

Process-Oriented Language Design for Distributed Address Spaces

PETER RICHARD BAILEY

A thesis submitted for the degree of
Doctor of Philosophy of
The Australian National University

January 1997

© Peter Bailey 1997

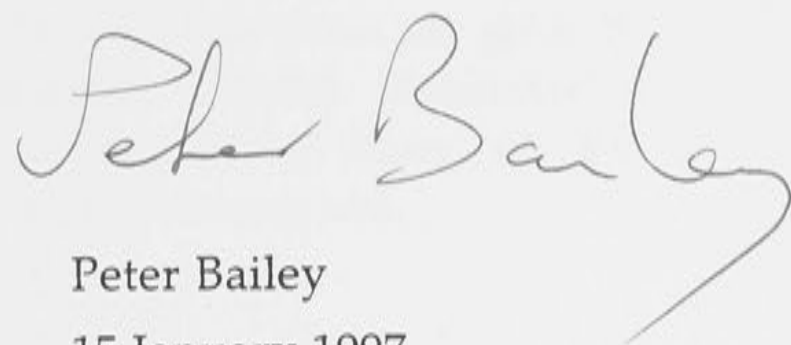
Earlier versions of some parts of this thesis have appeared in:

*Proceedings of the
Sixteenth Australasian Computer Science Conference*
© Australian Computer Science Association 1993

Proceedings of CONPAR 94 - VAPP VI
© Springer-Verlag 1994

*Proceedings of CATS'96:
Computing, the Australian Theory Symposium*
© Australian Computer Science Association 1996

Except where otherwise indicated, this thesis is my own original work.

A handwritten signature in cursive script that reads "Peter Bailey". The signature is written in dark ink and is positioned above the printed name and date.

Peter Bailey

15 January 1997

12-13-14

12-13-14

12-13-14

12-13-14

12-13-14

12-13-14

12-13-14

Abstract

Commercial and scientific applications exist with computational and communications requirements that far exceed the most powerful computers available today. Short of radical change in technologies, it is generally recognised that the way ahead to satisfy the demand for increased computational resources is restricted to parallel computer systems. Many such parallel computers present a distributed address space programming environment which assists in achieving scalable performance.

Finding safe and efficient language mechanisms for programming in the context of a distributed address space is a significant challenge for computer science. This thesis addresses several issues that arise in meeting the challenge through development of the process-oriented programming language paraML. The design choices for the language are motivated and realised as extensions to ML.

The paraML language is modelled theoretically by an untyped lambda calculus called λ_{pv} . The key results of the theoretical modelling are the formal specification of computation with reference variables which have been copied from one process to another. A proof of type soundness for the polymorphic type system is given which thus guarantees that any typable program will not produce a type-error at runtime.

A working version of paraML has been implemented as a proof-of-concept and used for applications. The performance metrics of the implementation are given to illustrate the scalable characteristics of the implementation. ParaML demonstrates that process-oriented mechanisms are valuable in simplifying many of the complications associated with programming in distributed address spaces.

Acknowledgments

Thanks...

- TO the members of my supervisory panel – Malcolm Newey, Robin Stanton and Chris Johnson: for their advice and encouragement.
- TO Mike Bailey, Joanne Evans, James Popple, Trevor Vickers and Greg Wilson: for their comments on and helpful critical reading of various parts of various drafts.
- TO the members of our weekly graduate student forum, particularly the regulars – Steve Blackburn, Stephen Fenwick, Bill Keating and Richard Walker: for their enthusiastic approach to research, constructive debates and general support.
- TO Dave Hawking, Paul Mackerras, David Sitsky, Andrew Tridgell and Dave Walsh: for their first-class work in creating and supporting the computing environments with which I've worked.
- TO the various members of the Department of Computer Science's academic, administrative and computing support staff – particularly Sally Begg, Kath Read and Wendy Novak.
- TO Alisdair Bruce, Simon Chapple, Lyndon Clark, Murray Cole, Neil Heywood, Andy Sanwell and Gordon Smith: for their help and support during six months visiting the Edinburgh Parallel Computing Centre.
- TO Andy Gordon, Thierry Le Sergent, Dave Matthews, Greg Morrisett, Benjamin Pierce and John Reppy: for their discussions and advice on matters concerning ML and concurrency.
- TO the staff at The Gods Cafe – particularly Warwick, Di and Bill: without whose coffee mornings would have been so much less productive.
- AND TO my family and friends – Margaret & Michael, Thomas & David, Reagan, Clare, Penelope, Charlie & Ingrid, Keiran, Kim, Nigel, Subho, Andy & Rachel, Felicity, Graham & Charlotte, Quintin, Simon & Heather, Greg, Don, Gavin & Emma, and Joey: for sustaining me throughout the years.

This research was supported by an Australian Postgraduate Award PhD Scholarship. Additional financial support was provided by a topup PhD Scholarship from the Advanced Computational Systems Cooperative Research Centre and by a TRACS Scholarship from the Edinburgh Parallel Computing Centre.

Contents

Part I – INTRODUCTION	1
Chapter 1. Introduction.....	3
1.1 Motivation	3
1.2 Focus	3
1.3 Process-oriented programming	7
1.4 Contributions of this thesis.....	8
1.5 Thesis structure.....	9
1.6 History	10
Chapter 2. Related work	11
2.1 Introduction.....	11
2.2 Meanings	11
2.3 Static versus dynamic invocation of processes.....	12
2.4 Styles of communication and synchronisation.....	12
2.5 Other ways of managing program partitioning	15
2.6 Specific comparisons	17
2.7 Conclusions.....	21
Part II – DESIGN	23
Chapter 3. Programming model.....	25
3.1 Introduction.....	25
3.2 Processes	25
3.3 Safe and efficient high performance programming facilities	29
3.4 Programming model	35
3.5 Abstract machine model.....	38
3.6 Physical machine model.....	39
3.7 Implications for programming.....	45
3.8 Summary	46
Chapter 4. Design	49
4.1 Overview	49
4.2 Design concerns.....	49

4.3 Core extensions.....	53
4.4 Derived operations.....	62
4.5 Input/output	70
4.6 Machine attribute operations	72
4.7 Concurrency within processes.....	72
4.8 Benefits and limitations.....	73
4.9 Conclusion	74
Part III – THEORY	77
Chapter 5. Theory background	79
5.1 Introduction and motivation	79
5.2 Related work.....	79
5.3 Notation	81
5.4 Formal semantics.....	82
5.5 Summary	99
Chapter 6. Operational semantics.....	101
6.1 Introduction	101
6.2 Syntax	101
6.3 Dynamic semantics.....	103
6.4 Traces	115
6.5 Fairness	117
6.6 Summary	118
Chapter 7. Typing	119
7.1 Static semantics.....	119
7.2 Type soundness.....	122
7.3 Conclusion	128
Part IV – PRACTICE	129
Chapter 8. Applications	131
8.1 Introduction	131
8.2 Algorithmic skeletons	131
8.3 Object stores.....	135
8.4 SIMPLE	141
8.5 Conclusion	143
Chapter 9. Implementation.....	145
9.1 Introduction	145
9.2 ParaML runtime system design.....	145
9.3 ParaML runtime system implementation.....	148
9.4 ML compiler and runtime system extensions.....	154

9.5 Concurrency within processes	157
9.6 MPI as a communication platform	158
9.7 Limitations of implementation	160
9.8 Conclusion	161
Chapter 10. Performance.....	163
10.1 Introduction.....	163
10.2 Multicomputer characteristics	163
10.3 MPI performance.....	164
10.4 ML performance.....	166
10.5 ParaML runtime system.....	168
10.6 ParaML operations.....	169
10.7 Mandelbrot benchmark.....	180
10.8 Optimisations	183
10.9 Conclusion.....	184
Part V – CONCLUSION	185
Chapter 11. Future research.....	187
11.1 Introduction.....	187
11.2 Design	187
11.3 Theory	188
11.4 Practice	188
Chapter 12. Conclusion	191
Appendices, Glossary, References.....	193
Appendix A -1	195
Proof of Lemma 5-6.....	195
Proof of Lemma 5-7.....	196
Proof of Lemma 5-8.....	198
Proof of Lemma 5-9.....	200
Proof of Lemma 5-10.....	202
Appendix A -2	205
Proof of Lemma 7-4.....	205
Proof of Lemma 7-5.....	206
Proof of Theorem 7-6.....	208
Proof of Theorem 7-7.....	209
Proof of Lemma 7-10.....	219
Proof of Lemma 7-11.....	223
Glossary	231
References	236

Figures

Figure 3.1 – Processes and ports in the programming model.....	37
Figure 3.2 – Shared memory MIMD computers.....	40
Figure 3.3 – Distributed memory MIMD computers.	40
Figure 3.4 – The multicomputer model.....	42
Figure 3.5 – Virtual processor model.....	43
Figure 3.6 – Simple sharing of memory in the programming model.	45
Figure 4.1 – A possible asynchronous message ordering; not possible in paraML. ...	59
Figure 4.2 – Synchronous communication.	63
Figure 4.3 – Outport communication.....	64
Figure 4.4 – Guarded choice.	65
Figure 4.5 – Processes used for results of functions.....	65
Figure 4.6 – Process group creation and execution.	66
Figure 4.7 – Basic port group creation and attribute operations.....	67
Figure 4.8 – Basic port group communications.	67
Figure 4.9 – Scattering and gathering to port groups.	68
Figure 4.10 – Synchronous port groups.....	68
Figure 4.11 – Reduction and scan operation interfaces.	69
Figure 4.12 – Barrier synchronisation for groups.	69
Figure 4.13 – Input/output for process groups.....	71
Figure 4.14 – Machine attribute operations.....	71
Figure 5.1 – Ground terms of λ_{ev}	82
Figure 5.2 – Basic syntactic definitions.....	82
Figure 5.3 – Grammar for expressions, exceptions and values.....	83
Figure 5.4 – Free variables in terms.	85
Figure 5.5 – Substitution in terms.....	86
Figure 5.6 – Grammars for evaluation contexts.	88
Figure 5.7 – Rules of reduction with E contexts in λ_{ev}	89
Figure 5.8 – Rules of reduction for references with R contexts in λ_{ev}	89
Figure 5.9 – Basic syntax for type constants and variables.	91
Figure 5.10 – Type rules for λ_{ev}	94
Figure 6.1 – New ground terms.....	102
Figure 6.2 – New expressions and values of the grammar.....	102
Figure 6.3 – Free variables in new terms.....	103

Figure 6.4 – Substitution in new terms.....	103
Figure 6.5 – Grammars for contexts.....	104
Figure 6.6 – Sequential evaluation relation.....	104
Figure 6.7 – Syntactic definitions for process configuration components.....	106
Figure 6.8 – Parallel evaluation rules for: (a) sequential evaluation; (b) exception binding; (c) proc; (d) self_id.....	107
Figure 6.9 – Parallel evaluation rules for prt.....	108
Figure 6.10 – Parallel evaluation rules for execute.....	108
Figure 6.11 – Parallel evaluation rules for send.....	110
Figure 6.12 – Parallel evaluation rules for recv.....	110
Figure 6.13 – Parallel evaluation rules for probe.....	110
Figure 6.14 – Definitions of mem and mem_gc	113
 Figure 7.1 – New type rules for λ_{pv}	 121
Figure 8.1 – Shared memory program execution.....	137
Figure 8.2 – Generic object store in shared memory.....	137
Figure 8.3 – Generic object store in a process-oriented system.....	138
Figure 8.4 – Object store in paraML.....	138
 Figure 9.1 – Programming model components.....	 146
Figure 10.1 – Performance for small messages under MPI.....	165
Figure 10.2 – Performance for large messages under MPI.....	165
Figure 10.3 – Performance for small messages under MPI for paraML.....	167
Figure 10.4 – Performance for large messages under MPI for paraML.....	167
Figure 10.5 – Times for process operation with differing numbers of processes.	172
Figure 10.6 – Times for port operation with differing numbers of processes.....	174
Figure 10.7 – Times for execute operation with differing numbers of processes.....	174
Figure 10.8 – Elapsed sending times for one process and different message sizes...176	
Figure 10.9 – Sending times for differing numbers of processes.....	177
Figure 10.10 – Sending times with standard deviation between runs.....	178
Figure 10.11 – Scatter plot of elapsed sending times.....	179
Figure 10.12 – Scatter plot of user sending times.....	179
Figure 10.13 – Times for Mandelbrot with differing numbers of processes.....	180
Figure 10.14 – Speed for Mandelbrot at size 64.....	181
Figure 10.15 – Speed for Mandelbrot at size 8192.....	182

Tables

Table 4-1 – Summary of extensions. 61

Table 9-1 – Categories of message requests accepted by the message handling
thread. 149

Table 10-1 – Object marshalling and unmarshalling..... 168

Table 10-2 – Times in *μsecs* for function loop and process descheduling and
rescheduling..... 169

Table 10-3 – Times and standard deviations in *μsecs* for *self_id*, *probe*,
recv and *self_port* operations..... 170

Table 10-4 – Times and standard deviations in *μsecs* for naive translation of
self_port operation..... 171

Table 10-5 – Costs of paraML operations per process in Mandelbrot program 182

Part I

INTRODUCTION

Part I

INTRODUCTION

1. Introduction

1.1 Motivation

Modern computing systems look radically different from those of thirty years ago. Computers are now ubiquitous, and immensely more powerful. Long gone is the time when it was possible to assert that all of a country's computing requirements could be supported with a single computer. As computers have become more and more widespread, the range of problems to which they have been applied has expanded similarly. And as computers have become more powerful, the degree of complexity in the problems being attempted has increased. Indeed, there exist many problems of interest which existing sequential computers are incapable of solving in a satisfactory time.

High-speed interconnection *networks*¹ are altering the possibilities for how computing services are provided. For instance, programs may reside on quite separate computers from the one in front of a user. These programs may execute either on the local computer after being sent across the network, or be invoked remotely with data from the user through some interface such as a Web browser. Computers connected by networks constitute a distributed computer system.

The realisation of programming environments to meet the changing nature of computer systems, particularly with respect to distributed computer systems, is a challenging task. The goal of this thesis is to show that process-oriented programming facilities provide an effective solution to some of the difficulties involved.

1.2 Focus

The three main concerns addressed by this thesis are:

- Supporting computation in the context of a *distributed address space*.
- Enabling *partitioning* and *coordination* of computation.
- Providing programming facilities that are both *efficient* and *safe*.

As usual in computing, many of these terms are open to different interpretations. The focus of the thesis will be better understood with clear definitions of the key terms as they are used in this thesis.

¹ Short definitions of italicised words are available in the Glossary, p. 231

1.2.1 Distributed address spaces

The essence of distributed address spaces is that the physical computer memory is partitioned among a number of *processing elements* (PEs) or *nodes*, and there is a one-to-one mapping between address spaces and memory partitions. Each PE typically consists of one or more *central processing units* (CPUs), memory, and network interface components which connect that PE to other PEs. Non-local access to a PE's memory must pass through a network which connects the PEs. At some layer of programming abstraction, the distribution of the address space, and consequently memory, becomes visible. The effect of this visibility means the programmer becomes aware of a difference between local and non-local memory access.

The distributed approach is in contrast to the traditional model of computing, where even if there are multiple CPUs, any of them may access any part of memory on an equal basis and there is a single *shared address space*. Some modern computers use physically distributed memory for scalability purposes, but provide a shared address space so that the same mechanism is used for both local and non-local memory access.

Computer systems with distributed address spaces are becoming increasingly common. Networked computers, from *multicomputers* to clusters of workstations to the Internet, have progressively wider memory distribution among nodes. As networked computers become more widespread, finding good mechanisms to program with the distributed address spaces they embody becomes more critical.

For sequential algorithms, a distributed address space requires a program to draw together disparate data entities into a computation. This action can be done by a layer which emulates a shared address space from the various memory components of the distributed address space. The merits of hiding the distribution of memory in layers of abstraction below programming languages are part of an ongoing debate. This issue is not central to this thesis, and its examination is deferred until later.

The network *bandwidth* and *latency* involved with distribution over the Internet can be highly unpredictable. For reasons of simplicity, this thesis examines only distributed address spaces over networks with high bandwidth and low latency, as may be found in multicomputers.

1.2.2 Concurrency: partitioning and coordination

Some problems are best described by partitioning the actions to be performed into separate tasks. For example, consider the problem of modelling electricity generation by steam turbines in a power station. Water must be piped to the power station from a reservoir. The water is turned into steam, which is used to drive the turbines. The turning turbines generate electricity, which must then be transmitted onto the electricity grid. Modelling this system is most naturally accomplished by coding each task and its interactions as a separate component. The area of *concurrent* computing addresses the issues involved with problem partitioning.

Many concurrent systems assume an underlying shared address space, since the goal of such systems is to introduce concurrency, not to manage issues of distribution. On other occasions, a problem is not naturally specified concurrently, but the performance of a sequential implementation is inadequate. In these instances, some partitioning of the problem is usual, to make efficient use of a computer with a distributed address space and multiple PEs.

Returning to the steam generation example, modelling of the turning turbines can be performed with varying degrees of precision. A very simple model will encode the rate of electricity generation by the amount of steam being delivered. However, the power station may wish to optimise its electricity generation. If a simple model is too inaccurate, a more complex simulation would become necessary, perhaps modelling the fluid dynamics of steam in the turbine. Fluid dynamics simulations are computationally expensive, and could require the resources of high performance computers with distributed address spaces. The space through which the fluid flows might be modelled with a 3-dimensional mesh decomposition, with different mesh elements mapped to different PEs.

In both cases, there is a notion that a single program is being executed, albeit composed of different parts. This characteristic is significant, since it distinguishes systems designed to support partitioned programming from those which support multiple distinct programs. The latter system type is often referred to as a *distributed system* or sometimes as a *coordination system*. Many of the issues involved in distributed systems are similar to those described for partitioned programs in distributed address spaces. However, there is considerably stronger coupling between the program components in a partitioned program than necessarily occurs in a distributed system. This thesis investigates the challenges of partitioning and coordinating the components of a program in a distributed address space. It does not concern itself with the issues of managing and coordinating a distributed set of distinct programs, which is the domain of distributed system languages or coordination languages.

Partitioning in the specification of a problem (either to achieve better performance or because the specification is inherently concurrent) usually requires some mechanism to coordinate the various activities. Coordination typically occurs by the communication of some information, either directly between the two (or more) program entities or through some intermediary. Typically information is communicated by either a message passing system or some common data structure manipulation.

In the power station model example, communication is required to carry information about rate of delivery of water to the power station, rate of production of steam and delivery to the turbine, and so on. Coordination is also required so that steam production does not occur prior to the water being delivered from the reservoir. In the fluid dynamics example, communication occurs between neighbouring elements about the state of steam movement through the spatial element in the previous step of

the simulation. Coordination is required to make sure that any steam which has migrated into the element during the previous step is incorporated into the calculations for the current step of the simulation.

1.2.3 Efficient programming facilities

There appears to be no limit to the desires for increased performance. One of the barriers to increased performance is access to memory. On shared memory computers, the access to memory becomes a bottleneck. Physically distributed memory offers a scalable solution to this bottleneck, and computers with multiple PEs embodying distributed address spaces will continue to be an effective means of achieving performance improvements. For example, the precision of a simulation is dependent on the resolution in the spatial decomposition. If the space can be modelled with a larger number of smaller components, then the margin for error diminishes since each component needs to make fewer assumptions about interactions with other components. Smaller spatial elements will also result in a closer match with the actual space being modelled. However, having more elements means more computation and more interactions overall, thus requiring more computing performance to solve the problem in the same time. Being able to model the fluid dynamics of the turbine more precisely may result in more cost-efficient operation.

Distributed address spaces and partitioned program specifications pose both a challenge and an opportunity for achieving high performance. The challenge arises because of the extra overheads and delays in coordinating concurrent actions in such an environment. The opportunity arises because of the potential for simultaneous or *parallel* execution of the different actions. Achieving maximum performance means the overheads of managing parallelism must be minimised, and thus the programming facilities must be efficient.

1.2.4 Safe programming facilities

Programs may fail in a number of ways. Some of these failures may be statically detectable before the program is ever run. Other failures are impossible to detect prior to execution, and occur while the program is running. The meaning of "safe" as it is used in this thesis is firstly that programs should not fail because the language permits semantically incorrect operations (such as adding strings to integers) which can be detected statically. Similarly, the design of the language should not permit avoidable system failures (such as those arising from memory leaks or null pointer dereferencing). If errors do arise which are undetectable prior to program execution (such as division by zero), there need to be clean mechanisms for identifying and recovering from such errors.

Of particular concern are ways to statically prevent and/or dynamically recover from failures in the mechanisms for partitioning and coordinating concurrent actions. For instance, if the destination of a communication action is no longer active, the

sender should be informed and have means of performing an alternative computation. More problematically, a system-wide deadlock situation should be identified and indicated to the user, even if there is no clear recovery state. Circumstances in which an otherwise safe program might work incorrectly would be because the programmer makes a logical error in the construction of the program (such as a non-terminating computation, which is generally undetectable).

For example, matching power generation to the demand for electricity in the grid is a problem that can be optimised by “just-in-time” simulations. These simulations are used to adjust the delivery of the various requirements for power generation according to changing demand just as the demand arises. Meeting this need requires both a very efficient simulation and one which is safe. Suppose the simulation is used to drive delivery of water to the steam generator to increase turbine speed (and thus electricity generation) when demand rises. If the reservoir suddenly drops below a critical level, the simulation must detect this condition, and prevent the steam generator from continuing at the previous rate. A critical failure of this kind could be addressed in the simulation using exception-handling language facilities.

The mechanisms which expose distributed address spaces and the mechanisms which specify problem partitioning necessarily introduce new complexities to a programming language. The need for these mechanisms to be as safe and robust as any other aspect of a language is just as strong as in sequential programming.

1.3 Process-oriented programming

The basic notion of a *process* is that of an entity that contains some code to perform an action or series of actions, while maintaining information on the state of computation, including what will execute next in the code. The process encapsulates any data (or memory) required by the code for its execution. If data is not available when the process commences, it may be communicated to the process through some defined interfaces. These interfaces may be established dynamically during the course of execution of the code of the process.

One of the best mechanisms to achieve high performance is by utilising the resources of *parallel* computers. For a program to make use of such parallel computers, there must be ways to partition the problem into processes that can execute simultaneously. Coordination of the processes occurs through communication. As such, processes are naturally suited to programming in a distributed address space, as each process may be mapped to a different PE and thus a different part of memory. The interconnection network would then be used for communication.

Coordination by communication among processes must be managed safely. At a purely textual level, it is helpful to distinguish different sorts of information to be communicated to a process. Such separation is achieved by providing multiple *ports* as

the destinations to which information is sent. Additional measures for achieving safety tend to be language-dependent.

In *object-oriented* computing, the primary structuring unit of computation is an object. Methods define the set of actions that may be performed by the object, and are invoked only when messages are sent to them. The term “process-oriented” is used here to capture the notion that the primary structuring unit is the process. Unlike object methods, where the semantic effect of message arrival is the invocation of some defined action, processes use ports just to receive information. The information may be used in whatever manner desired, as specified in the process code.

Process-oriented programming is useful for tackling the complications of efficient partitioned computation in distributed address spaces.

1.4 Contributions of this thesis

Supporting a process-oriented style of programming requires the development of some semantic notions for processes and ports in a programming language. The approach taken in this thesis is to extend an existing programming language, ML [MTH90].

The development of ML took place over a number of years, and represents the fusion of many efforts in building a rigorously defined, safe and high-level language. ML is based on the functional programming paradigm, but incorporates imperative features. It has a history of use in exploring new programming features: for example, concurrency, data parallelism, and distribution; as well as various developments in type systems. Several high-quality and efficient implementations of ML exist which are available for modification. ML is an excellent platform on which to develop the necessary semantic notions for processes and ports. ML is useful both as an implemented language for computation within processes and as an instance of a sound theoretical model.

The name given to the process-oriented extended ML language is paraML. ParaML’s extensions to ML are a compact set of requirements for supporting the process-oriented model within a strongly-typed programming framework. In paraML, processes provide services to create communication facilities and to execute some code. Evaluation takes place against an encapsulated data environment local to the process. Communication is performed by message passing to ports which hold queues of messages. Ports are created dynamically during execution and are local to individual processes.

A particular focus of the experiment has been to explore ways to achieve safe and efficient programming facilities. This thesis has thus developed the process-oriented extensions to ML in such a manner as to provide support for safe and efficient computing. The ability for paraML to address four major themes specific to high performance computing in the context of distributed address space computers and partitioned problem specifications is also explored. These themes are:

- Concurrency – ability to make use of multiple computing resources.
- Scalability – ability to utilise increases in the number of PEs.
- Portability – independence from particular makes of parallel computers.
- Locality – promotion of local data access through the programming model.

The design of the language also addresses a number of issues that are relevant to achieving safety, including:

- Simplicity and minimalism in language extensions.
- Type-safe communication during program execution.
- Formal modelling of the language.
- Non-determinism.
- Modularity of program components to permit composition and clean interfaces to their environment.

The extensions are formally characterised by an operational semantics complete with a proof of type soundness. Theoretical specification of computation with values that have been communicated from other memories, including mutable values, under a copying semantics has not previously been demonstrated. The value of the paraML language is demonstrated by its use in the development of derived operations, alternative high performance programming paradigms and applications.

1.5 Thesis structure

This thesis has five parts. The motivation and thesis are introduced in Part I – Introduction (Chapters 1-2), together with an overview of related work. The process-oriented programming model is used to motivate the choice of primitives in Part II – Design (Chapters 3-4). The language is then modelled formally by an operational semantics in Part III – Theory (Chapters 5-7). The design is demonstrated by its use as a support system for other programming models and its use in applications, as described in Part IV – Practice (Chapters 8-10). Some possibilities for future developments are discussed in Part V – Conclusion (Chapters 11-12).

It is assumed that the reader is familiar with the ML language; if not, there are a number of textbooks on programming with the language now available including *ML for the Working Programmer* [Pau91] and *Elements of ML Programming* [Ull93]. A formal definition of the language [MTH90, MT91] also exists, as do overviews of ML in other dissertations such as Reppy's *Higher-Order Concurrency* [Rep92]. The theoretical part of this dissertation does not assume familiarity with the style of semantics used, but a basic understanding of the λ calculus will be helpful; Barendregt's work provides a solid introduction [Bar84].

1.6 History

The thinking behind paraML has evolved over the last five years. Preliminary ideas were developed and explored with an initial implementation of the paraML language [BN93, BNS+94] culminating in a version 1.0 release for the Fujitsu AP1000 multicomputer in January 1994 [Bai94]. The early version of the language was used for a number of purposes, including as the implementation language for a parallel Hough transform application [Und95]. Since that time, work has progressed through a major evaluation and redesign of the language, and the concurrent development of a formal semantics [Bai96], leading to the current version as described here.

2. Related work

2.1 Introduction

As with any experiment in language design, the choices made in developing paraML have been conditioned by its intended use. The three major concerns of the experiment (distributed address spaces, partitioned computation, and safe and efficient high performance) overlap with many related experiments in concurrent, parallel, and distributed language design. It is helpful to examine some of the different options available and their impact. Four languages with similar concerns to paraML are examined in detail.

2.2 Meanings

The terms *parallel* and *concurrent* have frequently been used synonymously in computing. More recently, distributed computing has at times been used to imply parallel computing. The terms are also applied to computer hardware, programming languages and individual algorithms and programs. With respect to programming languages, the terms will be used as follows for this thesis:

- *Concurrent* means the language permits the user to specify a program as a collection of independent tasks, capable of interaction.
- *Parallel* means the language provides facilities to specify concurrency, and there is an expectation that when a program is executed, a subset (possibly all) of the tasks will be evaluated simultaneously. This parallel evaluation is achieved by parallelism in the hardware, usually involving several PEs.
- *Distributed* means the language provides facilities to specify concurrency, execution of a program's entities will exhibit parallelism, and the tasks will be executed on physically distributed (possibly heterogenous) computers. In particular, the user is made aware of the effects of distribution on program execution, including the possibility for failure in some parts of the system.

The term *parallel* will be used both in the manner defined above and to denote the superset of concurrent/parallel/distributed computation where the distinguishing aspect is simultaneous execution of the tasks. The issues that paraML explores are predicated on execution of programs on a parallel computer with a distributed address space, such as a multicomputer or tightly-coupled local area network of workstations. However, it is assumed that issues such as node reliability or heterogeneity are handled transparently. These simplifications enable paraML to focus on support for safe and efficient high performance computing in the context of a distributed address space.

2.3 Static versus dynamic invocation of processes

Parallel processes may be explicitly invoked either statically or dynamically. Static invocation establishes a configuration of processes executing in parallel within the boundaries of some defined start and end points. Frequently languages which allow only static invocation further restrict process creation to the start of the program.

Dynamic invocation establishes a new process operating in parallel with the invoking process. Dynamically invoked processes provide greater flexibility in accommodating algorithms which require changing computational requirements during the course of a program's execution. This is one of the reasons dynamic process creation is used in operating systems such as UNIX and in message passing systems for networks of workstations such as PVM [GBD94] and MPI-2 [MPI96]. Dynamic process invocation is adopted in paraML primarily due to the inflexibility of static process creation.

2.4 Styles of communication and synchronisation

Coordination among processes is most often achieved by some form of communication of information. There are a wide variety of ways to communicate information between processes, but these fall into basically two classes which reflect the characteristics of the underlying address space. Hennessy and Patterson remark that the shared address space form of communication is an implicit one involving loads and stores, while distributed address space communications are explicit, involving sends and receives of messages [HP94]. However, both of these can be layered on top of either shared or distributed address space computer systems.

2.4.1 Shared address space communications

Shared address spaces may be a property of the computer and/or its operating system, or they may be emulated in software from an underlying distributed address space. The latter form is often referred to as *distributed shared (virtual) memory* (DSM or DSVM) [LH89]. A shared address space permits processes to communicate via some common data structure. One process updates or writes some data value which is then readable by another process. Since more than one process may be active and attempting to access the same data value at any point in time there is a need to preserve the integrity or *coherency* of the data. Effectively this means that mechanisms must exist so that only one process may actually alter the data value at any time. These problems also arise in the context of multiple local caches of variables.

A wide variety of mechanisms exist to preserve data consistency. Low-level solutions include protection mechanisms by other data objects such as *locks* and *semaphores* [Dij68] or *critical regions* [Bri73] which indicate when a data structure is being accessed by some process. Such features are to be found in Ada [Ada83] and Modula [Wir77, Wir83] for example. Higher-level variants of this form of data sharing

include M-structures [BNA91] and the M-vars of Concurrent Haskell [PFG96]. Another alternative is to provide data structures that may only be modified once, though read many times, such as I-structures [ANP89]. An alternative to coherency of individual data values is to adopt a programming model that enforces a consistent set of data structure accesses and updates. Examples of this approach include *monitors* [Hoa74], *transactions* [LS83, Mos85], and *Linda* [ACG86]. Many variants of these different forms exist, often designed to relax the serialisation properties of data access (that is, in order to increase the achievable level of parallelism). Non-determinism with shared variable forms of communication is primarily a result of the scheduling policies of processes in the system.

2.4.2 Distributed address space communications

The distribution of an address space typically requires that information be explicitly communicated by means of messages. Since a process may have no access to any shared data with another process executing in a different part of the distributed address space, communication occurs by physically transmitting data between their local memories. In the simplest case, one process *sends* a message and another process *receives* it. Many extensions exist, which expand the number of senders (*scatter-gather*) or receivers (*broadcast, multicast*).

There are also choices to be made about the mechanisms used to send or receive data. For instance, should the message be *buffered* at the sender/receiver, should the message be sent *synchronously* or *asynchronously*, should the destination be named as a process or as a *channel/port*, and so on. Many of these alternatives are addressed and supported in the recent standardisation of a message passing interface for parallel computing – MPI [MPI94, GLS95].

Message passing may also be used on top of shared memory systems in order to support particular models of interaction, such as *pipelining* or *client-server* systems. Message passing models do not promote sharing of data, but have the advantage of making data locality very straightforward. If data is required by a process then either it already has the data or the data must be obtained by it in a message from some other process, after which the data is locally available. There have been some interesting results which indicate that a message passing programming model can achieve greater data locality and thus perform as efficiently as, or better than, a shared address space programming model on a shared memory computer system [NS93], but that is not of direct concern to this thesis.

2.4.2.1 Synchronous message passing

Synchronous message passing requires that both the sender and the receiver of a message are engaged simultaneously in the communication operation from the point of view of some observer. The act of communication is deliberately merged with the act of synchronisation between processes. Communicating sequential processes (CSP) [Hoa78] is the canonical model of this form of message passing. A major advantage

claimed for such systems is that the synchronous nature of message passing makes it easier to reason about programs. A number of process calculi (including CCS [Mil80] and the π calculus [MPW92]) exhibit synchronous communication by message passing.

2.4.2.2 Asynchronous message passing

Asynchronous message passing makes no guarantee about the synchronisation of the sender and receiver. Thus the sender may proceed past its communication operation without the receiver having completed or even initiated the matching part of the communication. An asynchronous version of the π calculus [HT91] exists which can be shown to be equivalent to the synchronous version. This result is not surprising given that it is clearly possible in synchronous systems with dynamic thread creation to model asynchronous communication by creating a new thread which accepts a send request and then attempts to complete the communication synchronously. The original thread that performed the send request is able to continue execution asynchronously.

Asynchronous message passing is better suited to distributed systems or parallel systems with distributed address spaces due to the increased latencies involved in message transmission. Since a sending process is not delayed waiting for the receiving process to actually perform a receive operation, the increased latencies are immaterial to the sender. This characteristic becomes important as the number of computing resources increases.

2.4.2.3 Buffering and blocking

The MPI standard has made explicit the implications of synchronous and asynchronous communications by providing different modes of communication that deal with memory and control requirements. In fact, MPI does not even provide an "asynchronous" mode of communication. There are four major modes for sending that MPI defines:

- synchronous – the operation does not complete until the matching receive operation completes;
- ready – the operation does not return control to the caller until a matching receive operation commences;
- buffered – the operation returns control once the message has been buffered for transmission (or a buffer is provided to a matching receive operation);
- standard – this mode is the generic mode, and may or may not buffer the outgoing messages according to the underlying implementation and runtime conditions. The operation is blocking in that it does not return control to the caller until the message has been stored for transmission or successfully delivered.

Each of these operations blocks control of the sender until the conditions of the mode are satisfied. However, it is also possible to return control immediately to the sender by the use of non-blocking variants of these calls. In these cases, a handle is returned

which may be used to test completion of the operation. Most asynchronous communication in existing systems is equivalent to buffered mode in MPI.

2.4.2.4 Naming of the communication destination

There are many different mechanisms for specifying the destination of a communication request. The simplest is merely the identifier of another process; it is up to the program to discriminate between the contents of different messages. Frequently this model is expanded to allow meta-data to be carried with message content, including the identifier of the sender and some integer tag field.

MPI provides an additional layer of abstraction, by allowing the creation of *communicators* by subsets of the processes in a system, which are a communication layer disjoint from any other communicators. This facility permits the construction of software which isolates messages between functionally different components within a program, as in library routines.

Another form of abstraction is that of *channels* or *ports*. Sometimes channels have a notion of restriction to use by only two processes – the sender and receiver. Sometimes channels have a notion of being bidirectional, with the implication that processes may both send messages on and receive messages from channels. Channels are also referred to as *sockets* in UNIX terminology. Ports by contrast tend to denote a communication destination only, owned by a particular process which is privileged to read from the port. Ports are sometimes referred to as *mailboxes*. Other processes may send messages to the port, but may not receive messages from it. It is feasible to create groups of channels and ports in instances where multiple senders or receivers are envisaged [Kru93]. In strongly-typed languages, channels and ports tend to be typed to restrict the kind of object which may be transmitted on them. For example, an integer-typed port will allow only integer messages.

2.4.2.5 Non-determinism in communication

With most forms of message passing, it is highly likely that some non-determinism in message arrival order will occur. As soon as two processes are capable of sending messages to a single receiver, unless there is some other synchronisation that takes place, it cannot be determined in advance which message will arrive first. Message passing can be made deterministic, as Foster and Chandy point out in the design of Fortran M [CF93, FC94], by restricting the use of a channel to a single sender and single receiver. Similar problems with non-determinism arise with communications by shared variables. In practice, most parallel programs are written to cope with non-determinism.

2.5 Other ways of managing program partitioning

The discussion so far has centred on program partitioning achieved through multiple processes: ways to initiate them and how they communicate. There exist other ways to

manage partitioning, including algorithmic skeletons, data parallelism, futures and actors.

