which to create NIP Tasklets;
- Determining and optimising locking requirements for objects referenced in Vorlon graphs.

Given that NIP itself is a prototype system with known limitations, there exist a second set of issues which must also be addressed, though these issues may not be directly addressed by features of the Vorlon language. These are:

- Memory management, and in particular the provision of garbage collection in the absence of any such scheme within the NIP DSM;
- Problem size must be carefully considered since NIP's internal data structures are known to continually consume memory as applications execute.

Since types form the basis of Vorlon applications, it is the generation of NIP-safe classes which constitutes the first stage in the code generation process. Fortunately, this is a relatively straightforward process since there is simple mapping between syntactic elements seen in the Vorlon class model, and components of the NIP system. This can best be shown by example. Consider the example class diagram of Figure 5-3.

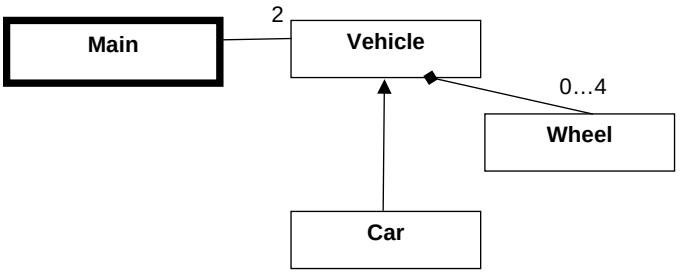


Figure 5-3 An Example Class Diagram

In Figure 5-3 a simple problem domain is shown whereby some algorithm embodied within the main class interacts with two instances of the Vehicle class, to run some kind of simulation with them. The Vehicle class itself is composed from between zero and four instances of the Wheel class, and is sub-classed by the Car class, which presumably addresses issues pertaining to motor cars rather than other types of vehicles (such as hovercraft). The NIP-safe type interfaces for this problem are shown Figure 5-4 below.

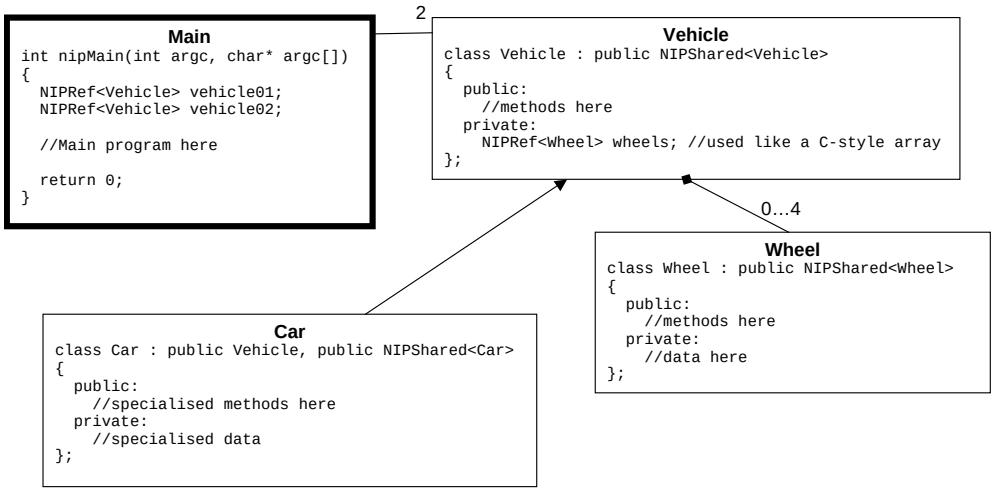


Figure 5-4 Generation of NIP-Safe Types from the Vorlon Class Diagram

The generation of NIP-safe interfaces is straightforward if certain rules are followed. In the first instance, all types (with the exception of the `Main` class) must sub-class the parametric NIP type `NIPShared`, which encapsulates the necessary functionality needed to store instances of those classes within the NIP DSM.

Secondly, since objects are stored within the NIP DSM rather than in the usual virtual memory address space, mechanisms other than virtual memory pointers are required to access those objects. The mechanism which is used to access objects in the DSM is the NIP Reference, or simply `NIPRef`, which appears in type interfaces where virtual memory pointers would normally be expected. There is one caveat in the use of `NIPRef`s in that only instances of non-primitive types are accessed through this

mechanism. Instances of primitive types are accessed directly and cannot be stored in the DMS unless they constitute part of the state of a non-primitive type object.

Apart from the global nature of the `NIPRef`, the semantics of the mechanism are similar to that of C-style pointers. A `NIPRef` may refer to either a single object as with each of the references to vehicles in the Main function, or may refer to a C-style array-like structure of multiple objects such as the array of `Wheel` objects in the `Vehicle` class. The mapping for uses and composition relations is therefore straightforward. When the extent of the relation is known (i.e. the developer has specified a single value for the number of related or contained instances in the class diagram) then that same number of `NIPRef`s to instances is generated. When the extent of the relation is unknown, an array, sizeable at run-time, is used to hold `NIPRef`s to maintain the relations. Although semantically different[11], both composition and uses relations are implemented using `NIPRef`s and it falls to the developer to ensure that correct use of each is maintained, by ensuring that details of encapsulated objects are not revealed to outside of their allocated scope.

Having established the means by which type interfaces can be automatically generated from the Vorlon class diagram, the next step is to devise a means by which the implementation of those types can be automatically generated from Vorlon method graphs. Once again explanation is facilitated by example. Consider the `Vehicle` problem domain, and in particular the `main(…)` program which is going to manipulate instances of the various problem domain types. For this example it is convenient to assume the overall goal of the application is to use parallelism to speed up a set of simulations involving vehicle collisions with other vehicles, though in this case the number of vehicles involved has been limited to two for simplicity (the figure governed by the qualified uses relation between `Main` and `Vehicle` in the class diagram). The assumption will be made that the `Vehicle` class will support a method called `collide(…)` which takes as its parameters another `Vehicle`:

```
void collide (NIPRef<Vehicle> vehicle)
```

[11] A composition relation implies that the contained objects are private to its container, whilst a uses relation implies that objects of one type hold references to objects of another type but which are not contained within the first object. Though both relations are implemented with NIP References, the class model should be used to enforce the notion of uses versus composition.

In this simple example, each of the `Vehicle`s needs to be collided with the other in order to ascertain what damage will be done. A possible Vorlon graph for the algorithm is depicted in Figure 5-5.

Figure 5-5 Main Method for Vehicle Simulator

Before continuing to show how parallelism can be automatically extracted from graphs, it is worth explaining how the graph in Figure 5-5 actually works. Firstly the parameters for the application are received from the environment through the uppermost source nodes. These source nodes provide handles to `String`s which describe each of the `Car` objects involved in the collision. Two `Car` objects are created and handles to those objects are released, copies of those handles are made, and sent to a number of method calls. The method calls invoke the `collision(…)` method on each of the `Car`s, passing the details of the partner in the collisions as a parameter. At

some point later the method for computing collision damage completes and the two cars are drawn to show the damage inflicted (encapsulated by a separate sub-graph). Figure 5-6 illustrates the extraction of potentially parallel activity from this method graph.

Figure 5-6 Main Method for Vehicle Simulator (Tasklet View)

Figure 5-6 shows each of the potentially parallel tasks (boxes labelled `Tasklet1` and `Tasklet2`) where each Tasklet is instantiated twice (since they perform the same operations, just on different data). The algorithm used to determine the potentially parallel sections of code is straightforward: wherever there are no dependencies between computational nodes, those nodes may execute in parallel. Taking `Tasklet1` as an example, there are clearly no interdependencies between the two New Object Nodes and as such they are potentially parallel. The following replicate nodes are graph management nodes and therefore have no bearing on the parallel activity within the graph. The method call nodes within `Tasklet2` are again mutually independent and are identified

as potentially parallel tasks. Subsequent nodes exhibit interdependencies and thus cannot be executed in parallel.

Having identified the required Tasklets for the graph, the translation process can then produce C++ source code. In this case, the `main(…)` method graph requires three distinct source code items, one for each of the Tasklets and one of the main program itself. Since the main program depends on instances of the two Tasklets, it is first useful to understand how the Tasklets themselves are represented. An example code fragment is given in Figure 5-7.

```
NIPRef<Car> Tasklet1Function(NIPRef<String> pString01)
{
  Car* car01 = new Car(pString01);

  ////////////////////////////
  // Interface Code
  ////////////////////////////
  return car01->registerObjs(car01, Mutable);
  ////////////////////////////
}

typedef NIPFunctionTasklet<NIPRef<String>, NIPRef<Car>,
Tasklet1Function> Tasklet1;
```

Figure 5-7 C++ Representation of Tasklet1

The source code in Figure 5-7 consists of a function (`Tasklet1Function`) which encapsulates the computation carried out by the Tasklet, and a type definition (`Tasklet1`) which is used to simplify code which uses this Tasklet. The Tasklet function takes as its parameter a handle to an instance of the `String` class, and using that parameter creates an instance of the `Car` class. The wrapper code (the code which deals with locking issues) is automatically generated, and in this case has the effect of changing the type of the pointer `car01` from a standard virtual memory pointer into a `NIPRef`, which is then returned to the caller.

```
class Tasklet2Args : public NIPShared<Tasklet2Args>
{
  public:
    NIPRef<Car> car01;
```

```
    NIPRef<Car> car02;
};

void Tasklet2Function(NIPRef<Tasklet2Args>
pTasket2Args01)
{
  ////////////////////////////
  // Interface Code
  ////////////////////////////
  Tasklet2Args* Tasklet2Args01 =
        pTasklet2Args01.lockRead();
  Car* car01 = Tasklet2Args01->car01.lockWrite();
  ////////////////////////////

  car01->collide(Tasklet2Args01->car02);

  ////////////////////////////
  // Interface Code
  ////////////////////////////
  Tasklet2Args01->car01.unlock();
  pTasklet2Args01.unlock();
  ////////////////////////////
}

typedef NIPFunctionTaskletArgument<NIPRef<Car>,
Tasklet2Function> Tasklet2;
```

Figure 5-8 C++ Representation of Tasklet2

Figure 5-8 shows a function which encapsulates the computational activity of `Tasklet2`, and a type interface for the argument that the function requires[12]. The first section of interface code for the Tasklet deals with the locking of parameters passed to the function so they can be operated upon safely, and return a virtual memory pointer to

[12] The NIP API imposes the restriction that a Tasklet's function can take only one argument, and hence a composite type is needed to pass multiple arguments.

one of those objects[13]. The main body of the function simply invokes a method on one of the objects via its virtual memory pointer and passes an object into the method call by its `NIPRef` (since the computation invoked by the method may not necessarily take place in the same memory address space of the call itself). Once the call has completed, the second layer of interface code releases any acquired locks and the function completes. A type definition is again provided to simplify the act of instantiating this Tasklet.

```
int nipMain(int argc, char* argv[])
{
  ////////////////////////////
  // Interface code
  ////////////////////////////
  String* param0 = argv[1];
  NIPRef<String> argument00 = param0->registerObjs(param0, Mutable);
  String* param1 = argv[2];
  NIPRef<String> argument01 = param1->registerObjs(param1, Mutable);
  ////////////////////////////

  Tasklet1 Tasklet10(argument00);
  Tasklet1 Tasklet11(argument01);

  Tasklet2Args* Tasklet2Args00 = new Tasklet2Args;
  Tasklet2Args* Tasklet2Args01 = new Tasklet2Args;

  Tasklet10.waitOrInline();
  Tasklet11.waitOrInline();

  Tasklet2Args00->car00 = Tasklet10.result();
  Tasklet2Args00->car01 = Tasklet11.result();
  Tasklet2Args01->car01 = Tasklet10.result();
  Tasklet2Args01->car00 = Tasklet11.result();

  NIPRef<Tasklet2Args> nrTasklet2Args00 =
    Tasklet2Args00->registerObjs(Tasklet2Args00, Mutable);
  NIPRef<Tasklet2Args> nrTasklet2Args01 =
    Tasklet2Args01->registerObjs(Tasklet2Args01, Mutable);

  Tasklet2 Tasklet20(nrTasklet2Args00);
  Tasklet2 Tasklet21(nrTasklet2Args01);
```

[13] A locking optimisation occurs here in that only objects which are to be operated upon within this graph are locked, whilst other objects (parameters to method calls) are not.

```
  Tasklet20.waitOrInline();
  Tasklet21.waitOrInline();

  //
  // Code for drawing cars would go here
  // eg, Tasklet20.result().lockRead()->draw();
  // Tasklet20.result().unlock();
  //

  return 0;
}
```

Figure 5-9 C++ Representation of the Main Program

The code in Figure 5-9 constitutes the `main(…)` function for the application. The first section of interface code merely maps the arguments to the application onto instances of the `String` type, and places those instances into the distributed shared memory. Once the command line arguments are available in the distributed shared memory, the actual computational work proceeds with the activation of a number of Tasklets. In Figure 5-9, each instantiation of a Tasklet corresponds directly to a Tasklet identified in the Vorlon graph shown Figure 5-6. Initially only instances of the `Tasklet1` type are executed whilst the `main` function continues its own work creating the parameter objects for the execution of the `Tasklet2` objects. When the instances of `Tasklet1` have completed their execution, the `waitOrInline(…)` method either forces the in-line execution of the Tasklet's function or waits until the function has completed within another thread of control before allowing further statements in `main` to be executed. Once the `waitOrInline(…)` methods have returned, the Tasklet is then ready to produce its result, which is obtained through the `result(…)` method call. Those results are then used to populate the parameters for the subsequent execution of two instances of `Tasklet2`. Once the execution of the `Tasklet2` functions have completed, other computation (such as drawing the damage of the collided cars) could then occur, before the program terminates via the final `return` statement.

Having established the process by which Vorlon graphs are mapped onto NIP mechanisms, the following sections detail two experiments and their test-beds designed to test Vorlon in typical situations from traditional parallelism areas and new application areas for parallel processing.

5.4 Experimental Environment

There were a total of three separate test bed environments used for performance analysis during this experiment. The first of these was a single four-way symmetric multiprocessor Linux computer, to examine application characteristics on a commonplace architecture. The second environment was the network of workstations environment, consisting of eight single processor Linux workstations connected by shared fast Ethernet, now commonly in use in scientific computing worldwide. The intention of using this equipment was to verify that Vorlon applications are suitable for execution on a traditional parallel architecture.

The final system used is somewhat more reflective of the average computer user, consisting of a Pentium II Mobile class PC connected via 10Mbit Ethernet to a more modern dual Pentium III class machine, where both machines ran an instance of the Windows 2000 operating system. This is certainly not a traditional parallel architecture, though it is an increasingly commonplace architecture, as new machines are purchased before old machines have failed. The intention here is to determine whether typical computing equipment could be re-used to provide performance increases to the user, by using the processing power of older hardware to bolster that of the new.

5.5 Matrix Multiplication

Matrix multiplication is habitually used as an example for parallel programming, because it is a well-known problem, which contains a great deal of potential parallelism – in fact it has been said of matrix multiplication that it is "embarrassingly parallel." Most previous work in the area of visual parallel programming has used parallel matrix multiplication as a benchmark, and to enable a direct comparison with previous research efforts, a Vorlon implementation of the matrix multiplication problem, by parallel matrix-vector multiplications, is implemented and analysed in the following sections.

An important point to note for the matrix multiplication application is that a design decision was made to use matrix*vector as the fundamental unit of parallelism. There were a number of reasons for this decision, including:

- Maintain parity with the matrix multiplication programs developed in Chapter 2;
- Ensure the grain size of tasks has a reasonable chance of achieving good performance (since it is known that fine grain-sizes will not

parallelise efficiently, and may not be efficient even after run-time optimisations).

Therefore, when reviewing method graphs it should not be surprising that there are sequential mechanisms, such as loop nodes, in places where there could have been parallel ones, such as parallel computation nodes. The upshot of this is that the run-time system is presented with tasks with coarser grain sizes to optimise, which should therefore result in more efficient performance at run-time.

5.5.1 Aims

The aim of the matrix multiplication experiment are to determine whether Vorlon can yield speedup given a problem which is known to be parallelism amenable.

5.5.2 Experimental Method

The first step in building any object-oriented application is the analysis phase. In the analysis phase, the developer considers the problem domain in terms of its constituent objects, and forms generalisations of those objects from which types are derived. In the case of matrix multiplication, the objects involved are matrices and vectors. Accordingly `Matrix` and `Vector` types can be derived from the problem domain, and the construction of a UML class diagram can then begin, as seen in Figure 5-10.

Figure 5-10 High-Level Analysis of the Matrix Multiplication Problem

Figure 5-11 Composition Relation Between Matrix and Vector Types

Once the types have been derived from the problem domain, the developer can then begin to examine the interrelationships between those types. In a typical application, there may be a high number of types, and complex interrelations between them, although in the matrix multiplication example things are somewhat simpler. Here, the only relationship that occurs in the problem domain is that a matrix object is composed from a number (≥ 2) of Vector objects (representing the columns of the matrix), which contain real numbers. A refined class diagram showing this relation can be seen in Figure 5-11.

The next development stage is high-level design. High-level design involves determining the internal structure (attributes) of each of the problem domain types, and the operations (methods) which those types are to support. Beginning with the `Vector` type, it is clear the vector dot product is fundamental to the application as a whole, and as such the Vector type is specified as supporting that method. In addition, `Vector` needs to provide subscripting operations, and that functionality is supported by the `get(…)` and `set(…)` methods. The other operations that the `Vector` type needs to support are two constructors, one which allows the size of the vector to be specified, and a copy constructor which is necessary in this prototypical version of the Vorlon programming language[14]. The `Vector` itself is composed from a single integer value which represents the size of the vector, and a pointer which is used to create a run-time array to hold the vector's contents.