2.5.1 Algorithmic skeletons

Algorithmic skeletons were developed by Cole [Col89] as a means to abstract from programmer involvement in the management of parallelism. Essentially, a skeleton provides a framework to perform all the creation of computation processes and communication and synchronisation required. The user is responsible for providing a set of operations that actually perform the work for a particular application. In many ways this is analogous to a sort routine in a library, where the user provides the datatype specification and a comparison function, and the sort routine actually implements the algorithm. The other major benefit from skeletons is that they are developed with a complexity metric to enable accurate performance predictions for applications across different computational platforms. The area of skeleton-based parallel programming has seen significant research interest in recent years (for example [BDO+95, DGT+95]), including a system for ML [Bra94].

2.5.2 Data parallelism

Data parallelism takes a completely different approach to the predominantly process parallel systems discussed above. The way parallelism is incorporated is to find operations that can be applied to many data elements simultaneously. A straightforward example is adding 1 to every member of an integer array. A large percentage of scientific applications have many opportunities for data parallelism, which has been one of the driving forces behind the development of High Performance Fortran (HPF) [HPF94]. Data parallelism has been investigated by extensions to many existing languages, including C [HQ91], C++ [LG91, LRV92] and ML [HF93], as well as inspiring the development of novel languages such as NESL [BCH+93] and SETL [Lev75], which is based on set comprehensions rather than parallel arrays. The popularity of data parallel languages was also supported through custom-designed parallel computers such as the Thinking Machines CM-2 and the MasPar.

2.5.3 Futures

Futures were developed by Halstead for MultiLisp [Hal85] as a way to exploit parallelism in functional programming languages. In essence, futures involve spawning a new thread of computation and binding the future result to a variable. Computation can proceed asynchronously with the variable representing the future result until such time as the actual value is required. If the future thread has completed execution, the result will already have been associated with the variable; otherwise computation will block until it becomes available. Futures neatly encapsulate a concept of result communication and synchronisation between threads. The main disadvantage of the model is that it leads to quite a fine-grained level of parallelism which relies on a programmer identifying parts of the computation which can proceed asynchronously.

2.5.4 Actors

The actor model of programming [AH87] involves another hybrid form of communication and program partitioning. Actors are both active computation entities and also communication messages. They predate the development of the higher-order distributed object-oriented systems such as Obliq [Car95], but bear strong operational resemblances. The main limitation on their performance has been in the cost of initiating threads of control, since the model lends itself to a fine-grained parallel approach and programs may end up with thousands of actors.

2.6 Specific comparisons

There are numerous examples of extensions to languages and language extensions which adopt a process-oriented model, using either shared variable and/or message passing forms of communication. These include imperative languages such as Fortran M [FC94], object-oriented languages such as Concurrent Eiffel [Car90], Modula-3 [Nel91] and Java [Sun95], and functional languages such as Concurrent Haskell [PFG96]. When combining object-oriented concepts with higher-order programming languages, object methods may accept arbitrary arguments including code represented by closures. Obliq [Car95] is a good example and in some ways the forms of communication are similar to those of a higher-order programming language with ports and dynamic process creation. However, to minimise the number of differences due to the nature of the underlying language, this section examines some specific examples of other mostly-functional languages that have been extended for parallelism (in the broad meaning of the term) with explicitly invoked processes (or their equivalent) and message passing. Exploring the reasons for their development and the ways in which they differ from paraML will help to clarify the distinctiveness of the approach taken in this thesis.

2.6.1 CML

Concurrent ML (CML) [Rep92] is perhaps the most elegant example of extensions to ML for concurrency. CML is representative of most of the concurrent extensions to ML such as Distributed Poly/ML [Mat91] and LCS [BlS94], and provides a higher-level model than the systems-level concurrency package ML-Threads [MT93]. Threads and synchronous message passing on channels with guarded choice are provided as the basic primitives. From these, Reppy developed the notion of an *event*, which is a synchronous operation that may be treated as a first-class abstracted value. Events can be used to construct alternative forms of concurrent operations by composing and manipulating them in different ways.

Although CML uses message passing for communication, this is done to support certain styles of concurrency, not because the language is based on a distributed address space programming model. CML assumes a shared address space programming model; its provision of channels and guarded choice are very difficult to

implement efficiently under a distributed address space programming model. Reppy has explored the possible development of a version of CML for distributed systems where he suggests the use of a Linda-style tuple space [Rep94], based loosely on an earlier development of ML-Linda [SC91]. Reppy remarks that CML's programming model is not well suited for distributed programming with remote communication and that individual processes may well be best structured as concurrent programs in their own right.

The operational semantics of CML [Rep91], developed as an extension of Plotkin's λ_v calculus [Plo75], does not actually prevent a distributed address space programming model, but neither does it address the issues that arise in a distributed address space implementation such as transmission of references in messages. Reppy provides an encoding for shared references by means of channels and threads, whereby a channel stores the current value, and messages sent to dedicated channels are processed by the thread to report or update the current value. The problem with sending normal reference values is that a reference identifies a mutable value in a particular address space. Thus if the reference is transmitted to another process in a different address space, it is no longer clear to what the reference refers.

Krumvieda's work on Distributed ML (DML) [Kru93] is based on CML, but incorporates asynchronous multicast transmission of messages on port groups. A number of other extensions were planned to allow DML to fully address the requirements of distributed computation (in a similar manner to the Isis toolkit developed for C at Cornell University [BJ87]), but none of the other projects proceeded and it is thus difficult to critique DML for these inadequacies.

Work by Matthews and Le Sergent [MLS95] and Steckler [Ste96] with Distributed Poly/ML and LCS describes mechanisms for efficient implementation of concurrent programming in a distributed computer system. The approach taken is to develop a DSM, using type information about mutable objects to optimise data consistency algorithms for the DSM. Static analysis of the locality of communications allows optimised implementations of the `channel` primitive, thereby avoiding message-passing overheads. Since a DSM is used, a shared address space, rather than a distributed address space is implied for computation.

2.6.2 Facile

Facile [TLP+93], developed by a group at the European Computer-Industry Research Centre, is another extension of ML, but unlike CML is designed to be used as a distributed programming language. Facile programs are structured as collections of dynamically-created processes which communicate synchronously over channels. Processes are executed on dynamically-created nodes, which correspond to the notion of a virtual processor. Processes conceptually have their own local environment, but are executed against a common heap for all other processes on the node, thus allowing reference variables to be shared. Several nodes may be executed by a single computer.

Facile also deals with node failure, heterogenous execution environments, and type-safe communication between processes from separately-compiled programs. The latter has required Facile to make some alterations to Standard ML's structure-level syntactic constructs. These allow the user to dynamically obtain structure/signature definitions from a structure server in the network.

The ability of a Facile programmer to specify distribution (through node creation) and concurrency (through process creation) is excellent. It is my assertion that the major weakness of the system is in the corresponding failure to distinguish concurrent communication from distributed communication. Both are handled through the synchronous channel mechanism and channels are separate entities, not owned either by a node or a process. Thus in a distributed communication three separate nodes may be involved (one for the sender process, one for the channel, and one for the receiver process).

Transmission of values on channels is not uniform in that references are shared when communicated between two processes and a channel all on the same node, but copied otherwise. The fact that distributed communication is not syntactically distinguished from concurrent communication means that it becomes a much more difficult (perhaps impossible) task for the programmer to inspect a program (or a part of one) and understand the outcome of communication involving references. The Facile group adopts the same theoretical solution as Reppy of encoding shared references as channels and threads. Such a translation has been shown to preserve the semantics of references [BMT92]. However it is arguable whether such shared references are actually desirable for programming in the context of a distributed address space. Neither do shared references capture the notion of copying. The operational semantics for Facile [Ama94], like that for CML, does not deal with references.

2.6.3 STING

The STING system [JP92, JP94] is aimed at providing efficient and flexible runtime support for concurrency experiments with programming languages such as Scheme and ML. Written in Scheme, the system provides facilities for:

- thread creation;
- customizable protocols for scheduling, migration, and load-balancing; and
- adjustable thread execution strategies and storage allocation policies.

The implementation makes extensive use of first-class procedures and continuations in its approach to concurrency support. All the elements of a typical parallel computer system (physical machines and processors, virtual machines and processors, and threads) are modelled abstractly and exposed as first-class objects in STING, and the user is able to adjust such parameters as virtual processor and thread scheduling policies.

STING assumes a shared address space, so that on distributed address space machines the system relies on a distributed shared memory (DSM) platform. Object references are thus defined no matter where the object is actually located. Although threads are responsible for managing their own private and public heaps, garbage collection is supported across thread boundaries. STING also supports message passing via ports, using an asynchronous send and a blocking receive semantics. The designers remark on the similarity between active messages [vEC92] and communication of first-class procedures to a thread-spawning process.

The STING system provides an example of how programming languages such as Scheme and ML can be used to experiment with parallelism in ways made significantly easier by the language support for first-class procedures and functions. The designers do not commit themselves to any particular model of parallelism, seeing the system instead as a way to experiment with different models. A formal semantics for the system has not been developed; it relies instead on the semantics for Scheme. While their results are impressive on shared memory multiprocessor systems, it is not clear how these would translate to a distributed address space system that would need to support a DSM.

2.6.4 Kali Scheme

Kali Scheme [CJK95] is a distributed extension to a dialect of Scheme called Scheme 48 [KR94], which in itself provides lightweight concurrent threads with condition variables and locks for synchronisation. Scheme 48 and Standard ML are quite similar, and it is worth noting that although type checking in Scheme is performed at runtime, rather than statically, finding ways to increase the level of static type checking is an active research area [JW95]. Kali Scheme extends Scheme 48 with facilities for message passing of objects between address spaces. Address spaces are first-class objects in their own right, and may be created dynamically. By default, most objects (including closures) transmitted between address spaces are structural copies of their value in the source address space. As a consequence, successive copies of mutable objects are not considered equal.

Kali Scheme adds a new form of object called a proxy, which consists of a record with two slots: the first holds a system-wide unique identifier (uid), and the second holds a value. When a proxy (or an instance of other classes of objects such as procedure templates and symbols which are uniquely identified) is transmitted between address spaces, only the uid component of the object is included. Operations are provided for determining the address space on which a proxy was created, determining the value component of the proxy on the creating address space, and setting the value component of the proxy to a new value on the executing address space. Local access to a proxy's value does not require any global synchronisation or communication, and proxies generally allow the programmer to distinguish local and remote values.

Kali Scheme also provides facilities for remotely executing code and migrating thread computations by way of continuations. The system performs mostly local garbage collection (for non-proxy objects), with some asynchronous global garbage collection to deal with proxies which may be globally referenced. Kali Scheme is expected to be executed on either loosely-coupled or tightly-coupled networks of workstations, although the system does not provide facilities for dealing with failure of nodes and communication channels. It is not clear whether the system executes on heterogeneous workstation networks; however, Scheme 48 is executed by a byte-coded interpreter, and thus has the potential to support heterogeneous program execution in the same manner as Java [Sun95].

As with STING, there is no formal semantics for the language at the present time. The strong point of the system is the notion of proxies, which allows users to adopt different styles of parallelism as appropriate to the application; proxies may be used to implement on-demand sharing of values across address spaces without the overheads of a DSM, and also to implement distributed data structures where values for a proxy object are different between different address spaces. The system also provides for automatically-marshalled data content in message passing forms of parallel program construction.

An interesting observation from the four systems studied above is that the ML heritage provides an impetus for a formal theoretical treatment while the Scheme heritage does not. All of the systems studied make use of the higher-order features of Scheme and ML to implement complex concurrency primitives in flexible and interesting ways.

2.7 Conclusions

Most efforts for dealing with distributed address space programming are addressed by distributed programming languages. Partitioning of computation is addressed more generally by concurrent, parallel and distributed programming languages, but the means of coordination vary enormously. Relatively few systems attempt to combine both safety and high performance. While the Facile project has many similar concerns to the paraML experiment, its emphasis on distributed computation rather than efficient high performance computation and its failure to formally model copied mutable reference values makes the paraML experiment important to pursue.

The design of paraML discussed in the next part of this thesis attempts to combine safety and efficiency in the manner of CML, while addressing programming in a distributed address space context in a manner similar to Facile. It also provides a semantics for copying of mutable values between these address spaces, although without providing the flexibility of Kali Scheme's shared and copied mutable objects. The separation of abstract notions for processors and virtual processors provided in STING is adopted for components of the programming model, although not exposed to become user-specifiable.

Part II

DESIGN

Part II

DESIGN

3. Programming model

3.1 Introduction

In examining the design of extensions to ML for process-oriented programming, it is essential to consider the programming model that the language is to support, and what goals the programming model is trying to achieve. The programming model is founded on the notion of *processes*, and it is worthwhile considering exactly what are processes, independent of any end use. Subsequently, the ways in which processes can support the goals of safety and efficiency in a high performance programming context are considered. The programming model of paraML is described with respect to the evaluation, memory and communication traits that are identified with processes. These traits are also used to motivate the abstract machine and physical machine models and the associated software characteristics of these layers. Lastly some of the implications for programming under such a model are discussed.

3.2 Processes

3.2.1 History

The concept of processes has been explored by many people during the history of computing. Two main uses for processes emerged almost immediately: operating system support for the management of multiple user programs, and concurrent programming specifications. The focus of concern in operating systems for supporting multiple user programs was on protection, particularly with regards to shared data. Processes were viewed as an elegant means to specify independence between different user programs. In concurrent programming, processes were seen as structuring mechanisms within a single program for disjoint, but possibly related, series of actions.

The definition of process used by Dennis and Van Horn was a “locus of control within an instruction sequence” [DVH66], and they described both of the above uses for processes. Dijkstra’s understanding of a process was similar, though he specialised it to be a sequential process where the program was a sequence of rules of behaviour [Dij68]. These definitions capture a primary property of a process: that it acts as a self-contained evaluation entity.

While parallel programming did not make an immediate impact, the expensive resources of available computers in the 1960’s meant that operating systems capable of supporting multiple users at the same time became extremely important. An example of this kind of use was the Multics system [DD68, Org72], which tied the notion of a process to the concept of an *address space* or *virtual memory*. Each process executed with respect to its own address space, created independently of all other address

spaces. Multiplexing of the available processor computation resources to each process was performed by the “traffic controller module” which would be termed the process scheduler of the operating system kernel in current terminology. The one-to-one relationship between a process and its address space is the second property that is crucial to the nature of processes.

Concurrent Pascal developed a more restricted notion of process, being that of a data structure and the sequential program that performed an operation on the data [Bri75]. However, this definition still conforms to the notion of a restricted address space that is in one-to-one correspondence with some sequential code acting on it. Mesa developed dynamic processes for concurrency and dynamic nested monitors for synchronisation [MMS79]. The interface to invoking a process was similar to invoking a procedure, but instead the process was executed concurrently with the caller.

The multiplicity of alternative program structuring devices (beyond the basic ones of repetition, conditional action, and sequencing) led to Hoare’s development of communicating sequential processes (CSP) [Hoa78]. This proposal adopted Dijkstra’s sequential processes, creation of parallelism and guarded commands [Dij75], and added inter-process communication by means of synchronous message passing. CSP is aimed directly at expressing concurrent computation, not at operating system support for multiple programs. The CSP model is basically that of a calculus for concurrency and communication. Considerable interest in the development and characterisation of models of concurrency and communication arose; notable among these are Milner’s calculus of communicating systems (CCS) [Mil80], Milner *et al*’s π calculus [MPW92], and various other process calculi [San92, Tho95]. The calculi have moved progressively towards the expression of computation itself by communication. The key to all of this work is that processes are seen to have a third major property, being the ability to communicate.

3.2.2 Issues

Throughout their history in computing, processes are considered to:

1. perform an evaluation,
2. have their own private address space or memory, and
3. communicate with other processes.

3.2.2.1 Evaluation

The process model is fundamentally concerned with isolating individual computations from each other. Irrespective of whether processes are to be used for parallel programming or operating system support, the capacity to specify a structural unit of active evaluation remains essential. The notion of evaluation as sequential execution of program statements should be read more generally to mean any standard evaluation mechanism; ranging across imperative, functional, logic, and object-oriented programming languages. The Multics system goes beyond this and argues that cases exist where “multipurpose processes” are appropriate [Org72]. In the terminology

defined in the previous chapter, this implies that the evaluation mechanism within a process may also be concurrent.

In his Turing Award lecture [Mil93] Milner characterises processes in the π calculus as capturing the concept of values, with names locating and accessing these values. Processes are active in as much as they interact with other processes through their names, and all evaluation is expressed by this interaction. This notion of evaluation is radically different from the sequential notion of evaluation represented by the λ calculus, which is based on the reduction of terms.

3.2.2.2 Sharing and protection

An essential aspect to avoiding confusion between processes is the ability to protect data from inadvertent access and/or alteration. (Data in this instance includes both executable code instructions and statically or dynamically created program data.) The simplest such mechanism is to provide every process with its own copy of all data. However, the consequence of such an action would be to forgo any possibility of sharing.

Eliminating unnecessary duplication of shareable procedure code data in operating system support for multiple user programs is clearly desirable. The Multics system's mechanisms to achieve this provide an excellent example, covering both a virtual memory system [DD68] and protection facilities [Gra68] that provided complete safety with respect to concurrent invocations of the same procedure. The mechanisms provided were sufficiently general to manage sharing of program data as well as procedure code data. The modern equivalent is with dynamically-linked shared libraries in UNIX-like operating systems.

The requirement for protection in parallel programs is somewhat different. Early investigations assumed that multiple processes would cooperate through common data structures, thus requiring mechanisms (such as locks and monitors) that would protect against other processes from destructively accessing the shared data. However, on parallel computers that do not provide shared memory it remains important to eliminate unnecessary duplication of the code of common procedures for processes executing on the same node.

Sharing of program data among processes in parallel programs is a more contentious issue. The primary problem that arises is that access to data structures becomes non-uniform for processes on the same node and processes on remote nodes. A system of access control to such shared data is then required, and some notion of data coherence must be developed. Simplifying the issues concerned, it will be assumed that the process model used for parallel computing in the context of a distributed address space does not share program data. Any sharing of either executable code or program data is transparent to processes, by virtue of some underlying operating system or middleware support layer.

3.2.2.3 Communication and naming

Hoare's work on CSP [Hoa78] was partially motivated by his belief that computers consisting of a number of self-contained processors, each with its own memory, would become prevalent. On such computers, it was no longer clear that data structures should be shared among processes. Yet it was still essential that multiple processes could coordinate and synchronise their actions as well as communicate information. Hoare believed that the communication of copies of data between processes would enable a model of parallelism that could be realised on either form of computer (monoprocessor or multiprocessor in his terminology) and that was sufficiently general to describe a wide number of alternative mechanisms for managing communication and synchronisation. The CSP model permits communication when one process is willing to output a value to a particular destination and the destination process is willing to input a value from the sender, in which case a copy of the sender's value is made in the destination.

A variety of mechanisms may be employed to realise inter-process communication. Among processes in an operating system, communication tends to operate by copying into a kernel memory space, and then copying from there into the user process's memory space. This is the low-level mechanism employed for most higher-level forms of inter-process communication in UNIX operating systems, such as pipes, sockets and streams. It is also possible to share pages of memory between processes and to map files into the memory space of a process.

In parallel programming systems, both shared variables and message passing are used for communication. The choice between the two remains a contentious area of investigation, with different algorithms being better suited to one or the other model of communication. (An example of some quantitative research comparing the two models for a range of applications and shared address space computers may be found in [NS93].)

While the CSP model does not specify the underlying mechanism that is used for communication, the fact that the value is copied and the explicit action of communication implies a notion of message passing. The model is very restrictive in that communication occurs between two processes in a synchronous fashion, although non-determinism is permitted through the alternative operation which selects one from a number of guarded communication commands to proceed. The only naming mechanism to identify communication targets in CSP was that of a process.

The notions of ports and channels were developed to allow a process to use multiple input and/or output facilities, each of which had a unique name to identify it. In some systems (for example, CCS [Mil80]), these names could not themselves be communicated to other processes. It turns out that this is a significant limiting factor in the expressivity of parallel programs since it prevents dynamic reconfiguration of communication facilities. The development of the π calculus [MPW92] explicitly addresses this restriction by allowing the communication of link names. In fact, the

only things that can be communicated are link names, and this ability, together with the rules of reduction for combining processes, provides computation facilities for expressing arbitrarily linked systems that may be dynamically reconfigured.

In the π calculus processes themselves cannot be transmitted on links; however, the links that activate processes can be transmitted, which is both more and less powerful than the ability to pass processes. (The subtleties of these distinctions are discussed on page 17 of [MPW92], but are outside the scope of this thesis.) Thomsen developed an alternative process calculus called CHOCS which does permit the transmission of processes [Tho95]. The main semantic question with process transmission is knowing what exactly is being transmitted. Milner *et al*'s understanding of transmitting a process is that it constitutes transmitting the *text* of a process; however, the definition of process that has been used earlier is not restricted to this static understanding of a process. Instead, there is a notion of dynamic evaluation of some self-contained fragment of code together with associated memory and communication facilities, not just the code fragment itself. Thus it is not at all clear how to ascribe meaning to the transmission of such an ongoing evaluation.

3.2.3 Summary

In essence, the concept of processes is intimately bound up with three things: evaluation, memory, and communication. A process performs evaluation (of some self-contained fragment of code or expression) with respect to some memory. The notion of memory encompasses the values of all the variables required for the process's evaluation, including the evaluation code itself. Communication of values is used to achieve cooperation and interaction with other processes.

This characterisation of the process model is independent of any end use of the model, although it was originally developed to enable parallel programming and multiple user program support in operating systems. It should be noted that the communication of values among processes is a stronger requirement for parallel programming than for operating systems. The precise way in which the process model is used in the design of paraML though is dependent on the end use of the language.

3.3 Safe and efficient high performance programming facilities

One major purpose in the design of paraML is to efficiently support safe high performance programming. The process model described above is generic in that it provides a useful model for various types of programming. High performance computing on distributed address space parallel computers requires support for concurrency, scalability, portability and locality. The emphasis in this section is on how the process model provides good facilities for supporting these attributes.

3.3.1 Concurrency

Inherent in the process model is the ability to support concurrent specification of distinct computation entities. Creation of a process relies on support from the underlying system to specify the computation to be performed and the communication facilities. The system may provide facilities to create new processes at any point in time with respect to an originating process (dynamic process creation). Alternatively, the system may only permit process creation at one point in time – resulting in a fixed group of processes (static process creation).

In a system permitting dynamic process creation, there is no inherent limit to the degree of concurrency which can be supported. Naturally, any actual computer system will impose some physical limits which cannot be exceeded; these limits are implementation-dependent. Creating a new process simply requires program code to be executed with respect to data (as specified by a parent process). There is no necessity for the process's execution to be reliant on any process other than the parent, and only then at creation time. However, processes may cooperate by the communication of information, provided there is some mechanism to identify the destination of the communicated information. Dynamic process creation is also useful in that programs tend not to be reliant on fixed configurations of resources. Thus there is less likelihood of programs using dynamic process creation failing due to a mismatch between the degree of concurrency in the program and the computer.

These attributes of the process model described above ensure that the first and most essential goal in supporting high performance programming on distributed address space computers – that is, the ability to express concurrency – is achieved.

3.3.2 Scalability

Scalability is a term in common use in the parallel computing community, but one which is subject to imprecise usage and difficulties in measurement. The term is used both with respect to hardware and to software.

Data parallel computers such as Thinking Machines' CM2 have been built with several tens of thousands of PEs, but the PEs being utilised are much simpler than for other parallel computers. Data parallel architectures are good for achieving scalability, at least for data parallel algorithms, but are not so effective for general programming problems. Without utilising off-the-shelf CPUs, they also have difficulty in tracking general hardware performance improvements. As a consequence of the need to track performance improvements in hardware, the majority of computers used for parallel computing today have standard RISC CPUs in the PEs. The total number of PEs in a computer typically ranges from around four or eight to several hundred. Shared memory computers are typically limited to at most 32 PEs, after which memory bus contention prevents the addition of more PEs from being effective. Without dramatic change to manufacturing technology, the expense of individual PEs means that

computers with several thousand PEs are likely to remain too expensive for the foreseeable future, other than for very specialised and well-funded agencies.

Put simply, scalability indicates the degree of efficiency in the use of increasing numbers of PEs. One motivation behind this approach is that the user of a parallel program to solve some problem wishes to know that if an n -PE parallel computer can execute the program in time t and the cost of this computer is $\$c$, then a $2n$ -PE parallel computer costing $\$2c$ will execute the program in a time less than t . Ideally, the time taken will be $t/2$, but this rarely happens in practice. However, there is clearly no point in purchasing a computer with more PEs if it fails to solve the problem faster. The exact point between $t/2$ and t at which the purchaser agrees to buy the more powerful computer is dependent on how much the ability to solve the same problem in a faster time is valued.

The software view of scalability is more complex. Scalability is intimately connected to the notion of *speedup*. Speedup is a measure of the performance improvement relative to the performance using only one PE, calculated as the inverse of the time taken multiplied by the time taken for a one-PE computer implementation. Linear speedup is where the speedup is the same as the number of PEs used. Reporting speedup as the only performance metric for parallel programs is problematic. For example, Hennessy and Patterson warn about the dangers of examining speedup on a parallel computer based on Intel 8086-based PEs when readily available sequential workstations easily outperform any such machine [HP94].

Efficiency is measured as the fraction of speedup achieved relative to linear speedup; thus achieving linear speedup yields an efficiency of 1. However, almost all problems have components which must be executed serially. Amdahl's Law [Amd67] states that as the number of processors approaches infinity, the maximum possible speedup is $1/\sigma$ where σ is the fraction of the problem which must be executed serially. The problem with this definition is that the user is unlikely to want to solve the same scale of problems on a single processor computer as on a parallel computer. In fact, it may not even be possible to determine the time taken to solve the problem on a single processor computer due to the size of the data involved.

One approach to providing a more balanced notion of speedup incorporates scaling the size of the problem to be solved as the number of PEs increases. The formulation of this notion of speedup is known as Gustafson's Law [GMB88], and relies on the time taken to perform the sequential part of the computation remaining unchanged regardless of the number of PEs and the size of the problem. With this formulation, it is possible to achieve efficiencies which approach 1 for large enough problem sizes. However, it is also unusual to find problems where the serial component of the problem does not depend on the size of the problem and the number of processors being used.

A final refinement is the notion of *isoefficiency* first developed by Kumar and others [KG91, KGG+93]. This concept defines a function by which the size of a problem must be scaled up as the number of PEs increases in order to maintain a specified constant efficiency. The isoefficiency function computes the relationship between the time taken for parallelism overheads and the total useful computation time. While this metric is the best one for characterising scalability of parallel programs, it is also the most complex. Since the metric is completely dependent on the algorithm used and the characteristics of the parallel computer on which the program is executed, it enables valid comparisons between different parallel computers for the same algorithm.

In summary, scalability can be characterised as the degree to which performance improves relative to increases in the degree of parallelism. Alternatively, it can be viewed as the degree to which problem size (which is typically dependent on the data size) must be increased to retain the same performance when the degree of parallelism is increased. Comparisons of scalability between different parallel computers need to use metrics such as isoefficiency to eliminate individual biases in the overheads of parallelism.

Having established a meaning for scalability, the next question is whether the process model helps or hinders scalability. The major issues for the process model with respect to scalability are:

- can it make use of increases in the number of PEs?
- are there any inherently serialising constraints that restrict increases in parallelism?
- do the overheads of process creation outweigh any benefits from increased parallelism?

Since the process model permits dynamic creation of processes, there is no inherent limit on the number of processes that can be created. Thus any increase in the underlying hardware can be utilised by creating more processes to be executed. Note that the computer's hardware or operating system will impose restrictions on the total number of processes which may be created, as a consequence of finite memory resources.

In the process model, there need be no global system resource which is required for process creation, and processes execute independently of others (other than where communications between processes imply synchronisation). Hence there is no inherent constraint in the model on increases in parallelism.

The overheads of process creation are dependent on a particular implementation of the process model, and may be dependent on problem size. However, the overheads for any single instance of process creation should not be affected by the total number of processes. Thus, overall process creation overheads should be dependent on the number of processes and problem size, but still finitely bounded.

Whether or not the overheads are offset by the increased parallelism will depend on the degree of parallelism in an individual application, as characterised by isoefficiency.

3.3.3 Portability

It is an oft-stated position that one of the main problems facing high performance parallel computing is that building efficient parallel programs is more complex than developing sequential programs. Wilson remarks on the lack of good software techniques for managing parallelism as a significant contributing factor to this problem [Wil95]. Hennessy and Patterson comment on the current need for detailed machine-specific knowledge to achieve efficient implementations, and this knowledge does not carry over to other architectures [HP94]. Regardless of the reasons, it thus becomes essential that once a solution has been developed, it can be reimplemented on a number of different parallel computers without having to replace large quantities of code, or worse still, redesign the entire algorithm. The procedure by which this reimplementation is carried out is known as *porting*, and if porting can take place with little or no alterations, perhaps only requiring recompilation of the program source, then the system is said to possess good *portability*. However, it is also desirable that the efficiency of the ported system remains comparable to its original implementation.

These requirements are similar to sequential computing, where a program, once developed, is expected to execute on a wide range of sequential computers requiring only minor modifications (to such things as operating system interface calls) but with similar levels of performance. One of the main reasons for the success of portability on sequential computers is that they all embody the same basic machine model (the von Neumann machine), and the components of these machines are fairly similar (CPU, cache, main memory, secondary storage and input/output devices). Portability has been more difficult to achieve across parallel computers due to the wide variation in their construction and the lack of a single programming model which performs well on different architectures for different types of problems.

The major issues for the process model with respect to portability are:

- can it be used to support parallel programming systems on different types of parallel computers with a degree of performance invariance?
- are there characteristics of the model which are likely to result in an inefficient implementation?

Operating systems for sequential computers are typically founded on the process model, so it is clearly possible to support processes on a sequential computer. Modern parallel computers are converging towards construction from a number of processing elements (PEs), where each looks like a sequential computer, although the available memory may be shared among all PEs or be disjoint. Where the memory is shared, the techniques to support the process model are well established, since the parallel computer appears basically the same as a sequential computer but with more available

PEs to execute processes simultaneously. Communication between processes occurs via the shared address space. Where the address space is distributed, each PE may support multiple processes, and communication occurs either through the shared memory on a PE, or through the inter-PE communication network. Sharing of common program code can occur for processes on the same PE, and each PE maintains a complete copy of all the executable program code. Thus the process model can be realised on different types of parallel computers, and used to support parallel programming systems.

Inefficient implementations of the process model would arise if there is some characteristic of the model which is actively limited by the underlying hardware. The main characteristics of the process model are its evaluation of separate encapsulated computations with respect to some memory, and communication between these entities. Any parallel computer with the ability to execute programs on each PE can support encapsulated computations. The main impediment to performance will be if the number of PEs is very low relative to the number of processes. In this case, the overheads in switching between processes on a PE may outweigh the benefits of having parallel computation. Another consequence of large numbers of processes relative to the number of PEs is that access to shared memory may become a bottleneck. The last impediment to performance will be if the communication medium (be it the access to shared memory or the inter-PE communication network) becomes saturated with outstanding communications, thus preventing evaluation progress within processes. This problem arises if the amount of communication relative to computation is high. All of these problems are somewhat application-specific, and are not direct consequences of the process model itself. Typically, it is possible to adjust the number of processes being created according to the number of PEs available, which tends to limit the impact of these problems.

3.3.4 Locality

The concept of locality is used in two ways:

- with respect to physical memory hierarchies (virtual memory to main memory to cache to processor registers), the smaller the latency involved in accessing a data value, the greater the locality of the data;
- with respect to software, whether the location of data values in the memory hierarchy can be directly influenced by the programmer.

Performance of programs can be increased by reduced the memory access latencies. In the case of sequential programs and shared address space computers, enormous amounts of research and development have been carried out to minimise the performance penalties arising from poor data locality during program execution. The goal of this research is to find automatic ways of moving data into a lower-latency position in the memory hierarchy before (first or repeated) use. The Stanford DASH and FLASH multiprocessor projects [LLG+92] [KOH+94], and the SUIF compiler

system [AAL95] are examples of such research. Some programming languages permit the programmer to provide direct clues as to the expected frequency of use of data (for instance, the register directive in the C programming language [KR78]) and thus the need to maximise the data locality. In the context of parallel programming, the affinity directives of COOL allow the specification of various associations between object data and task execution to maximise locality [CGH93]. Program structuring mechanisms are another implicit way of providing such clues to data use (for instance, loop count variables are frequently accessed). When computing in a distributed address space, data may be located in a completely different memory unit from that associated with the current evaluation, and subject to much greater access latencies.

The way in which data locality is promoted is important for good performance of parallel programs. The process model addresses both of the above aspects of data locality. Firstly, a process is an encapsulated evaluation with respect to some memory, so all data required as part of the computation will be physically resident within the process's memory at the time it is incorporated into the ongoing computation. This property is the result of the communication of values from process to process. However, a process may have to wait for this data to be communicated to it by another process before the data becomes local. In distributed address space computers, there is an additional complication. If some data value needs to be local to two separate processes simultaneously, a copy of the value must be resident in both places. If the data is mutable, then changes to one copy will not be seen by the other process, unless there is some mechanism for data coherence and sharing. However, as mentioned earlier, the process model being described makes the simplifying assumption that either data is not shared or any such sharing is transparent to the processes concerned.

Secondly, the location of data values is clearly exposed to the programmer in as much as it is either locally present or resident in a remote process. For the value to become local if it is currently remote, there must be a deliberate act to communicate the value. Further discussion of the impact of distinguishing local and remote data/memory access is provided in §3.6.3. The π calculus takes the concept of location one step further, by making names the sole mechanism of locating a value. Without a name, there is no mechanism to access a value. While this is an elegant and powerful way of viewing location in a concurrent system, its failure to provide any notion of distance or ownership becomes important in providing efficient high performance programming facilities in the context of distributed address spaces.

3.4 Programming model

The programming model of paraML is based on a process-oriented style of computation. Processes are good at supporting concurrency, scalability, portability and locality, which are essential to achieve high performance. The specialisation of the

process model for paraML is described with respect to its evaluation, memory and communication characteristics.

3.4.1 Evaluation

The programming model's organising unit of evaluation is the process. Each process is capable of executing some expression within an encapsulated data environment which includes some memory. The computation language within a process may be either sequential or concurrent; the precise details are not central to this design discussion although a concurrent language will allow more efficient implementations of certain problems, as discussed in §4.6. The coordination of processes is achieved through the extension operations that paraML defines, as described in the next chapter. The capacity to dynamically create multiple processes and for each to execute a unique expression provides the concurrency which is fundamental to a parallel programming system.

3.4.2 Memory

A defining feature of the programming model is that memory is not shared between processes at all. Thus processes must contain all information required for evaluation of an expression other than information communicated by other processes. The motivation for this approach is that it allows processes to be completely independent in their execution; they do not rely on the cooperation of any other process for memory access nor must they provide memory access services for other processes. The disjoint nature of process memory means that scalability is more easily achieved, since there are fewer shared resources to create barriers to performance scalability as the number of PEs in the physical machine increases.

3.4.3 Communication

Communication between processes takes place by message passing to ports. Ports are provided by and located on a process, and queue messages in order of arrival. Any process may send a message to the port, but just as the memory is disjoint, so messages in ports may only be removed by the process that created the port. The mailbox analogy is an appropriate one to characterise communications to ports. Successful delivery of a message to a port is reported to the sender, but this is no guarantee that the process will ever actually remove the message and incorporate its information into the ongoing computation. (In MPI terminology, this is like buffered mode communication, except that the buffer exists at the destination, not the source.) The motivation for this approach rather than fully synchronous message passing is that the latter binds together the progress of all processes in a program much more tightly. In particular, a sending process is unable to continue execution until the matching receive operation has been commenced. This characteristic increases the chances of inadvertent deadlock as well as possibly delaying the sending process until the receiving process calls its matching receive operation. The essence of the programming

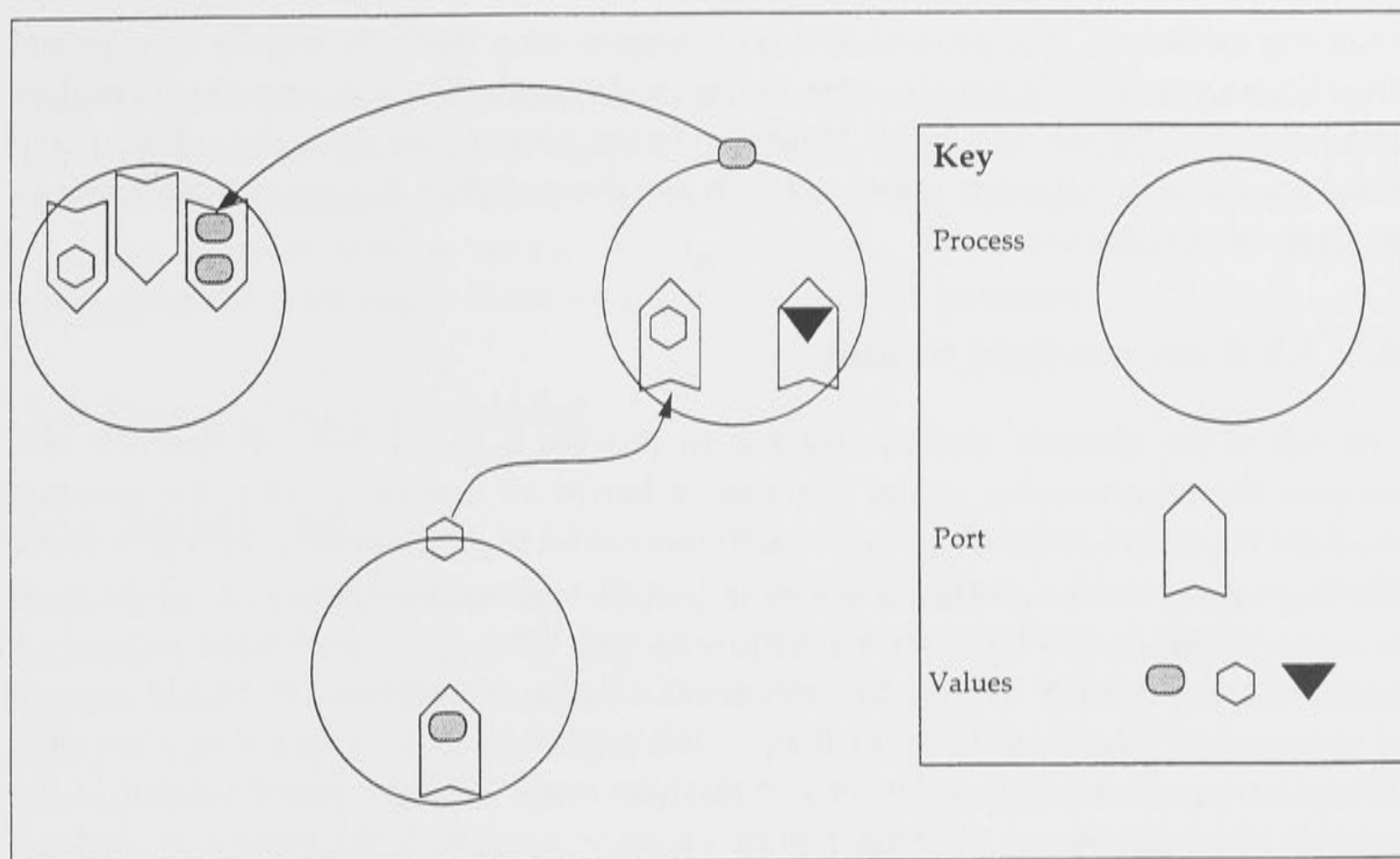


Figure 3.1 – Processes and ports in the programming model.

model is illustrated in Figure 3.1, where three processes are engaged in communicating two values.

The basic operations provided by the communication system are: sending, probing for, and receiving messages. The use of message passing to ports rather than some other form of communication (even message passing on shared channels) promotes locality by ensuring that any information required by a process must be local to its memory (in one of its ports) prior to incorporating it into the computation. The very simplistic model of communication is also easy to provide in any modern parallel computer, and thus does not pose any restriction on portability.

A degree of safety is achieved through the ability to differentiate messages by ports. Instead of a single buffer for all messages, identification of different classes of messages may be associated with different ports. It is still possible to overload use of a particular port with more than one form of information, but this is now chosen by the programmer rather than being imposed by a single message buffer approach.

Foster describes a programming model which is similar to the processes and ports model, but he uses the terms *tasks* and *channels* [Fos94]; the term *port* is also used to refer to the interface of a task to its environment, where output ports and input ports of two tasks may be connected by a channel consisting of a queue of messages. This use of terms was deliberately avoided in paraML since it leads to confusion with the channels of CML. Foster also uses the term *message passing* to refer to a programming model where processes communicate by messages to other processes, not to individuated ports within the process. However, message passing is used in this thesis to mean simply the communication mechanism that involves sending and

receiving messages. The process and port programming model is slightly less general than Foster's task and channel programming model, since there can never be more than one receiver at the end of a port. However, Foster prefers (but does not enforce) only single senders and receivers associated with any channel since this ensures determinism in communication.

3.5 Abstract machine model

The role of the abstract machine layer is to provide a model that can describe and support the programming model in terms of hardware concepts. Thus the abstract machine is a hardware analogy for the software model of programming with processes. The key concept in the abstract machine is that of a *virtual processor* (VP), where there is a one-to-one association between processes and VPs. A VP is not the same as a process however, since VPs are also responsible for the management of certain aspects of state and scheduling behaviours that are left implicit in the description of processes. The following description of the abstract machine model does not provide detail at the level of interpretation of machine operations, as is done with the functional abstract machine [Car83] for example. The abstract machine model is realised by a runtime system which executes on each node of the parallel computer.

3.5.1 Evaluation

A VP provides an evaluation service capable of executing the code of a process. In this manner, it acts like a normal computer processor. All the facilities required for evaluation, such as memory and interfaces to the communication facilities, are also provided. At most one process is executed by a VP, and new VPs can be created dynamically, disappearing when the evaluation is complete.

3.5.2 Memory

The memory of each VP is disjoint from that of all other VPs and is effectively unlimited, like the virtual memory system in an operating system such as UNIX, where each process apparently has an address space, regardless of either physical memory of the machine or the number of processes being executed. In practice, a VP's memory will be constrained by the available memory system, but as much as possible the process being executed by the VP should not be aware of this restriction.

3.5.3 Communication

Communication among VPs is provided such that any VP may send a message to any other VP. Unlike processes in the programming model, there is only a single destination for messages sent to a VP. The ordering on messages between VPs is preserved. There is no requirement for ordered broadcast facilities among VPs. The basic operations provided by the layer are to send and receive messages. Information in the messages is used to identify which port the message is destined for, and the VP

must provide queues for each distinct port that the process creates. The state of a message queue is sufficient to support the probe operation in the process layer. Additional information is carried in messages, which identifies the sender of a message, together with a unique message identity. This information is used in generating status messages about the success or failure of a request.

3.6 Physical machine model

3.6.1 Background

Prior to examining the evaluation, memory and communication traits of the physical machine model, it is useful to examine some background on parallel computer architectures. The classic computer taxonomy is that of Flynn [Fly66], which characterises computers by the number of instruction streams they are capable of processing and the number of data elements that the instruction streams can operate on simultaneously. Most modern parallel computers, such as the IBM SP2, Fujitsu AP1000+, Cray T3E and Silicon Graphics PowerChallenge, would all be classified as MIMD machines – those capable of executing multiple instruction streams and multiple data elements. The main advantage of this form of architecture is that the separate building blocks or processing elements of the computer are themselves very similar to a traditional monoprocessor, and thus are subject to economies of scale in production.

Flynn's taxonomy does not consider how memory is accessed by the instruction streams. In practice, this turns out to be one of the fundamental distinguishing characteristics of parallel architectures. As discussed in Chapter 2, parallel programming systems may embody either a shared address space or a distributed address space. Similarly, parallel architectures may provide either a single shared memory that is accessed by all PEs or many memory units, distributed across the machine, that are accessed by the PEs. The SGI PowerChallenge is an example of a shared memory MIMD machine (these are also known as *multiprocessors*), while the Cray T3E is an example of a distributed memory MIMD machine (these are also known as *multicomputers*).

The most common approach with distributed memory computers is for each memory to be *disjoint* from all others, and accessible by only a single PE. Some parallel computers, such as the Silicon Graphics Origin2000 series and the Kendall Square Research KSR-1, provide hardware support for access to distributed memory modules from other PEs whilst retaining memory coherency and cache coherency. The reason for such an approach is that distributed memory is a scalable technology, but the shared address space retains simplicity for programming without the overheads of a software DSM. Such memory interconnection networks are analogous to the PE interconnection network that is provided to allow communication and synchronisation between PEs. The current generation of distributed memory MIMD computers (for example, the Cray T3E and the Fujitsu AP1000+) provide hardware-supported

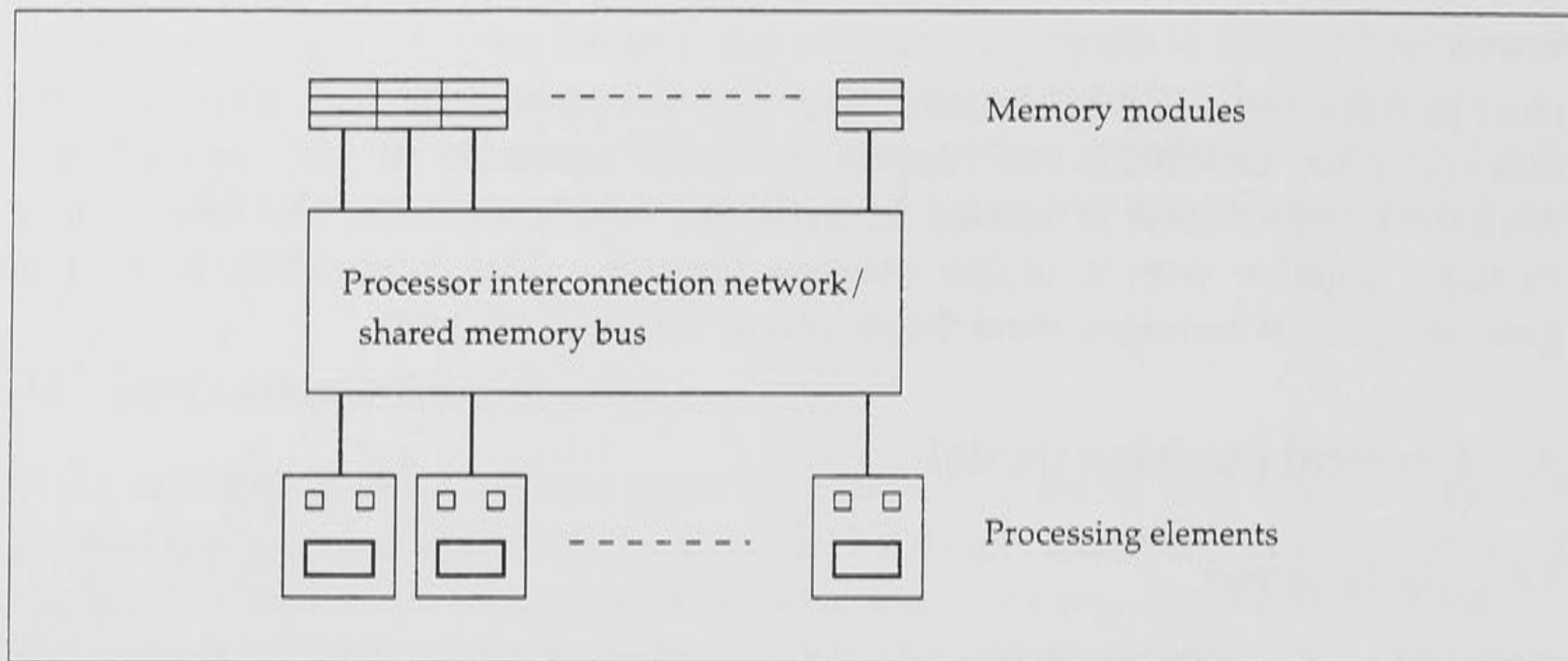


Figure 3.2 – Shared memory MIMD computers.

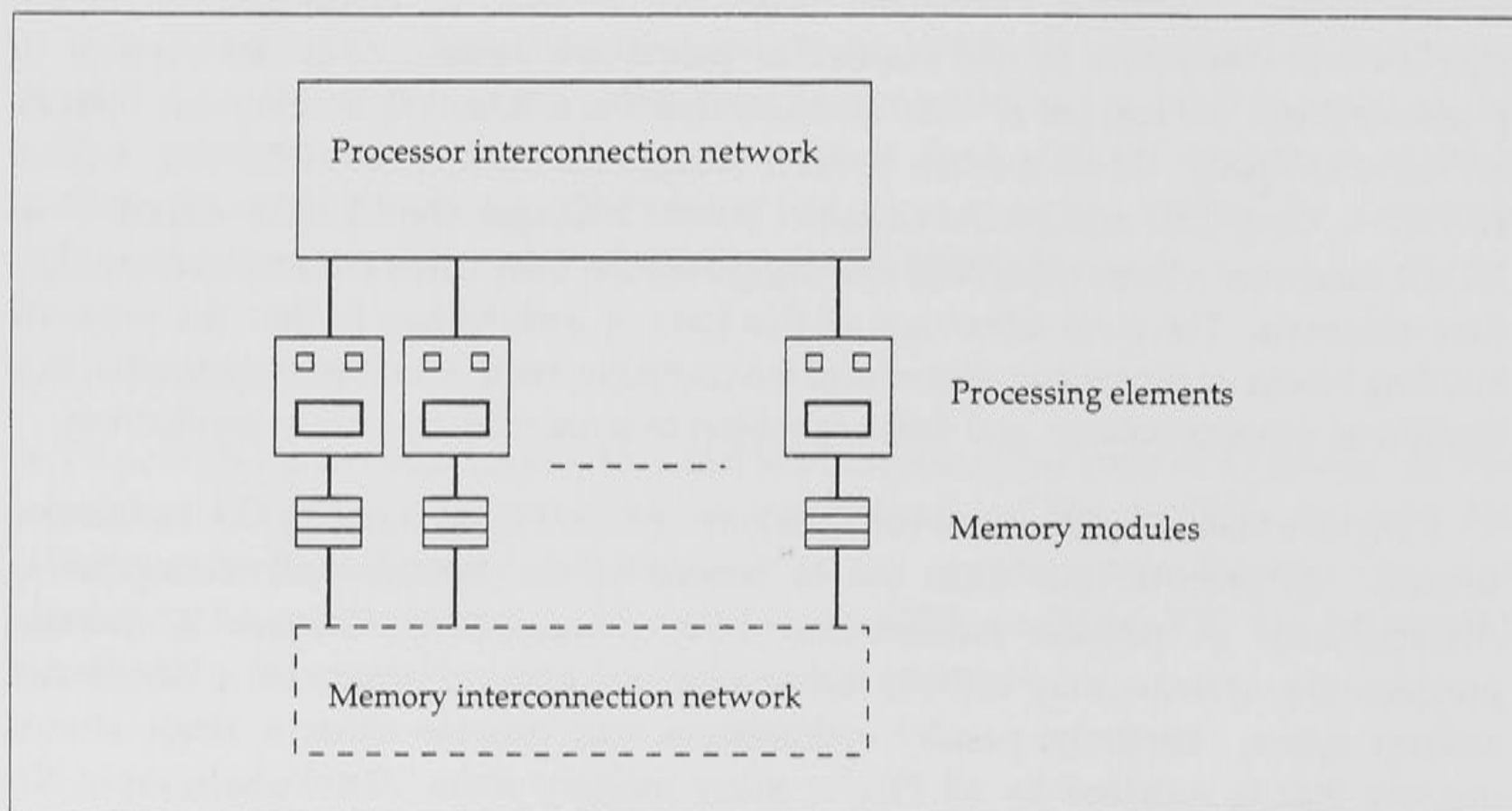


Figure 3.3 – Distributed memory MIMD computers.

operations that allow fetching and setting of memory locations in remote memory modules without involvement of the remote PE's processor. Such operations though are still performed through the PE interconnection network. The connection to memory in shared memory MIMD machines is shown in Figure 3.2, and for distributed memory MIMD machines in Figure 3.3.

The main benefit of distributed memory MIMD machines is that as the number of PEs grows, the technology for managing inter-PE communication is more scalable than with shared memory MIMD machines. Distributed memory machines need a faster switch for message routing, but shared memory machines require a faster memory bus which is shared among all PEs. The other problem is that the performance of the shared memory bus must increase linearly to retain performance but the switching technology does not have to increase at the same rate in order to retain the same

overall performance. Contention across the memory bus is higher since all non-cache memory accesses have to be mediated via the bus, while only communication requests require the communication network in a distributed memory machine. The overall result of these factors is that performance is more easily scalable on distributed memory machines.

The main disadvantage of distributed memory MIMD machines is that the time to access memory values (which may reside in a remote PE and require the remote PE's involvement) is non-uniform. This characteristic is common to all modern computers due to caches, virtual memory, or shared memory buses on multiprocessors. However, the degree of non-uniformity is more significant with multicomputers, and thus gives rise to the acronym NUMA, for non-uniform memory access; the converse is UMA, for uniform memory access. Most NUMA architectures require the programmer to be aware of the distinction between local and remote memory value access since the actual access operations are typically different. A shared memory program can ignore this level of detail since the hardware provides a single method of access to all values; awareness of the non-uniformity characteristics becomes significant usually only when performance tuning is required. The Silicon Graphics Origin2000 series computers retain the simplicity of a shared address space kept coherent in hardware, while the physical memory is distributed and subject to NUMA latencies. The degree of non-uniformity is kept smaller than other multicomputers by virtue of very high memory-to-memory bandwidth and high performance switching technology.

3.6.2 Physical machine model, operating system and runtime system

The physical machine model that paraML has been designed for is the *multicomputer*. Foster's distinction between multicomputers and distributed memory MIMD computers is adopted [Fos94]. Multicomputers are a collection of PEs consisting of von Neumann computers, which communicate through an interconnection network. They differ from distributed memory MIMD machines by the simplifying assumption that the time taken to send a message between two PEs is independent of the relative location in the network and other network communications, though the size of the message remains a factor. In this section, the key components of the multicomputer model are detailed: its evaluation, memory, and communication characteristics. The roles of the multicomputer operating system and communication library are also discussed.

The primary reason that the multicomputer model was chosen as the target architecture is that it is sufficiently general to be representative of a broad class of current high performance computational systems. These include distributed memory MIMD machines, clusters of multiprocessors, and workstations connected by high-speed interconnection networks. An alternative for characterising such classes of machines is the LogP model [CKP+93]; this model places greater emphasis on the cost metrics for communication. The canonical form of a multicomputer is illustrated in

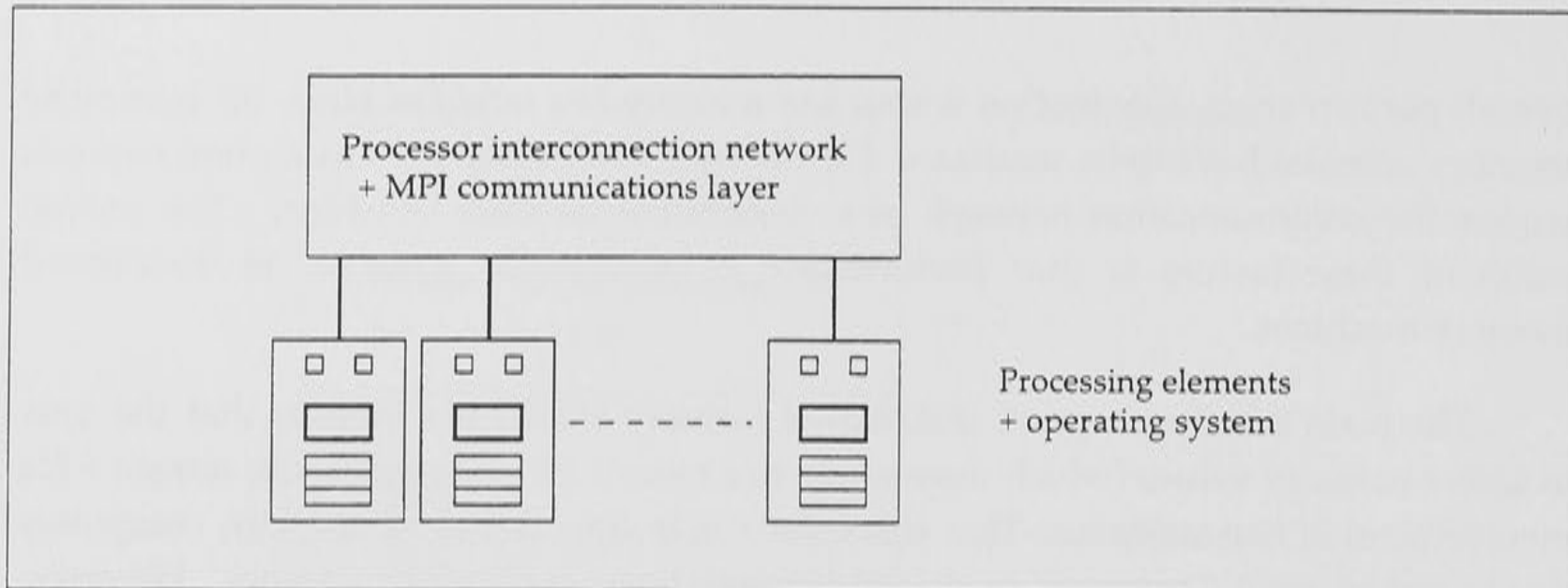


Figure 3.4 – The multicomputer model.

Figure 3.4, which shows that unlike the distributed memory MIMD machine, each memory module is contained within the PE. The same diagram represents equally well the abstract machine model of VPs which may communicate via an interconnection network.

3.6.2.1 Evaluation

The main limiting factor of the multicomputer as a general model for parallel programming is that the number of processing elements (PEs) is fixed for any particular machine. This restriction poses a serious problem for portability of programs that use either dynamic process creation or a fixed number of processes that differs from the number of PEs. The standard technique used for programming multicomputers is to assume a Single Program Multiple Data (SPMD) approach which executes the same program on each PE; effectively the program is partitioned relative to the number of PEs available. This approach is useful for adapting programs that require fixed numbers of processes but does not address those which are best modelled with dynamic process creation.

Evaluation services are provided by the processor(s) of the multicomputer's PEs. Each processor is capable of performing evaluation, accessing local memory, and initiating communication requests. In other words, the PEs are standard von Neumann-style computational devices. The multicomputer operating system provides facilities to allow the execution of at least one user process, and also supports file I/O operations. The filesystem may be a single global filesystem common to all PEs, a filesystem per PE, or a parallel filesystem that supports either or both of the two previous modes of access. There is no requirement for the operating system to provide support for dynamic process creation during the course of program execution. Minimally of course it must provide facilities to initiate a process on each PE of the computer; typically this will be a single program executable file.

The finite number of PEs available in a multicomputer must be capable of supporting a much larger and unbounded number of VPs which constitute the abstract machine. Each physical PE supports zero or more VPs, and VPs can be created

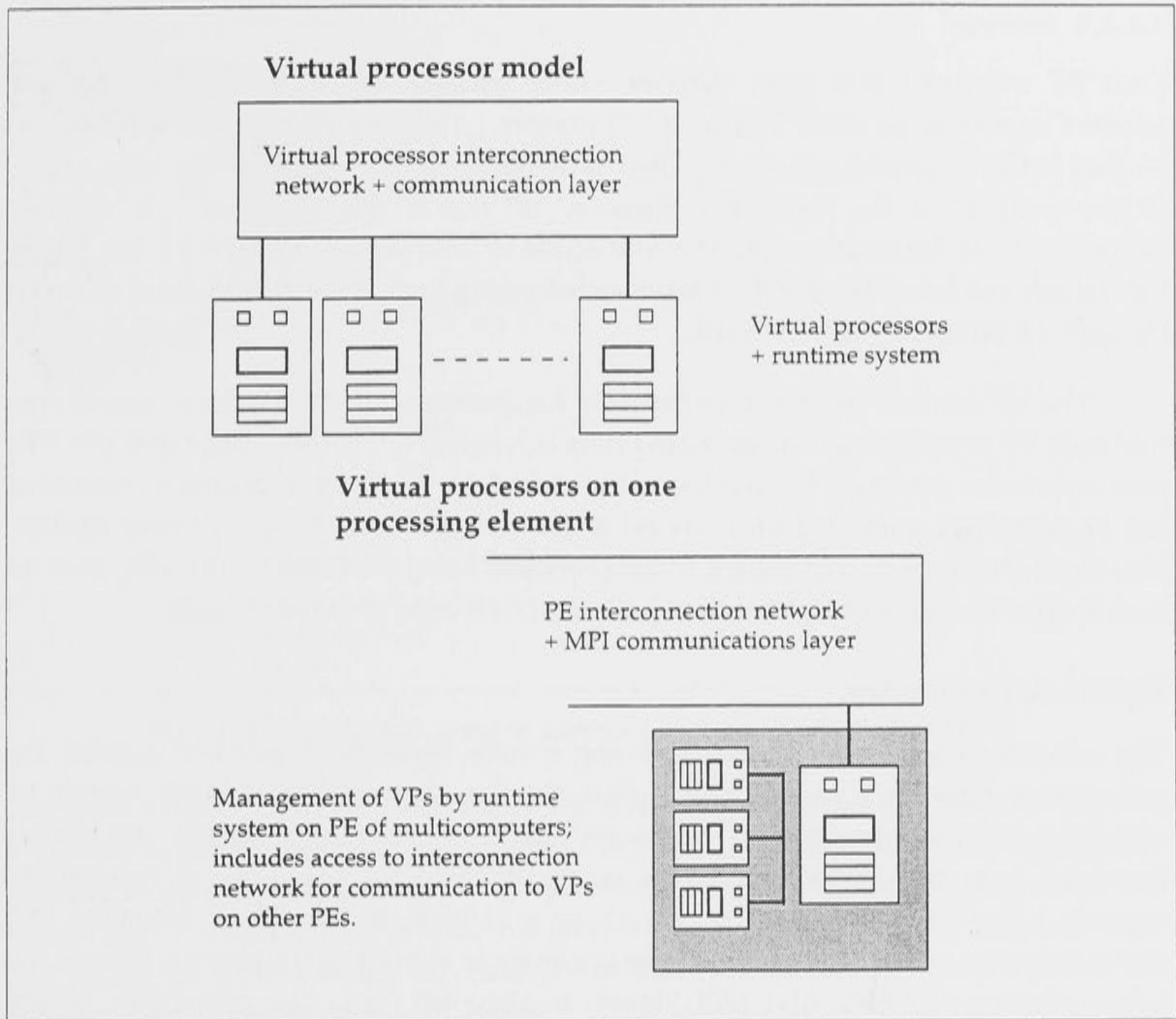


Figure 3.5 – Virtual processor model.

dynamically. A VP is very similar to a PE in that it performs evaluation of some program, has its own local memory, and can communicate with other VPs in the machine. The general construction of the abstract machine is shown in Figure 3.5, together with an expanded diagram of a PE supporting multiple VPs.

In many ways, the concept of a VP is analogous to that of a UNIX process: each has a protected address space and access to physical resources of the machine, as managed by the underlying operating system. On a sequential computer, the operating system kernel is responsible for providing a process management service to ensure that each process is given time for execution by the processor. On a multicomputer, a runtime system is responsible for ensuring that each VP is given time to execute its program by the PE's CPU and to maintain the illusion that the machine consists of an arbitrary number of VPs that may communicate through an interconnection network. The runtime system must obviously be concurrent in order to provide these evaluation and communication services. It must also be capable of creating new VPs. Each VP has its own system-wide unique identifier. Thus the restriction that each PE may only be able to execute one program is solved by making that program the runtime system for VPs.

3.6.2.2 Memory

Each PE incorporates a local memory whose address space is disjoint from the address spaces of all other PEs in the computer. The local memory is sufficient to execute entire sequential programs. There is no requirement that other PEs have access to the contents of the local PE's memory, but nor is this excluded. A defining characteristic of the multicomputer is that access to local memory by the PE has higher bandwidth and lower latency than memory belonging to a remote PE which is accessed through the communication network.

The VP runtime system is responsible for partitioning the available memory so that each VP is protected from accessing data belonging to another. Since multiple VPs may execute on the single PE, sharing of the code data of common program operations can be permitted, provided there are no problems with inconsistent memory update. This form of sharing is transparent to the processes being executed by the VPs, since to each it appears as if a completely unique copy of all code data is available.

3.6.2.3 Communication

The communication network and operating system layer must provide support for connections between arbitrary PEs. That is, the communication layer cannot be restricted to nearest-neighbour communications (as in, for example, transputer networks from the 1980's). There is no requirement for the physical network to support broadcast operations among all or a subset of the PEs. However, the communication software layer should be adequate to support an implementation of the MPI definition [MPI94]. The MPI library is also the basis for portability among different parallel computer systems with respect to the communication requirements. It is up to the MPI implementation to provide efficient inter-PE communications at the physical machine layer.

Communication among VPs is more complex than the communication layer among PEs. The main reason for this is that the MPI standard makes no provision for dynamic process creation and communication among such processes; in other words, the number of processes and the system-wide communication group or *communicator* (known as `MPI_COMM_WORLD`) is fixed at the start of the program. This communication layer is used for the runtime system executing on each PE, not for individual processes being executed by VPs. Thus the runtime system must also deliver messages bound for particular processes to the corresponding VP to queue appropriately. Depending on the location of two communicating VPs, messages may need to be transmitted between PEs over the PE interconnection network or simply copied between VP memories residing on the same PE.

3.6.3 Sharing

The main potential criticism of the programming model is its elimination of any form of sharing of memory. The lack of sharing is a deliberate choice made to achieve the goals

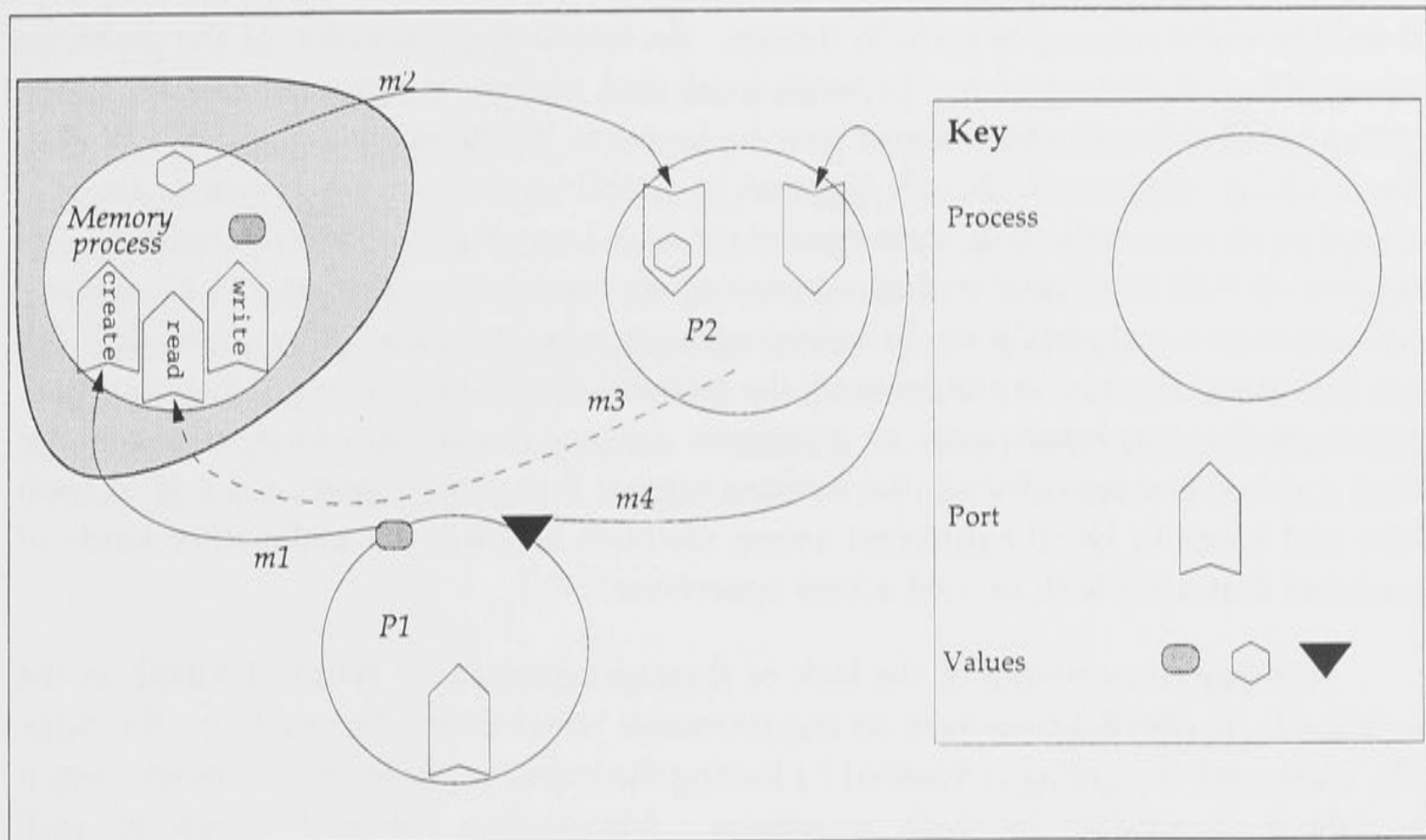


Figure 3.6 – Simple sharing of memory in the programming model.

discussed earlier. However, the issue of loss of sharing merits further discussion on its implications for programming; this is found in §3.7.1. Simplistically, sharing could be encoded in the model by having a single process manage any memory to be shared, and to provide a communication service that creates a new shared memory value and reads and writes the current value. An example of this type of sharing constructed in the programming model is shown in Figure 3.6. A single process (labelled the *memory process*) holds the memory for two types of values, and is about to receive a *create* request (the message *m1*) for a third from the process labelled *P1*. The process to the right of the diagram (labelled *P2*) has previously sent a *read* request (the message labelled *m3*), and is being answered by a copy of the current value (by the message marked *m2*). Normal message passing between the ports can still take place, as also shown in the diagram with the transmission of the black triangle value in message *m4*. However these *create*, *read* and *write* operations remain distinct from local memory access operations.

3.7 Implications for programming

The programming model described has various implications for the style of programming that may be used. Two issues in particular deserve some attention: how the loss of automatic sharing affects programs and the “plumbing” required to connect processes together with ports.

3.7.1 Loss of sharing

One consequence of not providing sharing is that local and remote value access operations are distinguished. The issue of whether or not local and remote operations

should be differentiated is open to debate. An interesting discussion of the problems inherent in not distinguishing between local and remote memory operations in the context of distributed object stores may be found in Waldo *et al*'s report [WWW+94]. The authors' arguments, though targeted at distributed computer systems, are still relevant with respect to their treatment of memory latency and access operations. The essence of their argument is that attempting to provide a unified model of memory access imposes cost penalties when the programmer knows the memory access is local. The penalties can only be eliminated if the memory access operations are distinguished. Although the implementation of programs where remote and local accesses are distinguished is more difficult, the authors suggest that programmers are able to learn how and when to use the different access methods and will not make other kinds of mistakes that arise with unified access operations.

Another consequence of the lack of sharing, criticised by Wilson [Wil95], is the overheads of copying messages among processes when their VPs reside on the same PE. However, if copying is avoided by having the message passed by reference, then it is subject to update by both processes. Maintaining coherent access to such information then incurs the overhead of a coherence protocol on all memory accesses where the memory has either arrived or been sent as a message, and increased compiler overhead to determine whether the coherence overheads can be avoided. In many ways, the tradeoffs involved are similar to those discussed in the preceding paragraph.

3.7.2 Plumbing

Wilson describes the quantity of connections between processes, which he refers to as "plumbing", as the single biggest drawback of using message passing for communication [Wil95]. In practice, this problem starts to arise when the number of either processes or ports becomes large and ensuring that processes communicate correctly with the right destination ports becomes problematic. There is no simple solution to this problem, but there are a number of advantages at the language level in paraML for minimising the difficulties, as discussed in §4.4.7. At the programming model level, the ability to dynamically create processes rather than being restricted to a fixed set of processes from startup goes some way to modularising the communication requirements of any individual process. Foster argues that the nature of processes as encapsulating data and operations on that data, and ports providing interfaces, is modular and advantageous [Fos94]. He likens the data encapsulation and interfaces to the object-oriented programming paradigm. A mechanism to manage some of the complexity of the connections is through the use of algorithmic skeletons, as discussed in Chapter 8.

3.8 Summary

This chapter has presented the foundations of the models which underlie the paraML language design. The programming model of self-contained evaluation environments represented by processes, and ports for message passing among processes, is designed

to map efficiently onto the abstract machine model, which in turn must map efficiently onto the physical machine model. There is a strong coupling between the programming model and abstract machine layer – the programming model is essentially the software description of the abstract hardware. The programming model uses processes to support concurrency, scalability, portability and locality in the context of efficient high performance programming for distributed address space computers. The physical machine model is the multicomputer, which is representative of a broad class of current and future high performance computers. Realising the process-oriented programming model is done by extensions to ML as discussed in the next chapter.

4. Design

4.1 Overview

The programming model described for paraML in the previous chapter develops an abstract view of process-oriented high performance programming facilities. This view is made concrete by the syntax and semantics of the operations for coordinating process-oriented constructs in paraML. This chapter starts by characterising the design concerns which these operations attempt to address and then introduces the core extensions. Alternative operations that may be derived from these core extensions are discussed, followed by a description of some useful operations to interrogate the multicomputer attributes. A discussion of the role played by concurrency within processes is included, along with a brief critique of the design.