The `Matrix` type, in addition to its `Vector`s, also requires a number of other attributes to support its functionality. In particular it needs two integer values to hold its two dimensions. `Matrix` supports more operations than `Vector`, including constructors (one of which is the compulsory copy constructor), two multiply methods (one for matrix-matrix multiplication and one of matrix-vector multiplication), methods for obtaining the dimensions of a matrix, and methods for extracting individual rows or columns from a `Matrix` object.

[14] The provision of a copy constructor (or equivalent mechanism) is good practice when building object-oriented software, though in Vorlon it is necessary for the sake of type safety in the run-time system. It has been found that the lack of a copy constructor may cause the certain aspects of the NIP run-time system to fail with certain compiler /operating system combinations. The simplest solution to this problem is to mandate the provision of the copy constructor either by the user, or by a default implementation being provided by the Vorlon to C++ translator.

Figure 5-12 Type Interfaces for the Matrix Multiplication Application

Once this stage has been reached, all of the principal operations that derive naturally from the problem domain have been identified. To complete the design of the matrix multiplication application, the start and end points to the application are added to yield the class diagram seen in Figure 5-13.

Figure 5-13 Matrix Multiplication Final Design

At this point the analysis and design phases are complete and what remains is the construction of method graphs for each of the operations specified in the class diagram of Figure 5-13.

Whilst there is no strict ordering in which the methods belonging to each class must be constructed, it is sometimes helpful to construct applications bottom-up, building smaller components first, and amalgamating those smaller components into larger ones. Following such an approach, the first type whose methods will be implemented is the `Vector` class.

Figure 5-14 Vector Constructor

Figure 5-14 shows the default constructor for the `Vector` class. This method receives an instance of the primitive type `int` through its start node which it uses to create an array of `double`s which will be used to hold values. The penultimate stage of the method involves two assignments within a computation node which assign the size of the instance to the attribute representing the vector's size, and the assignment of the newly created `double` array to the corresponding attribute within the `Vector`. Note

that the `Vector` constructor will initialise each of the elements of the `Vector` to a random (garbage) value, and that the user of the `Vector` class should therefore not assume that a Vector is ready to be used unless its values have been set.

The copy constructor follows a similar pattern to the default constructor of Figure 5-14, with the exception that as well as creating the internal data structures for the instance, it also duplicates the values held by the instance passed as a parameter to the constructor. The copy constructor can be seen in Figure 5-15.

Figure 5-15 Vector Copy Constructor

The major difference between the method shown in Figure 5-15 and that shown in Figure 5-14 is the use of the loop node. Within the loop, an assignment operation is repeated such that the contents of the `_data[]` attribute of one `Vector` object is copied to the new object. Once the necessary assignment operations have completed, the loop node returns a null handle to the halt node which indicates the termination of the graph.

Although the copy constructor could be made more parallel by replacing the loop node with a parallel computation node, it was not since years of use of sequential programming generally tends to force the notion of iterating over a data structure. In this case the amount of lost parallelism is minimised, since the assignment (a write operation) is naturally serialised. However, the issue of using iterative constructs where parallel ones would be better is recognised as being important, and is thus dealt with more thoroughly in a later chapter.

Figure 5-16 Vector Get Method Figure 5-17 Vector Set Method

Figure 5-16 shows the `get` method of the `Vector` class. The `_data[]` attribute is supplied by an attribute source node, and the element to be extracted is supplied as a parameter through the graph's start node. Both handles flow into a method call node which performs an array subscript operation and returns an instance of `double` to the calling graph via the halt node. The corresponding `set` method is shown in Figure 5-17.

Figure 5-18 Vector Size Method

The `size()`method shown in Figure 5-18 is the simplest of the methods that the `Vector` class supports. Its functionality is merely to return a copy of the value which represents the length of a `Vector` object. The receipt of a null handle (representing no parameters) triggers the execution of the graph, which in turn releases a handle to the `_size` attribute of the current object. Both then flow into the halt node which returns `_size` (by value) to the caller of the graph. Note that the graph will not return the null handle from the rightmost arc to the caller of the graph due to the special semantics of the halt node whereby non-void types have precedence.

Figure 5-19 Vector Multiply Method (Dot Product)

The method graph shown in Figure 5-19 shows the first of two graphs which constitute dot product. In this first graph, three parameters are passed to a loop node which iterates over the contents of two `Vector`s. The first parameter is the size of the `Vector`, used to control the number of iterations the loop node will execute. The second parameter is the vector which is to be multiplied with the current object. The

final parameter is a `double` which is used to hold the result of the dot product. Note that once again an iterative, rather than parallel, pattern is used here. However in this case it is reasonable since the design decision has already been made to sequentialise any aspect of the matrix multiplication of finer granularity than matrix*vector (discussed earlier).

Figure 5-20 Vector Dot Product Loop Node Sub-Graph

Figure 5-20 shows the loop node of Figure 5-19 decomposed. In this graph the i^{th} element (where i is determined by the control value of the loop node) of both the current object's internal array and of the parameter vector are extracted via a subscript operation and a method call respectively. Those values are then sent to a computation node where they are multiplied and their product added to the running total for the invocation. Once the total has been updated, the graph ends and control returns to the loop node of Figure 5-19 which, according to the control condition, may perform further iterations or send the result to that graph's halt node. The latter will end that graph's execution and return the vector dot product to the graph's caller.

Figure 5-21 Matrix Constructor

The `Matrix` type is essentially a container for `Vector`s in the same way that a `Vector` is a container for `double`s and as such the default constructor for the `Matrix` type is similar in principle to that of the `Vector`. The constructor shown in Figure 5-21 performs two (separate and potentially parallel) assignment operations before creating an array of `Vector`s[15] which the decomposed loop node of Figure 5-22 subsequently fills with `Vector` objects. Note that the loop node in the constructor

[15] A Vector of Vectors is not used here since the Vector class is a container for doubles, and not a generic container like the C++ STL Vector type. The built-in array type is used as an optimisation for multiple declarations.

serialises the construction of a `Matrix`, although like the constructor for the `Vector` class, the serialisation is not (fortunately) critical to performance.

Figure 5-22 Matrix Constructor Loop Node Sub-Graph

The `Matrix` copy constructor is similar in form to the default constructor, with the exception that instead of creating an array of empty `Vector`s, it copies the contents of the Matrix passed to it as a parameter. The top-level graph for this method is shown in Figure 5-23.

Figure 5-23 Matrix Copy Constructor

Figure 5-23 starts by obtaining the dimensions of the parameter Matrix by invoking the `getRows()` and `getColumns()` methods (in parallel), followed by the assignment of those values to the appropriate attributes in the new object. Once the `_rows` and `_columns` attributes of the new object are set, the loop node then copies each of the `Vector`s from the `_data` array (which holds columns) from the parameter object into `_data` in the new object. The sub-graph which the loop node encapsulates is shown in Figure 5-24.

Figure 5-24 Matrix Copy Constructor Loop Node Sub-Graph

The parameters for the graph shown in Figure 5-24 represent the current iteration of the loop, the internal array of the `Matrix` being constructed and the `Matrix` which is being copied. The appropriate element of the new matrix's array is accessed via the array subscript method call node, and the column to be copied from the parameter `Matrix` is extracted by calling the `getColumn(…)` method and subsequently used to create a new copy of the resulting `Vector` via a new object node. The final step in this graph is an insertion of the newly created `Vector` into the array of column vectors, performed by the array subscript method call node.

Figure 5-25 Matrix Multiply Method

The `multiply(…)` method of the `Matrix` class is shown in Figure 5-25. This graph instantiates a new `Matrix` object to hold the result of the matrix multiplication, and explicitly initiates a parallel computation node (using its non-canonical form) ready to perform parallel `Matrix*Vector` multiplications. The parallel multiplications are themselves executed by the sub-graph shown in Figure 5-26.

Figure 5-26 Matrix Multiply Parallel Computation Node Sub-Graph

Figure 5-26 shows two interdependent method calls. The first is a call on the current object (`this`) to the vector multiplication method (discussed subsequently), which returns the product of the current `Matrix` and the `Vector` passed as a parameter to this graph. The second method call takes that `Vector` and passes it as a parameter to the `setColumn(…)` method invoked on the result `Matrix` before the graph returns. The matrix-vector multiplication method, which is called by the graph shown in Figure 5-26, is itself shown in Figure 5-27.

Figure 5-27 Matrix-Vector Multiply Method

The matrix-vector multiplication method, shown in Figure 5-27, takes a `Vector` as its parameter, multiplies that `Vector` with the contents of the current `Matrix` in a series of row*column multiplications and returns a `Vector` as its result. The actual multiplication matrix*vector work is performed within the loop node[16] whose encapsulated sub-graph is shown in Figure 5-28. Note that although the graph is sequential due to dependencies and iterative activity, parallelism is achieved because instances of the graph will be created in parallel by dint of the fact that the graph's calling graph (Figure 5-26) is itself instantiated in parallel.

[16] The loop node is not the most parallel construct to use in this situation, however given the design decision to serialise any operation below the granularity of matrix*vector to maintain parity with previous implementations from Chapter 2, it is understandable.

Figure 5-28 Matrix-Vector Multiply Loop Node Sub-Graph

The sub-graph in Figure 5-28 begins by extracting the appropriate row from the current `Matrix` and multiplying it with the `Vector` supplied by the parent graph (Figure 5-27). Once the vector dot product operation has completed, the resulting double value is sent to the `set(…)` method which is invoked on the result `Vector` to store the result before the graph returns control to its parent.

Figure 5-29 Matrix Get Row Method

Figure 5-30 Matrix Get Column Method

The Vorlon graph representing the `getRow(…)` method of the `Matrix` class can be seen in Figure 5-29. Since the internal data structure for the `Matrix` type is represented by an array of column vectors, obtaining a row vector from that structure involves accessing each of the columns and extracting just one element. Having accessed that element, a copy of it can be made and stored in another `Vector` object which acts as the result of the operation. It is noteworthy that in the `getRow(…)` method the loop node is not decomposed into a sub-graph since the code to be repeated is simple enough to be written directly in C++. Although the loop node code operates upon objects which may be accessed concurrently, and which may not reside in the local machine's memory space, the developer does not have to contend with such issues which are taken care of by the loop node (and its subsequent translation).

In contrast to the `getRow(…)` method in Figure 5-29, the `getColumn(…)` method of Figure 5-30 is significantly simpler since the internal `Matrix` data is arranged by column. In this method, the column which is to be retrieved is accessed via an array subscript operation which obtains a handle to that column and returns it to the graph's caller. Note that both "get" methods return a read-only handle (`const`, as depicted in the class diagram). This maintains a consistent semantic to the user (and prevents direct access to the internals of the object in the case of `getColumn(…)`).

Figure 5-31 Matrix Set Row Method

Figure 5-32 Matrix Set Column Method

The `setRow(…)` and `setColumn(…)` methods of Figure 5-31 and Figure 5-32 respectively are the reciprocal of the get methods described earlier. Setting a new row in a `Matrix` composed of column vectors involves access to all columns. In contrast, setting a column in a `Matrix` composed from columns is straightforward in that one `Vector` merely replaces another in the `Matrix` and the replaced `Vector` can then be garbage collected if not reference elsewhere.

Figure 5-33 Matrix Get Rows Method

Figure 5-34 Matrix Get Columns Method

Accessing the dimensions of a Matrix is done through the `getRows()` and `getColumns()` methods which can be seen in Figure 5-33 and Figure 5-34. In both cases, the dimension to be accessed is introduced into the graph by an attribute source which immediately sends that dimension to a halt node. Once the null handle from the start node of the graph also reaches the halt node, the non-null handle (being either the row or column dimension) is returned to the caller of the graph.

The final method for this application is the `main(…)` from the `Main` class. The graph for `main(…)` can be seen in Figure 5-35, and is remarkably similar in shape to the

matrix multiplication graphs with previous visual parallel programming languages seen in Chapter 2. In this figure, four parameters representing the dimensions of two matrices are supplied via start nodes, and immediately used to create two matrices (potentially in parallel). Note that, since the `Vector`s from which the matrices are built are randomised at construction, in effect the matrices themselves will contain random values which is useful for experimental purposes, though for real work the matrices would have to be initialised with non-random data. Once the matrices are built, the `multiply(…)` method is invoked on the leftmost `Matrix` passing a handle to the rightmost as a parameter. The multiplication then proceeds and later a handle to a third `Matrix` is emitted from the bottom edge of the method call node. This handle is fed into a computation node which is responsible for the output of the result (to some output device whose characteristics are unimportant here). Once the output of the result has completed, the application terminates by sending a null handle to the halt node.

Figure 5-35 Matrix Multiplication Main Method

5.5.3 Matrix Multiplication Performance Results

Having followed the development of the matrix multiplication application, the next stage in the experimental procedure is to examine its performance. For the performance tests, all three test platforms described earlier were used together with a pure sequential C++ matrix multiplication program which formed the basis for measuring performance

characteristics. The results are presented with shared memory and uniprocessor cluster platforms first, followed by the less common ad-hoc parallel platform.

5.5.3.1. Shared Memory Multiprocessor

Single four-way Pentium III Xeon 500MHz (512KB cache), 512MB main memory, Ultra SCSI disk, running Red Hat Linux 6.2

The four-way PC SMP is typical of the recent multiprocessor trend the computing industry. Often used as powerful servers in corporate workgroup environments, such equipment readily lends itself to being utilised as a parallel computation platform. The main advantage that this machine provides for this test is size of both memory and CPU cache. The large memory means that larger problems can be investigated, whilst the large processor cache sizes reduce contention for that memory.

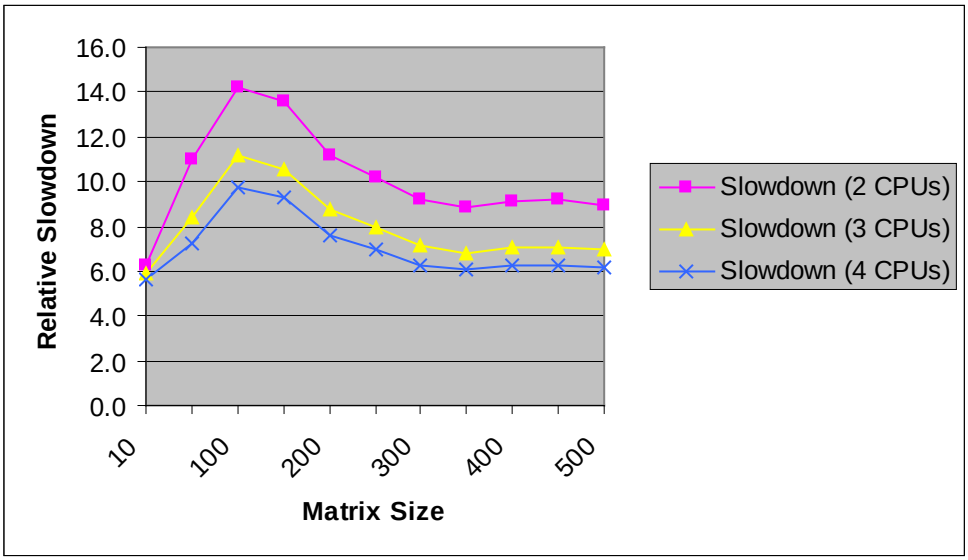


Figure 5-36 Slowdown for Matrix Multiplication Application on a Four Processor SMP Platform

The performance results for the matrix multiplication application on the SMP platform can be seen in Figure 5-36. The results are shown as relative *slowdown* when compared to a pure sequential C++ version of a matrix multiplication algorithm running on a single processor of the same computer for matrix sizes 10 by 10 through to 500 by 500.

It is important to understand that characteristics of the NIP run-time system play a key role in the results obtained. The SMP version of NIP cannot guarantee that the specified number of processors will be utilised at run-time since it does not perform its own thread-scheduling (instead leaving such matters to the operating system). NIP

instead creates a sufficient resource pool (worker threads, mutexes, etc.) for the specified number of CPUs (determined by heuristics), but the operating system is free to allocate any thread to any CPU. Thus if one thread suspends and another is active and then the suspended thread resumes, the operating system may choose to execute that thread on any available processor (and not necessarily on the processors the user believes are active). Thus when the following discussion uses terms like "*N*-processors" it actually means that the full complement of run-time resources for *N* processors is activated, and not physically that number of processors. In addition, NIP's lazy task creation mechanism will attempt to optimise the number of tasks that actually end up being executed by distinct threads (for the sake of grain-size). Thus, even though the resource may have been created for multiple processors, that it not to say that all of those resources (or processors) will be exploited.

Having understood that the underlying run-time system possesses such characteristics, the analysis of the performance results can proceed. For smaller matrices the slowdown is similar for all processor configurations, and this is due to the fact that the amount of work available is so small that it is executed sequentially by one processor – an optimisation caused by the lazy task creation mechanism of the underlying run-time system. The slight time differences between slowdowns are caused by the increased set up times needed for increasing numbers of processors and the slowdown itself caused by the fact that all object access is thread safe and thus incurs a significant locking cost compared to pure sequential C++ application.

For medium sized matrices, 50 by 50 through 150 by 150, the increased slowdown is caused by contention for matrix access. So while there exists enough work to instigate computation on more of the available processors, those active processors all try to access a comparatively small number of objects (particularly when accessing the result matrix which must be write-locked in places) and are thus in constant competition with, and waiting for, one-another.

The fact that the slowdown increases at first is due to a combination of increasing matrix sizes (and thus increasing workload) allowing the run-time system to start work on other processors (whereas previously the work available was in such short supply that only one CPU was used as an optimisation). As more processors come on-line the contention for access to matrix elements increases and correspondingly so do the slowdowns experienced.

As larger matrices were used, the contention for access to particular elements decreases since with more elements to be operated upon, the less is the likelihood of multiple processors requiring access to the same element at the same time. For larger matrices such as 300 by 300 and upwards, the performance graph settles into a more steady state where slowdown becomes a function of number of processors with four processors being quicker than either three or two.