4.2 Design concerns

The process-oriented programming model addressed issues of concurrency, scalability, portability and locality. These properties primarily facilitate efficient high performance programming. The other major aspect of the developments proposed in this thesis is safe high performance computing. The following concerns which address safety have influenced the design of the extensions. These concerns are:

- Error prevention.
- Simplicity and minimalism in language extensions.
- Type-safe communication during program execution.
- Formal modelling of the language.
- Modularity of program components to permit composition and clean interfaces to their surrounding system context.

Additional issues that require consideration, but which are not so directly tied to safety, include:

- Granularity of mapping program components to the machine model.
- Non-determinism in program execution.
- Garbage collection.
- Compiler alterations.
- Ability to define different programming abstractions.

4.2.1 Error prevention

The prevention of errors is greatly to be desired in all programming languages, and is particularly important in facilitating safe high performance programming. The costs of fixing errors once a program is working are considerably higher than those of fixing them at the design or implementation stage. Characteristics of the ML language have long been considered as some of the most useful in preventing errors [Mac92, App93]. These characteristics include the static type system, the exception handling mechanisms, automatic memory management and the module system. The design of extensions for paraML seeks to prevent any of these characteristics being lost, within the limits imposed by process-oriented computation.

4.2.2 Simplicity and minimalism

The benefits of achieving simplicity and minimalism in the design of the language extensions are manifest. Simple operations are understood more readily and there is a greater likelihood that they may also be implemented simply and efficiently. Providing a minimal set of operations for process-oriented programming is akin to axiomatising a logic or defining an abstract datatype. The more minimal the set and the simpler the operations, the easier it is to formally model the extensions. Whether or not the chosen set of operations is actually useful is observable from the degree of simplicity with which they can be used to define alternative programming abstractions. Unlike the earlier version of paraML, the current design attempts to keep the set of primitives to a bare minimum and to restrict the action of any operation to interactions between at most two processes.

4.2.3 Type-safe communication

The many benefits of strong typing in program construction are well known. In the context of parallel computing, where programs are considerably more difficult to debug and the tools for debugging are also less adequate, the desirability of eliminating as many forms of error as possible before program execution commences is even greater. Communication by message passing is the only means of interaction among executing processes, and thus it is crucial to ensure that such interaction is carried out without errors that can be eliminated through type checking. Since ML is a statically-typed language, it is highly desirable that communication to ports should also be statically type checked. Communication will be safer through the use of typed ports just as programming is safer with strong typing.

4.2.4 Formal modelling

There are a number of advantages that can arise from the ability to formally model a language. These include: a precise description of the semantic objects in the language and how evaluation of programs combines these semantic objects; the ability to construct checkable proofs about the outcome of particular programs; and a declarative specification of how any implementation of the language must behave.

Since Standard ML has a formal definition [MTH90], extensions of the language need to provide some justification that the extension operations do not discredit the existing definition. There are a number of problematic areas in any formal definition that builds extensions on ML, but these are left to the discussion of the theoretical modelling of paraML in Part III. Formal definitions also provide a well-founded description of implementation requirements for the extensions. The implementation is more likely to be sound as a result.

4.2.5 Modularity

Modularity is another desirable attribute of any programming language. ML has a well-developed module system that goes a long way towards simplifying this goal with respect to any extension of the language. The principal requirement with paraML is that the extension operations provide a straightforward mechanism to interact with some program component (constructed with a process or processes), and that these components can be composed together in a general but safe manner. Robust and reusable programming is enhanced through modular program construction, as independent components may be tested in isolation from the remainder of the program.

4.2.6 Other issues

4.2.6.1 Granularity

It is obvious that the design of the core operations must not hinder the mapping of processes to the machine model. Similarly, the granularity with which this mapping is performed should to some extent be embodied in the operations as otherwise the efficiency of the resulting program is likely to be poor.

4.2.6.2 Non-determinism

Most parallel systems accept non-determinism in the order of evaluation of different parts of a program. It has been shown by Chandy and Foster [CF93] that parallel programs using message passing can be deterministic provided certain restrictions are made on the use of the communication primitives. However, non-determinism need not be seen as an unpalatable characteristic depending on the algorithm being employed, and thus the design of paraML does not attempt to mandate that all communication be deterministic.

4.2.6.3 Garbage collection

Automatic memory management is an integral part of ML. The major part of such management is to automatically create memory for storing values as required, and then to garbage collect the memory once the values can no longer be accessed. High performance programming systems for parallel computers with a shared address space model require garbage collection algorithms to be implemented so that when a value is common to more than one process/thread, its memory is not garbage collected until all

references to the value are extinct. The typical solutions require either a “stop the world” approach where all evaluations are halted and garbage collection is performed globally, or an independent approach where a thread acts as a garbage collector autonomously from the other evaluations. The approach taken with paraML, enabled by the process model’s encapsulated memory attribute, is for each process to perform its own independent garbage collection. All the memory (other than some limited control information) associated with a process may be garbage collected once the process has completed execution. The solution is not quite as opportunistic as that achieved for CML [Rep92], which is able to garbage collect a thread (even if it is still executing) once it can be determined that it cannot interact with any other threads or perform input/output operations. The solution for paraML avoids the problems associated with global garbage collection in a distributed system.

4.2.6.4 Compiler alterations

Alterations to compilers are, wherever possible, to be avoided when a language is designed as an extension to an existing one. In the case of Standard ML, the compilers whose source code is available for modification are the subject of active research projects in their own right, and frequently undergo changes and improvements. Thus extensions to the language should be designed so that these compiler changes have minimal effect on the implementation of the extensions; the simplest way to achieve this is by implementing the extensions in ML itself. An extended discussion of this issue is included in §9.4.

4.2.6.5 Defining abstractions

It is difficult for an instance of any one programming model to provide all things to all potential users of the language. The goal of keeping the set of operations as small as possible contradicts any attempt to provide a large set of basic operations with different, possibly conflicting, functionality. The crucial requirement of the operations chosen is that they are useful for implementing alternative programming abstractions. Reppy’s work with CML provides an excellent example of how this may be achieved with concurrency. His choice of primitives is motivated by providing a higher-order view of concurrency. In paraML, the primitives are chosen with more concern for the safety and efficiency of their use for high performance programming in the context of distributed address spaces. It is important that alternative abstractions can be built from the paraML primitives and that their efficiency is not significantly worse than if these abstractions had been provided as primitives. A significant reason for building alternative abstractions from the existing primitives is that the formal meaning of these derived operations can then be given by their translation into the core primitives. If an abstraction is required to be more efficient than its implementation in terms of the core primitives, it is feasible to provide an optimised implementation.

In general, the design of paraML attempts not to mandate particular styles of high performance programming. It embodies a process-oriented programming model that maps easily onto multicomputers, but does not require that the software view of

programming be carried out in this manner at all levels. The ability to define alternative abstractions and the development of modules to embody alternative programming paradigms is important.

4.3 Core extensions

The core extensions can be split into two main groups: those required to capture the evaluation and memory characteristics as embodied by processes in the programming model and those required to perform communication between processes as embodied by message passing between ports in the programming model.

4.3.1 Processes

The first requirement in supporting the programming model is to capture the notion of processes. Processes provide services to evaluate an expression and to create new communication facilities. In particular, there must be some way to uniquely identify a process, create a new process, get it to evaluate an expression, and to obtain the process identity itself.

4.3.1.1 Process identification

A paraML system consists of a collection of processes. Initially this collection numbers only one. Processes are identified and referred to by name. Hence a new type of object is required for process names. This type is²:

```
type ProcessName
```

Values of type `ProcessName` are of course first-class objects (being objects that are denotable within the language; thus they may be returned as the result of function evaluation or incorporated into data structures). The use of the `Name` suffix is deliberate in that it emphasises that it is the process identifier which is a first-class object, not the process itself.

4.3.1.2 Creating a new process

Processes are created explicitly by the programmer in a paraML program. The choice of explicit process invocation rather than implicit invocation is so that the programmer is directly involved in the mapping of parallelism onto the machine, which may then be done at an appropriate level of granularity for efficient performance. Even the experiences reported with MultiLisp [Hal85] and QLisp [GMc84, GG88] illustrate that explicit invocation can yield far too much parallelism when the invocation mechanisms lend themselves to fine-grain parallelism. Explicit invocation also requires the user to be involved in managing the structure of a program at a textual level to identify

² All text written in the following font is a piece of ML code.

appropriate ways of partitioning a problem that aids understanding of the component tasks.

A new process is created by the `process` operation, which has the following type definition:

```
val process : unit -> ProcessName
```

The operational result of the `process` primitive is for a new process to be created (executing on a new virtual processor), which is identified by the process name value returned to the evaluation of the calling process. The operation requires no information, which is why it takes the null value as argument (often referred to as the unit value and written in ML as `()`; slightly confusingly this value has type `unit`). Note that the new process does not execute any expression, nor does it have any pre-defined communication facilities. The separation of these two activities (expression evaluation and provision of communication facilities) led to the decision for neither activity to be incorporated into the actual creation of a new process. Such a separation also aids in keeping every operation as semantically simple as possible. In object terminology, a process is an object which provides two methods: one for evaluating expressions and one for creating communication ports. The separation also arose out of experiences with earlier versions of `paraML` which incorporated both these activities into the actual creation of a process, and led to unnecessary confusion over the nature of creation of communication facilities.

4.3.1.3 Expression evaluation by a process

Naturally a process must be capable of evaluating some expression if it is to be of use. The first process in a `paraML` program is effectively evaluating the expression that is the program. Any newly created process may be requested to evaluate an expression by the following command:

```
val execute : ProcessName * (unit -> unit) -> unit
```

As can be seen from the type of the `execute` operation, the argument is a pair consisting of the name of the process and a thunk expression (a function taking a null argument). The null value is returned as a result of the application of the `execute` operation. An additional type constraint is that the thunk must also produce a `unit`-typed result. This restriction arises so that the user is made aware that a process is not like a function. Again, experiences with earlier versions of `paraML` where processes did have results which could be examined by other processes led to the present formulation. It now seems clear that processes should only communicate information by means of the communication primitives, and that a result of a process being available for inspection is an implicit form of communication. The developers of the `PICT` language illustrate a derived operation which allows the final value of a process to be reported as the "result" of that process [PRT93]. A similar derived function can be implemented using the `paraML` operations, as discussed in §4.4.4.

The type restriction also eliminates the semantic question of what happens to the value produced by a process execution – since the value is constrained to be of type `unit`, it can be thrown away since it carries no information. (In fact, the value is either `unit` or an unhandled exception. Again, if the process which initiated the `execute` operation wishes to know whether the expression was correctly evaluated or not, a final result should be reported using the communication facilities prior to yielding the `unit` value.) At most one thunk expression will be evaluated by a process. Exceptions are reported if the process is already evaluating (`ProcessExecuting`) or has completed evaluation of some other thunk (`NoSuchProcess`).

Operationally, the thunk expression is sent to the process together with any environment meanings for the free variables of the expression – in other words, a deep copy. Reference variables are (recursively) copied together with the value(s) they refer to. On receipt of this thunk closure, the local environment of the process is conceptually merged with that of the thunk (possibly resulting in the creation of some new memory if there are reference variables in the closure), and then the thunk is applied to a `unit` value. Once evaluation is complete (if the expression evaluation does not diverge), the entire memory used by the process may be garbage collected. The runtime system on the PE need only keep knowledge about the identity of the process and any of the communication facilities created by the process so that it may report errors to other processes attempting to send messages or perform some interaction with the now defunct process.

4.3.1.4 Determining a process's name

During the course of evaluation, it is sometimes convenient for a process to determine its own identity. This facility is useful if the process wishes to create some new communication facilities that reside on itself for instance. The operation has a type defined as:

```
val self_id : unit -> ProcessName
```

It would be feasible for a created process to have its own name included as part of the environment of free variables transmitted with the thunk expression to be evaluated or sent via the communication facilities. For example:

```
let val proc_name = process ()
    fun to_exec () = (... proc_name ... ; ())
in execute (proc_name, to_exec)
end
```

Such forms of name transmission are non-intuitive and arduous. Most importantly, it is singularly impossible for the first process in the system, which is not created by the `execute` operation. Due to the nature of port creation, this would have the effect either of making port creation impossible on the original process or of requiring an operation which creates a port on the calling process. The former is clearly undesirable, as is the latter, the reasons for which are explained in §4.3.2.2.

4.3.2 Communication

The essence of communication under the programming model of processes and ports is message passing. The primitives must define what a port is, how to send messages to a port and how to get messages from it. ParaML also defines an operation to test whether a port is empty or not.

4.3.2.1 Port identity

There are no predefined ports in a paraML program; all ports are created dynamically during the course of evaluation. Just as processes have identities, so too must ports. These names uniquely identify a port throughout the entire system of processes. Port names require the definition of a new type, parameterised by another type indicating what messages can be communicated to it.

```
type '1a PortName
```

The notation '1a indicates that the type parameter is *weakly polymorphic*. In brief, this means that before use, the type variable must be instantiated to a *monotype*. Monotypes are those which are either base types (like `int`, `bool`, etc.) or constructed from these using the function type constructor (`->`), union type constructor (`*`), or other datatype constructors. This restriction on the type of `PortName` values prevents conflicting instantiations of a polymorphic type from being accepted by the type checking algorithm, which would then allow processes to send values of different types (say integers and strings) to the same port. Clearly, this would contravene the goal of achieving type-safe communications between processes.

4.3.2.2 Port creation

Ports are created by a request to a named process, in much the same manner as the `execute` operation. The result of the operation is a new port name which identifies the port that has just been created. If the process on which the port is to be created has already finished execution of a `thunk` then the `port` operation fails, reporting an exception (`NoSuchProcess`). The type definition of `port` is as follows:

```
val port : ProcessName -> '1a PortName
```

Typically the new port will be bound to a variable and the type of the port name will be instantiated to a monotype. For example, a port to accept integer data might be created as follows:

```
let val procn = process ()
    val data_port : int PortName = port procn
    fun to_exec () = (... data_port ...; ())
in
  execute (procn, to_exec);
  ... data_port ...
end
```

As discussed above, ports are owned by particular processes – in the example above, by the process known as `procn`, not by the process that has requested the port to be created. The port is a queue, initially empty, of messages – in the example above, these messages can only be integers. Any process that knows the name of the port may send messages to it, but only the owning process may remove messages from the queue.

There are two issues that are worth discussing. The first is whether shared channels, accessible for sending and receiving by any process, would have been more appropriate. The second is whether the port creation mechanism should have been permitted only in the calling process and not on arbitrary processes within the system.

Shared channels, while appropriate within a concurrent system, seem singularly inappropriate in a distributed address space context. The problem is that data locality with shared channels is harder to achieve and also difficult to implement efficiently. For instance, it is not clear on which physical PE a channel should reside: the creating process's PE, the PE of the process that commonly receives from the channel, or on a completely different PE altogether? Similarly, when constructs such as the choice mechanism (permitting a process to select from a number of alternative communication possibilities) are required, efficient (or even correct) implementation of such constructs is problematic. Kieburtz and Silberschatz [KS79] discuss the difficulties that arise, while Bornat [Bor86] provides a correct implementation for Occam. In general, the choice of ports attached to processes appears to improve the goal of locality compared to shared channels.

Had a port creation mechanism been chosen such that ports were only ever created on the calling process, it would have meant a simpler formulation of the parallel evaluation rules for paraML (given in Chapter 6). However, the operational disadvantages of such an approach outweigh the advantages. While any paraML program can be rewritten so that creation of ports is only permitted on the calling process, it introduces the need for many more ports, used only for receiving the names of ports created by and on remote processes. (Such ports are used frequently anyway for some of the derived operations discussed in §4.4.) The principal reservation with this approach is that it leads to a lack of textual clarity in programs. It also requires a newly created process to spend some initial time sending the names of its ports to other processes who wish to communicate. The following example illustrates the problem, using the same example given previously (and the `send` and `recv` operations that are introduced in the next two sections). An operation `self_port` is used to indicate that the port is created on the calling process; this operation can be constructed trivially as: `self_port = port (self_id ())`.


```

let val procn = process ()
    val data_portname:int PortName PortName = self_port ()
    fun to_exec () =
        let val my_data_port:int PortName = self_port ()
        in
            send (data_portname, my_data_port);
            ... my_data_port ...; ()
        end
in
    execute (procn, to_exec);
    let val data_port = recv data_portname
    in
        ... data_port ...
    end
end

```

This example should clearly illustrate the code explosion that occurs with such an approach, particularly when considering multiple communication ports. The simplicity achieved with the preferred approach is partially due to the inclusion of port names in the closure constructed by the `execute` operation. Any such names that occur free in the thunk to be executed must have been subject to a binding operation in the surrounding lexical scopes, and thus are given meaning in the closure. In the example above, the variable `data_portname` occurs free in the definition of `to_exec`, but is `let`-bound in the previous statement. Effectively, this means that during the execution of a thunk by a process, any pre-existing values (including names of ports) referred to in the thunk can be used, rather than knowing only about values that are directly created during the course of the thunk's execution.

4.3.2.3 Sending messages

The sending of messages to ports is an operation that should be both efficient and also valid. In the early versions of `paraML`, messages were sent asynchronously to ports and execution continued in the sender as soon as the message had been successfully moved into the communication network. The drawback of this approach is that, although very fast, it gives no guarantee of successful delivery to the destination port. For instance, there might be no available memory to store the message or the process that owned the port might have terminated. That observation led to the problem of how such error conditions could or should be reported to the sender. In the current version, the sending operation is designed such that successful execution guarantees that the message has been delivered and queued in the destination port. Unsuccessful delivery is reported by an exception being raised; typically `NoSuchProcess` if the process has terminated. Naturally this does not guarantee that the process which owns the destination port will ever receive the message in the sense that it may not dequeue the message from the port's queue.

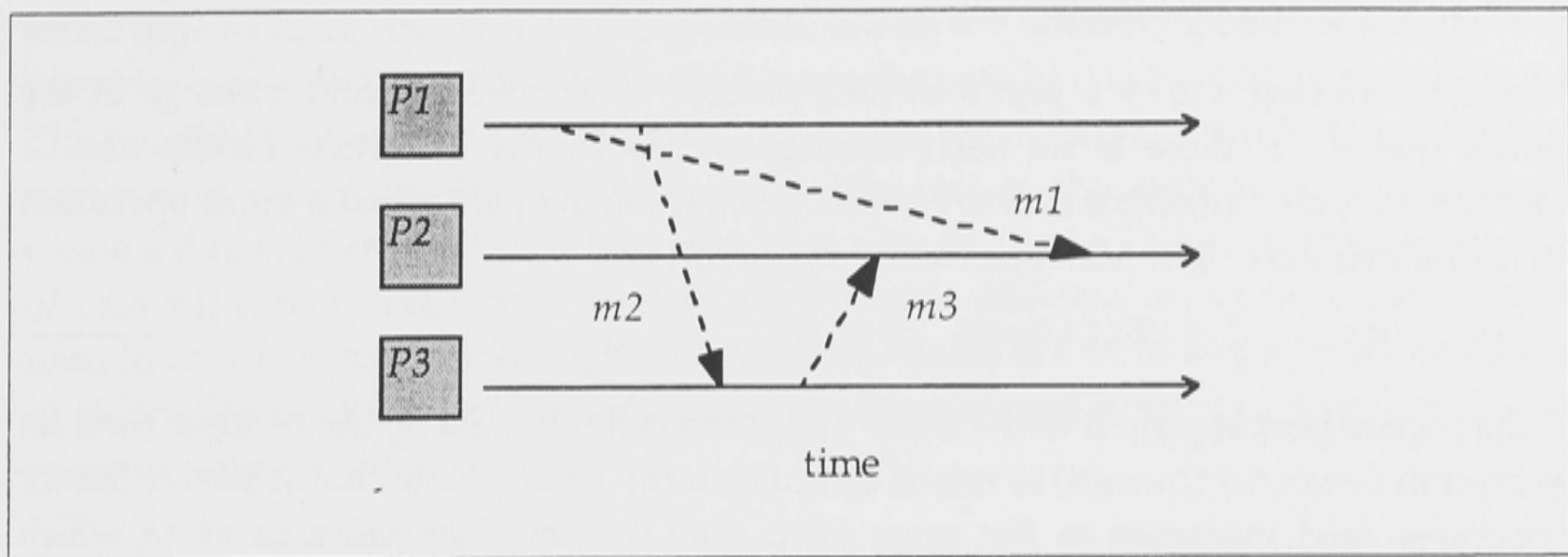


Figure 4.1 – A possible asynchronous message ordering; not possible in paraML.

The ML type of the operation is:

```
val send : 'a PortName * 'a -> unit
```

The operation requires a destination port identified by a port name, and a value of the appropriate type. The unit value is returned if the operation is successful. Note that the type of the operation uses a strongly polymorphic type variable. This is permitted since any port name object will already have been instantiated to a monotype due to the type rule for the `port` operation. The only guarantee made with respect to message ordering between two processes is that if a process successfully sends two messages to the same port, they will be queued and available for receipt in the order they were sent.

However, the confirmation of message receipt at a port allows a slightly stronger property to be achieved with communications between three processes. In a purely asynchronous system, there is no guarantee of message arrival order between three processes. Consider the example shown in Figure 4.1. In this situation, the process *P1* sends two messages, to ports on *P2* and *P3* respectively. *P3* sends a message also to the port on *P2*, but because of the vagaries in the communications network, message *m3* arrives at the port on *P2* before the message *m1*. ParaML prevents this from happening, as the message *m2* cannot be sent until the runtime system confirms that *m1* has been queued in the port on *P2*.

It is also worth emphasising that any first-class object can be sent to an appropriately-typed port. Thus functions (or more strictly, closures of functions) can be communicated from process to process. The ability to send such values stands in stark contrast to imperative programming languages, where it is extremely difficult to send functions or procedures. The ability to dynamically create new functions that incorporate newly-arrived information provides a very powerful mechanism for simplifying programs. The technique is used extensively in applications programming, as discussed in Chapter 8.

4.3.2.4 Receiving messages

Message receiving allows a port's owning process to remove the first message in the port's queue. If there is no such message available, the operation blocks until a message becomes available due to another process having performed a send operation to the named port. The ML type for the operation is:

```
val recv : 'a PortName -> 'a
```

If the named port supplied to the `recv` operation is not owned by the process, then an exception (`PortNotOwned`) is raised to indicate the error. Note that if two different processes send messages to the same port, then it is non-deterministic as to which message will be received first and which second.

As discussed earlier, receiving a message may result in new store fragments being created. For example, if the message type is an integer reference variable, then the message will actually consist of a reference variable and a copy of the integer value that the reference cell contained on the sending process. A new reference cell is created on the receiving process and initialised with the integer value, and the new reference cell's identifying variable is returned. A formal characterisation of this mechanism is given in the operational semantics defined in Chapter 6.

4.3.2.5 Checking for message arrival

Since the `recv` operation has a blocking semantics and `paraML` does not necessarily use multi-threading within processes, it is essential for the process to be able to check whether or not a message is actually available for receipt. The ML type for the operation is:

```
val probe : 'a PortName -> bool
```

The operation examines the port's message queue, and returns a boolean value indicating whether at least one message is present in the queue or not. As with `recv`, an exception (`PortNotOwned`) is raised if the port is not owned by the calling process.

Due to the typing restrictions in `paraML`, it remains necessary to provide `probe` rather than a choice (or guarded selection) primitive, despite the aesthetically undesirable nature of `probe` as a high-level parallel construct. In `paraML` it is not possible to define a choice primitive operation over a list of guards involving port names unless all the port names are of the same type without a fundamental change to the nature of ML, involving also an extensive change to the compiler. This is the general form of a choice operation:

```
choice [ (guard1, action1), (guard2, action2) ... (guardn, actionn) ]
```

Being able to choose only from a list of port names of the same type would either be unnecessarily restrictive or require user-specified type coercion of the port names. (Type coercion is not a practice that is encouraged in ML programs because of the

static type system, and is not even possible unless the underlying ML implementation provides a casting operation which transforms the type of an object at a user's behest.) The possibility of using an ML datatype constructor to package up differently typed port names into a single union type value does not apply, since it is impossible to create a datatype over all possible types that may be used. The alternative approach of creating a new datatype constructor for each different set of port name types used in choice operations throughout a program would not work as the implementation of choice would not know how to unpackage the different datatypes supplied in the guard arguments.

The `probe` operation allows choice operations to be constructed as derived operations from the available communication primitives (see §4.4.3). It must be noted that such choice operations will not be fair in that message delivery occurs autonomously from expression evaluation. The reason for this lack of fairness is that it is not possible to simultaneously test all guards supplied to a choice operation and then non-deterministically select one of the ones which is currently satisfied using the available primitives. The MPI standard does not provide any primitive choice operation, but it does provide probe operations. The implication of providing the probe operation in paraML is that repeated executions of the same program may result in differing behaviour if the program is written to rely on the outcome of a particular instance of probe, since the operation provides transient information about the state of a message queue which is not necessarily reproducible.

4.3.3 Summary

The design of paraML's seven core primitives and two new types has been described, and these extensions are summarised in Table 4-1 below.

<i>operation</i>	<i>ML type</i>
	ProcessName
	'la PortName
process	unit -> ProcessName
self_id	unit -> ProcessName
execute	ProcessName * (unit -> unit) -> unit
port	ProcessName -> 'la PortName
send	'a PortName * 'a -> unit
recv	'a PortName -> 'a
probe	'a PortName -> bool

Table 4-1 – Summary of extensions.

The design was performed with a view to keeping the set of primitives as small as possible while making the semantics of each operation very simple. Thus each operation performs a single action that affects the state of at most one other process. These semantics permit a relatively straightforward formal modelling as described in

Part III. The communication primitives have been constructed so that all communication permits static type checking to be performed. These core primitives can be used to construct alternative abstractions as discussed in the next section.

4.4 Derived operations

A goal of the design of the primitives for paraML was simplicity, which leads to minimisation of the number of primitives provided. Simplicity also has the effect of making the development of formal semantics more straightforward by reducing the number of possible interactions. Although paraML is representative of one style of process-oriented programming, there are other constructs for initiating processes and communicating that a user may find useful. The core paraML primitives may be used to derive these alternative constructs as discussed below.

4.4.1 Synchronous communication

The communication in paraML mixes both synchronous and asynchronous message passing. The ambiguity arises over whether the receipt and queuing of a message in a port constitutes an act of receiving, or whether only the `recv` operation counts. For certain applications requiring a strong degree of synchronicity between processes, programmers sometimes prefer synchronous message passing. The synchronous aspect of the communication is that when a sending operation completes, the receiving application has also completed its `recv` operation to incorporate the message into the ongoing computation. Synchronous communication can be encoded straightforwardly with the paraML primitives as shown in Figure 4.2. The implementation makes use of the ability to send port names as first-class values. A port is created transiently (see §10.8 for discussion of various optimisations in the use of transient ports in the current implementation) by the sender which is used to receive a confirmation that the actual value sent has been received. The confirmation need only be the unit value, since the exception handling mechanism associated with the communications primitives will report any failure such as the destination process having terminated. Note that syncports created on the local process are impossible to use except if the internal language for the process is capable of concurrency, as discussed in §4.6.

4.4.2 Outports

The basic communication model in paraML is predicated on processes always knowing who is to receive a particular piece of information, and then sending it in a message to the corresponding destination port. An alternative model is where the producer of some information is willing to release it, but has no idea which process would like to receive it. I have called the implementation of this form of communication “outports”. They can be modelled quite simply with the code in Figure 4.3. Once again, the implementation makes use of transient ports that are created solely to receive a value, and the name of the port is transmitted in a message. As with synchronous communication, a process is incapable of performing both a `place` and `retrieve`

```

(* a synchronous port is for synchronous communications,
   and consists of a port that accepts a value and a
   unit PortName, to which the confirm can be sent *)
type 'la SyncPortName = ('la * unit PortName) PortName

(* sync_port generates a new synchronous port and name *)
fun sync_port (procn:ProcessName):'la SyncPortName =
  port procn

(* the sync_send operation generates a confirmation
   port, and sends it and the value to the
   destination syncport; it then waits for a reply *)
fun sync_send (spn:'a SyncPortName, x:'a):unit =
  let val confirm_pn:unit PortName = self_port ()
  in
    send (spn, (x, confirm_pn));
    recv confirm_pn
  end

(* the sync_recv operation receives the value and the
   confirmation PortName on the syncport and sends
   unit to the confirmation port of the sender *)
fun sync_recv (spn:'a SyncPortName):'a =
  let val (x,confirm_pn) = recv spn
  in
    send (confirm_pn, ());
    x
  end

```

Figure 4.2 – Synchronous communication.

operation on one of its own outports except in the presence of concurrency. The implementation of both outports and syncports could be easily optimised to use only one send and receive pair at the runtime system level, rather than paying the overheads of two full paraML send/recv pairs.

4.4.3 Choice

The ability to choose from a number of alternative communication possibilities is an essential facet of most parallel programming languages. This ability was not constructed as a core primitive in paraML due to a number of reasons discussed earlier in §4.3.2.5. However, it is straightforward to implement such an abstraction from the core primitives as shown in Figure 4.4. The abstraction, called *choose*, accepts a list of pairs, where each pair consists of a guard (implemented as a thunk that yields a boolean value) and an action (implemented as a thunk that evaluates some arbitrary expression). Typically, the guard would be constructed as a thunk that uses the probe operation to test availability of a message for receiving. However it may be any arbitrary expression that yields a boolean value, and so can be constructed as a


```

(* output ports are a blocking output communication
   from a process, derived with the PortName type *)
type 'la OutPortName = 'la PortName PortName

fun out_port (procn:ProcessName):'la OutPortName =
  port procn

(* place "puts" a value onto the outport by waiting
   for an (input) PortName, to send the value to *)
fun place (opn:'a OutPortName, x:'a) =
  send ((recv opn), x)

(* the operation retrieve "gets" a value from an
   outport by generating a transient port, sending its
   name to the OutPortName, and then receiving a value *)
fun retrieve (opn : 'la OutPortName) =
  let val retrieve_pn:'la PortName = self_port ()
  in
    send (opn, retrieve_pn);
    recv retrieve_pn
  end

```

Figure 4.3 – Outport communication.

more complex guard. The implementation evaluates the guards in order until a guard expression returns the value `true`. The user needs to be wary of this property due to the non-deterministic nature of message delivery. In the case of two successive guards which test two different ports for message arrival, the first guard may fail and then the second succeed, even if the first guard would have succeeded had it been tested at the same time as the second guard. The only possible way to avoid such a situation would be to have primitives which lock out the delivery of messages to all ports until the lock is released, but no such primitives have been provided.

4.4.4 Result

As mentioned earlier, it is sometimes useful to obtain a final value computed by a process, which can be thought of as the result of the process evaluation. A clean abstraction of this notion is one where a process is used to execute some function, and the result of the function is demanded at some later point. Figure 4.5 illustrates an implementation of this abstraction. The `result` function accepts a function `f`, creates a new process and a local port to receive the result of the function. A new function `pf` is declared which sends the result of the function to the local port, and this is executed by the new process. The `result` function returns another function which, when applied to the unit argument, receives the value sent to the local port. In many ways, this is very similar to the notion of futures developed in MultiLisp [Hal85]. In the example that follows, an instance of the `result` abstraction is bound to the value

```

exception Choose

fun choose (nil:((unit -> bool) * (unit -> 'a)) list):'a =
  raise Choose
| choose (choices) =
  let fun choose' (nil) = choose' choices
      | choose' ((guard,action)::rest) =
          if guard ()
          then action ()
          else choose' rest
  in choose' choices
  end

```

Figure 4.4 – Guarded choice.

```

fun result (f:unit -> 'la):unit -> 'la =
  let val procn = process ()
      val result_pn:'la PortName = self_port ()
      fun pf () = send (result_pn, f ())
  in
    execute (procn, pf);
    fn () => recv result_pn
  end

let fun some_function () = ...
    val get_function_result = result some_function
in
  ... get_function_result () ...
end

```

Figure 4.5 – Processes used for results of functions.

`get_function_result`, and at some later point this is applied to a unit argument to incorporate the value computed by evaluation of `some_function`.

4.4.5 Groups and collective communication

Another area of practical parallel programming not addressed by the core primitives is that of collective communication facilities. Krumvieda identified this as the major weakness of CML when extending it to deal with distributed computing, and developed the notion of port groups in Distributed ML [Kru93]. Ports could be a source port, a destination port, or a meta port (which would receive information about other ports), and collections of these ports (defined by a set of source ports and a set of destination ports) were termed port groups, with an ordered multicast semantics for communication over such groups. The MPI standard [MPI94] develops its entire communication philosophy from the concept of a group of processes which have a *communicator* common to all members of the group. New communicators can be formed from sub-groups and efficient collective communications primitives are provided.


```

type GroupProcessName =
  { size          : int,
    processes      : ProcessName list}

fun group_process (n:int):GroupProcessName =
  { size = n,
    processes = iterator (process, (), n)}

fun processes_in_group ({ processes, ... } : GroupProcessName) =
  processes

fun group_execute ( { processes, ... } : GroupProcessName,
                    to_exec:unit -> unit):unit =
  (map (fn procn => execute (procn, to_exec)) processes;
   ())

(* determines the local process's rank in the
   process group *)
fun group_rank ({ processes, ... } : GroupProcessName):int =
  let val my_procn = self_id ()
  in
    isMemberNth (processes, my_procn)
  end

fun group_size ({ size, ... } : GroupProcessName):int = size

```

Figure 4.6 – Process group creation and execution.

The problems facing paraML are somewhat different in that there is no pre-defined group of processes that commence execution together at the start of the program – each process (other than the initial one) is created dynamically during the course of program execution. The creation of a group of processes is the first facility that must be provided through the paraML primitives. The semantic characterisation given in Figure 4.6 uses iterated calls, but it would be possible to implement these operations more efficiently by providing interfaces for some of the collective MPI operations in the runtime system layer. The implementation relies on two functions: *iterator*, which applies a function repeatedly to an argument, and *isMemberNth*, which determines whether a value is a member of a list and if so the position within the list (numbering from 0).

The operations to create a process group and to get each process in such a group to execute a function are straightforward. The operations for communication with groups are more complex. Just as there are process groups, there are also port groups (as defined in Figure 4.7; note these do not have the same semantics as those in Distributed ML [Kru93]) and variants of the standard communication primitives (given in Figure 4.8). The sending operation becomes effectively a multicast to all ports in the port group. An assumed function is *nth*, which finds the n^{th} element of a list.

```

type 'la GroupPortName =
  { size          : int,
    processes      : ProcessName list,
    ports          : 'la PortName list}

fun group_port ({processes,size}:GroupProcessName) :
  'la GroupPortName =
  { size = size, processes = processes,
    ports = map (fn procn => port procn) processes}

fun ports_in_group ({ports,...}):'a GroupPortName) = ports

(* determines the local process's rank in the port group *)
fun group_port_rank({processes,...}:'a GroupPortName) =
  let val my_procn = self_id ()
  in
    isMemberNth (processes,my_procn) : int
  end

fun group_port_size({size,...}:GroupPortName):int = size

```

Figure 4.7 – Basic port group creation and attribute operations.

```

(* group_send is essentially a multicast operation *)
fun group_send ({ports,...}:'a GroupPortName,x:'a):unit =
  (map (fn pn => send (pn,x)) ports; ())

(* group_recv receives from a multicast *)
fun group_recv (gpn as {ports,...}:'a GroupPortName):'a =
  recv (nth (ports,group_port_rank gpn))

(* group_probe probes on a group port *)
fun group_probe (gpn as {ports,...}:'a GroupPortName):'a =
  probe (nth (ports,group_port_rank gpn))

```

Figure 4.8 – Basic port group communications.

Note that there is no guaranteed ordering in the case of two overlapping multicasts. Implementing a broadcast or an ordered multicast by sending to a single process's port and then having it resend the message to all members would be possible.