Regardless of the fact that for larger matrix sizes, slowdown becomes more predictable, it is nevertheless slowdown and not speedup which is attained. This is clearly a source of some disappointment for a parallel solution to have performed an order of magnitude worse than its serial equivalent (and thus a likely two orders of magnitude worse than its successful parallel equivalents). The source of the slowdown is the amount of locking required to access elements of the matrices, and the severity is significantly influenced by contention for such locks. In further experiments carried out on the SMP platform, the per-element locking was removed by hand, and speedups as close to optimal as could reasonably be expected were obtained, and such issues are explored in more depth later.

5.5.3.2. Uniform Cluster

Eight uniprocessor Pentium II 233MHz, 64MB main memory, IDE disk, running Red Hat Linux 6.2 with exclusive access to shared 100Mbit Ethernet network.

The network of workstations (NOW) approach has become the flagship architecture for the parallelism community over the past few years. This "Beowulf" {Sterling, Salmon et al. 1999} class architecture has reached particular prominence through the widespread uptake of the PVM and MPI programming environments and is now commonly used as a parallel platform for scientific and engineering problems.

The advantage of this configuration for this experiment is the number of processors available to undertake work. Whilst the specifications of the machines and interconnection network are modest, having access to eight CPUs on which to execute work allows insights into scalability of the methods to be drawn. The drawback to using this configuration is its low memory per node which, given the current memory use pattern of the NIP run-time system, limits experimentation to somewhat smaller matrices than on the more powerful SMP platform. Although the total memory available on the cluster of workstations used is the same as the SMP platform, NIP's distributed shared memory abstraction does not yet support load balancing across nodes in a cluster. As

such objects are created in the memory of the node which happens to execute a Tasklet function that contains object declarations. In this experiment most of the objects (and the internal NIP data structures which maintain those objects in the DMS) end up being created on the primary node of the cluster (the node from which NIP initialises). Since there is relatively little computational work in creating objects, the primary node does not export such work to other nodes and so the experiment overall is limited by the memory of that node (64Mb).

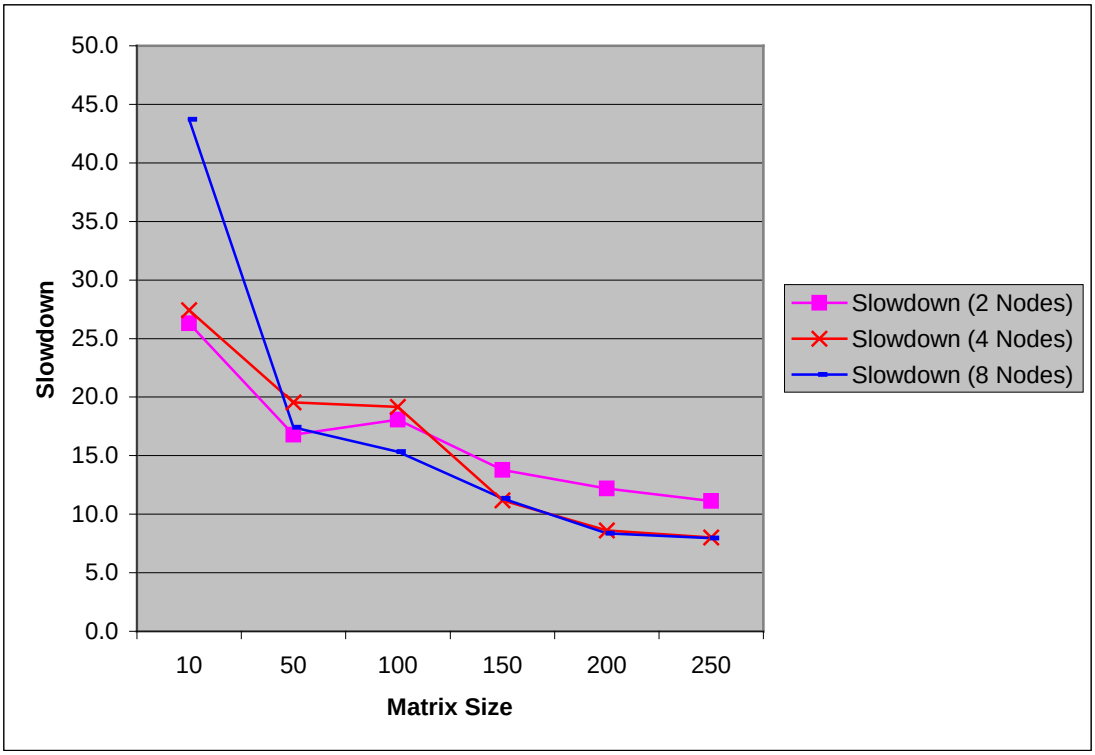


Figure 5-37 Slowdown for Matrix Multiplication Application on an Eight Node Computer Cluster

Once again before the analysis of the results can begin, it is helpful to understand the salient characteristics of the NIP run-time system on a distributed memory computing system. Although task creation is similar to that of the SMP (wherein tasks will only be created if there is sufficient work to justify it), unlike the SMP system the specified maximum number of processors involved in a computation cannot be exceeded. Thus where *N* nodes (and in a uniprocessor cluster, processors) are assigned to participate in a run, no more than *N* processors will ever participate (though through optimisations less than *N* might). Also important to these experiments is the fact that "page-based"

memory caching mechanisms were employed[17] (which improved performance by an order of magnitude), whereby when an object is accessed in the NIPDSM, a 2 kilobyte "page" is filled with subsequent objects with the idea being that such objects will probably be needed by the requesting task anyway, and thus optimises the number of requests made to the DSM.

Figure 5-37 shows the performance degradation of the matrix multiplication application on two to eight nodes of a distributed memory computing system, compared to a pure sequential C++ version of the program running on a single node of that system. Once again for small matrix sizes there is significant slowdown since the cost of initialising all nodes in the parallel platform is paid, though with little work to be done a single processor undertakes the whole of the computation.

For medium sized matrices the curve is somewhat different to that obtained using the SMP. In the distributed environment with NIP page-based caching enabled, copies of `Vector`s are distributed around the system which results in far less contention for locks on those `Vector`s since each processor is more likely to have its own copy cached in its local memory space. In this case since there is less contention for access to objects, the processors are able to work without considerable interruption of one-another. However, the cost of locking objects before accessing them remains and, coupled with the fact that transfers between nodes on the network are relatively slow, ensures that slowdown rather than speedup is the norm. For larger matrices, there is a similar pattern for that of the medium sized matrices where as grain size increases (without significant increase in contention for objects because of the caching), slowdown decreases.

It is interesting to note here that the best case slowdown for the distributed memory multiprocessor (7.4 times slower for 5 nodes) is not considerably worse than the best case slowdown for the shared memory architecture (6.1 times slower for 4 processors) due to the caching optimisation. This bolsters the argument offered by this thesis that optimisations are best left to the run-time system, and that introducing layers into an architecture for engineering reasons need not necessarily imply that performance will significantly suffer. Such performance issues are presented in more depth later in the chapter.

[17] The NIPDSM is completely object-based and does not use pages as its unit of storage. The so-called "page-based" caching is merely a convenient term for a group of objects which are spatially local.

5.5.3.3. Ad-Hoc Environment

Dual Pentium III 800 MHz SMP, 256MB main memory, IDE disk, and Pentium II Mobile 300MHz, 128MB main memory, IDE disk, both running Windows 2000 SP1 with non-exclusive access to shared 10Mbit Ethernet networking.

The ad-hoc computing environment is drawn together from computing peripherals that might be found in a typical office (in this case being the range of equipment available in the author's office). There are two particularly interesting aspects to this experiment, the first being the fact that a parallel computer is being built from non-specialist hardware, and the second is that an operating system not usually associated with high-performance computing (Microsoft's Windows 2000) is used to support the computational activity. Figure 5-38 shows the performance of matrix multiplication on this system.

The slowdowns seen in Figure 5-38 were measured using a pure sequential C++ program executing on one processor of the faster of the two machines in the "cluster," in order to give the most accurate interpretation of speedup (or in fact slowdown). As such it must be borne in mind that although the figures seem somewhat worse than those that have been previously presented, one of the machines involved is of a previous generation of processor and runs at less than half the clock speed of the other, which causes the bias.

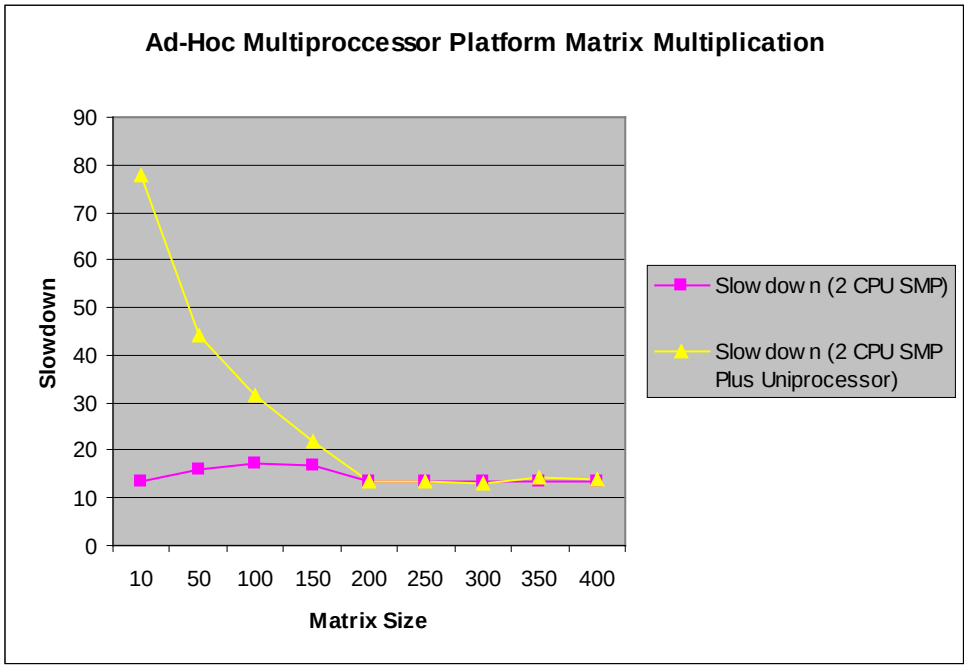


Figure 5-38 Slowdown for Matrix Multiplication Application on an Ad-Hoc Parallel Platform

If the curve for the two-way shared memory platform seen in Figure 5-38 is considered, it is similar (though considerably shallower) to those curves seen in for the four-way shared memory platform in Figure 5-36 where slowdown increases for medium sized matrices, but decreases for small and large matrix sizes. The reasons for the shape of the curve are the same, pertaining to contention and locking of objects and so forth.

The shape of the curve for the "cluster" of two machines is close to the curves witnessed in Figure 5-37. Once again the reason for the improvement in performance as matrix size is increased is due to the caching mechanism reducing contention between the two hosts for access to objects, and perhaps even lessening the effect of the slower machine on the faster. The initial large slowdown is exacerbated in this case by the slow networking between the two machines which compounds costs of contention between processors by acting as a serious bottleneck between the two hosts.

Thus the notion of bolstering newer hardware by adding in surplus (older) hardware in a kind of co-processor arrangement has failed for this application. Even for large grain sizes the dual processor computer is burdened by the attached uniprocessor. For small grain sizes the usual rules concerning efficiency of exploitation between shared and distributed memory architectures applies.

5.5.4 Matrix Multiplication Conclusions

The crucial factor in such poor performance figures is locking (and not communication costs since the performance degradation was consistent between platforms, due to caching in the distributed memory experiment). For matrix multiplication, the amount of locking required was of the order of three read locks for each element of the matrices to be multiplied, and two read locks (which each in turn acquire two operating system level locks) and one more expensive write lock for each element of the result matrix. For a matrix the potential amount of locking required is considerable, of the order of N^2 where N is the number of elements in the matrix.

Thus an unfortunate mismatch between the algorithm and the data structures that constitute the application has arisen. Although the algorithm stipulates matrix-vector multiplications (which is a relatively coarse grain size), the types that represent vectors still require locking at the element level (which is an extremely fine grain size). If Amdahl's law is applied, then the sequential parts of the application (the locking) far outweigh the parallel parts (the actual multiplications) and thus the chances of getting speedup are nullified.

In the serial matrix multiplication program, no such locking is required. Though only one processor is available to perform calculations, the lack of locking overhead made the serial program significantly quicker.

Matrix multiplication algorithms have of course been parallelised and achieved reasonable speedup - indeed previous visual parallel programming languages have all achieved this. Even using tools such as PVM which this thesis has claimed are primitive, matrix multiplication can achieve speedup with suitably sized matrices (of the order of 300 by 300 and upwards in experiments undertaken early in this research). One reason for the disparity between the Vorlon matrix multiplication application and others is that the design of the application involved the wrapping of instances of the `double` type inside instances of the `Object` type such that they could be stored in the distributed shared memory abstraction of the underlying NIP run-time system. Objects in the distributed shared memory *must* be locked before they can be operated upon, whilst with other approaches, since copies of values were used, locking was not necessary. Indeed the caching of copies of objects providing performance increases bolsters this argument.

There are thus four possible means of improving performance for this application:

1. Change the Vorlon language[18];
2. Change the design of the application;
3. Change the Vorlon to NIP translation strategy; or
4. Change (or optimise) the run-time system;

Since it is quite possible as NIP matures that it will improve and provide some modest performance increases, point (4) in the above list can be left aside for the purposes of this discussion (or in the unlikely case that NIP fails to improve, other run-time systems could be substituted to the same effect). Given that this thesis has argued that delegation to the run-time is a sensible notion, reasons for the lack of performance attributed to this delegation must be considered.

If Vorlon is to be changed to delegate less to the run-time system, careful consideration must be given to what aspects should be altered. Given that it is locking which has caused the disappointing performance figures, means of optimising locking at the language level could be introduced. One such scheme could take a similar form to the "actor folding" scheme proposed for the ParADE language to build grain-size by serialising a number of tasks. In Vorlon, locking could be optimised in a similar way by allowing the user to bound specific classes on the class diagram to indicate that they should be considered as one logical unit for storage, and thus when one of the objects is locked, all associated objects are implicitly locked, but at no extra cost.

A similar scheme could be employed if the mapping between Vorlon and NIP were to be changed (option 3). In an optimised mapping scheme, where the class diagram indicates a data structure composed of primitive types (or perhaps even non-primitive types with suitably small attribute sets), a single lock could be used to lock the entire structure, as opposed to locking individual elements as occurs now. In matrix multiplication, the benefits are obvious in that locking overhead would be substantially reduced. However, one possible drawback to this scheme is that there may be algorithms which exhibit far more complex data access patterns than merely by-row or by-column. If that is the case, then locking of such coarse granularity may increase the contention for access to data. Given that currently it is the locking cost itself (as opposed to waiting for locks) which is the significant penalty, coarse locking would seem to be favourable.

[18] In-depth discussion on changes to the language is offered in the next chapter.

If the application were to be changed, the developer would have to be aware of both the characteristics of the NIPDSM and of the mapping between Vorlon and NIP. Once those mappings are understood, the developer would then have to concoct data structures which avoided invoking the translator to use the costly parts of NIP. Whilst this is possible, it reduces Vorlon to the same level as its predecessors by exposing low-level (run-time) detail to the developer. As such, adopting this scheme should be opposed.

From the options identified, only (3) would result in performance benefit (for this particular application) without the need for further user intervention. Given the philosophy of Vorlon is to abstract details of parallel activity from the developer it is the most promising scheme. Whether this will eventually lead to further performance bottlenecks (because of non-regular access patterns to data structures) is unknown and can only be determined through further research.

This discussion raises a further important point for consideration - that of matching types and algorithms. In this experiment, because of the algorithm used it was implicitly assumed that coarse grain sizes would arise and that readily exploitable parallel activity would follow. Unfortunately, the locking strategy imposed by Vorlon is extremely simplistic (since no optimisations were made during the hand-translation of Vorlon graphs), and lead to a fine granularity of locks which in turn lead the poor application performance.

Particularly where containers of objects are being considered, the application developer must ensure that the algorithm being developed is compatible with the types available, and the developer of a (container) type should ensure that the types provide access mechanisms suited to a wide range of access patterns (which the `Matrix` and `Vector` types did not). Despite seeming contrary to the Vorlon philosophy of delegation of parallel processing issues to the run-time system, these are in fact higher-level software engineering issues, but in a context (parallel computing) which developers are not yet used to. In sequential software engineering, it is always thought of as good practice to harmonise the behaviour between data structures and algorithms and it is not therefore a huge step to ask that the same consideration be given to the development of classes to be used in a parallel application.

The fact that such considerations exist at all means that Vorlon in its current form does not implicitly support the construction of efficient parallel container types, and

relying on good software practice from programmers may be asking too much in this case. As such the only recourse for Vorlon at this point is to implement compiler-level optimisations, as have been outlined above, whereby for containers (particularly of primitive types) groups of elements can be locked with a single lock. Such a scheme would support novice users well enough (as has been proven with the hand-optimised locking scheme discussed earlier), and leave scope for advanced users to optimise application performance by exploiting their insights into likely access patterns for a particular container.

5.6 Parallel Compilation System

It has become the tradition for programming languages, that the one of the first applications to which they are applied is the construction of their own compilers, to ensure a language's self-sufficiency. In order to facilitate self-sufficiency of Vorlon applications, and to examine a more object-oriented problem, the second of the experiments presents a parallel compilation system for Vorlon applications.