Additionally, it is useful to provide an operation that scatters a list of values to a port group, and to gather a list of values from an output group. Generalised stride communication primitives are clearly feasible to provide also. The basic scatter/gather primitives are shown in Figure 4.9, together with definitions for output groups. Assumed in the implementation are: `zip`, which merges two lists into a single list; `length`, which reports the size of a list; and `nth`. A process may not scatter and gather on the same output group except in the presence of concurrency.


```

exception GroupArityError

type '1a GroupOutPortName = '1a PortName GroupPortName

fun group_out_port (gprocn:GroupProcessName)
    : '1a GroupOutPortName =
    group_port gprocn

(* group_scatter distributes a list among the
   port group *)
fun group_scatter ({ports,size,...}:'a GroupPortName,
    xs:'a list):unit =
    (if not (length x = size)
     then raise GroupArityError
     else map (fn pn_and_x => send pn_and_x)
              (zip (ports,xs)); ())

(* group_place makes available a value for collection *)
fun group_place (gopn:'a GroupOutPortName, x:'a):unit =
    send (recv (nth (ports,group_port_rank gopn), x))

(* group_gather assembles a list of values from a
   outport group; group_retrieve has the same
   functionality *)
fun group_gather ({ports,...}:'a GroupOutPortName)
    : 'a list =
    let val gather_pn:'1a PortName = self_port ()
    in map (fn pn => (send (pn,gather_pn);
                        recv gather_pn)) ports
    end

val group_retrieve = group_gather

```

Figure 4.9 – Scattering and gathering to port groups.

```

type '1a GroupSyncPortName =
    ('1a * unit PortName) GroupPortName

val group_sync_port (gprocn:ProcessName)
    : '1a GroupSyncPortName

val group_sync_send (gspn:'a GroupSyncPortName, x:'a):unit

val group_sync_recv (gspn:'a GroupSyncPortName):'a

```

Figure 4.10 – Synchronous port groups.


```

val reduce : (gopns:'a GroupOutPortName,
              result:'a GroupPortName,
              redf:'a * 'a -> 'a,
              x:'a) -> 'a

val scan : (partial_scan:'b GroupOutPortName,
            scan_to_include:'b GroupPortName,
            scanf:'a * 'b -> 'b,
            identity:'b,
            x:'a) -> 'b

```

Figure 4.11 – Reduction and scan operation interfaces.

```

fun barrier (barrier_gpns as {ports,size,...} :
             unit GroupPortName) : unit =
  let val rank = group_port_rank barrier_gpns
  in
    if size = 1 then ()
    else
      if rank = 0
      then (
        iterator (recv,(nth (ports,0)),size-1);
        group_send (barrier_gpns,());
        group_recv barrier_gpns )
      else (
        send (nth (ports,0),());
        group_recv barrier_gpns )
    end
  end

```

Figure 4.12 – Barrier synchronisation for groups.

Synchronous versions of port groups are clearly easy to construct also. The same restriction holds that synchronous communication by a process to a port group of which it is a member is not possible except in the presence of internal concurrency. Only the type signatures of the basic operations are given in Figure 4.10.

There remain two more interesting collective communication primitives: the reduction and scan (otherwise known as parallel prefix) operations. Both of these may be implemented with varying degrees of efficiency from the basic and derived operations outlined above. The most efficient implementations using point-to-point message passing revolve around the construction of binary trees of the participating processes' port groups. Since most of their implementation is taken up with the manipulation of these trees, Figure 4.11 only gives the type definitions for the `reduce` and `scan` operations. They both rely on having appropriately-typed port groups created prior to use, but can accept arbitrary functions to perform the reduction or scanning.

4.4.6 Barrier synchronisation

There are no explicit synchronisation primitives built into the core of paraML, but with the provision of process and port groups, it becomes natural to desire barrier synchronisation facilities. The semantics of this operation is that every member of the process group must call the `barrier` operation, and no process can proceed past the synchronisation point until all others have participated in the barrier operation also. The implementation is given in Figure 4.12. The essence of the implementation is for all processes other than the root process in the group to send a message to the root, and once the root process has received them, it multicasts a message back to the port group (it must also receive this message itself).

4.4.7 Summary

The development of derived operations from the core extensions of paraML is greatly facilitated by the ML language itself. ML provides excellent facilities for building clean abstractions through functions, user-defined types, and modules. Some of the problems of “plumbing” message passing connections between processes are simplified with these facilities as they aid in the differentiation of ports. Ports are typed and port names are bound to lexically-scoped identifiers, which makes it difficult for them to be accidentally confused with other port names. The module system allows whole libraries to be constructed utilising or providing a particular set of derived operations. Interfaces to these modules are type-safe, and completely abstract the implementation details and inter-process message passing “plumbing” required.

4.5 Input/output

Any process-oriented programming system must confront the issue of input/output (I/O), and how to manage potential interleaving of I/O operations. In a system which is constrained to a fixed number of processes, standard output is relatively easily supported by tagging all output with the process rank, and performing a post-execution sort on that value to collate each process’s output. Alternatively, writing to a single file can be performed according to the process rank in a round robin fashion.

The issue is somewhat more complex in paraML since process groups are created dynamically throughout the course of program execution, and thus the potential variety of interleavings is increased accordingly. One simple solution is to define an I/O process, which is capable of performing standard I/O requests, but buffers all output internally. When requested, the process can flush its buffer to a given file. When a process group is created, another process group of I/O processes can also be created of the same size. When the process group terminates, it can flush its buffers in turn. An example of this form of programming is given in Figure 4.13, which illustrates an I/O process group which accepts `write` and `close` requests. There are no `read` requests included in this example, but they would be straightforward to incorporate if there was a standard segmented input stream available.

```

type GroupIO = {write:string -> unit,
                close:unit -> unit}

fun group_io (gprocn as {size,...}:GroupProcessName)
    : GroupIO =
  let val io_gprocn = group_process size
      val write_gpn : string GroupPortName =
        group_port io_gprocn
      val close_gpn : unit GroupPortName =
        group_port io_gprocn
      fun io_exec () =
        let val out_buf = ref ""
            fun io_loop () =
              choose
                [ (fn () => probe write_gpn,
                  fn () => out_buf := (!out_buf) ^
                    (group_recv write_gpn);
                    io_loop ()),
                  (fn () => probe close_gpn,
                    fn () => write (stdout,!out_buf))
                ]
            in io_loop ()
          end
      val {ports=wports,...} = write_gpn
      fun write_op (to_write:string):unit =
        let val send_to_n = group_process_rank gprocn
            in send (nth (wports,send_to_n),to_write)
          end
      val {ports=cports,...} = close_gpn
      fun close_op ():unit =
        let val send_to_n = group_process_rank gprocn
            in send (nth (cports,send_to_n),())
          end
  in
    group_execute (io_gprocn,io_exec);
    {write=write_op,close=close_op}:GroupIO
  end

```

Figure 4.13 – Input/output for process groups.

```

(* this operation reports the total number of PEs in the
   multicomputer *)
val number_of_PEs : unit -> int

(* this operation reports the rank of the PE on which
   the calling process is being executed *)
val rank_of_PE : unit -> int

```

Figure 4.14 – Machine attribute operations.

4.6 Machine attribute operations

Optimal use of a multicomputer may be aided if the user is able to efficiently map problems to the available machine resources. Although the number of VPs is effectively unlimited, there are always only a finite number of PEs. In particular, it may be useful for a user's program to be able to query how many PEs there are available for use, and on which particular PE a process is executing. These two operations are shown in Figure 4.14. Effectively they correspond to the MPI operations `MPI_Comm_size` and `MPI_Comm_rank` [MPI94], where the communicator argument to the operations is `MPI_COMM_WORLD`.

The possibility of co-location of processes (that is, processes being executed on the same PE) is an obvious facility that programmers might like to utilise for particular algorithms. In general, the primitives have all been designed so that programmers do not work at the physical machine level. However, a process co-location primitive could be defined to create a new process on the same PE as an existing process; the operation would have the following type:

```
val colocate_process : ProcessName -> ProcessName
```

Since the formal semantics does not incorporate information about PEs on which processes are executed, any program that is correct with the standard `process` operation will be unaffected semantically if one or more occurrences of those operations are replaced by the `colocate_process` operation.

Co-located process groups could also be constructed easily using this primitive rather than the standard `process` operation. Such a facility becomes useful when large data structures are distributed among a number of *server* processes since the corresponding *client* processes can then be distributed in a similar fashion, which is likely to result in the majority of message passing between clients and servers occurring on the same PE, leading to a decrease in communication costs.

4.7 Concurrency within processes

There are many advantages if a process may internally utilise a concurrent extension of ML, rather than just sequential Standard ML itself. Many implementations of ML provide operations for manipulating continuations (in SML/NJ [AM91] these are `callcc` and `throw`). These facilities are sufficient for implementing multithreading. Appel [App92] gives an example of this, providing routines for forking, yielding, and dispatching of threads. The paraML runtime system layer which manages process/VP scheduling on an individual PE is built with a multi-threading system constructed by using continuations. Alternatively, the CML language [Rep92] can be used as the basis for a process's evaluation language, and the paraML primitives used to extend CML. The main restriction is that since threads exist within the shared address space of a process and may have certain thread runtime system state, they cannot migrate across different processes. It is possible for continuations to be transmitted, but they are

subject to the same restrictions as other communicated values. The formal semantics of paraML described in Part III of this thesis do not incorporate continuations or concurrency within processes.

There are a number of advantages in having a concurrent language within processes, which include the following:

- Concurrent code allows the development of more flexible solutions to particular problems.
- Concurrency allows a process to perform other actions whilst waiting for a blocking communication request to be completed, thus hiding network latency to a degree.
- Synchronous communication operations can be performed where a process is both sender and receiver (provided these are actions of different threads).
- Processes may become evaluation servers – forking new threads to execute expressions sent to them, thereby eliminating certain overheads associated with process creation rather than thread creation.

The reader may be concerned that having both parallel process and concurrency primitives would be unnecessary, since they appear superficially to perform the same actions of allowing independent computation to take place. However, the two sets of primitives have a critical distinction – the process-oriented operations are predicated on execution in a distributed address space while the concurrency primitives are predicated on execution in a shared address space. If the same set of primitives was used for parallelism and concurrency, it would become necessary to distinguish the sharing attributes on creation of a process, which in effect is semantically equivalent to specifying two different creation operations. The other problem that arises is that a single set of communication operations fails to distinguish whether a transmitted object would be shared or not. For these reasons, I believe it is crucial to have two sets of operations to distinguish the sharing attributes of programming a high performance system.

4.8 Benefits and limitations

The major benefit of the approach taken with paraML is that it maps cleanly onto the underlying machine model. This mapping is important as it helps in achieving the attributes of scalability, portability, and locality. The last goal of locality is particularly important for achieving efficient programs, and the issue of locality is made explicit to the programmer by requiring the use of message passing to transmit objects among processes. The number of extensions to ML is close to minimal while still embodying the programming model described, which makes the language simple to learn. These extensions are also useful in creating alternative abstractions for process-oriented programming as illustrated in §4.4.

The chief criticisms that are likely to be levelled at the type of programming system embodied by paraML are the lack of shared address space support (with the consequent requirement that the programmer needs to encode any such sharing through message passing) and the potential costs of copying objects on transmission between processes (which to some degree is a consequence of the lack of sharing). The main alternative on multicomputers to the distributed address space model supported by paraML is that of a distributed shared memory. Such a system provides the program with access to all (or most) of the memory in the multicomputer, regardless of its location relative to the current execution locus, and does not distinguish between local and remote access mechanisms. The advantage of this approach is that it requires only a single set of primitives in the programming language to initiate processes and to communicate and synchronise between processes. The disadvantages of the approach are:

- The issue of data locality is removed from the awareness of the programmer, and the runtime system is responsible instead for trying to achieve good data locality.
- Portability among different multicomputers becomes reliant on the provision of an equivalent DSM system, and such systems are much less available than implementations of MPI.
- Scalable performance of DSM systems becomes increasingly difficult without relaxing memory consistency models; even so, fine-grained memory contention remains particularly problematic.

An interesting critique of the current state of DSM research is available in [CKK95]. The consequence of these three disadvantages is that it becomes increasingly difficult to produce efficient programs that meet the stated goals for high performance computing. Recent advances in hardware technology, such as the Silicon Graphics Origin2000 computers, may help to alleviate some of these problems by providing efficient coherent distributed shared memory. However, this technology retains the non-uniform memory access latencies associated with distributed address space multicomputers. The issues involved in finding programming models which address the performance impediments that arise with distributed memories will remain in the foreseeable future.

4.9 Conclusion

While the programming model discussed in the previous chapter is an important factor in the design of paraML, the core operations that extend ML provide the full realisation of this design. These operations address a number of issues that are important for safe high performance programming. The flexibility of the operations determines how simple or complex it is to form alternative programming abstractions from them; a number of such alternatives have been presented. Understanding the core operations and the programming model is important since together they constitute the

distinctiveness of paraML. The central difference between paraML and CML is that the paraML operations must address the challenges arising from programming in the context of a distributed address space. The key to this is in capturing the notion of computation with values communicated from a different process's memory. This informal understanding is made concrete by the theoretical modelling of paraML in the next part of this thesis.

Part III

THEORY

Part II

THEORY

5. Theory background

5.1 Introduction and motivation

The sequential language Standard ML, on which paraML is based, has a rigorous and formal definition [MTH90]. The authors comment on the role of a language definer being to create a world of meanings (or semantic objects) appropriate for the language and a precise way of describing what the meanings are without resorting to a programming language notation. They also discuss the benefits of providing a theory of the meanings, which then enables reasoning about the equivalence of objects.

The purpose of developing a formal semantics for paraML has fewer goals than that for Standard ML. The primary goal is to characterise formally how evaluation of a paraML program may proceed. This goal is achieved by the development of sequential and parallel evaluation relations for λ_{pv} , an extension of Plotkin's call-by-value λ_v calculus [Plo75]. The λ_{pv} calculus and the definitions of evaluation it encodes provide an operational semantics that models the essence of paraML. A polymorphic type system is developed for the calculus and proved to be sound with respect to the operational semantics. The combination of the operational semantics and the type system capture the requirements for an implementation of paraML. The style of semantics used is identical to CML [Rep91, Rep92], which makes it simpler to distinguish paraML's treatment of communicated values.

Another purpose of developing a formal semantics for paraML is in the benefits arising from the interplay between the language design, formal semantics, and implementation. Working on these three aspects concurrently for paraML helped to inform each aspect individually in a way that would have been impossible if they were approached as unrelated activities.

The remainder of this chapter looks briefly at some related work in developing formal semantics for parallel extensions of existing functional languages, and then proceeds to review notation. The major component of the chapter is the development of an extended λ_v calculus called λ_{ev} to model sequential computation in paraML. The λ_{ev} calculus can be viewed as the computation language within processes. The λ_{pv} calculus is effectively a superset of λ_{ev} , although there are slight differences with respect to the treatment of exceptions. Chapter 6 develops the operational semantics for λ_{pv} and Chapter 7 covers the typing of λ_{pv} .

5.2 Related work

The formal semantics of paraML loosely follows the style developed by Reppy for CML [Rep92] and uses similar notation. In turn, Reppy's work is based primarily on

the style developed by Wright and Felleisen [WF91]. Reppy develops a concurrent extension of λ_v , which he calls λ_{cv} , and uses this to model the operations of CML. The choice of Wright and Felleisen's style of semantics is useful in proving type soundness for the resulting calculus in a purely syntactic manner. However this style of operational semantics is really only useful for reasoning about entire programs, not about program fragments. The development of a theory to enable equational reasoning about program fragments is more manageable with other styles of semantics, such as a labelled transition system (LTS) semantics. Reppy's remarks on the development of an algebraic theory for CML led to Ferreira, Hennessy and Jeffrey [FHJ95] developing a LTS for CML. Their work also proves an equivalence between their labelled transition system semantics and Reppy's formalism. A similar approach should be possible for the semantics presented here. Berry, Milner and Turner also present an alternative semantics for CML [BMT92]. A number of obvious differences between paraML and CML (such as the distributed address space versus shared address space models) result in significantly different calculi.

Formal semantics have been developed also for Facile [TLG92]. The authors grapple with many of the issues explored for CML, which are made more complex in Facile by the problems of distribution, time, and failure. The operational semantics take the form of two LTSs: one for an "evaluates" relation for evaluation of expressions and one for a "derives" relation for execution of processes. The concept of locality is addressed by adding node identifiers into the semantics, and providing a notation for indicating that a behaviour expression (effectively a thread of control or process) is located at a particular node; this concept is known as a distributed behaviour expression and requires yet another LTS. However, the semantics does not capture the transmission of reference variables in the two different ways that the informal characterisation describes. Shared references are modelled by an encoding into primitive thread creation and communication operations. The copied references are not modelled at all. The advantage of the approach taken in Facile is that the use of a LTS semantics should enable equational reasoning about program fragments.

The development of Concurrent Haskell (a concurrent extension of Haskell) has also seen the development of a formal semantics [PFG96]. Concurrent Haskell assumes a shared memory model of computation, with explicit process creation and atomically-mutable state (MVars) for communication and synchronisation between processes. The problems of regarding input/output as functional state transformation in Haskell led to Gordon developing a monadic characterisation for the semantics of I/O in purely-functional languages [Gor94]. These problems are even more serious when adding concurrency, which encouraged the use of operational semantics for Concurrent Haskell. The semantics are compact and simple, separating reduction into a deterministic reduction relation (specifying the computation language) and a non-deterministic reaction relation (specifying the coordination language). This separation is similar to the sequential and parallel evaluation relations developed for λ_{pv} . Locality is not addressed specifically, since their interest is with the concurrency aspects, not with distributed address space computation.

Another interesting language with formal semantics is PICT [PT94], but the semantics are obtained more through its nature as an implementation of the asynchronous π calculus than because a separate semantics has been developed for it. There is considerable interest currently in process calculi research. Some examples include the π calculus and its variants [HT91, MPW92], CHOCS [Tho95] and HO π [San92]. These are not directly related to the style of semantics developed for paraML, and so are beyond the scope of this thesis.

5.3 Notation

Most of the notation required for the presentation of the formal semantics is based on mathematical set notation, and should be familiar. Given two sets A and B , their union is $A \cup B$, their intersection is $A \cap B$, and their difference $A \setminus B$. The empty set is denoted \emptyset . The generalised union of a set of sets $\{A_1, \dots, A_n\}$ is written $\bigcup \{A_1, \dots, A_n\} \equiv A_1 \cup \dots \cup A_n$. The notation $A \xrightarrow{\text{fin}} B$ denotes the set of finite maps (partial functions with finite domains) from A to B . Given a map f the domain and range of f are defined by:

$$\begin{aligned} \text{dom}(f) &= \{ x \mid f(x) \text{ is defined} \} \\ \text{rng}(f) &= \{ f(x) \mid x \in \text{dom}(f) \} \end{aligned}$$

Finite maps are represented by the notation:

$$\{ a_1 \mapsto b_1, \dots, a_n \mapsto b_n \}$$

where the domain of the map is $\{ a_1, \dots, a_n \}$. The map with an empty domain is written $\{ \}$. It is also useful to consider maps as sets of ordered pairs in the usual way. If C is a set, then $C+x$ denotes $C \cup \{x\}$; if C is a map and $x = (a, b)$, then $C+x$ is defined only when $a \notin \text{dom}(C)$. The $+$ operator associates to the left; thus $C+x+y$ should be read as $(C+x)+y$. The *composition* of two finite maps is written $f \circ g$; *f modified by g* is also a map written $f \pm g$ whose domain is $\text{dom}(f) \cup \text{dom}(g)$, and is defined as:

$$(f \pm g)(x) = \begin{cases} g(x) & \text{if } x \in \text{dom}(g) \\ f(x) & \text{otherwise} \end{cases}$$

Note that if $\text{dom}(g) \cap \text{dom}(f) = \emptyset$ then the modification has a domain equal in size to the sum of the size of its component maps, but otherwise the size of the domain is less than the sum – thus the choice of the symbol \pm . The set of finite subsets of C is denoted $\text{Fin}(C)$. Lastly, if there is a binary relation μ , then μ^* is the transitive closure of μ . Throughout the theory chapters, text in *fixed-space font* is used to indicate value constants of the calculi, and also for paraML operations as they are represented in the λ_{pv} calculus.

x	\in VAR	variables
c	\in CONST = BCONST \cup FCONST	constants
	BCONST = {(), true, 0, 1 ...}	base constants
	FCONST = {+, -, fst, snd, ...}	function constants
ex	\in EXNNAME	exception names

Figure 5.1 – Ground terms of λ_{cv} .

e	\in EXP	expressions in the language
v	\in VAL \subset EXP	values in the language
exn	\in EXN \subset EXP	exception packets in the language

Figure 5.2 – Basic syntactic definitions.

5.4 Formal semantics

The rest of this chapter introduces an extension of Plotkin's call-by-value lambda calculus λ_v [Plo75]. The purpose of this extension known as λ_{cv} is to present the basis of sequential computation in paraML. The extensions that have been incorporated are derived from two main sources: Reppy's presentation of the sequential aspects of λ_{cv} (itself an extension of λ_v) [Rep92] and some of Wright and Felleisen's extensions to λ_v [WF91]. The style of semantics developed by Wright and Felleisen is adopted as it leads to a purely syntactic treatment of type soundness.

A number of extensions are included to faithfully characterise the sequential semantics of paraML. The extensions to the basic λ_v calculus include: pair values, exceptions, references, and a polymorphic type system. Pair values and exceptions are drawn from Reppy's presentation, references are adopted from Wright and Felleisen's work, and the polymorphic type system is that of Damas and Milner [DM82]. The syntax and dynamic semantics are presented first, followed by the type inference system, together with some of the theorems that connect the static and dynamic semantics.

5.4.1 Syntax

The ground terms of the λ_{cv} calculus are *variables*, *constants* (which are split into *base constants* like integers, and *function constants* like + for integer addition; there is no overlap between base and function constants), and *exception names*. The ground terms are given in Figure 5.1.

There are three syntactic classes of terms: *expressions*, *values*, and *exception packets* (which are raised exceptions). Values are the *canonical* terms in the dynamic semantics – since the semantics operates by subject reduction this means that they are *irreducible*. The exception packets are also irreducible terms, but are not values in the

e	$::=$	(e)	parenthesised expression
	$ $	v	value
	$ $	$e_1 e_2$	application
	$ $	$(e_1 . e_2)$	pair
	$ $	$\text{let } x = e_1 \text{ in } e_2$	let
	$ $	$\text{exception } x \text{ in } e$	exception
	$ $	$\text{raise } e_1 e_2$	raise exception
	$ $	$e_1 \text{ handle } x e_2$	exception handle
	$ $	exn	exception packet
	$ $	$\rho\theta.e$	ρ -expression – store fragment
v	$::=$	c	constant
	$ $	x	variable
	$ $	Y	fixed point combinator
	$ $	$(v_1 . v_2)$	pair value
	$ $	$\lambda x(e)$	λ -abstraction
	$ $	ex	exception name
	$ $	ref	reference creation
	$ $	$!$	dereference
	$ $	$:=$	assignment
	$ $	$:= x$	curried assignment
exn	$::=$	$[ex, v]$	exception packet
χ	$::=$	$\{ ex_1, ex_2, \dots, ex_n \}$	exception name set
θ	$::=$	$\{ \langle x_1, v_1 \rangle, \langle x_2, v_2 \rangle, \dots, \langle x_n, v_n \rangle \}$	$\langle \text{variable}, \text{value} \rangle$ set

Figure 5.3 – Grammar for expressions, exceptions and values.

language. These syntactic definitions are given in Figure 5.2, and the grammar for expressions, values, and exception packets is given in Figure 5.3. According to the grammar, pairs of values may be either expressions or values; the latter is chosen since generally the calculus tries to reduce expressions to values.

The reason for the formulation of exceptions being different from Wright and Felleisen's approach is that when the language is extended to deal with parallelism, the transmission of exceptions can lead to problems unless there is an implicit global environment for the exception names. The terms for exception packets and exception names are intermediate forms that do not appear in programs. Implicit from this is that there are no constant exception names in the calculus. Evaluation of the exception term results in substitution of a new exception name ex for x in the

expression e . At the same time, the exception name is added to the global environment χ containing exception names. This approach captures the generative nature of exceptions in Standard ML. The treatment of exceptions with a global exception name set is similar to the inclusion of such an exception name set in the state component (denoted s) of the inference rules for Standard ML's definition [MTH90]. However it should be noted that the semantics of exceptions is not identical to Standard ML. Handle expressions are evaluated and installed as handler functions before an exception is raised.

The formulation for references is adopted straightforwardly from Wright and Felleisen, which in turn is taken from ML. The store is represented by pairs of variables and values $\langle x, v \rangle$. The expression $\rho \langle x_1, v_1 \rangle \dots \langle x_n, v_n \rangle. e$ binds $x_1 \dots x_n$ in $v_1 \dots v_n$ and e , with the notation $\langle x_1, v_1 \rangle \dots \langle x_n, v_n \rangle$ used as shorthand for $\{\langle x_1, v_1 \rangle, \dots, \langle x_n, v_n \rangle\}$. The symbol θ is a finite map from variables to values; recall from the notation that θ is treated as a set of pairs whose first components are distinct. All ρ -expressions which differ only by a consistent renaming of bound variables are identified. The store can be thought of as a collection of (reference) cells with distinct names; each cell can store one value in it. The names of cells are represented by variables. The operation `ref` v creates a new cell with an initial value v , returning the name of the new cell x . The operation `!` x returns the current value in the cell, and the operation `:=` x v replaces the current value in the cell named x with v , returning the new value (which differs from Standard ML which just returns the unit value). Since assignment is a curried operation, the application of `:=` to a variable x is another value, which can be considered an ability to assign to the cell x .

Free variables of a term $FV(e)$ may be defined inductively over the structure of expressions, as given in Figure 5.4. *Closed* expressions and values are those with no free variables, that is, $FV(e) = \emptyset$. The set of closed values is denoted VAL° . The terms for `let`, λ -abstraction, `exception` and ρ are the only terms which declare variables. Occurrences of an identifier within the subsequent expression are then considered bound within the scope of the `let`, λ , `exception` or ρ expression which declares the variable identifier. For example, a term $(x \ y)$ contains two free identifiers. In the term $\lambda x(x \ y)$, x is bound and y is free. Terms are identified up to α -conversion of bound identifiers, thus:

$$\text{let } x = 1 \text{ in } + x \ 1 \quad =_\alpha \quad \text{let } x' = 1 \text{ in } + x' \ 1$$

The formulation of exceptions provides no binding mechanisms for exception names. Thus a closed value term may still contain free exception names. Free exception names may be defined as:

Definition 5-1 (Free exception names) The free exception names of an expression e are denoted $FEN(e)$, which is exactly the set of exception names ex_i that appear in e .

$FV(c)$	$=$	\emptyset
$FV(x)$	$=$	$\{x\}$
$FV(Y)$	$=$	\emptyset
$FV(\lambda x(e))$	$=$	$FV(e) \setminus \{x\}$
$FV(ex)$	$=$	\emptyset
$FV(\text{ref})$	$=$	\emptyset
$FV(!)$	$=$	\emptyset
$FV(:=)$	$=$	\emptyset
$FV(:= e)$	$=$	$FV(e)$
$FV(e_1 e_2)$	$=$	$FV(e_1) \cup FV(e_2)$
$FV(e_1 . e_2)$	$=$	$FV(e_1) \cup FV(e_2)$
$FV(\text{let } x = e_1 \text{ in } e_2)$	$=$	$FV(e_1) \cup (FV(e_2) \setminus \{x\})$
$FV(\text{exception } x \text{ in } e)$	$=$	$FV(e) \setminus \{x\}$
$FV(\text{raise } e_1 e_2)$	$=$	$FV(e_1) \cup FV(e_2)$
$FV(e_1 \text{ handle } x e_2)$	$=$	$FV(e_1) \cup FV(e_2) \cup \{x\}$
$FV(exn)$	$=$	\emptyset
$FV(\rho\theta.e)$	$=$	$(FV(e) \cup FV(\text{rng}(\theta))) \setminus \text{dom}(\theta)$

Figure 5.4 – Free variables in terms.

The `let` and λ -abstraction terms are also important due to the *substitution* of terms that is integral to the λ -calculus. (Substitution is also used for the exception term.) Substitution of a term e for variable x in a term e' is written $\{e/x\}e'$. However, x must not be bound in e' and no free variable of e must be bound in e' ; to avoid this, Barendregt's variable convention [Bar84] is used which, since bound variables can be renamed throughout a term, allows all the bound variables to be chosen different from the free variables, thus avoiding the problem of capture of free variables during substitution. The inductive definition of substitution for a term is given in Figure 5.5. In the four binding terms (λ -abstraction, `let`, `exception` and ρ) the bound variables are renamed to prevent any instance of $x = x'$ affecting the substitution.

5.4.2 Dynamic semantics

The purpose of the dynamic semantics is to establish an understanding of what the semantic objects of the language are and how they interact. In the style of operational semantics developed by Felleisen, Friedman and Hieb [FF86, FH92], the objects of the dynamic semantics are syntactic terms which are members of the class of expressions (EXP). Both Reppy and Wright and Felleisen use this approach because the syntactic nature of the semantic objects permits a relatively straightforward and extendable mechanism for the establishment of type soundness for a language.

$\{e/x\}c$	$=$	c	
$\{e/x\}x$	$=$	e	
$\{e/x\}x'$	$=$	x'	$x' \neq x$
$\{e/x\}(\lambda x'(e'))$	$=$	$\lambda x'(\{e/x\}e')$	
$\{e/x\}ex$	$=$	ex	
$\{e/x\}\text{ref}$	$=$	ref	
$\{e/x\}!$	$=$	$!$	
$\{e/x\}:=$	$=$	$:=$	
$\{e/x\}(:= e)$	$=$	$(:= \{e/x\}e')$	
$\{e/x\}(e_1 e_2)$	$=$	$\{e/x\}e_1 \{e/x\}e_2$	
$\{e/x\}(e_1 . e_2)$	$=$	$(\{e/x\}e_1 . \{e/x\}e_2)$	
$\{e/x\}(\text{let } x' = e_1 \text{ in } e_2)$	$=$	$\text{let } x' = \{e/x\}e_1 \text{ in } \{e/x\}e_2$	
$\{e/x\}(\text{exception } x' \text{ in } e')$	$=$	$\text{exception } x' \text{ in } \{e/x\}e'$	
$\{e/x\}(\text{raise } e_1 e_2)$	$=$	$\text{raise } \{e/x\}e_1 \{e/x\}e_2$	
$\{e/x\}(e_1 \text{ handle } x' e_2)$	$=$	$\{e/x\}e_1 \text{ handle } \{e/x\}x' \{e/x\}e_2$	
$\{e/x\}exn$	$=$	exn	
$\{e/x\}(\rho\theta.e')$	$=$	$\rho\langle x_1, \{e/x\}v_1 \rangle \dots \langle x_n, \{e/x\}v_n \rangle . \{e/x\}e' \quad x \notin \text{dom}(\theta)$	

Figure 5.5 – Substitution in terms.

In this form of operational semantics, *rules of reduction* are developed which are defined as binary relations that syntactically transform terms of the calculus. To restrict the order in which reductions are applied to terms, *contexts* are defined for the terms of the calculus. Contexts are expressions with one subexpression replaced by a *hole* marked by $[]$. Placing an expression e , or *redex*, in the hole of a context $C[]$ produces a new expression, written $C[e]$. Context grammars may be thought of as a higher order syntactic pattern matching of expressions. For example, the contexts of λ_v would be defined:

$$C ::= [] \mid C e \mid e C \mid \lambda x(C)$$

An example term in the calculus is:

$$(\lambda x(x) \lambda x(x)) (\lambda x(x) 1)$$

This term may be split into a context/redex pair in several ways, for example:

$$C = (\lambda x(x) \lambda x(x)) ([]) \quad e = \lambda x(x) 1$$

In turn, the subexpression e may also be split into a context/redex pair, again in more than one way:

$$C' = [] 1 \quad e' = \lambda x(x)$$

This time, there is only one form of context for the subexpression e' :

$$C'' = \lambda x([\]) \quad e'' = x$$

Generalised contexts pose two problems for successfully modelling computation for call-by-value calculi. First, as is seen in the example above, placing the subexpression e'' in the hole of the context C'' leads to capture of the free variable x . Unintentional capture of free variables can lead to serious problems for well known reasons (as shown by Barendregt, page 25 of [Bar84]). The second problem is that there are potentially multiple ways of partitioning terms into context/redex pairs. In order to describe evaluation when evaluation order is fixed, there needs to be some way of restricting the term partitioning.

Evaluation contexts are a restricted set of the full contexts for a language. The restrictions are carefully chosen to allow only certain pattern matching possibilities. In the case of strict call-by-value λ calculi, this has the effect of specifying the order of reduction and preventing capture of free variables when a redex is placed in a context hole. The evaluation contexts E for λ_v would be defined:

$$E ::= [\] \mid E e \mid v E$$

These evaluation contexts restrict term partitioning so that first the function term is evaluated to become a value (a λ abstraction), then the argument term is evaluated, and finally the function value is applied to the argument value. There is no possibility of free variable capture, because the hole never occurs inside the binding construct of the calculus. The evaluation of the example term given above would proceed as follows, with the context/redex boundary marked by $[\]$:

$$\begin{aligned} & [(\lambda x(x) \lambda x(x))] (\lambda x(x) 1) \longrightarrow \\ & \lambda x(x) [(\lambda x(x) 1)] \longrightarrow \\ & [\lambda x(x) 1] \longrightarrow \\ & [1] \end{aligned}$$

Rather than specifying the rules of reduction for each function constant, a partial function δ is assumed to exist which determines the result of applying function constants of the calculus to closed values. (If the closed values were always of the right type, the δ function is total, but since this cannot be guaranteed the function is partial.) In λ_{cv} , the partial function δ is defined:

$$\delta : \text{FCONST} \times \text{VAL}^\circ \rightarrow \text{VAL}^\circ \cup \{ [ex, v] \mid ex \in \chi, v \in \text{VAL}^\circ \}$$

The inclusion of exception packets (arising from raised exceptions) in the range of δ allows for the inclusion of function constants like integer division which do not produce closed values on all members of their input domain. Thus an expression such as $(/ 1 0)$ can raise an exception Div assuming such a defined exception name in the global exception name environment χ . The requirement that constant functions be defined on all members of their input domain is known as δ -typability, which is defined formally in **Definition 5-5**.

$ \begin{aligned} E &::= [] \mid E e \mid v E \mid (E . e) \mid (v . E) \mid \text{let } x = E \text{ in } e \mid \rho\theta.E \mid \\ &\quad \text{raise } E e \mid \text{raise } ex E \mid e \text{ handle } ex E \mid E \text{ handle } ex v \\ R &::= [] \mid R e \mid v R \mid (R . e) \mid (v . R) \mid \text{let } x = R \text{ in } e \mid \\ &\quad \text{raise } R e \mid \text{raise } ex R \mid e \text{ handle } ex R \mid R \text{ handle } ex v \end{aligned} $
--

Figure 5.6 – Grammars for evaluation contexts.

The evaluation contexts E for λ_{ev} are defined in Figure 5.6. Also defined are R contexts which are required for reductions involving reference variables. The R contexts ensure that there is a minimal ordering on reductions involving creation, dereferencing and assignment of reference variables. The R contexts do not include the exception term, in order to prevent exceptions escaping their binding site. The E and R contexts ensure that all operations in the calculus are evaluated in a leftmost-outermost order, which is consistent with Standard ML. For example, functions are evaluated first, then the function argument, and then the function is applied to the argument, resulting in call-by-value evaluation for λ_{ev} . Sequential evaluation of statements separated by semicolons can be encoded with some syntactic sugar by applying the operation `snd` to select the second member of a pair expression (which will firstly require evaluation of both components to become values).

$$(;)\quad e_1; e_2 \equiv_{def} \text{snd}(e_1 . e_2)$$

Lemma 5-1 If $E[e]$ is a closed term, then either e is a closed term, or E is of the form $\rho\theta.E'$ and $\rho\theta.e$ is closed.

Proof. Examining the definitions of evaluation contexts, it should be clear that if x is free in e , then, since $\rho\theta.E'$ is the only construct in E which could bind x , either x must also be free in $E[e]$ (which is a contradiction) or there exists $\langle x, v \rangle \in \theta$ which binds x in $\rho\theta.e$. The rules defining $FV(e)$ illustrate that the `let` expression does not bind x in E . ■

The rules of reduction involving evaluation contexts for the calculus are defined in Figure 5.7. In all the rules except the λ_{ev} -**exception** rule there is an implicit global exception name environment χ which remains unchanged during the reduction. The fixed point combinator Y is included to provide recursion at all types. The reduction involving Y introduces an abstraction around the $(Y v)$ term to ensure that v is applied to a value. The union of these reductions is denoted \mathbf{v} and a reduction in \mathbf{v} is written $\xrightarrow{\mathbf{v}}$. A large proportion of the rules are concerned only with propagating raised exceptions through terms, discarding any further computation until a matching handler is found. Wright and Felleisen's formulation for exceptions performs this rather more elegantly through the use of contexts for raising and handling exceptions. Reductions involving references are defined in Figure 5.8; the union of them is denoted \mathbf{r} and a reduction is written $\xrightarrow{\mathbf{r}}$. The union of both of these is referred to as \mathbf{vr} and $\xrightarrow{\mathbf{vr}}$ is written for a reduction in \mathbf{vr} .

$E[c\ v]$	\xrightarrow{v}	$E[\delta(c,v)]$	$(\lambda_{cv}\text{-}\delta)$
$E[\lambda x(e)\ v]$	\xrightarrow{v}	$E[\{v/x\}e]$	$(\lambda_{cv}\text{-}\beta)$
$E[\text{let } x = v \text{ in } e]$	\xrightarrow{v}	$E[\{v/x\}e]$	$(\lambda_{cv}\text{-}\text{let})$
$E[Y\ v]$	\xrightarrow{v}	$E[v\ (\lambda x((Y\ v))\ x)]$	$(\lambda_{cv}\text{-}Y)$
$\chi, E[\text{exception } x \text{ in } e]$	\xrightarrow{v}	$\chi + ex, E[\{ex/x\}e] \quad ex \notin \chi$	$(\lambda_{cv}\text{-}\text{exception})$
$E[\text{raise } ex\ v]$	\xrightarrow{v}	$E[ex, v]$	$(\lambda_{cv}\text{-}\text{raise})$
$E[v_1 \text{ handle } ex\ v_2]$	\xrightarrow{v}	$E[v_1]$	$(\lambda_{cv}\text{-}\text{nohandle})$
$E[ex, v \text{ handle } ex\ v']$	\xrightarrow{v}	$E[v'\ v]$	$(\lambda_{cv}\text{-}\text{handle})$
$E[ex_1, v \text{ handle } ex_2\ v']$	\xrightarrow{v}	$E[ex_1, v] \quad ex_1 \neq ex_2$	$(\lambda_{cv}\text{-}\text{reraise})$
$E[exn\ e]$	\xrightarrow{v}	$E[exn]$	λ_{cv} exception propagation rules
$E[v\ exn]$	\xrightarrow{v}	$E[exn]$	
$E[(exn\ .\ e)]$	\xrightarrow{v}	$E[exn]$	
$E[(v.\ exn)]$	\xrightarrow{v}	$E[exn]$	
$E[\text{let } x = exn \text{ in } e]$	\xrightarrow{v}	$E[exn]$	
$E[\text{raise } exn\ e]$	\xrightarrow{v}	$E[exn]$	
$E[\text{raise } ex\ exn]$	\xrightarrow{v}	$E[exn]$	
$E[e \text{ handle } ex\ exn]$	\xrightarrow{v}	$E[exn]$	

Figure 5.7 – Rules of reduction with E contexts in λ_{cv} .

$R[\text{ref } v]$	\xrightarrow{r}	$R[\rho\langle x, v \rangle . x]$	$(\lambda_{cv}\text{-}\text{ref})$
$\rho\theta\langle x, v \rangle . R[!x]$	\xrightarrow{r}	$\rho\theta\langle x, v \rangle . R[v]$	$(\lambda_{cv}\text{-}\text{deref})$
$\rho\theta\langle x, v_1 \rangle . R[:= x\ v_2]$	\xrightarrow{r}	$\rho\theta\langle x, v_2 \rangle . R[v_2]$	$(\lambda_{cv}\text{-}\text{assign})$
$R[\rho\theta_1 . \rho\theta_2 . e]$	\xrightarrow{r}	$R[\rho\theta_1 \theta_2 . e]$	$(\lambda_{cv}\text{-}\rho_{\text{merge}})$
$R[\rho\theta . e]$	\xrightarrow{r}	$\rho\theta . R[e] \quad \text{if } R \neq []$	$(\lambda_{cv}\text{-}\rho_{\text{lift}})$

Figure 5.8 – Rules of reduction for references with R contexts in λ_{cv} .

Definition 5-2 (\xrightarrow{vr}) The sequential evaluation relation is the smallest relation “ \xrightarrow{vr} ” satisfying the rules of reduction given in Figure 5.7 and Figure 5.8. The transitive closure of \xrightarrow{vr} is \xrightarrow{vr}^* .

The partial function *eval* is defined on closed expressions to be:

Definition 5-3 (eval) $eval(e) = a$ iff $\chi, e \xrightarrow{vr}^* \chi', a$.

The *answers* that may be produced as a result of evaluation are defined to be:

Definition 5-4 (answers) $a ::= \{\rho\theta.\} v \mid \{\rho\theta.\} [ex, v]$ where $ex \in \chi'$

(where the phrases within $\{\}$ may be omitted).

Essentially, this statement may be read as being that answers consist of either a value (possibly bound within some memory) or an exception packet (again with the value possibly bound within some memory). The global bindings of exception names χ and χ' may be empty in the value case but χ' must contain at least the name of the exception that is raised when the answer is an unhandled exception packet.

5.4.3 Type system

The motivation for proving type soundness of a type system is that evaluation of a program for which a type has been inferred using the type system does not ever produce a “wrong” answer. The concept of a “wrong” answer in the context of λ_{ev} includes attempting to apply a function to incorrectly-typed arguments or attempting to apply a non-function to some value. The basic approach followed in proving type soundness for the polymorphic type system of λ_{ev} is that developed in [WF91] and adopted by Reppy for CML [Rep92]. The type system is a deductive proof system which assigns types to λ_{ev} terms. Polymorphism is achieved through the rules for `let` constructs. The system developed by Tofte [Tof88] is used to prevent overly generous type assignments for references and exceptions that could lead to runtime errors.

Subject reduction is a method of reducing terms of the calculus according to some rules of reduction. Each term is rewritten and since the program itself is a term, each reduction of the program term is also a program in its own right. The rewritten terms correspond to intermediate states of evaluation of the program. The rules for reducing terms are purely syntactic. At each intermediate state of the evaluation, if subject reduction can be proved to hold, then the type of the rewritten term remains unchanged.

The basic definitions for the type system are introduced, including the type inference rules. Then the standard proof strategy of [WF91] is followed, which proceeds by demonstrating subject reduction for the calculus, characterising the possible answers from evaluation and the faulty expressions, and finally proving that faulty expressions are untypable. Notions of weak and strong soundness are defined, which characterise the results which may be produced by typable expressions.

5.4.3.1 Definitions

The types for λ_{ev} are formed from type constants and type variables, whose basic syntax is given in Figure 5.9. Following Tofte [Tof88], type variables are split into imperative type variables and applicative type variables, such that an imperative type may not contain any applicative type variables. Only values with imperative types will be capable of being stored in reference cells. Similarly, only parameters with imperative types will be permitted for raised exceptions. The distinctions between

$\iota \in \text{TYCON} = \{\text{unit}, \text{bool}, \text{int}, \dots\}$	type constants
$u \in \text{IMPTYVAR}$	imperative type variables
$t \in \text{APPTYVAR}$	applicative type variables
$\alpha, \beta \in \text{TYVAR} = \text{IMPTYVAR} \cup \text{APPTYVAR}$	type variables

Figure 5.9 – Basic syntax for type constants and variables.

imperative and applicative type variables are a solution to problems arising from overly generous type assignments in the polymorphic type system.³

The set of *types* ($\tau \in \text{TY}$) is defined:

$\tau ::= \iota$	type constant
α	type variable
$(\tau_1 \rightarrow \tau_2)$	function type
$(\tau_1 \times \tau_2)$	pair type
$\tau \text{ ref}$	reference cell type
$\tau \text{ exn}$	exception type

Type schemes bind type variables in types, similar to the way in which a λ -abstraction binds a variable throughout the subsequent expression. The set of type schemes, $\sigma \in \text{TYScheme}$, is defined as:

$\sigma ::= \tau$
$\forall \alpha. \sigma$

The type scheme $\forall \alpha_1 \dots \forall \alpha_n. \sigma$ is abbreviated as $\forall \alpha_1 \dots \alpha_n. \sigma$, and the type variables $\alpha_1, \dots, \alpha_n$ are considered bound. The type scheme $\forall. \tau$ is identified with τ , so all types are also type schemes. Type schemes represent the set of types τ that can be obtained by substituting for the bound type variables α_1 to α_n . Type schemes are required because of the polymorphic type system, which allows variables to be bound to polymorphically-typed functions. Type schemes then allow the variable to be used with a set of different types. The simplest example is the identity function, $\lambda x(x)$, bound to a variable i . For values of type *real*, i can be used as a function of type $\text{real} \rightarrow \text{real}$, while for values of type *bool*, i can be used as a function of type $\text{bool} \rightarrow \text{bool}$. The set of such types is represented by the type scheme $\forall \alpha. \alpha \rightarrow \alpha$. If a type variable is not bound in σ then it is regarded as free. The free type variables of a type scheme σ are written $\text{FTV}(\sigma)$.

³ In the recently revised definition of Standard ML, these distinctions are eliminated and replaced with a *value restriction* [SML96]. This restriction permits generalisation of the type of a variable binding only in circumstances where the expression on the right hand side is constructed from constants, other variables, lambda expressions, or combinations of these built with base-level combining operators. Since the semantics being described here were developed to model paraML, built with a version of Standard ML which predates the revised definition, the new restrictions were not adopted.

The set of *imperative types* is defined:

$$\psi \in \text{IMPTY} = \{\tau \mid \text{FTV}(\tau) \subseteq \text{IMPTYVAR}\}$$

Note that if the free type variables of a type is the empty set, $\text{FTV}(\tau) = \emptyset$, then τ is a *monotype*. From the definition of imperative type, it is clear that all free type variables of an imperative type are also imperative.

The use of imperative and applicative type variables requires a restriction on substitutions that they are only permitted to map imperative type variables to imperative types. This substitution leads naturally to the notion of *generalisation* of a type τ by a type scheme σ , written $\sigma \succ \tau$. A generalisation exists if there is some substitution S of the bound type variables of σ which yields τ , in which case τ is said to be an instance of σ . The substitution S is a partial function from type variables to types and is capture avoiding.

$$(\succ) \forall \alpha_1 \dots \alpha_n. \tau' \succ \tau \quad \text{iff} \quad \exists S. S\tau' = \tau \text{ and } \text{dom}(S) = \{\alpha_1, \dots, \alpha_n\}.$$

Generalisation is also applicable to type schemes, and provides a partial ordering on them.

$$(\succ) \sigma' \succ \sigma \quad \text{iff} \quad \text{wherever } \sigma \succ \tau \text{ then } \sigma' \succ \tau.$$

Type environments are used for giving type information about variables in open expressions. Since the semantics is based on syntactic rewriting of terms, it is necessary to assign types to intermediate stages of computation. Thus the type system must be able to assign types to terms such as exception names. A *type environment* is a pair of finite maps. The first is a map from variables to type schemes, defined as:

$$\text{VT} \in \text{VARTY} = \text{VAR} \xrightarrow{\text{fin}} \text{TYSCHEME}$$

The second is a map from exception names to imperative types (of the parameters that are passed to the raised exception of the corresponding name):

$$\text{ET} \in \text{EXNTY} = \text{EXNNAME} \xrightarrow{\text{fin}} \text{IMPTY}$$

Type environments are defined:

$$\text{TE} = (\text{VT}, \text{ET}) \in \text{TYENV} = (\text{VARTY} \times \text{EXNTY})$$

The free type variables of variable typings and exception typings are denoted $\text{FTV}(\text{VT})$ and $\text{FTV}(\text{ET})$ respectively. The free type variables of a variable typing are the union of free type variables of all the type schemes within the range of the map, defined:

$$\text{FTV}(\text{VT}) = \bigcup_{\sigma \in \text{rng}(\text{VT})} \text{FTV}(\sigma)$$

Note that there are no bound type variables in exception typings, and also that $\text{FTV}(\text{ET}) \subseteq \text{IMPTYVAR}$. The free type variables of a type environment are thus defined:

$$\text{FTV}(\text{TE}) = \text{FTV}(\text{VT}) \cup \text{FTV}(\text{ET})$$

Type environment modifications can be conveniently expressed as follows:

$$\begin{aligned} \text{TE} \pm \{x \mapsto \sigma\} &\equiv_{\text{def}} (\text{VT} \pm \{x \mapsto \sigma\}, \text{ET}) \\ \text{TE} \pm \{ex \mapsto \psi\} &\equiv_{\text{def}} (\text{VT}, \text{ET} \pm \{ex \mapsto \psi\}) \end{aligned}$$

The separation of imperative and applicative type variables leads to two forms of closures of types with respect to type environments. The *closure* of type τ with respect to type environment TE is written:

$$\text{CLOS}_{\text{TE}}(\tau) = \forall \alpha_1 \dots \alpha_n. \tau \text{ where } \{\alpha_1 \dots \alpha_n\} = \text{FTV}(\tau) \setminus \text{FTV}(\text{TE})$$

The *applicative closure* of type τ with respect to type environment TE is written:

$$\begin{aligned} \text{APPCLOS}_{\text{TE}}(\tau) &= \forall \alpha_1 \dots \alpha_n. \tau \\ &\text{where } \{\alpha_1 \dots \alpha_n\} = (\text{FTV}(\tau) \setminus \text{FTV}(\text{TE})) \cap \text{APPTYVAR} \end{aligned}$$

Two facts may be deduced from the definitions of type closures and generalisation which will be used in later proofs.

Lemma 5-2 The following properties hold for any TE, σ , σ' , τ , and x :

- If $\sigma' \succ \sigma$, then $\text{CLOS}_{\text{TE} \pm \{x \mapsto \sigma'\}}(\tau) \succ \text{CLOS}_{\text{TE} \pm \{x \mapsto \sigma\}}(\tau)$.
- If $x \notin \text{dom}(\text{TE})$, then $\text{CLOS}_{\text{TE}}(\tau) \succ \text{CLOS}_{\text{TE} \pm \{x \mapsto \sigma\}}(\tau)$.

Proof. Both of these properties arise from observing that if $\text{FTV}(\text{TE}') \subseteq \text{FTV}(\text{TE})$, then $\text{CLOS}_{\text{TE}'}(\tau) \succ \text{CLOS}_{\text{TE}}(\tau)$. ■

Type judgements are the sentences which may be inferred from the type system's rules. These judgements are of the form $\text{TE} \vdash e : \tau$, which are read as meaning that the expression e has type τ under the assumptions of type environment TE. If the type environment is empty, then judgements may be written $\vdash e : \tau$. *Well-typed programs* are closed expressions where a judgement exists in the form $\vdash e : \tau$. The type rules form the basis of the type system, and allow sentences of the form $\text{TE} \vdash e : \tau$ to be inferred.

A function `TypeOf` is assumed to exist, which is used to abstract the details of typing every individual base and function constant.

$$\text{TypeOf} : \text{CONST} \rightarrow \text{TYScheme}$$

An important property of the δ function (described earlier in §5.4.2) is δ -*typability*. This property guarantees that δ is defined on all constants of function type and arguments of matching type. Formally, δ -typability is defined:

Definition 5-5 (δ -typability) If $\text{TypeOf}(c) \succ (\tau' \rightarrow \tau)$ and $\vdash v : \tau'$, then $\delta(c, v)$ is defined and $\vdash \delta(c, v) : \tau$.

$\frac{\text{TypeOf}(c) \succ \tau}{\text{TE} \vdash c : \tau}$	$\tau\text{-const}$
$\frac{\text{VT}(x) = \tau}{(\text{VT}, \text{ET}) \vdash x : \tau}$	$\tau\text{-var}$
$\frac{\text{ET}(ex) = \psi}{(\text{VT}, \text{ET}) \vdash ex : \psi}$	$\tau\text{-ex}$
$\text{TE} \vdash Y : ((\tau_1 \rightarrow \tau_2) \rightarrow \tau_1 \rightarrow \tau_2) \rightarrow \tau_1 \rightarrow \tau_2$	$\tau\text{-Y}$
$\frac{\text{TE} \vdash e_1 : (\tau' \rightarrow \tau) \quad \text{TE} \vdash e_2 : \tau'}{\text{TE} \vdash e_1 e_2 : \tau}$	$\tau\text{-app}$
$\frac{\text{TE} \pm \{x \mapsto \tau\} \vdash e : \tau'}{\text{TE} \vdash \lambda x(e) : (\tau \rightarrow \tau')}$	$\tau\text{-abs}$
$\frac{\text{TE} \vdash e_1 : \tau_1 \quad \text{TE} \vdash e_2 : \tau_2}{\text{TE} \vdash (e_1 . e_2) : (\tau_1 \times \tau_2)}$	$\tau\text{-pair}$
$\frac{\text{TE} \vdash v : \tau' \quad \text{TE} \pm \{x \mapsto \text{CLOS}_{\text{TE}}(\tau')\} \vdash e : \tau}{\text{TE} \vdash \text{let } x = v \text{ in } e : \tau}$	$\tau\text{-app-let}$
$\frac{\text{TE} \vdash e_1 : \tau' \quad \text{TE} \pm \{x \mapsto \text{APPCLOS}_{\text{TE}}(\tau')\} \vdash e_2 : \tau}{\text{TE} \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}$	$\tau\text{-imp-let}$
$\text{TE} \vdash \text{ref} : \tau \rightarrow \tau \text{ ref} \quad \text{if } \tau \text{ is imperative}$	$\tau\text{-ref}$
$\text{TE} \vdash ! : \tau \text{ ref} \rightarrow \tau$	$\tau\text{-deref}$
$\text{TE} \vdash := : \tau \text{ ref} \rightarrow \tau \rightarrow \tau$	$\tau\text{-assign}$
$\frac{\text{TE} \pm \{x_1 \mapsto \tau_1 \text{ ref}, \dots, x_n \mapsto \tau_n \text{ ref}\} \vdash e : \tau \quad \text{TE} \pm \{x_1 \mapsto \tau_1 \text{ ref}, \dots, x_n \mapsto \tau_n \text{ ref}\} \vdash v_i : \tau_i \quad \tau_i \text{ is imperative} \quad 1 \leq i \leq n}{\text{TE} \vdash \rho \langle x_1, v_1 \rangle \dots \langle x_n, v_n \rangle . e : \tau}$	$\tau\text{-rho}$
$\frac{\text{TE} \pm \{x \mapsto \tau \text{ exn}\} \vdash e : \tau' \quad \tau \text{ is imperative}}{\text{TE} \vdash \text{exception } x \text{ in } e : \tau'}$	$\tau\text{-exn}$
$\text{TE} \vdash \text{raise} : \tau_1 \text{ exn} \rightarrow \tau_1 \rightarrow \tau_2$	$\tau\text{-raise}$
$\frac{\text{TE} \vdash e : \tau_2}{\text{TE} \vdash e \text{ handle} : \tau_1 \text{ exn} \rightarrow (\tau_1 \rightarrow \tau_2) \rightarrow \tau_2}$	$\tau\text{-handle}$

Figure 5.10 – Type rules for λ_{ev} .

The provision of exceptions in the range of the δ function allows the inclusion of functions like division, since the answer may be a raised exception, not just a closed value. The type of a raised exception matches the type of the value that would be returned in unexceptional circumstances, as is seen from the type rules.

5.4.3.2 Type rules

The typing rules for λ_{cv} are given in Figure 5.10. These rules cover all the expressions defined in the calculus, and from these rules, proofs of the types of arbitrary closed expressions can be deduced. Worth noting are the τ -**app-let** and τ -**imp-let** rules, which are defined separately to prevent the generalisation of variables in the store. The basic premise is that if the right hand side of the variable being bound is known to be a value, then evaluation of it cannot generate new memory cells, and thus it is safe to generalise any imperative type variables in its type since they cannot also generalise the type of a value in the memory. However, if the right hand side is an expression, then unrestricted generalisation of the type variables may lead to problems, and thus generalisation is conservatively restricted to applicative type variables.

5.4.4 Type soundness

The type system given above is only one aspect of the typing for λ_{cv} . The next step is to establish type soundness of the system with respect to the dynamic semantics given in §5.4.2. As mentioned earlier, the approach taken is that set out in [WF91] and [Rep92].

5.4.4.1 Supporting lemmas

Before establishing subject reduction for λ_{cv} , there are several important lemmas required in the proof. The first lemma establishes that variables or exception names in the domain of the typing environment which are not free in an expression e can be ignored when typing e . The variable convention ensures that $x \notin \text{FV}(e)$ whenever the lemma applies.

Lemma 5-3 If $x \notin \text{FV}(e)$ then $\text{TE} \vdash e:\tau$ iff $\text{TE} \pm \{x \mapsto \sigma\} \vdash e:\tau$. Likewise, if $ex \notin \text{FEN}(e)$ then $\text{TE} \vdash e:\tau$ iff $\text{TE} \pm \{ex \mapsto \psi\} \vdash e:\tau$.

Proof. The proof is straightforward, and follows from induction on the height of the typing deduction. ■

The next lemma is known as the replacement lemma, since it allows the replacement of a complete subexpression in a term with another of the same type, without having any effect on the type of the whole term.

Lemma 5-4 (Replacement) Let $C[\]$ be a context with a hole. Provided the following conditions hold:

1. D is a type deduction which concludes $\text{TE} \vdash C[e_1] : \tau$,

2. D_1 is a subdeduction of D which concludes $TE' \vdash e_1 : \tau'$,
3. D_1 occurs in D in the position which corresponds to the hole in C ,
4. $TE' \vdash e_2 : \tau'$,

then $TE \vdash C[e_2] : \tau$.

Proof. The proof comes from [HS86] and is echoed in [WF91]. Consider the deduction D as a tree, with the conclusion as the root. The deduction D_2 which concludes $TE' \vdash e_1 : \tau'$, may also be viewed as a tree, and it can replace the subtree D_1 in D . All occurrences of e_1 in D can then be replaced by e_2 in the resulting tree. This tree is then a valid deduction also, but concludes $TE \vdash C[e_2] : \tau$, by induction on the height of the tree. ■

The substitution lemma is used to show that types are preserved under β -reduction. The proof of this lemma relies on an additional two lemmas, which are stated here. The first of these lemmas extends substitution on types to substitution on type judgements.

Lemma 5-5 If S is a substitution and $TE \vdash e : \tau$, then $S(TE) \vdash e : S_\tau$.

Proof. A proof of this lemma for a similar system may be found in [Tof90]. ■

The next lemma is used to show that generalising the typing assumptions (the information in the type environment) has no effect on the typing outcome of an expression.

Lemma 5-6 If $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$ and $\sigma' \succ \sigma$, then $TE \pm \{x \mapsto \sigma'\} \vdash e : \tau$.

Proof. The proof of this lemma is by induction on the height of the typing deduction of $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$, and case analysis on the shape of e for the last step. The details of this proof are given in Appendix A-1. ■

Lemma 5-7 (Substitution) If $x \notin FV(v)$, $TE \vdash v : \tau$, and $TE \pm \{x \mapsto \forall \alpha_1 \dots \forall \alpha_n. \sigma\} \vdash e : \tau'$, with $\{\alpha_1, \dots, \alpha_n\} \cap FTV(TE) = \emptyset$, then $TE \vdash \{v/x\}e : \tau'$.

Proof. Just as in the previous lemma, the proof of the substitution lemma is by induction on the height of the typing deduction, and case analysis on the shape of e for the last step. The details of the proof are given in Appendix A-1. ■

5.4.4.2 Type preservation

The core result for establishing syntactic soundness is type preservation. Essentially, this states that if an expression can be assigned a type, then that type is preserved through reduction of the expression.

Theorem 5-8 (Type preservation) For any type environment TE , expression e_1 , and type τ , such that $TE \vdash e_1 : \tau$, if $e_1 \xrightarrow{vr} e_2$ then $TE \vdash e_2 : \tau$.

Proof. Let $E[e]=e_1$ and $E[e']=e_2$, and assume that $TE' \vdash e:\tau'$ with $TE'=(VT',ET')$. Then by **Replacement (Lemma 5-4)**, it is sufficient to show that $TE' \vdash e':\tau'$. This is done by case analysis of the definition of \xrightarrow{vr} . Details of this proof are found in Appendix A-1. ■

5.4.4.3 Stuck expressions

The type preservation theorem guarantees that if an expression is typable, any series of reductions will fail to produce an untypable expression. However, this is not sufficient to establish type soundness. The crucial property to establish is that typable expressions do not become *stuck*.

Definition 5-6 (Stuck expressions) The evaluation of an expression e is stuck if e is not an answer and there is no e' such that $e \xrightarrow{vr} e'$.

5.4.4.4 Faulty expressions

The concept of stuck expressions is a semantic one. In particular, it is not decidable whether an expression will eventually reduce to a stuck expression. A useful approximation of the set of stuck expressions is a set of *faulty* expressions. These faulty expressions may become stuck (though they may not). Hence faulty expressions are a superset of the stuck expressions.

Definition 5-7 (Faulty expressions) The faulty expressions of λ_{ev} are those expressions containing a subexpression of the form:

$(c\ v)$	where $\delta(c,v)$ is undefined
$((v_1\ .\ v_2)\ v)$	
$(!\ v)$	where $v \notin \text{VAR}$
$(:=\ v)$	where $v \notin \text{VAR}$
$\rho\theta\langle x, v_2 \rangle. C[x\ v_1]$	
exception x in $C[!\ x]$	
exception x in $C[:=\ x]$	
exception x in $C[x\ v]$	
$(\text{raise}\ v_1\ v_2)$	where $v_1 \notin \text{VAR} \cup \text{EXNNAME}$
$(e\ \text{handle}\ v_1\ v_2)$	where $v_1 \notin \text{VAR} \cup \text{EXNNAME}$
$\rho\theta\langle x, v \rangle. C[\text{raise}\ x\ v']$	
$\rho\theta\langle x, v \rangle. C[e\ \text{handle}\ x\ v']$	

where

$$\begin{aligned}
 C ::= & [] \mid C e \mid e C \mid (C.e) \mid (e.C) \mid \text{let } x = C \text{ in } e \mid \text{let } x = e \text{ in } C \mid \\
 & \lambda x(C) \mid \rho\theta.C \mid \rho\theta(x,C).e \mid \text{exception } x \text{ in } C \mid C \text{ handle } ex \ v \mid \\
 & e \text{ handle } ex \ C \mid \text{raise } C e \mid \text{raise } ex \ C
 \end{aligned}$$

5.4.4.5 Uniform evaluation

Evaluation can be characterised by the following uniform evaluation lemma. Firstly the notation $e \Uparrow$ for the divergence of e is defined:

Definition 5-8 (Divergence \Uparrow) If $e \xrightarrow{\text{vr}} e'$ for some e' and for all e' such that $e \xrightarrow{\text{vr}} * e'$ there exists an e'' such that $e' \xrightarrow{\text{vr}} e''$, then $e \Uparrow$.

Lemma 5-9 (Uniform evaluation) For closed expressions e , if no e' exists such that $e \xrightarrow{\text{vr}} e'$ and e' is faulty, then either $e \Uparrow$ or $e \xrightarrow{\text{vr}} * a$ where the answer $a = \{\rho\theta.\} v \mid \{\rho\theta.\} [ex, v]$ with $ex \in \chi$.

Proof. By induction on the length of the reduction sequence. The proof requires that one of the following holds for any expression e : e is faulty, $e \xrightarrow{\text{vr}} e'$ and e' is closed, or e is an answer a . See Appendix A-1 for details of this proof. ■

5.4.4.6 Results

A number of results can now be proven that establish type soundness for λ_{ev} .

Lemma 5-10 (Faulty expressions are untypable) If e is faulty, then there are no TE, τ such that $\text{TE} \vdash e : \tau$.

Proof. It is sufficient to show that the subexpressions e' of e that cause e to be faulty are untypable. The proof proceeds by case analysis on the form of the subexpression e' . See Appendix A-1 for the full details of this proof. ■

Theorem 5-11 (Syntactic soundness) Let e be a program, with $\vdash e : \tau$. Then either $e \Uparrow$ or $e \xrightarrow{\text{vr}} * a$ and $\vdash a : \tau$.

Proof. By **Uniform evaluation** (Lemma 5-9), either $e \xrightarrow{\text{vr}} e'$ and e' is faulty, or $e \Uparrow$, or $e \xrightarrow{\text{vr}} * a$. Since $\text{TE} \vdash e : \tau$, **Type preservation** (Theorem 5-8) implies $\text{TE} \vdash a : \tau$ and $\text{TE} \vdash e' : \tau$. Suppose that $e \xrightarrow{\text{vr}} e'$ and e' is faulty. But faulty expressions are untypable by Lemma 5-10, and thus $\text{TE} \vdash e' : \tau$ is a contradiction and cannot occur. Hence either $e \Uparrow$ or $e \xrightarrow{\text{vr}} * a$ and $\text{TE} \vdash a : \tau$. ■

The definition of evaluation is refined to permit statements about strong and weak soundness. Specifically, characterisations of “wrong” answers are necessary, to distinguish between programs that diverge and those which result in type errors. Weak soundness guarantees that a typable program does not produce a “wrong” answer, while strong soundness states that if an answer is produced, then its type is the same as the type of the program.

Definition 5-9 (eval')
$$eval'(e) = \begin{cases} \text{WRONG} & \text{if } e \xrightarrow{vr}^* e' \text{ and } e' \text{ is faulty;} \\ a & \text{if } e \xrightarrow{vr}^* a. \end{cases}$$

Strong and weak soundness now follow as corollaries of **Syntactic soundness** (Theorem 5-11).

Theorem 5-12 (Strong soundness) If $\vdash e : \tau$ and $eval'(e') = a$ then $\vdash a : \tau$. ■

Theorem 5-13 (Weak soundness) If $\vdash e : \tau$ then $eval'(e) \neq \text{WRONG}$. ■

5.5 Summary

The purpose of this chapter has been to present the complete development of an extended λ_v calculus with respect to both its static and dynamic semantics. This λ_v calculus is used as the sequential language within processes, and is very similar to ML. The central result of type soundness establishes that typable programs do not result in runtime type errors.

The λ_v calculus, the style of semantics and general notation are used throughout the next two chapters as the basis in developing a formal semantics to model paramL. Although the dynamic operational semantics is extended considerably to model the process-oriented aspects of paramL, the general structure of the proof of type soundness remains essentially unaltered.

6. Operational semantics

6.1 Introduction

This chapter extends the λ_{ev} calculus introduced in the previous chapter to model evaluation in paraML. The sequential aspects of the calculus remain essentially unchanged, but a parallel evaluation relation is introduced which models the extension operations provided in paraML. This new calculus is known as λ_{pv} . The development of an operational semantics to characterise parallel evaluation is similar to the work of Reppy [Rep92] in developing an operational semantics for the concurrency aspects of CML. Parallel evaluation proceeds by transitions between process configurations, which draws from the Chemical Abstract Machine (CHAM) model of parallel evaluation developed by Berry and Boudol [BB92]. The λ_{ev} calculus does not completely model Standard ML, and hence λ_{pv} does not completely model paraML. However, the core extension operations of paraML are captured. The interaction of exceptions and references with these process-oriented aspects of the λ_{pv} calculus is particularly novel.

6.2 Syntax

The description of the syntax for λ_{pv} is given just as extensions to the syntax for λ_{ev} . The syntax is extended firstly with new ground terms for process names and port names as given in Figure 6.1. Process names and port names appear only as intermediate results in evaluation.

The new expressions in the calculus (Figure 6.2) include expressions which match each of the extension operations in paraML and the new values are process names and port names. Notice that the extension operations are defined as expressions, not function constants, since they will be given meaning by the parallel evaluation relation, not by the sequential evaluation relation like the other function constants.

The main differences between paraML and λ_{pv} are the formulations for process and port creation. These formulations have been chosen to assist in the typing rules of λ_{pv} described in the next chapter. Note that `pvt` is a contraction of `port` (not “print”), and should be pronounced the same way. As mentioned in the previous chapter, there are no constant exception names. The exception identifiers used for indicating failure in the paraML operations are `NoPortCreated`, `ProcessExecuting`, `NoSuchProcess`, `NoSuchPort`, and `PortNotOwned`.

π	\in PROCESSNAME	process names
ϕ	\in PORTNAME	port names

Figure 6.1 – New ground terms.

e	$::=$	<code>proc x in e</code>	process creation
		<code>prt x on π in e</code>	port creation
		<code>execute e</code>	execute request
		<code>self_id e</code>	own process name
		<code>send e</code>	send value to port
		<code>recv e</code>	receive value from port
		<code>probe e</code>	test ability to receive from port
v	$::=$	π	process name
		ϕ	port name

Figure 6.2 – New expressions and values of the grammar.

The paraML forms of the operations `proc` and `prt`, together with these exceptions, can be achieved by embedding a λ_{pv} program e in the following context:

```
let process =  $\lambda x(\text{proc } p \text{ in } p)$  in
  let port =  $\lambda x(\text{prt } f \text{ on } x \text{ in } f)$  in
    exception NoPortCreated in
      exception ProcessExecuting in
        exception NoSuchProcess in
          exception NoSuchPort in
            exception PortNotOwned in
              [e]
```

The definition of free variables remains as before, with the addition of those for the new expressions and values. There are no free variables in the terms for process names and port names. The free variables of the new expressions are just those of the argument expressions e , as shown in Figure 6.3.

As in λ_{ev} the set VAL° is the set of closed value terms. However, closed value terms may contain both free process names and free port names, as well as containing free exception names as before. The free process names are denoted $\text{FPN}(e)$ and the free port names are denoted $\text{FFN}(e)$. As with exception names in λ_{ev} , since there are no process name or port name binding forms, $\text{FPN}(e)$ is exactly the set of process names that appear in e and $\text{FFN}(e)$ is exactly the set of port names that appear in e . A *program* is a closed term which does not contain any subterms in the syntactic classes EXN, PROCESSNAME or PORTNAME. Thus programs do not contain intermediate values.

$FV(\pi)$	$=$	\emptyset
$FV(\phi)$	$=$	\emptyset
$FV(\text{proc } x \text{ in } e)$	$=$	$FV(e) \setminus \{x\}$
$FV(\text{prt } x \text{ on } \pi \text{ in } e)$	$=$	$FV(e) \setminus \{x\}$
$FV(\text{execute } e)$	$=$	$FV(e)$
$FV(\text{self_id } e)$	$=$	$FV(e)$
$FV(\text{send } e)$	$=$	$FV(e)$
$FV(\text{recv } e)$	$=$	$FV(e)$
$FV(\text{probe } e)$	$=$	$FV(e)$

Figure 6.3 – Free variables in new terms.

$\{e/x\}\pi$	$=$	π
$\{e/x\}\phi$	$=$	ϕ
$\{e/x\}(\text{proc } x' \text{ in } e')$	$=$	$\text{proc } x' \text{ in } \{e/x\}e' \quad x \neq x'$
$\{e/x\}(\text{prt } x' \text{ on } \pi \text{ in } e')$	$=$	$\text{prt } x' \text{ on } \pi \text{ in } \{e/x\}e' \quad x \neq x'$
$\{e/x\}(\text{execute } e')$	$=$	$\text{execute } \{e/x\}e'$
$\{e/x\}(\text{self_id } e')$	$=$	$\text{self_id } \{e/x\}e'$
$\{e/x\}(\text{send } e')$	$=$	$\text{send } \{e/x\}e'$
$\{e/x\}(\text{recv } e')$	$=$	$\text{recv } \{e/x\}e'$
$\{e/x\}(\text{probe } e')$	$=$	$\text{probe } \{e/x\}e'$

Figure 6.4 – Substitution in new terms.

Substitution in the new terms is straightforward. As before, the bound variables in the `proc` and `prt` rules are renamed to prevent any instance of $x = x'$ affecting substitution according to the conventions of Barendregt [Bar84]. The definitions are given in Figure 6.4.

6.3 Dynamic semantics

The λ_{pv} calculus is defined by a sequential evaluation and a parallel evaluation relation. The sequential evaluation relation " \longrightarrow " defines what goes on inside a process. The relation " \longrightarrow " is " $\xrightarrow{\tau}$ " with some additional contexts and reduction rules. The parallel evaluation relation " \Rightarrow " extends sequential evaluation to finite sets of processes as well as defining how processes come into existence and how they exchange information.

$ \begin{aligned} E &::= [] \mid E e \mid v E \mid (E.e) \mid (v.E) \mid \text{let } x = E \text{ in } e \mid \\ &\quad \text{raise } E e \mid \text{raise } ex E \mid e \text{ handle } ex E \mid E \text{ handle } ex v \mid \rho\theta.E \mid \\ &\quad \text{execute } E \mid \text{self_id } E \mid \text{send } E \mid \text{recv } E \mid \text{probe } E \\ R &::= [] \mid R e \mid v R \mid (R.e) \mid (v.R) \mid \text{let } x = R \text{ in } e \mid \\ &\quad \text{raise } R e \mid \text{raise } ex R \mid e \text{ handle } ex R \mid R \text{ handle } ex v \\ &\quad \text{execute } R \mid \text{self_id } R \mid \text{send } R \mid \text{recv } R \mid \text{probe } R \end{aligned} $

Figure 6.5 – Grammars for contexts.

$E[\text{execute } exn]$	\mapsto	$E[exn]$
$E[\text{self_id } exn]$	\mapsto	$E[exn]$
$E[\text{send } exn]$	\mapsto	$E[exn]$
$E[\text{recv } exn]$	\mapsto	$E[exn]$
$E[\text{probe } exn]$	\mapsto	$E[exn]$

Figure 6.6 – Sequential evaluation relation.