As Vorlon is a visual language which targets an existing textual language (C++), the meaning of a "compiler" is ambiguous. It could be taken to mean the translation of graphs into the intermediate language, or the transformation of intermediate language code into machine code – though both are simply transformations. For purposes of experimentation, the latter of the two options has been chosen. The former, though equivalent in terms of overall methodology (translating one form of encoding to another) requires additional software support such as a graphical development environment in which Vorlon graphs could be built and parsed, which has not been developed during this period of research. Instead the approach takes a Vorlon project as might be produced by such a software development environment which consists of a number of C++ source file representations of Vorlon graphs and some meta-data on how those graphs are related, and performs a parallel compilation of all necessary source files into machine code.

It should be noted that the purpose of this experiment is not to explicitly create tool support for Vorlon, though tool support for the language would be valuable. Instead this experiment tests whether Vorlon can be used to explore an unknown problem domain, and via object-orientation, successfully carry forward a solution for that problem domain through to a software release. That being said, it is also important that performance gains

are achieved, but this is of lesser importance than simply being able to build parallel object-oriented software.

5.6.1 Aims

To develop a parallel *Make*-like {Oram and Talbott 1993} system from inception through to release underpinned by object-orientation. A secondary aim is to investigate the performance characteristics of the application developed.

5.6.2 Experimental Environment

The experimental environment for this set of experiments was the same as that introduced in Section 5.4. The only significant difference was the issue of disk access, since the compilation system is dependent upon ready access to source files. The times to access disk files were not identical, and on the shared memory multiprocessor and home user architectures, file access was found to be particularly expedient or particularly slow respectively.

5.6.3 Experimental Method

An abbreviated version of the development of the compilation system is presented here. The complete development of the application can be found in Appendix 1.

As with any object-oriented application, the starting point in the construction of the parallel compilation system is the analysis phase. In this problem domain, there exist a number of source code files, a project (which is essentially meta-data describing the relations between source code files and the final executable) and a compiler. In addition there were found to be a number of other data structures and types present to support the problem domain-specific types. Having gone through the high level analysis and design phases, the final class diagram for this problem was developed and is presented in Figure 5-39.

Figure 5-39 Class Diagram for the Parallel Compilation System

The class diagram of Figure 5-39 captures the overall composition of the application and shows the relationships between types in the problem domain. There are three primary types (`Project`, `Compiler`, and `SourceFile`), and when considering the model it is these types which encapsulate the functionality of the underlying system. The other types in the diagram (`List`, `String`, and `Object`) are there to support the problem domain-derived types and do not occur naturally within the problem.

The next development stage is to implement each type's methods. Since there is little in the way of compositional relationships as were apparent in the matrix multiplication application, there is no need to build the parallel compilation application bottom-up. Instead, it was felt that a top-down approach was more appropriate, and as such the starting point for this application is its `main(…)` method which can be seen in Figure 5-40.

Figure 5-40 Parallel Compilation System Main Method

The `main(…)` method of the compilation system contains a great deal of potentially parallel activity. Once an instance of the `Project` class has been created, everything until the `link` operation is potentially parallel. These parallel operations include obtaining a list of out-of-date files for the build, and information on which compiler is to be used and which compiler switches are to be activated in order to create an instance of the `Compiler` class. Once the `Compiler` object has been instantiated, and the list of out-of-date files has been produced, the parallel compilation of each of those files is undertaken by a parallel method call node which uses each element of the input `List` to spawn the `compile(…)` method of the `Compiler` class. Once all compilation has

finished, the final stage is to link the necessary object code files together into an executable program which is done by the `Compiler` type's `link(…)` method.

Although several method calls are made from within the `main(…)` method, the two which are most interesting directly pertain to the compilation of files (as opposed to setting up the environment for the compilation to run). The first of these is the `getOutOfDateFiles(…)` method from the `Project` class which returns a `List` of `SourceFile` objects that is then consumed by the `compile(…)` parallel method call node which runs a (potentially) parallel compilation on each out-of-date source file. The `getOutOfDateFiles(…)` method is shown in Figure 5-41 and its decomposed sub-graph in Figure 5-42.

Figure 5-41 Obtaining Out of Date Files for Compilation

The method graph shown in Figure 5-41 performs the action of building a `List` of source files whose last modified date is newer than that of the target executable. It does this by first creating an empty `List` of out-of-date files and sending that empty `List` to a parallel sub-graph, shown in Figure 5-42, where it is filled with instances of the `SourceFile` type which are out-of-date. Note that in Figure 5-41 the explicitly parallel form of the parallel computation node is used since not all of the input `List`s are to be decomposed and as such the automatic decomposition semantic of the parallel

computation node is unhelpful here. Once the sub-graph has finished its (parallel) operations, the method completes and returns a `List` of all out-of-date files to the caller.

Figure 5-42 Obtaining Out of Date Files for Compilation Sub-Graph Decomposition

Although the sub-graph shown in Figure 5-42 is sequential because of its pattern of dependencies, it must be borne in mind that executing multiple copies of this graph will be activated thus achieving parallelism. It is also important to remember that although the sub-graph node takes `List`s as its parameters, those `List`s are not automatically decomposed, and it is the responsibility of the sub-graph itself to access the required elements from the data structures.

The overall functionality of the sub-graph of Figure 5-42 is straightforward. A `List` structure is presented as a parameter and a `SourceFile` object is retrieved from that `List` according to which instance of the node program is running (from zero to one less than the size of the `List`). The age of the `SourceFile` object is compared with the age of the target application, and if the `SourceFile` object is found to be younger than the target application it is added to a second `List` which stores the `SourceFile` that need re-compilation. If the current `SourceFile` is not out-of-date then the sub-graph returns immediately. Note that there may be many parallel calls to add SourceFile objects into the `List` of out-of-date files, and as such the order in which they are added is undefined. However, this does not affect the program as a whole since order of compilation is unimportant.

The `compile(…)` method from the `Compiler` class is the second of the more important method calls in the `main(…)`, and is shown in Figure 5-43.

Figure 5-43 The compile(…) Method from the Compiler Class

The `compile(…)` method is invoked via a parallel method call node from the `main(…)` method of Figure 5-40. Although the method itself is largely serial because of dependencies, it still has the potential to exhibit parallel behaviour. The functionality of the graph is straightforward and largely involves string concatenation. The string concatenation simply builds a compilation command which can be sent to the shell command interpreter in order for the compilation of a source file to occur. An example of a typical string might be something like:

```
g++ -c –I/home/me/include –L/home/me/lib –lmylib myfile.cpp
```

The final node in the graph simply calls a C library function (the `system(…)` call) in order to pass that compilation command to the shell. In an ideal situation, each available processor in the underlying parallel machine will be used to execute one compilation command at a time in order to achieve faster build times for software.

5.6.4 Parallel Compilation System Performance Results

For the performance tests, all three types of test platforms described earlier were used to build the resulting C++ source files from the Vorlon graphs, and the performance results were compared to an equivalent serial build using the Unix tool Make.

5.6.4.1. Shared Memory Multiprocessor

Single four-way Pentium III Xeon 500MHz (512KB cache), 512MB main memory, Ultra SCSI disk, running Red Hat Linux 6.2

The shared memory multiprocessor used for this experiment is designed as a server computer, and as such is optimised for typical server applications, importantly including file access. Since a great deal of work for this problem involves reading in data from files so that it can be compiled (in parallel), an optimised disk system is advantageous.

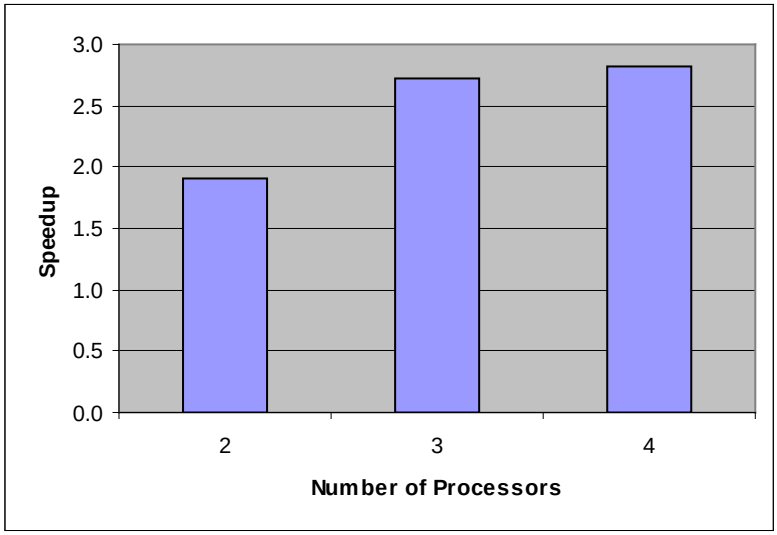


Figure 5-44 Speedup for Compilation System on a Four Processor SMP Platform

The performance data shown in Figure 5-44 shows speedup for all processor configurations. For two and three processors, speedup is as good as could be expected in practice, taking Amdahl's Law into account. However, the four processor speedup is not in line with what would have been expected if the two and three processor results had been projected to four processors. The performance increase in adding the fourth processor should have resulted in an execution time being around 3.5 times faster than the sequential version, yet the actual figure is more like 2.8 times. The reason for the more limited speedup is contention for access to files on disk. Had the source files been located on separate disks then, because of the characteristics of the SCSI interface,

concurrent access to files would have been supported and speedup, particularly for the four processor experiment, should have been more similar to the projected speedup.

5.6.4.2. Uniform Cluster

Eight uniprocessor Pentium II 266MHz, 64MB main memory, IDE disk, running Red Hat Linux 6.2 with exclusive access to shared 100Mbit Ethernet network, where one host acts as a fileserver.

The cluster computing environment provides a larger number of processors than the SMP, and given sufficient numbers of source files to process, should logically be able to complete more compilations in parallel than the SMP. However, unlike the SMP, the cluster computer is not optimised for disk access even at the individual host level, since IDE disks are used, and certainly not at the cluster level where NFS is used to transport file systems between hosts. It is unfortunate that the same network which applications use to swap messages via the NIP DSM is also as a route via which NFS traffic, in this case the C++ source files, moves between hosts. This situation is further compounded by the fact that one of the processing nodes acts as a fileserver and the fact that the network is a shared, as opposed to switched, medium which prevents more sensible routing of the two types of traffic (communication and file movement). Thus the results gained from the cluster experiments represent what might occur in a typical user cluster, rather than on a cluster dedicated to parallel processing.

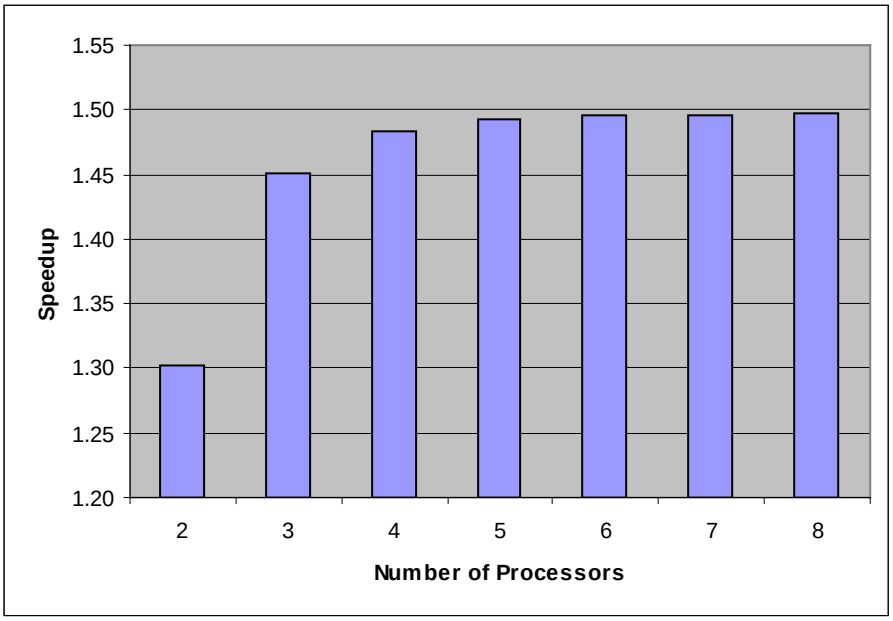


Figure 5-45 Speedup for Compilation System on an Eight Node Cluster Computer

The performance data presented in Figure 5-45 is symptomatic of the I/O bottleneck that cluster computing suffers from. Although speedup is attained throughout, the fact that access to the source files was performed via NFS meant that, even with large numbers of processors, performance was modest at best.

For two nodes, network contention is low and so overall performance is not significantly hindered. As the number of nodes increases, so does contention for resources, including the network, file services on the file server, and access to the disk containing the source files itself. At each successive increase in processors (up to and including six nodes) the performance increases are successively less because of the contention.

At seven nodes the performance flattens out. This is not because of reduced contention, but because the compilation system application is composed from only six source files (which derive from the class diagram of Figure 5-39) and as such there is no more parallel activity from the problem domain that the computer can exploit. On eight nodes there is a slight performance increase which is attributed to the fact that the eighth node is the fileserver, and is thus logically close to the source files. Since the eighth node has fast access to the source files, it is in a position to compile more quickly than the other nodes, which is does in addition to performing its duties as a fileserver.

5.6.4.3. Ad-Hoc Environment

Dual Pentium III 800 MHz SMP, 256MB main memory, IDE disk, and Pentium II Mobile 300MHz, 128MB main memory, IDE disk, both running Windows 2000 SP1 with non-exclusive access to shared 10Mbit Ethernet networking.

The ad-hoc parallel platform suffered from precisely the same problems as the cluster, only more acutely. In the ad-hoc platform not only was the fileserver itself an active computational node, but the network joining the two hosts is slow (10Mbits per second) and shared amongst a number of users.

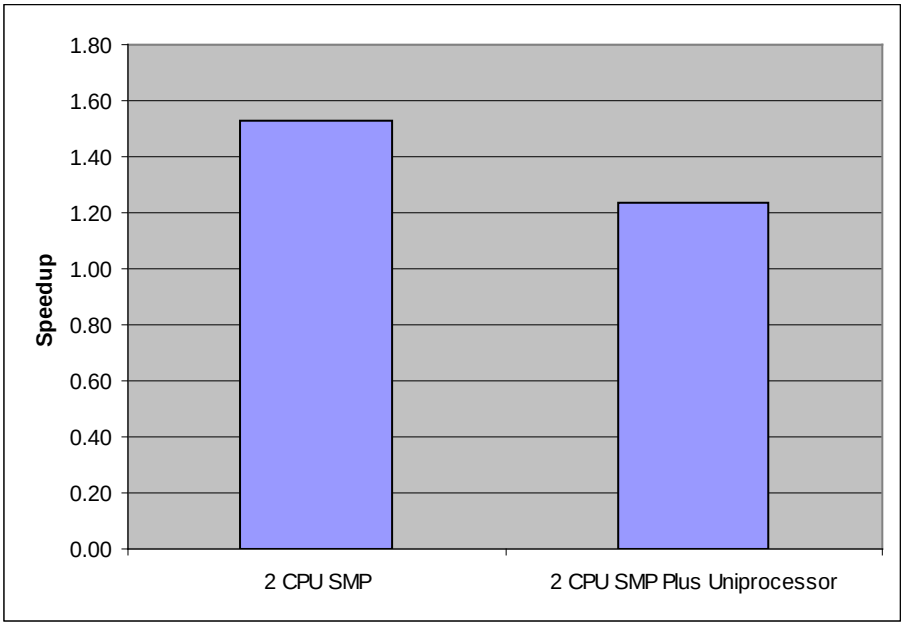


Figure 5-46 Speedup for Compilation System on an Ad-Hoc Parallel Platform

Within the two-way SMP performance is quite acceptable. For two processors a speedup of 1.5 times that of the sequential application running on one of the processors in the same machine. In this case the theoretical ~2 times speedup is not attainable since the disk architecture of the machine (IDE) is not well suited to efficiently supporting concurrent disk access.

When the second (slower) host is added into the system, performance degrades from 1.5 times to around 1.2 times speedup. This reduction in speedup happens because of the fact that the two-way SMP has to spend some of its time acting as a fileserver, rather than as a processing node, and because file access is slow across a low-bandwidth and high latency interconnection network. Once again adding older hardware to a configuration to support newer hardware has had the opposite affect to that desired.

5.6.5 Parallel Compilation System Conclusions

In the case of the compilation system, performance is generally good insofar as speedups are attained across all configurations, and follows a predictable pattern where shared memory performance is best, whilst the distributed memory platforms perform less well. In this case since the number of objects was much lower compared to the matrix multiplication experiments (typically of the order of a few tens per application) and the granularity of most of the method calls higher (especially the callouts to the C++ compiler), locking was a much less significant aspect of the system. In this experiment it was disk access that was the most important factor in determining speedup.

Even with very coarse grained parallelism such as that present in this problem, adding older hardware to help with the computational workload does not necessarily mean that better performance will be obtained. Whilst it is true that better networking may have helped the ad-hoc cluster a little, the cost benefits of such a scheme may outweigh the benefits.

In order to change the performance characteristics of the application, it is the issue of disk access which would need to be addressed. Since disks cannot be made faster by software applications, nor network bandwidth higher, the only way in which performance can be improved at the application (as opposed to hardware) level is by reducing contention for access to such resources. In this case it is not necessary to look to changes to the language, or improvements in the run-time system, but instead to look at the application structure. One straightforward improvement that could be made is to perform a file copy operation before compilation so that when compilation occurs the source file is being read from a local disk rather than from a network volume. This could be relatively simply achieved in Vorlon by sub-classing the `Compiler` type in the application to include facilities to perform file copies before compilation occurs. This is a clear benefit of object-orientation in that all of the work previously invested in building the `Compiler` class is not wasted, but re-used in a structured fashion to build a type that more closely suits the device access pattern of the application. Such a scheme would mean less contention for the network volume (and less thrashing of that disk) and less contention for the network, and given the less contentions state, better performance should be achieved.

With the benefit of expert knowledge of the underlying system, such changes as have been suggested above may seem a viable route to take to improve application

performance. However, this is in some sense difficult to completely reconcile with the Vorlon philosophy of abstracting even the virtual parallel machine that the language translator targets. Furthermore, a normal Vorlon developer would be shielded from this detail and would not have access to such hardware architectural details. Thus whilst object-orientation provides richly for the higher-level aspects of the system development (from analysis through to implementation), and parallelism was attained throughout the application, there was no optimised support at the execution level for the actual file operations that the application was performing. This is not a vastly different problem from that experienced with matrix multiplication where the algorithm design and the particular type implementation were incompatible. Here a similar mismatch between the logically specified parallelism and the physical execution of that parallelism occurs, since the only resource taken into consideration for the execution of a task is CPU availability, and not other resources such as access to I/O devices.

This does not suggest that an amendment to the language syntax is required as had been contemplated for the matrix multiplication experiment[19], but if expert knowledge is not available, that better support for device access is needed. This is an issue which should most likely be addressed at the Vorlon library level, whereby components optimised for a typical parallel hardware architecture could developed (in Vorlon) and used in preference to the external language library calls which are non-optimal for parallel computing problems. If such components were available for the experiments, then it is entirely possible that the true benefits of building object-oriented parallel software (full lifecycle support, improved performance) would have shone through, as they did on the SMP platform where I/O was much less of a problem.

5.7 Summary

This chapter has shown the development of two distinct classes of applications using Vorlon. Importantly, is has been shown how Vorlon supports the development of an application throughout the software lifecycle, and has demonstrated how the language syntax works.

This chapter has presented the performance of the two applications developed. In the case of the matrix multiplication application, slowdown was observed, whilst in the case of the compilation system reasonable performance was obtained. Reasons for the

[19] Language issues arising from both experiments are dealt with in-depth in the next chapter.

performance figures were discussed and means of avoiding those conditions which incur performance penalties were also alluded to. In particular, the importance of minimising locking from information available at compile-time was identified from the matrix multiplication experiment, whilst the cost of file access was identified as a particular problem for the parallel compilation system.

The engineering and language aspects of the work have not been commented upon, and will instead be dealt with in the next chapter.

Chapter 6 Reflections on the Vorlon Language

This chapter examines the Vorlon language using the taxonomy criteria that have been applied to other visual parallel programming languages in Chapter 3. Language issues arising from the experience of developing applications using Vorlon are also presented, including possible amendments to the language and future directions of some of the techniques developed in this thesis.

6.1 Introduction

Chapter 5 introduced two applications developed with Vorlon and presented experimental procedure and performance results across a range of hardware platforms. It was found that as expected performance varied between applications and across hardware platforms, although performance was found to be within reasonable limits. However, it is not only performance which is crucial to the success of Vorlon. For the language to be successful it must also address higher-level software engineering concerns, something that Chapter 2 showed was a significant weakness in prior visual parallel programming languages. To see whether the same is true of Vorlon, this chapter examines the software engineering aspects of Vorlon, and identifies key strengths and weaknesses of the syntactic elements of the language and the execution model.

6.2 Applying the Taxonomy Criteria to Vorlon

A taxonomy for visual parallel programming languages was presented in Chapter 3, where it was used to characterise four previous languages. The taxonomy provided some key insights into the execution models that particular types of syntactic elements support. It is fitting at this point to subject Vorlon to the same criteria that its predecessors faced. As with the earlier use of the taxonomy, each of the important classes of syntactic elements will be scrutinised individually, starting with the graph.

6.2.1 Graph

Graph Semantics			
Sub-Graphs	**Structure**	**Node Dependencies**	**Recursion**
Decomposition and Method Graphs	Directed Acyclic	Implicit	Permitted (through recursive method calls)

Table 6-1 Vorlon Graph Semantics

Vorlon supports sub-graphing both for both managing visual complexity (computational node decomposition) and for structuring the application to suit the

problem domain (multiple method graphs per type). Graphs are acyclic and directed, and all dependencies between nodes on a graph are explicit in that, to be mutually dependent, nodes must be connected by arcs. Recursion is supported through recursive method calls, and does not warrant specific language syntax.

6.2.2 Nodes

Node Semantics		
Type	**Purpose**	**State**
Computational Graph Management	Special General	Cannot be retained

Table 6-2 Vorlon Node Semantics

Vorlon supports both computational (method call, computation) and graph management (replicate, merge) nodes. Graph management nodes are used to circumvent the need for textual graph annotations and to maintain the notion that all control issues are dealt with graphically. Special (predefined) and general (user-defined) computational nodes are also supported where nodes such as the method call node have a predefined action, whilst nodes such as the computational node may encapsulate user-defined activity. State cannot be retained by nodes since it is the objects from which applications are built that maintain state, and not syntactic elements from the language.

6.2.3 Arcs

Arc Semantics					
Direction	**Tokens Carried**	**Structure**	**Capacity**	**Consumption**	**Connection**
Supply	Handle	Single item	Single item	When used	1 : 1

Table 6-3 Vorlon Arc Semantics

Vorlon arcs are supply driven in that they only carry elements once the producer of the element has produced, and not when the consumer of the element demands it. Arcs in Vorlon, in keeping with the Parallel Object-Flow paradigm, carry handles (or references) to objects, though they are not equipped to carry structures such as arrays of references, as previous languages were. This is in fact not a disadvantage in Vorlon since the a similar semantic to passing structures is maintained, since arcs can carry handles to arbitrary data types including any kind of structure.

Since Vorlon graphs are acyclic, and nodes can only activate once per graph, arcs carry only a single item per graph instantiation. Once an object handle reaches its destination, it is consumed from the arc upon activation of that destination node. Arcs

themselves always connect two nodes and as such exhibit a 1:1 mapping, leaving more complex schemes to be implemented by special purpose nodes (mentioned earlier).

6.2.4 Node: Firing Rule

Node: Firing Rule Semantics
Fixed

Table 6-4 Vorlon Firing Semantics

Since all dependencies in Vorlon graphs must be explicitly maintained via arcs connecting nodes, all Vorlon nodes are subject to the firing rule. There are no implicit dependencies. An important consideration in the design of Vorlon was to keep control flow issues entirely within the graphical element of the language, firing rules were kept deliberately simple, and fixed to an all-inputs rule whereby a node would activate only when all of its input arcs were carrying handles. The only exception to this rule is the merge node which is used where conditional execution is present in a graph.

6.2.5 Node: Inputs and Outputs

Node: Data Input Semantics
Shared
Implicit

Table 6-5 Vorlon Data Input Semantics

Node inputs and outputs in Vorlon consist solely of handles. It is permissible for any number of handles to refer to the same object, and although the handles themselves cannot be shared (though they can be copied) the overall semantic is one of object sharing. Since the graphical syntax of the language encapsulates low-level detail such as locking, this sharing is implicit, though the developer must ensure correct sequencing of accesses to shared objects through the dependencies in the graph.

6.2.6 Node: Program

Node: Program	
Language	**Variable Mapping**
Standard	Implicit

Table 6-6 Vorlon Node Program Semantics

For those nodes which permit user programming with textual source code (the computation and parallel computation nodes), the main issues are whether a standard or proprietary language is used to program those nodes, and whether there is support for automatic mapping between graphical and textual objects. In Vorlon, the language used to program computation nodes is standard C++. Mappings between textual and graphical objects is predominantly a tool support issue, but since the environment of Vorlon nodes is dictated solely by the arcs which are incident upon them, automating

such mappings would be straightforward to implement. As such they are thought of as being implicit.

6.2.7 Node: Output Token Production Rule

Node: Outputs		
Arc Coverage	**Explicit Send/Implicit Send**	**Streamed Output**
All	Implicit	No

Table 6-7 Vorlon Node Output Semantics

Since Vorlon nodes only have one outgoing arc, the arc coverage for the output production rule is all arcs. Even in the single exceptional case, the conditional node, the arc from one distinct output area of the node carries an output token, though it is admittedly a fine distinction. All sends are implicit since nodes are only permitted to output handles onto arcs once they have completed their execution, and as such the end of a node's execution implies an output on its output arc.

6.2.8 Support for Parallel Modes of Execution

Behaviour	Node
Data parallel	Parallel Computation Node Parallel Method Call Node
Task Parallel	Implicit in Graph Structure
Pipeline	Not supported
Iteration	Loop node
Conditional	Conditional Node
Recursion	Method Call Node

Table 6-8 Syntactic Support for Parallel Execution in Vorlon

Explicit data parallelism is instantiated with either the parallel computation or parallel method call nodes, depending on whether its is a textual computation (or a sub-graph) which is to be executed in parallel, or a method call on a number of objects. Though the developer may be expected to notice the opportunity for data parallel style activity, typing can be used to implicitly drive that activity where appropriate.

There is no specific syntactic element for task parallelism, since mutually independent, and thus potentially parallel, tasks can be identified from the overall graph structure. There is no syntax for inducing pipeline parallel activity since Vorlon graphs do not support pipelines (for reasons discussed later).

Iteration is not a parallel construct in Vorlon, though the type-driven aspects of the explicitly parallel nodes could be seen as a kind of `foreach` construct. The loop node is instead a purely iterative construct as recommended by ParADE {Allen 1998}.

Recursion is supported by the Vorlon language insofar as any method graph is free to invoke itself via a method call node. If the recursive method does not update the state of its associated object (it is a read-only method) then parallelism may occur since multiple instances of that method may run concurrently. It is noteworthy that recursion at the sub-graph level is not supported by Vorlon since there is no naming convention for sub-graph nodes which means they cannot be called from any graph node (unlike methods which, by definition, are named).

6.2.9 Notable Omissions from the Vorlon Language

Compared to contemporary visual parallel programming languages, Vorlon has some significant omissions from, and restrictions inherent in, its syntax. That is not to say that a range of other syntactic elements was not considered for adoption, but simply that they were either incompatible with the Vorlon model, or were found to be unsuitable abstractions for application-level parallelism. Specifically, Vorlon provides no mechanisms for pipeline parallelism, or for streams of handles to occur either in its syntactic elements or its graph semantics. The next sections discuss why such facilities were deliberately left out of Vorlon.

6.2.9.1. Pipelined Parallelism and Related Syntactic Elements

Mechanisms and execution models supporting pipelined parallelism have been the norm in visual parallel programming languages to-date. Value has been placed on the ability to exploit parallel behaviour whilst maintaining a strict sequential ordering of tasks. This type of parallel activity has been proven in everyday use, from a car production line through to pipelines within computer processors. So, given that pipelining is seemingly valuable it begs the question as to exactly why it is not supported in Vorlon.

The answers to that question are manifold, and rest on the premise that software is a logical, not a physical entity. In the first instance, imagine a car plant or CPU pipeline. The designers of that plant or CPU have gone to great lengths to ensure that the available real-estate is used to its best potential. That is they seek to address the question, "Given a limited physical space, how is maximal concurrent activity achieved?". In such cases the pipeline is useful, providing for a fixed level of parallel activity which given careful design can be maximised for the space available.

Software does not have a physical dimension. It is then a strange exercise that developers impose a parallelism abstraction designed for real-estate economy in the physical world onto a world where physical space has little significance (like the idealised parallel machine which Vorlon targets where physical computer characteristics are unimportant). This is the first problem with application-level pipelined parallelism: it does not take into account the "physics" of the environment in which software resides.

Secondly, again consider the traditional pipeline, be it either software or physical. The maximum level of parallelism which can be exploited is equal to the number of stages in that pipeline. If sufficient input tokens exist to feed the pipeline then that maximal parallelism will be achieved, but no more. This makes perfect sense in the physical world where machinery cannot be added and removed at will. In software where "virtual" machinery can be added and removed relatively quickly and inexpensively, obeying physical world rules for pipelines seems wrong, and yet it is done and furthermore encouraged by the abstractions available in some visual parallel programming languages.

Again consider a pipeline and imagine it turned into a piece of sequential code. If the pipeline is executed, there is no potential for parallelism. However, now imagine a situation where there are two tokens available to the first stage of the pipeline. If the pipeline is sequential again no parallelism is available, but in the virtual environment of computer software, another (zero-parallelism) pipeline can be constructed to accept the second input token, and immediately there exists parallel activity. Now imagine M available input tokens to a the pipeline, where M is greater than the number of stages in the original pipeline. In this case, logically there exists parallelism of M, which cannot be achieved by the standard pipeline (whose parallelism is restricted to N, the number of segments in the pipeline). This is the second problem with application-level pipelined parallelism: stifling potential parallel activity (Figure 6-1).

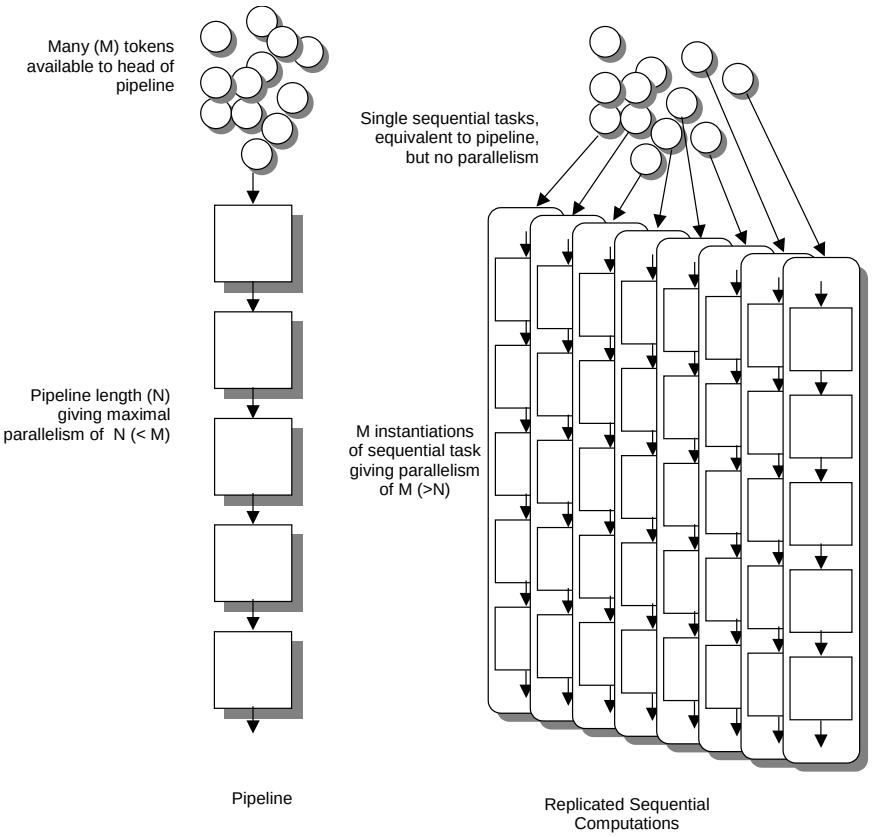


Figure 6-1 Pipeline Reduces Potential for Parallelism

Once again consider a physical pipeline. In such a pipeline each stage is significantly shorter than the length of the entire pipeline, taking approximately $1/T$ the duration of a pipeline which takes time T to complete. In a software pipeline the same is true.

In a parallel computing environment the issue of grain size is paramount. In a pipeline grain size is reduced since one coarse-grained task is split into several finer-grained tasks. Since coarser grain sizes give a much better chance of efficiently exploiting parallel activity than finer grain sizes, developers should be looking to increase, not decrease grain size. This is the third problem with application-level pipelined parallelism: it reduces grain size.

Parallel software executes within a somewhat unpredictable environment where, depending on hardware conditions, identical tasks may take differing amounts of time to complete. In a physical pipeline this situation cannot be allowed, and significant effort goes into ensuring that each stage of the pipeline takes almost exactly the same amount of time. Given that in a parallel computing system there will be contention for resources, and that such contention is largely unpredictable, application-level pipelines are far more susceptible to underlying hardware conditions than other mechanisms. Imagine a situation in a pipeline where one stage requires access to some data which is not currently

available, for example it could be residing on another node in the computer system and thus require network transfer. Waiting for that data will cause the pipeline to stall, and incur associated performance penalties. Compared to a "data parallel" approach, the pipeline has multiple points of failure in terms of performance (one at each pipeline stage). Here lies the final problem with application-level pipelined parallelism: it requires that the developer understands each stage of the pipeline to ensure efficient implementation.

Given the number of drawbacks to logical pipelines, and that more coarsely grained tasks with an equivalent behaviour can be obtained using data parallel activity, the argument against directly supporting application-level pipelines is strong. Thus Vorlon does not support pipelining.

6.2.9.2. Streaming Nodes and Streamed Graphs

Streaming occurs in a graph when a node is permitted to produce output repeatedly. In the normal case such repeated outputs would be used to invoke more activity in a graph and potentially increase parallelism. Streaming is somewhat similar to pipelining, except where in a formal pipeline stages are strongly coupled together in terms of functionality, in a streamed graph they are not.

Streaming is clearly a convenient way of keeping active what would otherwise be dormant graph nodes in order to obtain a state of maximal parallelism. However, there are a number of problems with streaming, as an investigation into the actual semantics of the ParADE loop actor revealed. The functionality of the ParADE loop actor was found to have unwittingly played a dual role, in that it bears both the responsibility for iterative behaviour and for instantiating streaming activity. This duality was not acknowledged in the ParADE work, save for the fact that it was known that there were problems in halting ParADE applications where streaming was exploited.

The problems with the ParADE loop actor stemmed from the fact that a loop node could output data values on each iteration of the loop. Although the purpose of the per-iteration output in the loop actor was clearly out of place in a mechanism designed for iteration, from a parallelism point of view, it was thought to be very useful. The output of values on a per-iteration basis allowed the ParADE programmer to stream data values out along an arc, and thus to invoke the activity of nodes further along the graph whilst current nodes were still working.