6.3.1 Sequential evaluation

As before, reductions in λ_{pv} involving function constants are defined abstractly using the partial function δ . Similarly, the answers a that may be produced as a result of sequential evaluation of terms are defined as before. Recall too that the answers may possibly be ρ -bound in some memory.

Definition 6-1 (answers) $a \in \text{ANS} = \text{VAL}^\circ \cup \{ [ex, v] \mid ex \in \chi, v \in \text{VAL}^\circ \}$

The sequential evaluation order is controlled by the use of evaluation contexts. The E and R contexts in λ_{pv} are extended to include the new terms (shown in Figure 6.5), excluding those which introduce new process names or port names. Terms involving name creation (such as the `exception`, `proc` and `pvt` terms) do not appear in the evaluation contexts to prevent possible capture of free variables in the hole of the context. As a consequence, the proof of **Lemma 5-1** remains the same, allowing its restatement here:

Lemma 6-1 If $E[e]$ is a closed term, then either e is a closed term, or E is of the form $\rho\theta.E'$ and $\rho\theta.e$ is closed.

Proof. Examining the definitions of evaluation contexts, it should be clear that if x is free in e , then, since $\rho\theta.E'$ is the only construct in E which could bind x , either x must also be free in $E[e]$ or there exists $\langle x, v \rangle \in \theta$ which binds x in $\rho\theta.e$. ■

The sequential evaluation relation from λ_{ev} is extended solely by new exception propagation rules, as shown in Figure 6.6. The additional change is that the λ_{ev} -**exception** rule is removed from the sequential evaluation rules for λ_{pv} . This

alteration is necessary to preserve the global nature of the exception name environment χ , thus making it visible to all processes. Hence the λ_{pv} -**exn** rule for exceptions is found in those for the parallel evaluation relation.

Note that the type system will guarantee that the arguments to the operations involving processes and ports are of the correct type.

6.3.2 Parallel evaluation

Reppy's concurrent evaluation relation is built around a CHAM model [BB92], except without any heating or cooling transitions. The machine is embodied at any one time by a *configuration* which consists of finite sets of *process states* and channel names. The same general approach is taken for λ_{pv} – each process is tagged with a unique name; similarly all ports are tagged with a unique name. A process state is given as the 3-tuple consisting of the process name, the current state of any ports attached to the process, and the *evaluation state* of the process. Tagging each process state with a unique process name avoids having to use the multisets of a CHAM. Configurations in λ_{pv} consist of finite sets of exception names, χ , and process states, \mathcal{P} . The definitions for the various aspects of configurations are given in Figure 6.7, and explained in the following paragraphs.

There are multiple ways to formulate the semantics for exceptions. Reppy's suggested mechanism for modelling exceptions was adopted in λ_{cv} with the express purpose of making their formulation straightforward in λ_{pv} . This technique of providing an implicit global environment for exception names guarantees that they are unique across all processes, despite an ability to escape their binding site in message transmissions. The requirement of a global environment for exceptions is unfortunate as they have no bearing on parallelism; ideally only process names and port names would have constituted an implicit global environment. The set of exception names χ includes every exception name in the set of process states \mathcal{P} .

Communication in CML is synchronous; in paraML, messages are delivered to ports, which act as queues of messages destined for a particular process. This requires some additional notation to describe the state of communication for a port. Queues of messages are denoted $q = [m_1, \dots, m_N]$ and the empty queue is given by \perp ; the infix operation $@$ is used to indicate insertion into a queue. The normal semantics of FIFO queue operations are observed.

A process can be in one of three different evaluation states:

1. awaiting execution to take place (denoted by ε);
2. evaluating a term e (often written as $E[e']$, the sequential evaluation contexts defined earlier in Figure 6.5); or
3. having completed evaluation (such that there is an empty evaluation context) to a closed basic value v or unhandled exception exn , which are collectively known as answers and written a .

χ	$\equiv \{ex_1, ex_2, \dots, ex_N\} \subset \text{EXNNAME}$	exception names
q	$\equiv [m_1, m_2, \dots, m_N] \in \text{QUEUE}$	queues of values
Φ	$\equiv \{\phi_1 \mapsto q_1, \phi_2 \mapsto q_2, \dots, \phi_N \mapsto q_N\}$	a set of maps from port name to queue; $\text{dom}(\Phi) = \{\phi_1, \dots, \phi_N\}$
es	$\in \varepsilon + \text{EXP} + \text{ANS}$	evaluation states
μ	$= \langle \pi; \Phi; es \rangle \in \text{PROC}$	process states
\mathcal{P}	$= \{\mu_1, \mu_2, \dots, \mu_N\}$	sets of process states
χ, \mathcal{P}	$\in \text{Fin}(\text{EXNNAME}), \text{Fin}(\text{PROC})$	configurations
$\text{Terminal}(\mathcal{P})$	$\subseteq \mathcal{P}$, such that $\langle \pi; \Phi; es \rangle \in \text{Terminal}(\mathcal{P})$ iff $es = [a]$ and $a \in \text{ANS}$	
$\text{Awaiting}(\mathcal{P})$	$\subseteq \mathcal{P}$, such that $\langle \pi; \Phi; es \rangle \in \text{Awaiting}(\mathcal{P})$ iff $es = \varepsilon$	

Figure 6.7 – Syntactic definitions for process configuration components.

Terminal processes of a configuration are those which have evaluated to an answer, and *awaiting* processes are those still awaiting an `execute` request. Implicit in the definition of possible evaluation states is that processes will only service at most one `execute` request. The definitions for these attributes of process configurations are given in Figure 6.7. By definition, there are no free variables, free exception names, free process names, or free port names in the awaiting evaluation state. Thus $\text{FV}(\varepsilon) = \emptyset$, $\text{FEN}(\varepsilon) = \emptyset$, $\text{FPN}(\varepsilon) = \emptyset$ and $\text{FFN}(\varepsilon) = \emptyset$.

The set of process names from the process states of a configuration is given as $\text{PROCNAME}(\mathcal{P})$, and the set of port names is given as $\text{PORTNAME}(\mathcal{P})$. The initial configuration of a paraML system is an empty exception name set \emptyset and the singleton process state set $\{\langle \pi_0; \emptyset; E[e] \rangle\}$, where no port names have been allocated, and $E[e]$ is the user program. The initial process name is given as π_0 to distinguish it from processes created by the `proc` operation. This configuration is well-formed.

Parallel evaluation proceeds by transitions between configurations, according to the parallel evaluation relation rules. For some of the rules, particularly those connected with the `execute` and `send` operations, there is an additional operation needed. The `copy` operation captures the semantics of copying a value complete with any information needed to interpret it in a different evaluation context. (It is not a new expression of the language, just a technique for describing value transmission.) Thus the `copy` operation takes a tuple argument consisting of an evaluation context and the value itself, and produces a ρ -bound expression consisting of a set of memory cells and the value. Since this expression is a term in the λ_{pv} calculus, it can be placed in the hole of a different evaluation context with appropriate alterations. Since there are no free variables in a copied closed value, there are no variable capture problems. Memory cell identifiers copied with the value may be consistently renamed in the usual way to avoid capture as defined with the merge and lift rules for references. An extended discussion and definition of `copy` is given in §6.3.2.10.

(a)	$\frac{e \mapsto e'}{\chi, \mathcal{P} + \langle \pi; \Phi; e \rangle \Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; e' \rangle}$	$(\lambda_{pv}\text{-seq})$
(b)	$\frac{ex \notin \chi}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{exception } x \text{ in } e] \rangle \Rightarrow \chi + ex, \mathcal{P} + \langle \pi; \Phi; E[\{ex / x\}e] \rangle}$	$(\lambda_{pv}\text{-exn})$
(c)	$\frac{\pi' \notin \text{PROCN}(\mathcal{P}) \cup \{\pi\}}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{proc } x \text{ in } e] \rangle \Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\{\pi' / x\}e] \rangle + \langle \pi'; \emptyset; \varepsilon \rangle}$	$(\lambda_{pv}\text{-proc})$
(d)	$\frac{}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{self_id } ()] \rangle \Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\pi] \rangle}$	$(\lambda_{pv}\text{-self_id})$

Figure 6.8 – Parallel evaluation rules for: (a) sequential evaluation; (b) exception binding; (c) `proc`; (d) `self_id`.

The general form of the parallel evaluation rules shows preconditions above a horizontal line. Two configurations are connected by the parallel evaluation relation \Rightarrow below the line. This corresponds to a transition between the two configurations provided the preconditions are satisfied. One or two processes are *selected* on the left-hand side of the relation, and their process state is given in full. No other transitions occur simultaneously with this indicated transition. An implicit precondition exists in every rule that for the selected process π (and for π' if applicable) $\pi \notin \text{PROCN}(\mathcal{P})$ (and $\pi' \notin \text{PROCN}(\mathcal{P})$ if applicable), where \mathcal{P} is used to denote the unselected processes in the current configuration.

6.3.2.1 Sequential evaluation

The first rule in Figure 6.8 states that sequential evaluation can be deduced in the presence of parallel evaluation. The name of the selected process in this transition is π . Note that if $e' = [a]$, then after the parallel evaluation rule has completed, $\pi \in \text{PROCN}(\text{Terminal}(\mathcal{P} + \langle \pi; \Phi; [a] \rangle))$.

6.3.2.2 Exception binding

The operation of choosing a new exception name is defined in the parallel evaluation rules since the new exception name must be unique and no other process may be allowed to choose the same name in an exception reduction. The new name is added to the global environment of exception names χ , and substitution of the new exception name for the identifier is performed, as given in Figure 6.8 (b).

$\frac{\phi' \notin \text{PORTN}(\mathcal{P}) \cup \text{dom}(\Phi)}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{prt } x \text{ on } \pi \text{ in } e] \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto \perp); E[\{\phi' / x\}e] \rangle$	(λ_{pv} -prt1)
$\frac{\phi'' \notin \text{PORTN}(\mathcal{P}) \cup \text{dom}(\Phi \cup \Phi') \quad \pi' \notin \text{PROCN}(\text{Terminal}(\mathcal{P})) \quad \pi \neq \pi'}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{prt } x \text{ on } \pi' \text{ in } e] \rangle + \langle \pi'; \Phi'; es \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\{\phi'' / x\}e] \rangle + \langle \pi'; \Phi' + (\phi'' \mapsto \perp); es \rangle$	(λ_{pv} -prt2)
$\frac{\pi' \in \text{PROCN}(\text{Terminal}(\mathcal{P}))}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{prt } x \text{ on } \pi' \text{ in } e] \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise NoPortCreated } \pi'] \rangle$	(λ_{pv} -prt3)

Figure 6.9 – Parallel evaluation rules for prt.

$\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{execute } (\pi'. v)] \rangle + \langle \pi'; \Phi'; \varepsilon \rangle$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[(\)] \rangle + \langle \pi'; \Phi'; [(\text{copy } (E, v)) (\)] \rangle$	(λ_{pv} -exec1)
$\frac{\pi' \in \text{PROCN}(\text{Terminal}(\mathcal{P}))}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{execute } (\pi'. v)] \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise NoSuchProcess } \pi'] \rangle$	(λ_{pv} -exec2)
$\frac{\pi' \notin \text{PROCN}(\text{Awaiting}(\mathcal{P}) \cup \text{Terminal}(\mathcal{P}))}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{execute } (\pi'. v)] \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise ProcessExecuting } \pi'] \rangle$	(λ_{pv} -exec3)

Figure 6.10 – Parallel evaluation rules for execute.

6.3.2.3 proc

Process creation requires picking a new process name and creating a new process without any ports initially. The evaluation state of the new process is *awaiting* an execution request to be sent. The evaluation rule is given in Figure 6.8 (c).

6.3.2.4 self_id

The `self_id` operation, given in Figure 6.8 (d), is straightforward, simply filling the evaluation context hole with the process identifier.

6.3.2.5 prt

Port creation requires picking a new port name, and a new mapping in the designated process from port name to queue. There are three basic situations (given in Figure 6.9): rule $(\lambda_{pv}\text{-prt1})$ describes what happens when the designated process is the same as the port requester; rule $(\lambda_{pv}\text{-prt2})$ describes what happens when the designated process is not the same as the port requester and is still executing; and rule $(\lambda_{pv}\text{-prt3})$ describes what happens when the designated process has terminated.

6.3.2.6 execute

There are three rules also for the `execute` operation (see Figure 6.10). These cover the following situations: rule $(\lambda_{pv}\text{-exec1})$ is where an execution request is accepted by another process; rule $(\lambda_{pv}\text{-exec2})$ describes what happens if the process belongs to the terminal set of the configuration; and rule $(\lambda_{pv}\text{-exec3})$ describes what happens if the designated process (possibly itself) is already evaluating some expression.

6.3.2.7 send

There are three different rules for sending: rule $(\lambda_{pv}\text{-send1})$ covers the situation where a message is sent to a port in the same process; rule $(\lambda_{pv}\text{-send2})$ details sending to a port on a remote process; and rule $(\lambda_{pv}\text{-send3})$ is the error situation when the port belongs to a process which has terminated. The use of `copy` in rule $(\lambda_{pv}\text{-send1})$ is important as it permits a uniform treatment of values in a port's queue by the `recv` operation. The three rules for the `send` operation are given in Figure 6.11.

6.3.2.8 recv

There are two essential rules for receiving messages. The first situation is when the named port has a non-empty queue, in which case the first message is dequeued, given in rule $(\lambda_{pv}\text{-recv1})$. Rule $(\lambda_{pv}\text{-recv2})$ covers the error condition, where the port is owned by another process (or was before it terminated). The two rules for the `recv` operation are given in Figure 6.12.

Rule $(\lambda_{pv}\text{-recvblock})$ is a derived rule. It covers the case where a process is *blocked* awaiting input on a named port. Evaluation can only proceed if there is a matching send to the same port. Rule $(\lambda_{pv}\text{-recvblock})$ is constructed by the successive application of rules $(\lambda_{pv}\text{-send2})$ and $(\lambda_{pv}\text{-recv1})$. Rule $(\lambda_{pv}\text{-recvblock})$ is given in Figure 6.12.

6.3.2.9 probe

The rules for the operation `probe` look similar to the rules for receiving, except that a simple boolean truth value is returned, with `true` returned if the port queue is non-empty, and `false` if it is. Error conditions are again detected, as shown in rule $(\lambda_{pv}\text{-probe3})$. The three rules are given in Figure 6.13.

$\frac{\phi' \notin \text{dom}(\Phi)}{\chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto q'); E[\text{send}(\phi'.v)] \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto q' @(\text{copy}(E, v))); E[()] \rangle$	$(\lambda_{pv}\text{-send1})$
$\frac{\phi'' \notin \text{dom}(\Phi') \quad \pi' \notin \text{PROCIN}(\text{Terminal}(\mathcal{P}))}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{send}(\phi''.v)] \rangle + \langle \pi'; \Phi' + (\phi'' \mapsto q''); es \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[()] \rangle + \langle \pi'; \Phi' + (\phi'' \mapsto q'' @(\text{copy}(E, v))); es \rangle$	$(\lambda_{pv}\text{-send2})$
$\frac{\phi' \in \text{PORTN}(\text{Terminal}(\mathcal{P}))}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{send}(\phi'.v)] \rangle \Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise NoSuchPort}()] \rangle}$	$(\lambda_{pv}\text{-send3})$

Figure 6.11 – Parallel evaluation rules for send.

$\frac{\phi' \notin \text{dom}(\Phi) \quad M \geq 1}{\chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto \lceil m_1, m_2, \dots, m_M \rceil); E[\text{recv } \phi'] \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto \lceil m_2, \dots, m_M \rceil); E[m_1] \rangle$	$(\lambda_{pv}\text{-recv1})$
$\frac{\phi' \notin \text{dom}(\Phi)}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{recv } \phi'] \rangle \Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise PortNotOwned}()] \rangle}$	$(\lambda_{pv}\text{-recv2})$
$\frac{\phi'' \notin \text{dom}(\Phi)}{\chi, \mathcal{P} + \langle \pi; \Phi + (\phi'' \mapsto \perp); E[\text{recv } \phi''] \rangle + \langle \pi'; \Phi'; E[\text{send}(\phi''.v)] \rangle}$ $\Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi + (\phi'' \mapsto \perp); E[\text{copy}(E', v)] \rangle + \langle \pi'; \Phi'; E[()] \rangle$	$(\lambda_{pv}\text{-recvblock})$

Figure 6.12 – Parallel evaluation rules for recv.

$\frac{\phi' \notin \text{dom}(\Phi) \quad q' \neq \perp}{\chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto q'); E[\text{probe } \phi'] \rangle \Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto q'); E[\text{true}] \rangle}$	$(\lambda_{pv}\text{-probe1})$
$\frac{\phi' \notin \text{dom}(\Phi)}{\chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto \perp); E[\text{probe } \phi'] \rangle \Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto \perp); E[\text{false}] \rangle}$	$(\lambda_{pv}\text{-probe2})$
$\frac{\phi' \notin \text{dom}(\Phi)}{\chi, \mathcal{P} + \langle \pi; \Phi; E[\text{probe } \phi'] \rangle \Rightarrow \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise PortNotOwned}()] \rangle}$	$(\lambda_{pv}\text{-probe3})$

Figure 6.13 – Parallel evaluation rules for probe.

6.3.2.10 copy

Various of the preceding rules rely on the **copy** operation. This operation defines how values are copied from one evaluation context so that they may be placed in the hole of a different evaluation context. The need for this operation is the inclusion of reference variables in the calculus and the semantics attached to transmission of such variables under a distributed address space model. Informally, any reference variable is sent together with a copy of the value it maps to in the memory of the sending process. At the destination process, a new memory cell is created (with a new identifying variable) and initialised with the copied value.

Reppy's semantics for CML do not require any such operation as CML is based on a shared address space model, and thus reference variables are common between threads. In fact, Reppy does not introduce references into his calculus at all, and instead models them by an encoding with threads and channels, which is known to be semantics-preserving [BMT92]. Morrisett discusses how his dynamic type dispatch compilation mechanisms can be used to marshal objects into a transmissible form [Mor95]. In particular, this is demonstrated for functions represented by proxies in a heterogenous execution environment. Knabe discusses marshalling and unmarshalling in the context of mobile agents over distributed heterogenous environments for Facile, but does not include a formal semantics [Kna95]. Nettles provides a Larch specification of copying garbage collection [Net92], which is akin to the actions which take place in the implementation of paraML's marshalling operations, but with only a single object. His specification is an abstract one for nodes connected by pointers, rather than for a full λ calculus-like language.

The **copy** operation relies on the **mem** operation, which recursively identifies any memory cells required by the value being copied. These memory cells are then used when the value is ρ -bound. The **mem** operation proceeds by case analysis on the possible structure of closed values and evaluation contexts. The definition of **copy** is as follows:

Definition 6-2 (copy) $\text{copy}(E, v) \equiv \rho(\text{mem}(E, v)).v$

The definition of **mem** is given in Figure 6.14. The input to the **copy** and **mem** operations are an evaluation context E and a value v . In those cases where the contents of memory cells in an evaluation context are unimportant, the evaluation context is written simply E . Where the contents of memory cells, or the very presence of memory at all, is important, then it is written in the usual way with a ρ -expression. The output, m , from the **copy** operation is a ρ -expression of the λ_{pv} calculus, consisting of a set of memory cells θ produced by the **mem** operation and the value v .

The values for which memory in an evaluation context is important are memory cell identifiers, λ -abstractions, and the curried form of reference update. In these cases, there are two rules – one for when there is memory and one for when the content of

memory is unimportant. The other cases are fairly self-explanatory, but the ones involving memory copying deserve some extra explanation.

Copying memory cell identifiers proceeds by creating a memory consisting of a memory cell with the memory cell identifier and a copy of the value it maps to in the existing memory. The cell and the union of any additional memory that results from copying the value the memory cell identifier maps to becomes the set of memory cells associated with the memory cell identifier. The value v that is bound with the set of memory cells for transmission is the memory cell identifier itself. Note that as a memory cell is identified as required, it is removed from the ρ -expression binding in the evaluation context passed to the recursive use of the **mem** operation, thus preventing infinite cycles in copying. Once a memory cell identifier has been encountered once, a copy of the value will be present in the ρ -bound memory to be transmitted, and hence if the identifier is discovered again there is no further need to copy the value. Note also that although the identifier x for variables is used, the only variables that can be discovered are memory cell identifiers since any other variable will have been replaced by substitution throughout the structure of the value – a consequence of transmitting only closed values.

If a λ -abstraction is to be copied, then if there is any memory bound by a ρ -expression in the evaluation context, the entire contents of memory are included in the memory to be transmitted. This definition may seem slightly surprising, but is necessary to make sure that any free (memory cell identifier) variables within the λ -abstraction are available when the ρ -expression is placed in the hole of the evaluation context at the destination. However, since it is possible to identify the free variables of the λ -abstraction, garbage collection can be performed to eliminate all but the essential memory cells which correlate with free variables of the expression before transmission. The essence of this is captured in the **mem_gc** definition, given below the definition for **mem**. This definition states that the memory cells from a **mem_gc** operation are the generalised union of the memory cell sets produced for each free variable x of the expression e . Note that the generalised union operation eliminates any duplicates arising from the potentially multiple copies of memory cells. It is worth noting that although these definitions suffice to characterise the form of the object to be transmitted, they only provide a specification of the outcome; the definitions do not necessarily reflect the structure of the algorithm used in the paraML runtime system to perform the same action. Felleisen and Hieb describe explicit garbage collection in their work on state [FH92].

The last case involving memory copying is that of the curried update operation, $:= x$. In the λ_{ev} calculus, this denotes an ability to assign to the memory cell identified by x . It does not allow any examination of the current value stored in the memory cell. Thus when such an operation is copied and the memory cell identifier has not been the subject of an earlier **mem** operation, then all that is required is to include the memory cell's current contents in the memory to be transmitted, but not to perform a recursive copy on this value as it can never be accessed by the curried update operation. If some

mem (E, v) \equiv		
case (E, c)	$=$	\emptyset
case ($\rho\theta\langle x, v \rangle.E', x$)	$=$	$(\mathbf{mem}(\rho\theta.E', v)) \cup \{\langle x, v \rangle\}'$
case (E, x)	$=$	\emptyset E is not in the form $\rho\theta\langle x, v \rangle.E'$
case (E, Y)	$=$	\emptyset
case ($E, (v_1 . v_2)$)	$=$	$(\mathbf{mem}(E, v_1)) \cup (\mathbf{mem}(E, v_2))$
case ($\rho\theta.E', \lambda x(e)$)	$=$	$(\mathbf{mem_gc}(\rho\theta.E', \lambda x(e)))$
case ($E, \lambda x(e)$)	$=$	\emptyset E is not in the form $\rho\theta.E'$
case (E, ex)	$=$	\emptyset
case (E, π)	$=$	\emptyset
case (E, ϕ)	$=$	\emptyset
case (E, ref)	$=$	\emptyset
case ($E, !$)	$=$	\emptyset
case ($E, :=$)	$=$	\emptyset
case ($\rho\theta\langle x, v \rangle.E', := x$)	$=$	$\{\langle x, v \rangle\}$
case ($E, := x$)	$=$	\emptyset
mem_gc ($\rho\theta.E, e$)	\equiv	$\bigcup_{x \in \text{FV}(e)} \mathbf{mem}(\rho\theta.E, x)$

Figure 6.14 – Definitions of **mem** and **mem_gc**.

other part of the value being copied provides access to the memory cell, then appropriate recursive copying of the memory cell value will be performed by the rule being applied to the other part, and the set union operations will ensure that sufficient memory is copied.

Since the copied value m is a term in the λ_{pv} calculus, it can be inserted into the hole of the current evaluation context in another process. In the instances where the set of memory cells is not empty and the value is one of those which may require creation of memory, then the value is bound within a ρ -expression of the set of memory cells. The rules for dealing with references, in particular the $\lambda_{pv}\text{-}\rho_{\text{merge}}$ and $\lambda_{pv}\text{-}\rho_{\text{lift}}$ rules, are used to move the memory cells into any surrounding memory. The requirements on disjoint sets of memory cell identifiers in these rules are used to ensure that completely new memory cell identifiers are chosen if necessary to avoid any possibility of variable capture and consistently replaced throughout the subexpression. Thus there is no requirement for an explicit inverse of **copy**.

A short discussion of the implications of the definition of **copy** is merited. Although it has been stressed above that process-oriented computing in distributed address spaces breaks sharing, this definition makes the loss of sharing absolute. Consider the case of sending a message between two processes π_1 and π_2 , where the value to be sent is a reference x , which identifies a value in memory, say the integer 1. On receipt of the message (consisting of $\rho\langle x, 1 \rangle.x$), the $\lambda_{pv}\text{-}\rho_{\text{merge}}$ and $\lambda_{pv}\text{-}\rho_{\text{lift}}$ rules will

choose a new memory cell identifier, say y , place the pair $\langle y, 1 \rangle$ in π_2 's ρ -bound memory, and the expression y will remain in the hole of the evaluation context. If the same message is sent by π_1 again, the λ_{pv} - ρ_{merge} rule chooses a completely new memory cell identifier, say z , places the pair $\langle z, 1 \rangle$ in π_2 's ρ -bound memory, and returns the expression z . Thus, performing an operation such as $(recv\ \phi) = (recv\ \phi)$ will return `false`. Of course, in π_1 , $x = x$ will return `true`. Conversely, if π_2 sends back the value it first received ($send\ (\phi'. (recv\ \phi))$), then π_1 , on receiving the value will create a new memory cell also. Thus, performing an equality test such as $x = (recv\ \phi')$ will return `false`.

The only way to avoid such problems would be to uniquely identify the source process's address space. For example, the representation of memory could be changed so that memory identifiers consisted of an identifier (say x) and the process name π . In the previous example, the reference to an integer 1 might instead be represented as $\langle (x, \pi_1), 1 \rangle$. No renaming of the identifiers would be required since (x, π_1) would uniquely identify the memory location in π_1 . The consequence of such a choice would be how to share memory across processes, including issues such as coherency, access protocols, and global garbage collection. These considerations are not the focus of this thesis, which is finding effective mechanisms to support programming in distributed address spaces.

6.3.2.11 Well-formed configurations and the parallel evaluation relation

Definition 6-3 A process state set \mathcal{P} is *well-formed* if for all $\langle \pi; \Phi; es \rangle \in \mathcal{P}$ the following hold:

- $FV(es) = \emptyset$ (es is closed), and
- if $\langle \pi; \Phi'; es' \rangle \in \mathcal{P}$, then $es' = es$ and $\Phi' = \Phi$.

The definition requires that any process within the set is unique and that no free variables exist in its evaluation state. Note that it may contain free exception names, free process names and free port names.

Definition 6-4 A configuration χ, \mathcal{P} is *well-formed* if for all $\langle \pi; \Phi; es \rangle \in \mathcal{P}$ the following hold:

- \mathcal{P} is well-formed,
- $FEN(es) \subseteq \chi$
- $FPN(es) \subseteq PROCN(\mathcal{P})$, and
- $FFN(es) \subseteq PORTN(\mathcal{P})$

Definition 6-5 (\Rightarrow) The *parallel evaluation relation* is the smallest relation " \Rightarrow " satisfying the rules: $(\lambda_{pv}$ -seq), $(\lambda_{pv}$ -exn), $(\lambda_{pv}$ -proc), $(\lambda_{pv}$ -self_id), $(\lambda_{pv}$ -prt1), $(\lambda_{pv}$ -prt2), $(\lambda_{pv}$ -prt3), $(\lambda_{pv}$ -exec1), $(\lambda_{pv}$ -exec2), $(\lambda_{pv}$ -exec3), $(\lambda_{pv}$ -send1), $(\lambda_{pv}$ -send2), $(\lambda_{pv}$ -send3), $(\lambda_{pv}$ -recv1), $(\lambda_{pv}$ -recv2), $(\lambda_{pv}$ -probe1), $(\lambda_{pv}$ -probe2), and $(\lambda_{pv}$ -probe3). The transitive closure of \Rightarrow is \Rightarrow^* .

Under these rules, processes always remain in the configuration, even if a process evaluates to some basic value or an unhandled exception (in which case it is a member of $\text{Terminal}(\mathcal{P})$). As discussed in the theoretical modelling of CML [Rep92], it would be feasible to include a rule that would remove processes from configurations. This is not done however as it is easier to state and prove some properties if the process set increases monotonically.

6.4 Traces

It is useful to consider some properties about configurations and evaluations. A λ_{pv} evaluation proceeds by progressive transitions of the configuration χ, \mathcal{P} , commencing with the initial well-formed configuration of $\emptyset, \{\langle \pi_0; \emptyset; e \rangle\}$, where e constitutes the program. Clearly it may be possible for many different transitions to occur from a single configuration. It is also possible for interesting programs not to terminate. Reppy's general formulation of *traces* is adopted here also. Traces are a useful terminology for describing possible evaluation sequences and for stating fairness and type soundness results.

Lemma 6-2 If χ, \mathcal{P} is well-formed and $\chi, \mathcal{P} \Rightarrow \chi', \mathcal{P}'$ then the following also hold:

- χ', \mathcal{P}' is well-formed
- $\chi \subseteq \chi'$
- $\text{PROCN}(\mathcal{P}) \subseteq \text{PROCN}(\mathcal{P}')$
- $\text{PORTN}(\mathcal{P}) \subseteq \text{PORTN}(\mathcal{P}')$

Proof. By examination of the rules for \Rightarrow . ■

Corollary 6-3 The properties of **Lemma 6-2** hold for \Rightarrow^* .

Proof. By induction on the length of the evaluation sequence. ■

The preservation of well-formedness under \Rightarrow^* implies that parallel evaluation preserves closed terms.

Definition 6-6 A *trace* T is a sequence (possibly infinite) of well-formed configurations, where each configuration is obtained through application of the rules for \Rightarrow . (The notation $\langle\langle a; b; \dots \rangle\rangle$ is used to denote a sequence.) Traces are written:

$$T = \langle\langle \chi_0, \mathcal{P}_0; \chi_1, \mathcal{P}_1; \dots \rangle\rangle$$

such that $\chi_i, \mathcal{P}_i \Rightarrow \chi_{i+1}, \mathcal{P}_{i+1}$ (for $i < n$, if T is finite with length $n+1$). The head of T is χ_0, \mathcal{P}_0 .

Clearly if χ_0, \mathcal{P}_0 is well-formed, then any possible sequence of evaluation steps proceeding according to the rules governing " \Rightarrow " and starting with χ_0, \mathcal{P}_0 is a trace.

Process states incorporate an evaluation state that is either *awaiting* (prior to execution), *terminal* (after evaluation to an answer), or evaluating some term. The status of a process with respect to a configuration is defined as following:

Definition 6-7 Let \mathcal{P} be a well-formed process set and let $\langle \pi; \Phi; es \rangle \in \mathcal{P}$. The *status* of π in \mathcal{P} is either *awaiting*, *terminated*, *blocked*, or *ready*. These are characterised as follows:

- if $es = \varepsilon$, then π is *awaiting*,
- if $es = [a]$, then π is *terminated*,
- if $es = E[\text{recv } \phi_i]$, $\phi_i \mapsto q_i \in \Phi$, $q_i = \perp$, and there does not exist a $\langle \pi'; \Phi'; E'[\text{send}(\phi_i.v)] \rangle \in \mathcal{P}$, then π is *blocked* in \mathcal{P} ,
- otherwise, π is *ready* in \mathcal{P} .

The status of process configurations can be characterised according to the status of their constituent processes.

Definition 6-8 The set of *ready* or *enabled* processes in a process state set \mathcal{P} is defined by:

$$\text{Ready}(\mathcal{P}) = \{\pi \mid \pi \text{ is ready in } \mathcal{P}\}$$

The following definitions about a process configuration χ, \mathcal{P} then hold.

1. If $\text{PROCN}(\mathcal{P}) = \text{PROCN}(\text{Terminal}(\mathcal{P}) \cup \text{Awaiting}(\mathcal{P}))$, then the configuration χ, \mathcal{P} is *complete*.
2. If $\text{Ready}(\mathcal{P}) = \emptyset$ and the configuration χ, \mathcal{P} is not complete, then the configuration χ, \mathcal{P} is *deadlocked*.
3. Otherwise, the configuration is *running*.

Definition 6-9 A trace T is a *computation* if it is infinite, or if it is finite and the final configuration is complete. In either of these cases, the computation is maximal. If e is a program, then the computations of e are defined to be:

$$\text{Comp}(e) = \{T \mid T \text{ is a computation with head } \emptyset, \langle \pi_0; \emptyset; e \rangle\}$$

Definition 6-10 The set of processes of a trace T is defined

$$\text{Procs}(T) = \{\pi \mid \exists \chi, \mathcal{P}_i \in T \text{ with } \pi \in \text{PROCN}(\mathcal{P}_i)\}$$

The definitions of convergence and divergence that were established for sequential evaluation are inadequate in the context of many different computations for a single program. Thus convergence and divergence are defined relative to a specific computation of a program.

Definition 6-11 A process $\pi \in \text{Procs}(T)$ *converges* to an answer a in a computation T , written $\pi \Downarrow_T a$, if there is a configuration $\chi, \mathcal{P}_i + \langle \pi; \Phi; [a] \rangle \in T$. A process π *diverges* in T , written $\pi \Uparrow_T$, if for every $\chi, \mathcal{P}_i \in T$, with $\pi \in \text{PROCN}(\mathcal{P}_i)$, π is ready or blocked in \mathcal{P}_i .

In addition to capturing processes executing forever, divergence may include deadlocked processes and would-be terminating processes which do not evaluate

enough to terminate. The possibilities of the latter occurring are described in the next section, which discusses fairness with respect to traces.

6.5 Fairness

Unfair traces – those in which some processes fail to make progress despite the fact that they are enabled – can exist given the definitions above. The characterisation of fair traces proposed draws on Reppy's solution to similar problems in characterising evaluation in λ_{cv} . ParaML implementations should not exhibit such properties of unfair traces, and thus a characterisation of acceptable traces is necessary.

Definition 6-12 A computation T is *acceptable* if it ends in a configuration that is complete, or if T satisfies *strong process fairness* constraints as defined in Kwiatkowska's survey of fairness issues [Kwi89]. This definition requires that any process that is *enabled* infinitely often is selected infinitely often. The enabled processes in a trace are those which are ready in a configuration.

The actions that can be taken by an enabled process are defined by the parallel evaluation rules and the selected processes are those that are the subject of one of the transitions described by these rules. Reppy faced two problems, since not only did he require strong process fairness, but also *strong event fairness*. The latter property is used to guarantee progress in communications on channels in λ_{cv} . However, since all communications in λ_{pv} are captured in the parallel evaluation rules as ready processes, there is no such requirement here.

Reppy also makes the point that in practice a stronger set of fairness requirements are needed for finite traces, since the infinite selection property may be inadequate. This additional refinement is captured by Reppy using *k-bounded fair* traces. The property for λ_{pv} is expressed as follows.

Definition 6-13 A finite trace T of length n is *k-bounded fair* (for k a fixed positive integer) if every intermediate configuration χ_i, \mathcal{P}_i satisfies one or other of the following, with $m = i + k \mid \text{Ready}(\mathcal{P}_i) \mid$:

- $m > n$, or
- for every $\pi \in \text{Ready}(\mathcal{P}_i)$, π is a selected process at least once in the trace subsequence $\chi_i, \mathcal{P}_i \Rightarrow \dots \Rightarrow \chi_m, \mathcal{P}_m$.

An infinite trace T is *k-bounded fair*, if every finite prefix of T is also *k-bounded fair*.

In practice, with the implementation of this language it is fairly straightforward to guarantee fairness properties by pre-emptive scheduling of processes, with FIFO queues of ready processes. The time slice between pre-emptions must be long enough for a process to make continue making progress.

6.6 Summary

The definition of λ_{pv} as an extension to λ_{cv} simplifies the specification of an operational semantics to model paraML. The combination of sequential and parallel evaluation rules neatly distinguishes actions that go on inside a process and those that result in interactions between processes. The general form of semantics draws strongly from Reppy's theoretical specifications for CML, which is not totally suprising since similar problems are faced by both languages. This form of semantics is particularly useful in specifying the operational requirements on implementations of paraML.

One of the major difference between Reppy's presentation and mine is that Reppy models recursion and references by encodings into the λ_{cv} calculus. I have chosen instead to build them into the underlying sequential λ_{cv} calculus on which λ_{pv} is based, so that the description of how these features interact with parallel evaluation is made explicit. The definition of **copy** carries these interactions in the context of communications across distributed address spaces. The next chapter presents a formal proof of the type soundness of λ_{pv} .