However, the way in which the ParADE loop actor was used as a means of invoking pipeline parallelism was found to be unsafe. In particular, due to the fact that that there is no means of checking for the end of any streaming activity invoked by the loop actor, there is the possibility of introducing a race condition into a ParADE graph. Imagine a situation, such as that depicted in Figure 6-2, where a loop node positioned towards the top of a graph begins streaming out data values, which feed a number of subsequent nodes[20]. At the end of this graph is a halt actor which terminates the program, activated when an item of data is transmitted to it. At some point in the execution of the program, several nodes in the stream are processing and the node nearest the halt actor emits an element of data along its outgoing arc. At this point the program stops, despite processing still occurring within the other nodes. Whilst the conscientious programmer will be prudent enough to monitor the number of elements streamed by the loop actor, and count that number of elements through the stream before allowing the program to close, this is not mandated in ParADE.

Figure 6-2 Race Conditions with ParADE Streaming

Having discovered that the streaming semantics of the ParADE are weak, other mechanisms of inducing parallelism had to be investigated. If the other dataflow-like languages surveyed in Chapter 2 are recalled, only one other streaming approach was found. In the CODE language {Newton 1993} a single computation node is free to

[20] For the sake of clarity, a pipeline-like structure is used in the example, though in reality streams of data could permeate deep and wide in a graph.

produce multiple outputs per activation, which may subsequently instigate graph streaming.

This approach is not easily reconciled with the Vorlon language philosophy. In Vorlon, control flow issues are specified purely in graphical form which immediately discounts the CODE approach of embedding control flow commands in the textual node program.

Since streaming seemed an unnecessary complication, it was not adopted by Vorlon. Initially it was feared that parallelism would suffer, but since most parallelism is based upon concurrent invocation of methods (task parallelism), the lack of streaming has not been a significant drawback. However, there have been benefits in terms of graph readability attributed to the fact that there is no streaming. Since each node activates once only, it is possible to see all possible control-flows on a graph at compile-time and be certain that the same will occur at run-time, just as was advocated by the execution model for HeNCE.

6.3 Software Engineering Issues from the Experiments

The single most important feature demonstrated by the experiments developing Vorlon applications was the fact that, through object-orientation, support for the entire software lifecycle was attained. Each application was developed from early analysis phase through to design, development and release within proper engineering framework, which allowed seamless transition from one stage to the next (and back again as necessary). No such engineering facilities have been seen in a visual parallel programming language to-date, which are highly implementation-centric.

Aside from the lifecycle support that object-orientation provides, perhaps the next most obvious factor that differentiates Vorlon applications from those developed with previous visual parallel programming languages is their size. If any of the matrix multiplication programs from Chapter 2 are considered alongside the Vorlon equivalent, it is clear that the Vorlon application is considerably larger than any of the others, stretching to several tens of nodes across several graphs. When compared to even the largest of the previous matrix multiplication programs (CODE, composed from two graphs) the Vorlon program seems quite unwieldy.

Although there are obvious size costs in using Vorlon to implement such applications, these are offset to some degree by other benefits. The fact that Vorlon is object-oriented,

and thus requires the implementation of types which constitutes the majority of the work of an application programmer, also means that it is extensible and encourages re-use.

Extensibility is an aspect which no other visual parallel programming language has been able to offer, outside of limited functional re-use. Once the set of supported abstractions was fixed in any of the previous languages, it was fixed permanently. Whilst the abstractions are often extremely useful, such as the automatic array decomposition of ParADE's depth actor, they cannot be extended. Being object-oriented, the Vorlon language offers extensibility. If there is no suitable abstraction available from the language directly, then the developer is free to extend the language by developing types to suit the problem under consideration.

Vorlon does not natively support matrices and must therefore be extended to provide them. In the other languages, and most notably ParADE, the notion of a matrix (or at least arrays of arrays) is built into the language syntax. The point here is that once the Vorlon language has been extended to include matrices by developing a `Matrix` type, the actual implementation of the matrix multiplication program is as straightforward in Vorlon as it is in any of the other languages. In fact, the `main(…)` method in the Vorlon matrix multiplication is of the same order of magnitude in terms of number of nodes of any of the other matrix multiplication programs seen in Chapter 2.

Vorlon is extensible since it supports necessary abstractions through typing, and not via language syntax. Whilst it is true that the implementation of types may be time consuming, it is also true that once a type is implemented it will not need to be re-implemented often. This is the second key advantage offered by object-orientation – type re-use. The re-use of types allows re-use much earlier in the software lifecycle, and is a more powerful model of re-use than function libraries since types can be specialised. If, for example, a developer needed to investigate another numerical analysis problem using matrices then the Vorlon approach pays dividends since the now-developed `Matrix` type can be re-used early in the lifecycle, and where appropriate simply specialised to suit the new problem domain. The other visual languages offer no such means of re-use - all written code is inapplicable to the new problem unless that new problem specifically uses a matrix multiplication subroutine, and even then re-use can only occur much later at the implementation stage.

In short, because of its object-oriented nature, Vorlon provides significant benefits in terms of re-use and extensibility. Whilst it is true that Vorlon developers have to develop

types for any new problem domains, in the long term the benefits afforded by object-orientation should outweigh costs, and development cycles should be shortened.

6.4 Syntactic and Semantic Improvements for Vorlon

Vorlon is a prototype language for visual object-oriented parallel programming. As such, it is by no means perfect but is valuable as a means of exploring new approaches. Having gained experience developing applications with Vorlon, it is clear that there are a number of aspects of the language which could be improved, removed, or included. The following sections discuss such language amendments, including:

- Possible improvements to the current language.
- The removal of particular language elements.
- The possible inclusion of new language elements.

6.4.1 Improvements for Current Language Syntax

There are two main aspects of the Vorlon programming language which it is felt would warrant further refinement in future versions of the language. These are:

Conditional execution; and

Explicit data parallelism.

The problem with the conditional execution node is simply that it is confusing to use. Whilst the node itself always executes when activated by handles on all input arcs, the node program or sub-graph which the node encapsulates may not. This is not consistent with other nodes in the language where activation implies the execution of the node's functionality.

As a replacement, conditional execution could be subsumed into the semantics of the computation node, whereby multiple outputs could be attached to the node and subsets of these outputs could be activated depending on certain conditions within the node program. Note that more than one output may also be allowed in a future version of the node since such permits multiple return paths which may be desirable for a parallel language (as first noted in Chapter 3), though for conditional execution some of these multiple outputs would be mutually exclusive. A possible form for the future computation node is depicted in Figure 4-1.

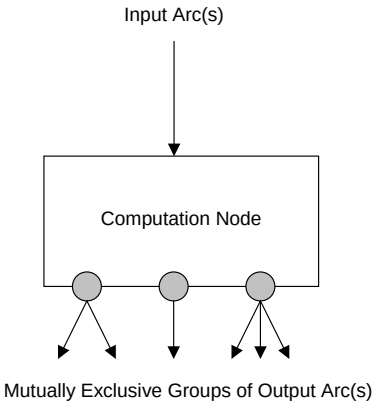


Figure 6-3 Possible Future Syntax for Conditional Execution Supporting Multiple Return Paths

Although the conditional execution node shown Figure 4-1 is an improvement over the current node, it does not solve all the problems associated with conditional execution. The need to alter particular firing rules to accommodate conditional behaviour remains, and the only solution proposed in this thesis has been the adoption of the merge node for "gathering together" arcs from multiple conditionally executed sub-graphs, which would have to either remain or be supplanted by additional semantics added to (groups of) halt nodes.

The other aspect of Vorlon which requires refinement is the explicitly parallel aspects of the parallel computation node. This node and the corresponding parallel method call node have a dual personality whereby they can be explicitly parameterised to perform data-parallel operations, or can use the type information from their input arcs to implicitly perform operations in parallel. Given that implicit parallelism is the better choice since it does not burden the developer, and that one of the aims of Vorlon was to present the user with an environment which does not require that parallelism be directly identified, it is proposed that the explicitly data-parallel form of the node should be removed from the language. Instead, the parallel computation node should be reduced to its canonical form where data-parallelism can be automatically extracted from the types of the handles carried by the input arcs.

It is not a significantly bigger step to suggest that the node itself could be removed from the language, and its decomposition and data-parallel semantics be added to the standard computation node. In this case, if the computation node's function signature takes individual arguments of a certain type as its parameters, and the input arcs provide linear data structures (such as lists, vectors, arrays and so forth) composed from instances of that type, then it should be possible to automatically decompose the input structures

(and correspondingly re-compose the output into a similar structure). This is shown in Figure 6-4.

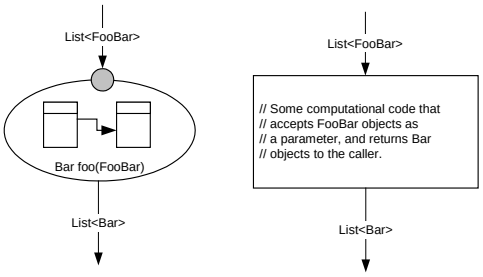


Figure 6-4 Data Structure Decomposition to Facilitate Type-Driven Data-Parallelism

The graph shown in Figure 6-4 shows a computational node whose function prototype specifies an instance of the type `FooBar` as its arguments, and `Bar` as its return type. The input arc is able to carry a handle not to an instance of `FooBar` but to an instance of `List<FooBar>` whose contents will automatically be extracted and a single instance fed to each parallel instance of the computational node. When each instance completes, the output from the node is built into a `List<Bar>` from the individual instances of `Bar` produced by the parallel node programs.

Whilst such decomposition semantics may be useful, they are not especially generic in that they rely details of the decomposable data structures being exposed to the language. This is quite contrary to the extensible nature of Vorlon, and so whilst this simple decomposition mechanism offers a "quick-fix" to the problem (as did list decomposition in the current version of the language) it is clear that any major language revision should tackle this issue at a more fundamental level.

6.4.2 Possible Future Additions to and Removals from Vorlon

The fact that data-parallelism, which is traditionally explicitly invoked by the user, may be attained implicitly through the language's type system is interesting. The discussion in the previous section noted that the parallel computation node should, if its functionality is retained in the language, rely solely on type information to decompose linear data structures into a form suitable for data-parallel processing. However, it may be possible to extend that automatic decomposition of data structures to arbitrary types[21]. Specifically, in a class diagram, there is a great deal of information available which has not

[21] With thanks to Savas Parastatidis for his constructive input in formulating these ideas.

been capitalised on in the current Vorlon language save for producing skeleton class interfaces, but which could be used to facilitate more automated decomposition of data structures. Take for example a tree data structure such as that modelled in Figure 6-5.

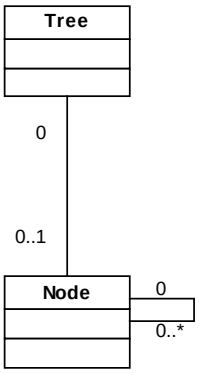


Figure 6-5 A Tree Data Structure

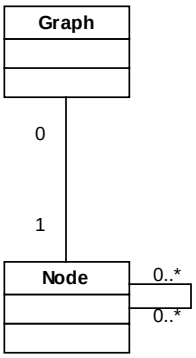


Figure 6-6 A Graph Data Structure

Figure 6-5 shows a typical UML representation of a tree. In this case, the `Tree` class is associated with zero or one instance of the `Node` class, and the node class is associated with zero or more instances of itself. A similar pattern is observed in Figure 6-6, where the `Graph` class is associated with zero or one instance of the `Node` class, and any instance of the `Node` class is associated with any number of other instances of itself.

Neither of the patterns seen in Figure 6-5 and Figure 6-6 are especially different to the interrelationship pattern exhibited by the `List` class in Figure 6-7, which has already been shown to be an implicitly decomposable data structure that can be used to drive data parallel activity.

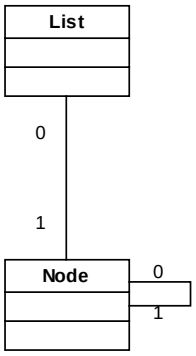


Figure 6-7 A List Data Structure

Given that there are known algorithms for traversing data structures {Sedgewick 1992}, it is possible to decompose those structures into their individual elements. Once

individual elements have been obtained it is straightforward to use them to drive data parallel activity.

In order to facilitate implicit data parallel activity from the automatic decomposition of data structures, it is likely that a syntactic element will need to be introduced. Whilst the introduction of syntax to explicitly support parallel activity is contrary to the goals of reducing such syntax (Section 6.4.1), it is possible to think of such a mechanism without necessarily exposing the fact that it exists to support data parallel activity. Furthermore the introduction of such a syntactic element would allow data structures to be processed serially by "non-parallel" nodes when required. Inspiration for such a syntactic element comes from the Visual Basic language {Deitel, Deitel et al. 1999}, where the `Collection` data structure is commonly used with the `foreach` statement for iterating over the contents of a `Collection`. Since Visual Basic does not (natively) support concurrent activity, there is no notion of executing each of the iterations of a `foreach` loop concurrently - it is instead seen as an optimised syntax for unordered iteration.

A `foreach` node could be added to the Vorlon language whereby one of the inputs contains a data structure which is then decomposed and used to drive parallel instances of the node's program. The Vorlon developer would not think of that node as an explicitly parallel mechanism, but as a kind of non-ordered iterative construct as does the Visual Basic programmer. If such a construct could be supported by the Vorlon language then it could remove all notion of explicit parallel programming. Both task- and data-parallelism would be totally implicit within the structure of a graph, whereas now only task parallelism is completely implicit, and data parallelism only partially implicit.

Whilst adding features into the language to support implicit parallelism is certainly a good idea, it is also a good idea to remove those features from the language which explicitly sequentialise code. This is particularly prevalent in the case of the loop node which has been (mis-) used in many places in the matrix multiplication application, where parallel mechanisms could, or perhaps should, have been used. With the benefit of hindsight it is clear that the provision of a sequential control-flow abstraction in a parallel programming language is the wrong thing to do. Instead implicitly parallel mechanisms (such as the `foreach` mentioned above) should be used to support iterative activity, and ordering of iterations, should the need arise, could be dealt with in the textual language element of a computational node. Thus the purely sequential iteration would be

supported, though the developer would be less tempted to use that feature since it would not be supported at the graph level.

In addition to parallelism-oriented features, future visual parallel programming languages should seek to accommodate new features which modern textual languages support. In particular, the ability to handle events would be valuable such that programs where there is no top-to-bottom execution could be built, and exceptions and exception handling could be implemented. An event handling mechanism would replace the current use of loops and busy waiting for conditions in Vorlon, and exceptions and exception handling would help to increase the robustness of the produced software.

The final aspect of the Vorlon language which it is felt needs amendment is the replicate node. Whilst the behaviour of the replicate node (making copies of handles) is useful in graphs, it is felt that such a semantic could instead be assimilated into the arc – i.e. 1:M arcs. This would help to make Vorlon graphs more readable since there would be fewer nodes per graph. An example of the upgraded arc can be seen in Figure 6-8 alongside the equivalent replicate node pattern.

Figure 6-8 Reducing Visual Complexity by Removing Replicate Nodes

6.5 Closing Remarks

This chapter has presented Vorlon within the taxonomy framework introduced in Chapter 3, and identified areas where the Vorlon language improves on its peers, and areas where it lacks. The costs and significant benefits of developing within an object-oriented framework have been identified - initial implementation costs versus extensibility and re-use – and it has been argued that although Vorlon applications tend to be large, this is simply because extensibility demands that abstractions are added through types, and are not hard-coded into the language syntax.

The syntax and semantics of the language has also been considered in light of experience developing with Vorlon, and suggestions have been made in both tightening

the language syntax, and for developing the parallel processing model for future Vorlon revisions.

Chapter 7 Conclusions and Further Thoughts

This final chapter draws together the work presented by this thesis. An overview of the work is presented which briefly covers the salient features of each of the chapters and thus sets the context for the final evaluation of the programming technology developed. The results obtained from experiments with the developed language technology are recapped and suggestions for aspects of current practise to be upheld in future work are also offered.

7.1 Overview

The fundamental tenet of this thesis is that the parallel computing will become widespread in the near future. Chapter 1 described Moore's Laws and the software stretch which showed how reliant software is on ever-increasing uniprocessor hardware performance, and how those performance increases will eventually be limited by economic factors; at that point parallel computing platforms could be used to compensate for the expected shortfall in computational power. Chapter 1 also pointed out that parallel programming is a difficult discipline, complicated by not only a wide variety of computing platforms, which is set to get wider as parallel computing moves into the commodity computing range, but also by the intricate nature of parallel programs themselves.

Chapter 2 examined four existing visual parallel programming languages whose existence is testimony to the fact that parallel programming is an intricate and time-consuming process. Each of the four surveyed languages uses computer graphics to abstract some of the detail of the underlying parallel computing system in an attempt to make those systems easier to exploit. Examples of developing parallel programs with those languages were given, and discussion on the suitability of each language was presented.

The work presented in Chapter 3 built on Chapter 2, and a taxonomy of the surveyed languages was presented and their strengths and weaknesses analysed. The results from the taxonomy process, in conjunction with experience from developing examples with each of the four languages, lead to the belief that the current generation of visual parallel programming languages provide little more than current textual parallel programming languages, in that they support *parallel programming*, and not *parallel application development*. To address this problem a new paradigm for parallel application development, *Parallel Object-Flow*, was developed to deal with both problem domain and architectural

complexity. Importantly, object-orientation was proposed as a means of supporting the software lifecycle, whilst visual programming languages were proposed as a means of abstracting the detail of an idealised parallel computer.

Chapter 4 introduced the main work developed for this thesis, the Vorlon visual parallel programming language which implemented the Parallel Object-Flow paradigm. The syntax and semantics of the language were introduced, alongside the development methodology and software architecture of a Vorlon application.

The next stage in the development of this thesis was the collection and analysis of results. Chapter 5 presented the construction of two applications (one numeric, one non-numeric) developed from conception through to release with Vorlon. The results obtained from these applications were in the form of practical experience developing with Vorlon, and timing data from executing the applications. The performance results obtained were mixed and revealed that, in the first version of the Vorlon language, applications which exhibit a pattern of fine granularity of locking of objects (such as numerical applications using matrices) did not perform well, whilst other types of application (in this case a compilation system) performed satisfactorily.

The penultimate chapter of this thesis was a period of reflection on the work undertaken. Chapter 6 took the work from Chapters 4 and 5 and subjected Vorlon to the same taxonomy process as was presented in Chapter 3 for previous visual parallel programming languages. From this taxonomy, reflections on both linguistic and software engineering aspects of the language were presented, and the reason why Vorlon applications tend to be larger than their equivalents was explained as a necessary symptom of an extensible language that encourages structured re-use.

7.2 Reflections on Software Engineering Aspects

The fact that Vorlon is an object-oriented language provides significant software engineering benefits. Since object-orientation supports the whole of the software lifecycle, so too can Vorlon applications enjoy such support. Problems can be analysed, solutions designed, and applications implemented within a common framework – in short because of object-orientation, applications can be *engineered*.

However, the engineering benefits come at a cost. In this case the cost is the size of the applications compared to the equivalent written in previous visual parallel programming languages. The overwhelming majority of this additional cost of

developing with Vorlon is invested in building types to match the problem domain requirements. Since previous visual parallel programming languages were not object-oriented, the development of such types was not an issue since all the available abstractions came built into the language syntax.

However, it was noted that there are significant benefits associated with the ability to extend a language through types. Importantly, this extensibility allows the language to adapt to new situations which the language designer may not have foreseen, which had the abstractions been built directly into the language would not be possible. Furthermore, the re-use of types (as opposed to the functional re-use pattern seen in previous languages) means that re-use can occur much earlier in the lifecycle (at the analysis and design phases) which will have the effect of shortening development times for subsequent software projects.

7.3 Vorlon Language Issues

The main problem with Vorlon was the fact that parallelism is explicit in some of the syntactic elements whereas the goal of the language was to encapsulate parallel activity away from the developer. In particular, the explicit data-parallel aspects of the parallel computation node and parallel method call node were cited as mechanisms which should be removed from the language in favour of implicitly parallel mechanisms. Improvements to the current language which would enable a purely implicitly parallel style of programming were suggested, but it was noted that such approaches would not be extensible. Instead, it was suggested that a more fundamental re-appraisal of the data-parallel mechanisms was required and that generic, extensible, type-driven mechanisms should be employed to implicitly control data-parallel activity.

The prototypical nature of the language meant that the overall look-and-feel of the language was basic, and that some of the syntactic elements (and conditional execution nodes in particular) were poorly designed. Thus, although the overall style of execution was found to be acceptable, the construction of applications with Vorlon was harder than it should have been, such problems could be rectified given time since they are not fundamental.

7.4 Performance Results

The performance results presented in Chapter 5 showed a clear distinction in performance between the matrix multiplication and compilation system applications. The

matrix multiplication program performed badly on all test platforms, including shared memory multiprocessor, whereas the compilation system performed reasonably well on all platforms and particularly well on the shared memory system.

For the matrix multiplication application, it was found that the costs of memory access (predominantly locking of fine-grained objects) as the cause of the poor performance. It follows that if locking can be minimised, Vorlon programs may perform significantly better, and it was suggested that adopting message-passing under certain circumstances may help to avoid locking overheads. The problem of efficiency would then become one of grain size – communication costs version computation time – and mechanisms such as NIP's lazy task creation and load balancing would help under such circumstances.

For the compilation system, the problem was not with Vorlon, nor the NIP run-time system. Whilst both Vorlon and NIP concentrate on memory and processor abstractions, neither pays any attention to I/O device abstractions which would work efficiently in a parallel environment. Since such mechanisms are outside of the research area for this thesis, no solution was developed. As such it would be fair to say that applications which are heavily I/O dependent do not scale well using the techniques outlined in this thesis, in the absence of appropriate library support such as that discussed in Chapter 5.

7.5 Future Work

When offering suggestions for future work, it is often the case that less well received aspects of the current work are presented and purified as if that action would resolve all outstanding issues. It is not the intention here to follow suit, since aspects of the current work which could be improved upon have already been discussed. Instead, more generic aspects of parallel application development are considered and, where germane, opinions based on experience with the Vorlon language are offered.

It is wholeheartedly felt that a visual approach to parallel application development is an eminently practical and sensible approach to take. During the period of research for this thesis a range of visual and textual parallel programming languages/libraries were investigated. Even those textual approaches which were considered to be at the cutting edge of their field (in particular the NIP run-time system which Vorlon utilised) were found to be significantly more difficult to develop with than a visual language. As for scalability of visual languages (a problem which is often quoted as being the downfall of visual programming), experience demonstrates that textual parallel programming scales

little better since the source code can be incredibly intricate. In short, the visual approach to parallel application development is good and should be pursued.

Saying that a visual approach is desirable is, however, an incomplete argument. In Chapter 2, several visual parallel programming systems were surveyed and each exhibited it own semantics and style – and there are still other visual programming systems which were not considered in the survey which have still different styles. However, in the parallel programming arena where issues of synchronisation are of paramount importance, the style of programming language can have a serious impact on the overall performance of the developer. Specifically, it was felt that the notion of flows of items (be those items data or object handles in the case of Vorlon) was valuable since it provided implicit communication and synchronisation, whilst giving the developer a clear picture of dependencies. It was thus felt that flows are a useful syntactic element which should be preserved in future languages.

Following on from the suggestion that flows are a useful means of implicitly describing communication and synchronisation patterns, and therefore of implicitly describing task- and (where supported) pipeline-parallelism, there is the more general implication that mechanisms which implicitly support parallel activity are good. Languages which support the implicit exploitation of parallelism from an application's structure remove the "parallelism burden" from the developer and thus free the developer to concentrate on more important issues like ensuring the software meets its specification.

A complementary argument to the fact that languages should support implicit exploitation of parallel activity is the fact that they should not permit explicit mechanisms for introducing parallel activity. Whilst this may seem an obvious, if not superfluous, point to make, it is not. Previous work, notably ParADE, and this work both acknowledged the fact that a suitably intelligent developer may want to be able to specify parallelism at will, rather than allowing the underlying development environment to control such issues. Both ParADE and Vorlon were languages which attempted to support the novice parallel programmer to write parallel programs, whilst permitting the expert to optimise. Only at this point in the work it is possible to acknowledge that such a philosophy is ultimately wrong. An expert developer should not be expected to shoulder the parallelism burden any more than a novice. Instead an expert developer should be expected to better apply proper software engineering principles to the problem

than a novice, and if the programming language is suitably advanced (such as Vorlon with the suggested type-driven modifications), then the application of those principles will implicitly invoke more parallel activity. That is, explicit parallelism, even in the hands of an expert developer, may not prove to be a particularly good idea, and future visual parallel programming languages should not support it.

Given that typing is thought to be key in the future success of visual parallel programming, the final and most important suggestion for future work is that it continue to support object-orientation. Not only will object-orientation provide the primary means of exploiting parallelism from an application, but it will continue to provide full lifecycle support for the development process which will be fundamental in the shift from parallel programming to parallel application development.

7.6 Closing Remarks

The main contribution of this thesis has been to demonstrate that parallel applications can be engineered, and that paradigms which support both performance and software engineering aspects of application development can be developed. Vorlon has made a positive contribution to the field of visual parallel programming. Certainly Vorlon does not solve all the difficulties of parallel programming – it is not a universal panacea – but it is a step in the right direction.

Appendix A Development of a Parallel Compilation System

The most salient graphs from the parallel compilation system were presented in Chapter 5, where those graphs provided the context within which performance results could be investigated. For completeness, the whole development of the application (excluding performance analysis) is presented in this appendix.

A.1 Introduction

The UNIX make tool is a utility for managing software builds. It is particularly useful for projects where an application may be dependent on a number of source files, and where not all source files are changed between successive application builds. In such cases make only re-compiles object code for those source files that have changed. At present, most implementations of make use an iterative approach to compilation, in that the date of a file is checked, and if found to be out-of-date it is compiled, before the next file is checked, and so on.

However, given that the only dependency in a build operation is between the production of object code and the link phase of the build, and that the production of the object code from each source file is an independent task, there is the potential to parallelise the build process. This appendix discusses the construction of a simple parallel implementation of a make-like tool which uses a data-parallel, rather than an iterative approach in the compilation of source code files.

A.2 High Level Analysis

As with any software project, the first development stage is to perform an analysis of the problem domain. In an object-oriented analysis, the goal is to look for objects in the problem domain, and reason about those objects in order to extract a set of types. In the parallel compilation system problem domain, there are three types which are immediately obvious:

The intermediate language (C++) compiler;

The project which describes the relation between Vorlon graphs in an application;

The (intermediate language) source files, derived from the Vorlon graphs, which are built to produce the final executable application.

As the first step in the analysis of the parallel compilation system, it is helpful to place those types on a class diagram, and add any obvious operations (methods) and attributes (data members) to their interfaces. This can be seen in Figure A-1.

Figure A-1 Initial Class Diagram for the Parallel Compilation System

The class diagram shown in Figure A-1 depicts the three main types for this problem domain, along with their corresponding operations and attributes. At this point in the development, what is understood about the problem is as follows:

1. There exist a number of named `SourceFile`s, and each has an associated age;
2. The `Compiler` compiles source files, and may be parameterised via compiler flags to link code with appropriate libraries as specified by the user of the application;
3. The overall activity of the system is coordinated by a `Project` which is responsible for determining out-of-date source files, and compiling those files in order to build the resultant application.

Having established the functionality of each of the primary types in the problem domain, the development process can progress to the high-level design stage.

A.3 Application Design

The real beauty of the object-oriented methodology is its seamlessness. The work undertaken in the analysis phase is carried forward directly into the design phase of the development cycle. The primary distinction between the analysis and design phases is the fact that the design phase begins the process of "concreting" the components from the analysis phase such that they can be used as a basis in the construction of the final software artefact.

In the design presented in Figure A-2, the types needed to support those introduced in Figure A-1, are shown and the relations between those types have been added.

Figure A-2 Initial Design-Stage Class Diagram

The class diagram in Figure A-2 presents a full static view of the types arising from the problem domain and the types which the problem domain types require in order to fulfil their own functionality. This diagram presents a number of important relations, including:

1. The `Project` type is dependent on the `String` type, which itself contains references to a number of instances of the `Object<char>` type.
2. A Project object contains a list of source files (`List<SourceFile>`).
3. The `List<SourceNode>` type contains an instance of the `ListNode<SourceFile>` type.
4. All instances of the `ListNode<SourceFile>` type contain a reference to another instance of `ListNode<SourceFile>` which is used to link `ListNode<SourceFile>` objects into a `List`.
5. The `List<SourceFile>` class is aware of the interface of the `SourceFile` class, as is the `Compiler` class.
6. The `SourceFile` class depends on the `String` class to fulfil its responsibilities, and is aware of the `String` type's interface.

Having introduced the overall structure of the types in the application, it is worthwhile discussing some of the responsibilities and behaviours of those types.

- The `Project` type maintains data on the build state of the source files that comprise the application. Most of this data is kept in the form of text string representing filenames and other compiler options, and numerical values which represent the last build time of the application, along with some other housekeeping data. Apart from the get /set methods, the only other method supported by the type is the `getOutOfDateFiles(…)` method which returns a list of out-of-date source files to the caller.
- The `List<SourceFile>` and `ListNode<SourceFile>` are used to simply build lists of `SourceFile` objects. They support a standard compliment of list methods and attributes to facilitate this functionality.

- The `SourceFile` type is a very simple representation of a physical file, containing a filename and age as attributes and a pair of get/set methods for those attributes.
- The `Compiler` type is a representation of an external C++ compiler. It supports only one method called `compile(…)` which is used to invoke the external compiler on the specified source file. Its attributes are set on construction on include the compiler flags and the fully qualified path to the external compiler.
- `String` is simply a NIP-safe array of `Object<char>` objects. It supports two concatenation methods (one for `char` objects and one for `String` objects), and one method called `c_str()` (convention adopted from the C++ STL) which is used to produce strings in a form suitable for passing to the environment via the C `system(…)` call, so that the external compiler can be invoked.

The final stage in the development of the application is to refine the full static view of all programmatic components, including the starting point for the application and its relations with the other types. This can be seen in Figure A-3.

Figure A-3 Final Application Design

The class diagram shown in Figure A-3 shows the `Main` class and the `main(…)` method in conjunction with the other types. The `Main` class contains one instance of the `Project` class, and one instance of the `Compiler` class, and is dependent on the `List<SourceFile>` type to fulfil its responsibilities. The `main(…)`method interrogates the Project object for a `List<SourceFile>` object which it then uses to call the `compile(…)` method from the Compiler object that it contains. Note that is parallel calls on the `compile(…)` method that is the main source of parallelism in this application.

Having introduced the responsibilities of, and the operations supported by each of the types that constitute the application, it is now possible to implement those types, which is the subject of the following section.

A.4 Implementation

Once the static structure of the application has been determined, the next stage in the development of the software is to build the functionality of the application by implementing the types from the design phase. In the case of the compilation application, it makes most sense to tackle the problem top-down. That is, implement the most important classes first, and the more minor classes later. As such, the `Main` type's `main(…)` method is the first to be implemented, followed by the `Project` class, the `Compiler` class, `SourceFile` class, the `List` class (and associated `ListNode` class), and finally the `String` class.

A.4.1 The Main Class

The `Main` class is composed from only a single method called `main(…)` the implementation of which can be seen in Figure A-4.

Figure A-4 The main(…) Method

The `main(…)` method depicted in Figure A-4 is responsible for the majority of the work undertaken by the application. It first accepts an instance of the `String` type as a filename of a file that contains the project data. This is then used to create an instance of the `Project` type.

The instance of the `Project` type is then used to service three method calls in parallel. Two of the method calls are simple get methods, which retrieve the compiler name and the compiler flags to be activated, which are then used to instantiate an instance of the `Compiler` type. The third of the method calls requests a list of out-of-date source files, and causes the `Project` object to search its internal data structures

looking for files which are more recent than the target executable. It is the resultant list of objects that this method call releases which drives the majority of the subsequent parallel activity within the graph, when the `List` is sent as a parameter to a parallel method call node. The parallel method call node invokes the `Compiler` type's `compile(…)` method on each of the `SourceFile` objects contained within the input `List` of `SourceFile`s. Having compiled each of the source files into object code, the final stage of the build process is to perform a link to produce the final executable. This functionality is supported by the `link(…)` method in the `Compiler` class.

A.4.2 The Project Class

The `Project` class contains a total of four methods and one constructor. Three of those methods are trivial get methods which return particular project attributes such as the compiler name to the caller. The implementation for get methods that the `Project` class supports can be seen in Figure A-5,Figure A-6, and Figure A-7.

Figure A-5 getCompilerName(…) Method

Figure A-6 getCCFlags(…) Method

Figure A-7 getTarget(…) Method

In addition to the get methods, the other method that the `Project` type supports is `getOutOfDateFiles(…)` whose implementation can be seen in Figure A-8 and Figure A-9 below.

Figure A-8 getOutOfDateFiles(...) Method

Figure A-9 getOutOfDateFiles(...) Method Sub-Graph

The `getOutOfDateFiles(…)` method has already been discussed in depth in the main text, and it is not the intention to repeat that discussion here, save for the fact that the parallel computation node is used to build the list of out-of-date files, but it is not used in its canonical form since not all inputs are to be decomposed. This raised several syntactic issues about decomposition of structures and implicit data parallel activity which are themselves tackled in Chapter 6.

The final operation supported by the Project type is its constructor (the provision of a copy constructor can be left to the translator in this case). The implementation of the default constructor is shown in Figure A-10.

Figure A-10 The Default Constructor of the Project Type

The graph shown in Figure A-10 is straightforward with only an assignment and a piece of sequential C++ code running concurrently. The sequential C++ code merely reads in a number of parameters from the file and assigns them to appropriate places within the `Project` object's internal data structures. The sequential C++ was used in preference to a Vorlon graph because C++ has far better facilities for handling I/O than Vorlon does (since Vorlon does not have an I/O library), and since file I/O is sequential in this case there would be little added performance benefit by developing the routines in Vorlon.

A.4.3 The Compiler Class

Figure A-11 The compile(…) method of the Compiler Class

The `Compiler` type's `compile(…)` method has already been discussed in the main text. It suffices to say here that it merely forms a command-line argument which represents the compiler options and the source file to be compiled, and passes that argument to the command-line interpreter (via a C standard library `system(…)` call). For a fuller description of the activity of the graph, refer to Chapter 5.

Figure A-12 The link(…) method of the Compiler Class

The `link(…)` method of the `Compiler` type simply issues a call to the command-line interpreter to link all source files together to build the target executable. In this case, the command-line argument passed is simply the name of the compiler with the link switch set and passing the name of all object code files, an example of which can be seen here:

```
g++ -o *.o
```

Although more complex behaviour might be expected from an industrial-strength software build system, for purposes of experimentation all object files in the build directory (and only that directory) are used in the link process.

A.4.4 The SourceFile Class

Figure A-13 SourceFile Class Constructor

Figure A-14 SourceFile Class Copy Constructor

The standard and copy constructors for the `SourceFile` class can be seen in Figure A-13 and Figure A-14 respectively. In both cases the majority of the work performed is the assignment of attributes passed as arguments to the constructor call to local attributes within the new object. In the default constructor, the `_filename` attribute is set from the parameter passed to the constructor call and the `_age` attribute is set by a call to the operating system to retrieve the age of the file. In the copy constructor a similar process occurs except that the attributes are retrieved from a method call on the parameter object, rather than directly from the parameter or from the operating system.

Figure A-15 SourceFile Class getName(...) Method

Figure A-16 SourceFile Class getAge(...) Method

Figure A-15 and Figure A-16 show the two get methods supported by the `SourceFile` class. Both get methods simply return a handle to one of the attributes of the class.

A.4.5 The List and ListNode Classes

The `List` class acts as a repository of `SourceFile` objects which represent the source code files of the application which is being built. The `ListNode` class is used to hold the actual `SourceFile` objects, and to link the `List` together in a classical linked list structure.

Figure A-17 List Class Default Constructor

The constructor for the `List` class is shown in Figure A-17. The functionality of the default constructor is straightforward in that it simply initialises the attributes of the `List` object to their starting values. In this case the reference to the first item (a `ListNode` object) in the `List` is set to `null` (since the `List` is empty on creation), and the length of the `List` is set to zero.

Figure A-18 List Class addNode(…) Method

The `addNode(…)` method of the `List` class is the means by which items are added to the tail of a `List`. In this method, two distinct activities take place in parallel. In the first of these activities, the current last node in the `List` (the node at position `_totalNodes–1` in the `List`) is retrieved through a call to the `nodeAt(…)` method, and that node's `_next` reference (the reference to the next node in the `List`) is accessed via a method call node. At the same time, the `SourceFile` object passed as a parameter to the method call is used to create a new `ListNode` object. Once the `ListNode` object has been created, the `_next` reference obtained from the current last node in the `List` is updated to refer to the new `ListNode` object, and the number of nodes in the `List` is incremented.

Figure A-19 List Class nodeAt(…) Method

The `nodeAt(…)` method is used to retrieve data from a `List`. The implementation of the top-level graph of the method is shown in Figure A-19. There is no parallelism in this method since it simply involves iterating over a number of `ListNode` objects until a counter value is met, at which point the attribute `_data` (a reference to a `SourceFile` object) of that node is returned to the caller of the method. The decomposed loop node sub-graph of the `nodeAt(…)` method can be seen in Figure A-20.

Figure A-20 List Class nodeAt(…) Method Sub-Graph

The sub-graph shown in Figure A-20 is used to access subsequent `ListNode` objects from a `List`. It simply assigns the `_next` attribute of the current `ListNode` object to an object handle called `current` which is maintained between successive executions of the graph. Once all executions of the graph have completed, the `current` handle references the node that the caller of the method requires.

Figure A-21 List Class noOfNodes() Method

The `noOfNodes(…)` method is a simple get method which returns a handle to the attribute `_noOfNodes` from the current object. Its implementation is shown in Figure A-21.

Figure A-22 ListNode Default Constructor

Figure A-23 ListNode Copy Constructor

Figure A-24 ListNode Constructor

The `ListNode` class is used to store objects within a `List`. It is an extremely basic data type, consisting only of constructors and public attributes. The constructors of the class can be seen in Figure A-22, Figure A-23, and Figure A-24 above. The default constructor simply sets both of the attributes of the new object to `null`. The copy constructor sets the `_data` attribute to be identical to that of the `ListNode` object passed in as its parameter, and sets its `_next` attribute to be `null` (since the `_next` attribute is set by a `List` object as part of an insertion operation). The final `ListNode` constructor takes a handle to a `SourceFile` object, and a handle to another `ListNode` object as its parameters, and simply uses them to initialise its attribute set.

A.4.6 The String Class

The `String` class is an abstraction of a sequence of characters, supporting common options such as concatenation. In this application, `String` objects are used to hold filenames, compiler switches, and other project data.

Figure A-25 String Class Default Constructor

The default constructor for the `String` class can be seen in Figure A-25. It creates a zero-length array of `Object<char>` objects which is then assigned to the `_data` attribute of the new object, and sets the `_length` attribute of the String object to zero.

Figure A-26 String Class Copy Constructor

The copy constructor of the `String` class is shown in Figure A-26. It sets the `_length` attribute of the new object to be identical to that of the parameter object, and in parallel makes a copy of the parameter object's internal array structure and assigns that copy to the local `_data` attribute.

Figure A-27 String Class Constructor

The final constructor for the String class, shown in Figure A-27, takes a single char as its input parameter, and uses that input to create an instance of `Object<char>`. In parallel with the creation of the `Object<char>` instance, an `Object<char>` array of length one is created. Once the array and the single instance have been instantiated, the single instance of `Object<char>` is inserted into the array at location `0` and the `_length` attribute of the new `String` object is set to `1`.

Figure A-28 String Class concat(String) Method

Although the `concat(…)` method shown in Figure A-28 is a highly-connected graph, its functionality is actually straightforward. The `_data` and `_length` attributes are extracted from the parameter `String` object, and the `_length` attribute is then added to the `_length` of the current object and the value yielded is used to create a new array of type `Object<char>`, big enough to hold the contents of both `String` object's data. Once created, the new array is operated upon in parallel by two loop nodes which iterate over the current object's `_data` attribute, and that of the parameter object, copying element each into an appropriate position in the previously instantiated (larger) array structure. Once the copying process has finished, the new, large array of `Object<char>` objects is assigned to the `_data` attribute of the current object, leaving the old structure to be garbage collected. The final action performed is to update the `_length` attribute of the current object to reflect the increased size of the `String` after the concatenation.

Figure A-29 String Class concat(char) Method

The other concatenation method supported by the `String` class is the that of a single character to an existing `String`. The implementation for this method can be seen in Figure A-29. The character concatenation method is not dissimilar to the `String` concatenation method with the exception that one of the iterated copy

operations present in the `String` concatenation is replaced by a single array insertion for the `char`-oriented version. In the method graph of Figure A-29, a new array which is one element larger than the current array referenced by the `_data` attribute of the object is created, and the contents of the current internal state are copied into the new array. Once the copy has finished, the instance of `Object<char>` created from the parameter passed to the method is inserted at the final position in the new array, and it is assigned to the current `_data` attribute, leaving the old array to be garbage collected.

Figure A-30 String Class length(…) Method

The `length(…)` method of the String class is a simple get method which returns a handle to the `_length` attribute of the object that it is invoked on. Its implementation can be seen in Figure A-30 above.

Figure A-31 String Class c_str() Method

The `c_str()` method of the `String` class, shown in Figure A-31, is used to extract a C-style null-terminated array of characters from a `String`, to be used where elements of the C or C++ standard libraries requiring such structures as parameters are to be invoked. The functionality of the method is straightforward (if a little crude), in that a computation node is used to create a C-style array (since the Vorlon new object nodes create Vorlon, not C, arrays) which is then passed onto a loop node which iterates over that array and the array referenced by the `_data` attribute of the current object performing a copy operation on each element. Once the copy operation has finished, the C-style char array is returned to the caller of the method. It should be noted that this method is "unsafe" insofar as it is not guaranteed to work in a distributed memory environment if subsequent operations do not occur on the same processing node. It should not be called if there is any parallelism below the call to this method further down in the calling graph. Furthermore since C-style memory mechanisms are used, the objects created are not managed by the run-time support and will not benefit from garbage

collection, transparent distribution, caching, and so forth[22]. However, since it has been acknowledged that external library support is required until Vorlon library support is available, such mechanisms are unavoidable, and the developer must simply be cautious in their use.

A.5 Summary

This appendix has shown the development of the parallel compilation system for building Vorlon applications. The project was developed from its inception through to implementation with each step in the process discussed. It has not shown performance figures for the application, nor drawn any claims as to the suitability of the Vorlon language for developing such applications, since those topics were covered in the main text (Chapter 5 and Chapter 6). However, it has shown the complete construction of a fully object-oriented application, including the implementation of a classical data structure with the Vorlon language, which adds some credibility to the approach.

[22] Such notions of "managed" and "unsafe" code are also supported by textual languages such as C# {Wille 2000}, and similar warnings about usage abound.

References

Agha, G. and C. Hewitt (1987). Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. Research Directions in Object-Oriented Programming. B. Shriver and P. Wegner (editors), MIT Press**:** 49-74.

Allen, R. S. (1998). A Graphical System for Parallel Software Development. Ph.D. Thesis, Department of Computing Science, The University of Newcastle upon Tyne.

Amdahl, G. (1967). The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS.

Annot, J. K. and P. A. M. d. Haan (1990). POOL and DOOM: The Object-Oriented Approach. Appearing in: Parallel Computers: Object-Oriented, Functional, Logic. P. C. Treleaven (editor), Wiley**:** 49-79.

Babaoglu, O., L. Alvisi, et al. (1992). Paralex: An Environment for Parallel Programming in Distributed Systems. 6th ACM International Conference on Supercomputing, ACM Press.

Barnes, J. and P. Hut (1986). A Hierarchical O(N log N) Force Calculation Algorithm. Nature 324: 446.

Beguelin, A., J. Dongarra, et al. (1994). HeNCE: A User's Guide. http://www.netlib.org/hence/hence-2.0-doc-html/hence-2.0-doc.html

Bik, A. J. C. and D. B. Gannon (1997). Automatically Exploiting Implicit Parallelism in Java. Concurrency: Practice and Experience 9(6): 579-619.

Booch, G. (1994). Object-Oriented Analysis and Design, Benjamin/Cummings.

Booch, G. (1996). Object Solutions: Managing the Object-Oriented Project, Addison-Wesley.

Booch, G., J. Rumbaugh, et al. (1999). The Unified Modelling Language User Guide. Addison-Wesley.

Browne, J. C., S. I. Hyder, et al. (1995). Visual Programming and Debugging for Parallel Computing. IEEE Parallel and Distributed Technology (Spring 1995).

Bruegge, B., J. Blyth, et al. (1992). Object-Oriented Modeling with OMT. OOPSLA '92, ACM Press.

Burnett, M. M., A. Goldberg, et al (editors). (1995). Visual Object-Oriented Programming, Manning.

Carriero, N. and D. Gelernter (1989). How to Write Parallel Programs: A Guide for the Perplexed. ACM Computing Surveys 21(3): 322-357.

Citrin, W., M. Doherty, et al. (1995). The Design of a Completely Visual OOP Language. Appearing in: Visual Object-Oriented Programming: Concepts and Environments. M. M. Burnett, A. Goldberg and T. G. Lewis, Prentice-Hall**:** 67-93.

Coad, P. and J. Nicola (1993). Object-Oriented Programming, Yourdon Press.

Coad, P. and E. Yourdon (1991). Object-Oriented Analysis, Yourdon Press.

Coad, P. and E. Yourdon (1991). Object-Oriented Design, Yourdon Press.

Cox, P. T., F. R. Giles, et al. (1995). Prograph. Appearing in: Visual Object-Oriented Programming: Concepts and Environments. M. M. Burnett, A. Goldberg and T. G. Lewis, Prentice Hall**:** 45-66.

Deitel, H. M., P. J. Deitel, et al. (1999). Visual Basic 6: How To Program. Prentice Hall.

Dozsa, G., P. Kacsuk, et al. (1997). GRADE: A Graphical Programming Environment for PVM Applications. 5th Euromicor Workshop on Parallel and Distributed Processing, London.

Flanagan, D. (1997). Java Examples in a Nutshell, O'Reilly & Associates.

Flanagan, D. (1997). Java in a Nutshell, O'Reilly & Associates.

Fowler, M. and K. Scott (1997). UML Distilled: Applying the Standard Object Modelling Language, Addison-Wesley.

Geist, A., A. Beguelin, et al. (1994). PVM 3 User's Guide and Reference Manual, Oak Ridge National Laboratory. http://www.epm.ornl.gov/pvm/pvm_book.html

Geist, A., W. Gropp, et al. (1996). MPI-2: Extending the Message-Passing Interface. Europar '96, Springer-Verlag.

Gelernter, D. (1985). Generative Communication in Linda. ACM Transactions on Programming Languages and Systems 7: 80-112.

Gurd, J. R., C. C. Kirkham, et al. (1985). The Manchester Prototype Dataflow Computer. Communications of the ACM 28(1): 34-52.

Harley, J. W. (1993). Dataflow Development of Medium Grained Parallel Software. Ph.D. Thesis, Department of Computing Science, The University of Newcastle upon Tyne.

Hoare, C. A. R. (1985). Communicating Sequential Processes, Prentice-Hall International.

Jacobson, I., M. Christerson, et al. (1992). Object-Oriented Software Engineering: A Use Case Driven Approach, ACM Press / Addison-Wesley.

Kacksuk, P. and S. Forrai (1999). GRADE - Graphical Environment for Parallel Programming. ERCIM News(36): 30-33.

Kacsuk, P., J. C. Cunha, et al. (1997). A Graphical Development and Debugging Environment for Parallel Programs. Parallel Computing Journal 22(13): 1747-1770.

Kacsuk, P., T. Dozsa, et al. (1997). A Graphical Environment for Message-Passing Programs. 2nd International Workshop for Parallel and Distributed Systems, Boston.

Kale, L. V. and S. Krishnan (1993). Charm++: A Portable Concurrent Object-Oriented System Based on C++. OOPSLA '93, ACM.

Keleher, P., A. L. Cox, et al. (1994). TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. 1994 Winter USENIX Conference, San Francisco, CA, USA.

Kennedy, K., C. F. Bender, et al. (1997). A Nationwide Parallel Computing Environment. Communications of the ACM 40(11): 63-72.

Koelbel, C. H., D. B. Loveman, et al. (1994). The High Performance Fortran Handbook, MIT Press.

Kramer-Fuhrmann, O. and T. Brandes (1991). Gracia - A Software Environment for Graphical Specification, Automatic Configuration and Animation of Parallel Programs. ACM International Conference on Supercomputing, Cologne, Germant.

Lavender, R. G. and D. C. Schmidt (1996). Active Object: An Object Behavioural Pattern for Concurrent Programming. Appearing in: Pattern Languages of Program Design 2. J. Vlissides, J. Coplien and N. Kerth, Addison-Wesley.

Layzell, P. and P. Loucopoulos (1989). Systems Analysis and Developement, Chartwell-Bratt.

Loques, O., J. Leite, et al. (1998). P-RIO: A Modular Parallel-Programming Environment. IEEE Concurrency (Spring 1998): 47-56.

Microsoft (1998). Visual J++ 6.0 Programmer's Guide, Microsoft Press.

Mohr, E., D. A. Kranz, et al. (1991). Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. IEEE Transactions on Parallel and Distributed Systems 2(3): 264-280.

Moore, G. E. (1965). Cramming More Components Onto Integrated Circuits. Electronics Magazine. 38**:** 114-117.

Moore, G. E. (1997). An Update on Moore's Law. San Francisco, Intel Developer Forum Keynote Speech.

Moore, G. E. (1998). What is Moore's Law?, Intel Museum Online. http://www.intel.com/intel/museum/25anniv/Hof/moore.htm

Morrison, M. (1997). Java 1.1 Unleashed, Sams.

Newton, P. and J. Dongarra (1994). Overview of VPE: A Visual Environment for Message-Passing Parallel Programming. The University of Tennesse, Knoxville.

Newton, P. W. (1993). A Graphical Retargetable Parallel Programming Environment and its Efficient Implementation. Ph.D. Thesis, Department of Computer Sciences. The University of Texas at Austin.

Oram, A. and S. Talbott (1993). Manging Projects with Make. Sebastopol, CA, O'Reilly.

Protic, J., M. Tomasevic, et al. (1998). Distibuted Shared Memory: Concepts and Systems, IEEE Computer Society.

Sargeant, J. (1993). United Functions and Objects: An Overview. Manchester, University of Manchester, Department of Computing Science.

Schaller, R. R. (1997). Moore's Law: Past, Present, Future. IEEE Spectrum 34(6): 52-59.

Sedgewick, R. (1992). Algorithms in C++, Addison Wesley.

Snir, M., S. Otto, et al. (1996). MPI: The Complete Reference, MIT Press.

Sterling, T., D. Becker, et al. (1998). An Assessment of Beowulf-class Computing for NASA Requirements. IEEE Aerospace.

Sterling, T. L., J. Salmon, et al. (1999). How to Build a Beowulf. Cambridge, MIT Press.

Stroustrup, B. (1991). The C++ Programming Language (Second Edition), Addison-Wesley.

Stroustrup, B. (1997). The C++ Programming Language (Third Edition), Addison-Wesley.

Treleaven, P. C. (1990). Parallel Computers: Object-Oriented, Functional, Logic. London, Wiley.

Treleaven, P. C., D. R. Brownbridge, et al. (1982). Data-Driven and Demand-Driven Computer Architecture. ACM Computing Surveys 14(1): 93-143.

Veen, A. H. (1986). Dataflow Machine Architectures. ACM Computing Surveys 18(4): 366-396.

Watson, P. and S. Parastatidis (1999). The NIP Parallel Object-Oriented Computational Model. Network-based Parallel Computing: Communication, Architecture, and Applications. Orlando, Florida, USA, Springer-Verlag.

Watson, P. and S. Parastatidis (1999). An Optimised Lazy Task Creation Technique for Iterative and Recursive Computations. International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA.

Webber, J. (1998). Vorlon: A Visual Object-Oriented Approach to Parallel Application Development. Doctoral Symposium, Automated Software Engineering, Honolulu, Hawaii, IEEE.

Webber, J. (2000). Visual, Object-Oriented Development of Parallel Applications. IEEE VL 2000 Workshop on Visual Methods for Parallel and Distributed Systems, Seattle, IEEE Press.

Whitley, K. N. (1997). Visual Programming Languages and the Empirical Evidence for and Against. Journal of Visual Languages and Computing(8): 109-142.

Wille, C. (2000). Presenting C#, Sams.

Winbald, A., L., S. Edwards, D., et al. (1990). Object-Oriented Software, Addison-Wesley.

Wolf, K. and O. Kramer-Fuhrman (1996). An Integrated Environment to Design Parallel Object-Oriented Applications. Europar '96, Springer-Verlag.

Yonezawa, A. and M. Tokoro (1987). Object-Oriented Concurrent Programming, MIT Press.