7. Typing

7.1 Static semantics

The static semantics for λ_{pv} are developed in a similar fashion to the dynamic semantics. The polymorphic type system of λ_{ev} is extended to characterise the extensions made in λ_{pv} . This type system is then subject to a proof of type soundness relative to process configurations rather than just sequential evaluation. Once again, the form of extensions to the type system draws on the style developed by Reppy for λ_{ev} . The central result in the proof of type soundness is that a well-typed λ_{pv} program will never produce a runtime type error. This result is especially important given that programs may involve communications between processes, and the values that are communicated are transferred among distributed address spaces. In other words, type soundness provides a guarantee that receiving and computing with a value will be sound with respect to the type of the value. The type system for λ_{pv} is essentially identical to that for paraML.

7.1.1 Definitions

The types for λ_{pv} are formed from type constants and type variables, which include all of those found in λ_{ev} . The addition of *ProcessName* types and *PortName* types in paraML is captured in λ_{pv} by changes to the definitions of types and type constants to include *ProcessName* and *PortName*. The restrictions imposed on reference cell values and raised exceptions also apply with respect to ports. Thus only values with imperative types will be capable of being transmitted to ports.

The set of types ($\tau \in TY$) is defined:

$\tau ::= \iota$	type constant
α	type variable
$(\tau_1 \rightarrow \tau_2)$	function type
$(\tau_1 \times \tau_2)$	pair type
τref	reference cell type
τexn	exception type
$\tau PortName$	port name type

The set of type constants ι (consisting of such things as *bool*, *int*, etc. in λ_{ev}) is extended with a new type constant, *ProcessName*. Note that the operational semantics do not allow there to be either process name constants or port name constants. This restriction makes the proof of some of the subsequent type soundness properties simpler. The absence of process name values as constants should not be confused with *ProcessName* being a type constant.

Type environments are used for giving type information about variables in open expressions. It is also necessary for the type system to assign types to intermediate stages of computation. Just as it was necessary for the type system to be able to assign types to exception names, the same is true for process names and port names. Since process names are always of type *ProcessName*, there is no requirement to map process names to types, other than through a type rule. However, port names are similar to exception names and a similar approach is used to assign types to port names. A type environment in λ_{pv} is a 3-tuple of finite maps. The first two maps are for variables and exception names, defined:

$$\begin{aligned} VT \in \text{VARTY} &= \text{VAR} \xrightarrow{\text{fin}} \text{TYSCHEME} \\ ET \in \text{EXNTY} &= \text{EXNNAME} \xrightarrow{\text{fin}} \text{IMPTY} \end{aligned}$$

The third map is from port names to imperative types:

$$PT \in \text{PORTTY} = \text{PORTNAME} \xrightarrow{\text{fin}} \text{IMPTY}$$

Type environments are defined:

$$TE = (VT, ET, PT) \in \text{TYENV} = (\text{VARTY} \times \text{EXNTY} \times \text{PORTTY})$$

The free type variables of port name typings are denoted $\text{FTV}(PT)$, and those for exception name typings $\text{FTV}(ET)$. There are no bound type variables in port name or exception name typings, and $\text{FTV}(PT) \subseteq \text{IMPTYVAR}$ and $\text{FTV}(ET) \subseteq \text{IMPTYVAR}$.

The free type variables of a type environment are thus defined:

$$\text{FTV}(TE) = \text{FTV}(VT) \cup \text{FTV}(ET) \cup \text{FTV}(PT)$$

Type environment modifications for variables, exception names and port names can be conveniently expressed as follows:

$$\begin{aligned} TE \pm \{x \mapsto \sigma\} &\equiv_{\text{def}} (VT \pm \{x \mapsto \sigma\}, ET, PT) \\ TE \pm \{ex \mapsto \psi\} &\equiv_{\text{def}} (VT, ET \pm \{ex \mapsto \psi\}, PT) \\ TE \pm \{\phi \mapsto \psi\} &\equiv_{\text{def}} (VT, ET, PT \pm \{\phi \mapsto \psi\}) \end{aligned}$$

The two forms of closures of types with respect to type environments remain the same. The closure of type τ with respect to type environment TE is written:

$$\begin{aligned} \text{CLOS}_{TE}(\tau) &= \forall \alpha_1 \dots \alpha_n. \tau \\ &\text{where } \{\alpha_1 \dots \alpha_n\} = \text{FTV}(\tau) \setminus \text{FTV}(TE) \end{aligned}$$

The applicative closure of type τ with respect to type environment TE is written:

$$\begin{aligned} \text{APPCLOS}_{TE}(\tau) &= \forall \alpha_1 \dots \alpha_n. \tau \\ &\text{where } \{\alpha_1 \dots \alpha_n\} = (\text{FTV}(\tau) \setminus \text{FTV}(TE)) \cap \text{APPTYVAR} \end{aligned}$$

The two facts about generalisation expressed in **Lemma 5-2** remain true for λ_{pv} . Similarly, the notions of type judgements and well-typed programs as closed

$\overline{\text{TE} \vdash \pi : \text{ProcessName}}$	τ-procvar
$\frac{\text{PT}(\phi) = \psi}{(\text{VT}, \text{ET}, \text{PT}) \vdash \phi : \psi}$	τ-prtvar
$\frac{\text{TE} \pm \{x \mapsto \text{ProcessName}\} \vdash e : \tau}{\text{TE} \vdash \text{proc } x \text{ in } e : \tau}$	τ-proc
$\frac{\text{TE} \vdash \pi : \text{ProcessName} \quad \text{TE} \pm \{x \mapsto \psi \text{ PortName}\} \vdash e : \tau}{\text{TE} \vdash \text{prt } x \text{ on } \pi \text{ in } e : \tau}$	τ-prt
$\frac{\text{TE} \vdash \pi : \text{ProcessName} \quad \text{TE} \vdash v : \text{unit} \rightarrow \text{unit}}{\text{TE} \vdash \text{execute } (\pi . v) : \text{unit}}$	τ-execute
$\text{TE} \vdash \text{self_id} : \text{unit} \rightarrow \text{ProcessName}$	τ-self_id
$\frac{\text{TE} \vdash \phi : \psi \text{ PortName} \quad \text{TE} \vdash v : \psi}{\text{TE} \vdash \text{send } (\phi . v) : \text{unit}}$	τ-send
$\frac{\text{TE} \vdash \phi : \psi \text{ PortName}}{\text{TE} \vdash \text{recv } \phi : \psi}$	τ-recv
$\text{TE} \vdash \text{probe} : \tau \text{ PortName} \rightarrow \text{bool}$	τ-probe

Figure 7.1 – New type rules for λ_{pv} .

expressions with a judgement in the form $\vdash e : \tau$ remain identical. The δ -typability property is also unchanged.

7.1.2 Type rules

The typing rules for λ_{pv} which extend those from λ_{ev} are given in Figure 7.1. These new rules cover all the new expressions defined in the calculus. From these rules, proofs of the types of arbitrary closed expressions can be deduced solely from the syntactic form of the expression. Note that although the rule for exceptions moved from the sequential evaluation relation to the parallel evaluation relation in λ_{pv} , the type rules for exceptions (τ -ex and τ -exn) remain unchanged.

7.1.3 Configuration typings

A *configuration typing* is a finite map from process names to types:

$$\text{CT} \in \text{CONFIGTY} \quad = \quad \text{PROCNAME} \xrightarrow{\text{fin}} \text{TY}$$

Typing judgements extend to process configurations according to the following definition:

Definition 7-1 A well-formed configuration χ, \mathcal{P} has type CT under an exception name typing ET, and port name typing PT, written:

$$ET, PT \vdash \chi, \mathcal{P} : CT$$

if the following hold:

- $\chi \subseteq \text{dom}(ET)$,
- $\text{PORTN}(\mathcal{P}) \subseteq \text{dom}(PT)$,
- $\text{PROCEN}(\mathcal{P}) \subseteq \text{dom}(CT)$, and
- for every $\langle \pi; \Phi; es \rangle \in \mathcal{P}$, $(\{\}, ET, PT) \vdash es : CT(\pi)$.

The type of a process in a configuration with evaluation state ε is defined to be *unit*. For λ_{pv} , since *execute* requires a $(\text{unit} \rightarrow \text{unit})$ argument for evaluation by the remote process π , the configuration typing is $CT(\pi) = \text{unit}$ for all $\pi \in \text{PROCEN}(\mathcal{P})$, other than for the initial process π_0 . The configuration typing $CT(\pi_0)$ may have any type whatsoever, which is considered to be the type of the program.

7.2 Type soundness

The type system given above is only one aspect of the typing for λ_{pv} . The next step is to establish type soundness of the system with respect to the dynamic semantics given in Chapter 6. In this I follow the alterations suggested in [Rep92], which focus more on stuck expressions than on faulty expressions. This is the major difference between the type soundness proofs for λ_{pv} and those for λ_{cv} .

7.2.1 Supporting lemmas

Before establishing subject reduction, there are several important lemmas required in the proof. The first lemma establishes that variables or exception names in the domain of the typing environment which are not free in an expression e can be ignored when typing e . The variable convention ensures that $x \notin \text{FV}(e)$ whenever the lemma applies. Some proofs are essentially identical to the equivalent ones in Chapter 5, and are indicated as such.

Lemma 7-1 If $x \notin \text{FV}(e)$ then $TE \vdash e : \tau$ iff $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$. Likewise, if $ex \notin \text{FEN}(e)$ then $TE \vdash e : \tau$ iff $TE \pm \{ex \mapsto \psi\} \vdash e : \tau$. And if $\phi \notin \text{FFN}(e)$ then $TE \vdash e : \tau$ iff $TE \pm \{\phi \mapsto \psi\} \vdash e : \tau$.

Proof. As for Lemma 5-3. ■

The next lemma is known as the replacement lemma, since it allows the replacement of a complete subexpression in a term with another of the same type, without having any effect on the type of the whole term.

Lemma 7-2 (Replacement) Let $C[]$ be a context with a hole. Provided the following conditions hold:

1. D is a type deduction which concludes $TE \vdash C[e_1] : \tau$,
2. D_1 is a subdeduction of D which concludes $TE' \vdash e_1 : \tau'$,
3. D_1 occurs in D in the position which corresponds to the hole in C ,
4. $TE' \vdash e_2 : \tau'$,

then $TE \vdash C[e_2] : \tau$.

Proof. As for Lemma 5-4. ■

The substitution lemma, which is used to show that types are preserved under β -reduction, relies on an additional two lemmas. The first of these lemmas extends substitution on types to substitution on type judgements.

Lemma 7-3 If S is a substitution and $TE \vdash e : \tau$, then $S(TE) \vdash e : S\tau$.

Proof. As for Lemma 5-5. ■

The next lemma is used to show that generalising the typing assumptions (the information in the type environment) has no effect on the typing outcome of an expression.

Lemma 7-4 If $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$ and $\sigma' \succ \sigma$, then $TE \pm \{x \mapsto \sigma'\} \vdash e : \tau$.

Proof. The proof of this lemma is by induction on the height of the typing deduction of $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$, and case analysis on the shape of e for the last step. The additional cases from Lemma 5-6 that must be considered are given in Appendix A-2. ■

Lemma 7-5 (Substitution) If $x \notin FV(v)$, $TE \vdash v : \tau$, and $TE \pm \{x \mapsto \forall \alpha_1 \dots \forall \alpha_n. \sigma\} \vdash e : \tau'$, with $\{\alpha_1, \dots, \alpha_n\} \cap FTV(TE) = \emptyset$, then $TE \vdash \{v/x\}e : \tau'$.

Proof. Just as in the previous lemma, the proof of the substitution lemma is by induction on the height of the typing deduction, and case analysis on the shape of e for the last step. The additional cases from Lemma 5-7 to be considered in this proof are given in Appendix A-2. ■

7.2.2 Type preservation

The core result for establishing syntactic soundness is type preservation. Essentially, this states that if an expression can be assigned a type, then that type is preserved through reduction of the expression. Type preservation must be established both for the sequential evaluation relation and for the parallel evaluation relation.

Theorem 7-6 (Sequential type preservation) For any type environment TE , expression e_1 , and type τ , such that $TE \vdash e_1 : \tau$, if $e_1 \longrightarrow e_2$ then $TE \vdash e_2 : \tau$.

Proof. Since $e_1 \mapsto e_2$, then it must be because a rule of the form $E[e] \mapsto E[e']$ was used, with $E[e] = e_1$ and $E[e'] = e_2$. Assume that $TE' \vdash e : \tau$ with $TE' = (VT', ET', PT')$. Then by the **Replacement Lemma (Lemma 7-2)**, it is sufficient to show that $TE' \vdash e' : \tau$. This is done by case analysis of the definition of \mapsto , effectively on the structure of e . The proof is given in Appendix A-2. ■

The second subject reduction theorem states that parallel evaluation preserves configuration typing.

Theorem 7-7 (Parallel type preservation) If a configuration χ, \mathcal{P} is well-formed with

$$\chi, \mathcal{P} \Rightarrow \chi', \mathcal{P}'$$

and, for some exception name typing ET and port name typing PT,

$$ET, PT \vdash \chi, \mathcal{P} : CT$$

Then there is an exception name typing ET', port name typing PT', and configuration typing CT', such that the following hold:

- $ET \subseteq ET'$,
- $PT \subseteq PT'$,
- $CT \subseteq CT'$,
- $ET', PT' \vdash \chi', \mathcal{P}' : CT'$, and
- $ET', PT' \vdash \chi, \mathcal{P} : CT'$

Proof. The final property follows from the others. The proof of the first four properties proceeds by case analysis of the left hand side of the parallel evaluation relation \Rightarrow . Since the proof is quite lengthy, full details are found in Appendix A-2. ■

The corollary of this theorem with respect to traces of configurations is immediate:

Corollary 7-8 Let $\langle\langle \chi_0, \mathcal{P}_0; \dots; \chi_n, \mathcal{P}_n \rangle\rangle$ be a finite trace, with

$$ET, PT \vdash \chi_0, \mathcal{P}_0 : CT$$

Then there is a exception name typing ET', port name typing PT', and configuration typing CT', such that:

- $ET \subseteq ET'$,
- $PT \subseteq PT'$,
- $CT \subseteq CT'$, and
- for $i \in \{0, \dots, n\}$, $ET', PT' \vdash \chi_i, \mathcal{P}_i : CT'$.

Proof. This follows by a straightforward induction on n . ■

7.2.3 Stuck expressions

Parallel type preservation (Theorem 7-7) guarantees that if a configuration is typable, any series of reductions will always yield a typable configuration. However, this is not sufficient to establish type soundness. The crucial property to establish is that typable configurations do not become *stuck*. The focus in the type soundness for λ_{ev} was to establish that faulty expressions are untypable, since faulty expressions are a conservative approximation to those which may become stuck. Faulty expressions are still useful in helping to determine which processes may become stuck in a configuration.

Definition 7-2 A process π with process state $\mu = \langle \pi; \Phi; es \rangle$ is stuck if $es \notin \{\epsilon\} \cup \{[a]\}$, and there do not exist well-formed configurations $\chi, \mathcal{P} + \mu$ and χ', \mathcal{P}' such that $\chi, \mathcal{P} + \mu \Rightarrow \chi', \mathcal{P}'$, with π a selected process. A well-formed configuration is stuck if one or more of its processes are stuck.

This definition requires that if a process is evaluating an expression and is yet to produce an answer, it must have an ability to continue evaluating (possibly in conjunction with other processes). The concept of a process being stuck is a semantic one. In particular, it is not syntactically decidable whether a process's evaluating expression could eventually become stuck. The crucial result will be to establish that stuck processes are untypable. To prove this results requires a statement about uniform evaluation and a definition of stuck expressions.

Lemma 7-9 (Uniform evaluation) Let e be a program, $T \in \text{Comp}(e)$, and $\pi \in \text{Procs}(T)$. Then either $\pi \Downarrow_T a$, $\pi \Uparrow_T$, or $\langle \pi; \Phi; e' \rangle \in \mathcal{P}_i$ and π is stuck for some $\chi, \mathcal{P}_i \in T$.

Proof. The uniform evaluation result follows directly from the definitions. ■

The next lemma provides a definition of stuck processes based on the syntactic form of an evaluating expression.

Lemma 7-10 (Stuck processes) A process π with process state $\langle \pi; \Phi; e \rangle$ and e closed, is stuck iff e has one of the following forms:

1. $E[c \ v]$ where $\delta(c, v)$ is undefined
2. $E[v \ v']$ where v has the form $(v_1 . v_2)$, ex , π , or ϕ
3. $E[! \ v]$ where $v \notin \text{VAR}$
4. $E[: = v]$ where $v \notin \text{VAR}$
5. $\rho\theta\langle x, v_2 \rangle. E[x \ v_1]$
6. $\text{exception } x \text{ in } E[! \ x]$
7. $\text{exception } x \text{ in } E[: = x]$
8. $\text{exception } x \text{ in } E[x \ v]$
9. $E[\text{raise } v_1 \ v_2]$ where $v_1 \notin \text{VAR} \cup \text{EXNNAME}$
10. $E[e \ \text{handle } v_1 \ v_2]$ where $v_1 \notin \text{VAR} \cup \text{EXNNAME}$

11. $\rho\theta\langle x, v \rangle. E[\text{raise } x]$
12. $\rho\theta\langle x, v \rangle. E[e' \text{ handle } x]$
13. $E[\text{prt } x \text{ on } v \text{ in } e']$ where $v \notin \text{VAR} \cup \text{PROCESSNAME}$
14. $E[\text{execute } (v_1 . v_2)]$ where $v_1 \notin \text{VAR} \cup \text{PROCESSNAME}$ or v_2 has the form $Y, (v_1 . v_2), \text{ref}, !, :=, := x, ex, \pi$, or ϕ
15. $E[\text{send } (v_1 . v_2)]$ where $v_1 \notin \text{VAR} \cup \text{PORTNAME}$
16. $E[\text{recv } v]$ where $v \notin \text{VAR} \cup \text{PORTNAME}$
17. $E[\text{probe } v]$ where $v \notin \text{VAR} \cup \text{PORTNAME}$
18. $\rho\theta\langle x, v \rangle. E[\text{proc } x \text{ in } e']$
19. $\rho\theta\langle x, v' \rangle. E[\text{prt } x \text{ on } v \text{ in } e']$
20. $\rho\theta\langle x, v \rangle. E[\text{execute } (x . v_2)]$
21. $\rho\theta\langle x, v \rangle. E[\text{send } (x . v_2)]$
22. $\rho\theta\langle x, v' \rangle. E[\text{recv } x]$
23. $\rho\theta\langle x, v' \rangle. E[\text{probe } x]$

Proof. The *if* component of the *iff* is proved by case analysis on the possible forms of e' where $e = E[e']$. This is given in Appendix A-2. The *only if* component follows directly from the definitions. ■

Lemma 7-11 (Stuck configurations are untypable) If π is stuck with process state $\langle \pi; \Phi; E[e'] \rangle$ in a well-formed configuration χ, \mathcal{P} , then there do not exist $ET \in \text{EXNTY}$, $PT \in \text{PORTTY}$, and $CT \in \text{CONFIGTY}$, such that

$$(\{\}, ET, PT) \vdash E[e'] : CT(\pi)$$

In other words, χ, \mathcal{P} is untypable.

Proof. Let π be stuck with process state $\langle \pi; \Phi; E[e'] \rangle$ in χ, \mathcal{P} , and assume that there exist $ET \in \text{EXNNAMETY}$, $PT \in \text{PORTNAMETY}$ and $CT \in \text{CONFIGTY}$, such that $(\{\}, ET, PT) \vdash E[e'] : CT(\pi)$. It suffices to show that e' is untypable, which is a contradiction. Let τ be the type of e' ; that is, $TE' \vdash e' : \tau$, for some TE' . Note that since χ, \mathcal{P} is well-formed, $E[e']$ is closed; and thus **Lemma 7-10** gives the possible forms of e' . The proof proceeds by case analysis on the form of the subexpression e' , showing that e' is untypable in each case. The details are given in Appendix A-2. ■

7.2.4 Soundness

The key results can now be proven that establish type soundness for λ_{pv} .

Theorem 7-12 (Syntactic soundness) Let e be a program, with $\vdash e : \tau$. Then for any $T \in \text{Comp}(e)$, $\pi \in \text{Procs}(T)$, where χ, \mathcal{P}_i is the first configuration in T containing π , there exist an ET , PT and CT , such that

$$ET, PT \vdash \chi, \mathcal{P}_i : CT$$

and $CT(\pi_0) = \tau$. Then either

- $\pi \uparrow_T$, or
- $\pi \Downarrow_{\tau} a$ and there exists an extension ET', PT' of ET, PT with $(\{\}, ET', PT') \vdash a : CT(\pi)$.

Proof. The existence of ET, PT and CT follows from **Parallel type preservation** (**Theorem 7-7**). By **Uniform evaluation** (**Lemma 7-9**), it follows that either $\pi \Downarrow_{\tau} a$, $\pi \uparrow_T$, or π is stuck with process state $\langle \pi; \Phi; E[e'] \rangle \in \mathcal{P}_i$ for some $\chi, \mathcal{P}_i \in T$.

Assume that π is stuck with process state $\langle \pi; \Phi; E[e'] \rangle \in \mathcal{P}_i$ in χ, \mathcal{P}_i . By **Lemma 6-2**, χ, \mathcal{P}_i is well-formed and by **Lemma 7-11** it must be untypable. But, since the initial configuration $(\emptyset, \{\langle \pi_0; \emptyset; e \rangle\})$ of the trace is typable, by **Parallel type preservation** (**Theorem 7-7**), there is an $ET' \in \text{EXNTY}$, $PT' \in \text{PORTTY}$, and $CT' \in \text{CONFIGTY}$, such that $ET', PT' \vdash \chi, \mathcal{P}_i : CT'$. Which means that $(\{\}, ET', PT') \vdash E[e'] : CT'(\pi)$, hence π cannot be stuck and either $\pi \Downarrow_{\tau} a$ or $\pi \uparrow_T$.

If $\pi \uparrow_T$, then there is no more to do.

Assume that $\pi \Downarrow_{\tau} a$ and let $\chi, \mathcal{P}_i \in T$ such that $\langle \pi; \Phi; [a] \rangle \in \mathcal{P}_i$. **Parallel type preservation** (**Theorem 7-7**) means that there exist extensions ET', PT' and CT' of ET, PT and CT respectively such that $ET', PT' \vdash \chi, \mathcal{P}_i : CT'$. Since CT' is an extension of CT , $CT'(\pi) = CT(\pi)$, and hence $(\{\}, ET', PT') \vdash a : CT(\pi)$. ■

The definition of evaluation must distinguish between programs that produce a result and those which have type errors.

Definition 7-3 For a computation T , define the *evaluation* of a process π in T as

$$eval_T(\pi) = \begin{cases} \text{WRONG} & \text{if } \pi \text{ has process state } \langle \pi; \Phi; E[e'] \rangle \in \mathcal{P}_i \text{ for some } \chi, \mathcal{P}_i \in T \\ & \text{and } E[e'] \text{ causes } \pi \text{ to be stuck;} \\ a & \text{if } \pi \Downarrow_{\tau} a. \end{cases}$$

For sequential programs, this is essentially the same as the result for λ_{ev} . The definition of $eval_T$ enables the proof of strong and weak soundness properties for λ_{pv} .

Theorem 7-13 (Soundness) If e is a program with $\vdash e : \tau$, then for any $T \in \text{Comp}(e)$ and any $\pi \in \text{Procs}(T)$, the following two statements hold:

(Strong soundness) If $eval_T(\pi) = a$, and χ, \mathcal{P}_i is the first configuration in T containing an occurrence of π , then for any ET, PT and CT such that $ET, PT \vdash \chi, \mathcal{P}_i : CT$ and $CT(\pi_0) = \tau$, there are extensions ET' and PT' of ET and PT respectively, such that $(\{\}, ET', PT') \vdash a : CT(\pi)$.

(Weak soundness) $eval_T(\pi) \neq \text{WRONG}$.

Proof. The proof follows directly from **Syntactic soundness** (**Theorem 7-12**) and the definition of $eval_T$. ■

7.3 Conclusion

The importance of establishing type soundness for λ_{pv} is that it guarantees that well-typed paraML programs will never result in runtime type errors. This property is significant since it extends the strong static type checking of ML to process-oriented computation, despite the added complications of process initiation and inter-process communications. To my knowledge, this result has not been established previously for languages where communication takes place between processes each with its own evaluation environment, where communication includes transmission of copied mutable values. Previous formulations of semantics for languages with distributed execution environments, such as Facile [TLG92], have encoded references with process and communication primitives into global shared objects. Such encodings do not capture the nature of transmission of non-shared mutable reference values between memories.

The other major goal of λ_{pv} has been to capture the operational requirements on an implementation of paraML. This goal has been achieved without recourse to encoding common programming facilities in ML as derived operations in the concurrency and communication operations of the calculus. In the next part of this thesis, the implementation and use of paraML in practice is considered.

3. Application

3.1. Introduction

The following is a brief introduction to the application of the theory of the previous chapter to the case of the simple harmonic oscillator. The theory of the simple harmonic oscillator is a special case of the more general theory of the harmonic oscillator. The theory of the harmonic oscillator is a special case of the more general theory of the harmonic oscillator. The theory of the harmonic oscillator is a special case of the more general theory of the harmonic oscillator.

Part IV PRACTICE

The first of these is the fact that the
the second is the fact that the
the third is the fact that the
the fourth is the fact that the
the fifth is the fact that the
the sixth is the fact that the
the seventh is the fact that the
the eighth is the fact that the
the ninth is the fact that the
the tenth is the fact that the

Part IV

FRAGMENTS

8. Applications

8.1 Introduction

The ultimate test of any language design is its use in practice. The description of derived operations in Chapter 4, while valuable, does not provide convincing proof that the language mechanisms are actually worthwhile. This part of the thesis describes the practical use of paraML. In this chapter, paraML's use in the development of alternative programming paradigms and applications is discussed. The next chapter describes the actual implementation of paraML on the Fujitsu AP1000+ multicomputer. Chapter 10 details performance measurements of the implementation.

Just as the choice of language paradigm may be functional, imperative, object-oriented or logic-based, high performance programming has a number of differing paradigms that may be adopted. The paradigm embodied by paraML is that of processes with message passing to ports. However, this paradigm may not be best suited for certain applications, or individual programmers may prefer to work with an alternative. In this chapter, two such alternatives are considered: *algorithmic skeletons* and *object stores*. With each, the goal is to find out how to layer their implementation on paraML's process-oriented model and what aspects are limited or enhanced. The use of paraML as an implementation language for parallelising the SIMPLE hydrodynamics benchmark is also discussed.

8.2 Algorithmic skeletons

8.2.1 Overview

A frequent criticism of high performance parallel programming is that the programmer does not wish to be involved in the explicit details of managing the parallelism. Although there is undeniable merit in involving the programmer in specifying possible concurrency in the program design, there are also situations where utilising pre-existing sequential algorithms is a much simpler option than hand-coding a new solution to an old problem. In sequential computing, this is borne out by the large numbers of libraries (for classes, routines and so forth) and the general emphasis on software reuse and modularity. Cole developed a similar concept for parallel computing with what he termed algorithmic skeletons [Col89]. These skeletons allow the user to specify a number of routines which perform the essence of an individual program, but leave the details of making best use of an underlying parallel machine to the skeleton's implementation. The skeleton provides all the necessary management of parallelism, including decomposition, communication and synchronisation.

Cole's motivation was twofold: the first aspect of a skeleton was an informal description of a particular algorithm together with detailed implementation suggestions. The second aspect was to provide a formal analysis of the performance characteristics of the implementation. A degree of abstraction of the underlying computer architecture was required, but within these parameters, an algorithmic skeleton should be guaranteed to perform with the same relative performance across different platforms. This approach was designed to remove the need to retune or totally redevelop an algorithm when porting parallel programs. Skeletons for high performance computing are an active area of research. The work of Darlington's group at Imperial College [DGT+95] and the P³L group at the University of Pisa [DP93, BDO+95] represent mature developments in the field. A skeleton system for the functional subset of ML was also developed by Bratvold [Bra94]. Cole's four original skeletons are used as the basis for an experiment in paraML's process-oriented framework. The implementation characteristics of the skeletons are discussed, together with some comments as to the strengths and weaknesses of this approach to managing parallelism in paraML.

8.2.2 Which skeletons

Cole's original four skeletons [Col89] were:

- divide and conquer – successive subdivision of a problem into manageable parts;
- cluster – clustering of objects according to depth in a solution tree;
- iterative combination – iteratively find best partner relationships of objects in a set, until only one object is left or no best partner may be found;
- worker farm – fragment the problem into separate tasks, and farm tasks out to workers, combining the results until no tasks remain.

Cole's original solutions assumed an underlying architecture that supported only fixed numbers of PEs and whose communications could only occur between neighbouring PEs. These restrictions are clearly not necessary in the context of paraML. The assumptions were primarily utilised in making guarantees of performance complexity. However, it is not possible to do the same with paraML since it may be impossible to predict how many processes execute on the same PE.

8.2.3 Implementation

8.2.3.1 The need for abstractions

During the design and implementation of the algorithmic skeletons in paraML, the value of collective operations was made apparent. This need contributed to the definition of the derived operations for groups described in §4.4.5. Process-oriented programming facilities benefit from an ability to refer to collections of processes or ports when used for parallel programming. Depending on how frequently these

facilities are used in practice, it may be appropriate to build optimised versions of these operations.

8.2.3.2 Divide and conquer

The divide and conquer skeleton is one of the simplest to express in a language like *paraML*. A problem is divided into successively smaller sub-problems, until such point as a sub-problem is solved directly. The results of combining solved sub-problems are passed back up a level, combined with other solutions, and so on. Each sub-problem is solved by creating a new process, and outport communications make the result available to the process which will combine a number of results.

The solution is an almost exact translation of the functional specification for the skeleton, with a small number of primitives for creating processes and collecting results. An ability to create solutions which closely match a functional definition is encouraging, since it simplifies the translation phase from problem specification to solution, and thus is less likely to produce errors.

8.2.3.3 Cluster

The cluster skeleton is one which Cole invented for the purpose of working backwards from an interesting idea to an algorithmic definition. Objects are combined together, incorporating information about their depth in the solution tree. Cole's proposed implementation splits object sub-sets across processes, in a manner similar to the iterative combination skeleton discussed in §8.2.3.4. However, the motivation for doing this is an orientation towards the particular architecture that Cole envisaged being used, which bears little resemblance to that of *paraML*'s abstract machine model. In addition, since the object spaces to be distributed would have to be small enough to fit into the top-level process at the commencement of the problem, object distribution is not being used for reasons of space restrictions.

The problem specification looks moderately close to the divide and conquer skeleton. With appropriate manipulations of the user-supplied functions and data types, it was possible to transform the cluster skeleton into an instance of the divide and conquer skeleton. The transformation approach to skeleton construction is somewhat similar to the explorations of skeletons by Darlington's group at Imperial College [DGT+95].

8.2.3.4 Iterative combination

The iterative combination skeleton proved to be the most complex of all four skeletons constructed. The problem is to find the best partners for all objects from a set, combine the best partners to reduce the size of the set, and iterate until there is either a single combined object (representing a solution for the data set and cost metric of combination), or there are no more best partners and hence no solutions for the problem.

The basic imperative and declarative specifications of the algorithm are straightforward, and a data parallel solution would probably be the most natural mechanism to build a parallel solution. However, the paraML solution closely follows Cole's suggested implementation, which requires splitting the set of objects across a group of processes. During each phase of computation, each object requires access to every other object in the set, and this requires constructing a local form of identification for the objects, and providing a mechanism by which sub-sets of objects can be communicated around the ring of processes. Since the number of objects can also shrink to less than the number of processes in the ring, it is necessary to dynamically re-map the communication boundary points.

The solution is approximately an order of magnitude more complex (in terms of number of lines) than any of the other solutions. Nevertheless, this approach to solving a problem, while complex, illustrates that standard forms of problems involving a fixed number of communicating processes in a distributed space can be used to provide the illusion of a shared address space, and illustrates some techniques to manage collections of objects.

8.2.3.5 Worker farm

The basic algorithm is an explicitly parallel one, designed to solve the problem of load balancing a system where any form of static decomposition could lead to unacceptable load imbalance. A master process generates a number of tasks that are handed out to workers as they make requests. Each worker solves its own task, reports the result to a sink for the output (which will combine the results as required), and makes a request of the master for more work. When there is no more work to be done, the workers are killed off on completion of their last task.

The facilities available in paraML make this a straightforward algorithm to implement – processes are dynamically created for the master and workers, specialised by the user-provided functions for task generation and so forth, and left to communicate on appropriate ports for the delivery of new tasks, results, and requests for more work. The paraML solution is significantly more compact than an equivalent worker farm, the Parallel Utilities Library Task Farm module [CFT+94] developed at the Edinburgh Parallel Computing Centre, though it is necessary to be wary of drawing direct comparisons between C and ML.

8.2.4 Strengths

The module system of ML is an excellent basis for providing skeleton abstractions. Users are completely separated from the details of process creation and inter-process communication. The functions required by a skeleton are specified in a signature file, and the user provides a corresponding structure module. This structure module is then used to parameterise the skeleton functor module, producing a specific instance of the skeleton for use. The ability to send functions as messages between processes makes the implementation of the skeletons more flexible. Similarly, since processes are

created dynamically, it is practical to compose skeletons. The work of Darlington's group is an example of how such skeleton composition could be made more efficient.

8.2.5 Limitations of using algorithmic skeletons

The major weakness of using paraML with skeletons is that it remains difficult to guarantee the performance complexity of algorithms. The reason for this is that the performance complexity metrics of Cole's skeletons are predicated on having only one process execute on each processor. Given these restrictions, paraML can provide the same performance guarantees. However, unless the skeletons make use of the knowledge of the number of PEs available in a multicomputer, the implementation may execute several processes on a single PE, thereby invalidating the performance complexity metrics.

8.2.6 Summary

The algorithmic skeleton approach to providing an alternative high performance programming paradigm is cleanly supported by paraML. This arises from the correspondence between the original conception of the machine architecture targeted by algorithmic skeletons and the programming model supported by paraML. Algorithmic skeletons in general provide a useful means of structuring and managing parallelism, and may be optimised by a skilled programmer. The benefits of software reuse and leveraging such knowledge for other programmers is as clear for high performance programming as it is for sequential programming.

8.3 Object stores

8.3.1 Overview

One of the major distinguishing features of paraML is the distribution of processing resources encapsulated by processes. Similarly, all data is managed by processes and hence distributed. These design choices are made to promote locality of data access, given the latencies involved in accessing data in a distributed address space. An alternative paradigm for data storage and management is a logically shared global store of objects. This object store may be accessed by any computing entity with the appropriate permissions. In this investigation, the mechanisms for providing access to individual objects and maintaining coherence are of particular interest. The key issues that an object store must address are coherent mutable objects, the reintroduction of sharing among processes, and reduced copying of objects between processes.

Various efforts have been made at extending ML with objects [RV96, RR96]. The extensions are typically interested in developing object-oriented notions, such as object data encapsulation with methods. In this exploration, objects are treated more simply, just as data values which may be updated. In this sense, they are similar to reference

variables in ML, except that these objects are accessible to any process, not just locally within a process.

Object stores grew out of research in two areas: *persistent* programming languages such as Napier [MBC+89], which did not desire the overheads of database storage, and toolkits for database construction, such as EXODUS [CDV88]. Object stores take a position in the spectrum of data management research which includes: relational databases [Sto94]; object-oriented databases (OODBs) [ZM90]; object stores; and tuple stores [ACG86]. ML has also been used with tuple stores in the development of ML-Linda [SC91], and in exploring transaction programming models [HKM+93, WFH+93].

8.3.2 Basic model

An object store returns to the notion of a single shared address space (or environment) that is accessible by the computation. In many languages, memory management is performed explicitly by the program, and thus memory is treated simply as a sequential block of bytes. Concurrent languages permit multiple threads of control to access this single environment. Figure 8.1 represents the situation of a program executing in this way.

An object store provides a slightly higher level notion of what this environment provides, since it must also define what constitutes an object. If the object store is intended to be language-independent, only very weak definitions of what constitutes objects are possible, typically consisting of some control information, a collection of pointers and a contiguous byte sequence [BM92]. If the object store is tied into a language-specific framework, that language's definitions of what constitutes values in the languages can be utilised. Either way, object stores hold collections of objects. Operations exist for individual objects to be added or removed from the store, and for their values to be made accessible for computation. Objects are typically identified by a name. Figure 8.2 illustrates this form of simple object store.

A process-oriented construction of an object store can be characterised as a single process which other processes know about. Objects are stored by this process, and access by other processes is controlled through whatever inter-process communication system exists. The primary difference of this form of object store is that the memory is now conceptually disjoint, and thus access to objects involves access to remote environments. Access protocols which specify how data or computation are moved between these environments are necessary for the implementation. Figure 8.3 illustrates a process-oriented object store. If the system uses dynamic process creation, the object store must be created dynamically too, and information about its existence communicated to other processes. Static process configurations would assign the object store role to one process at configuration time.

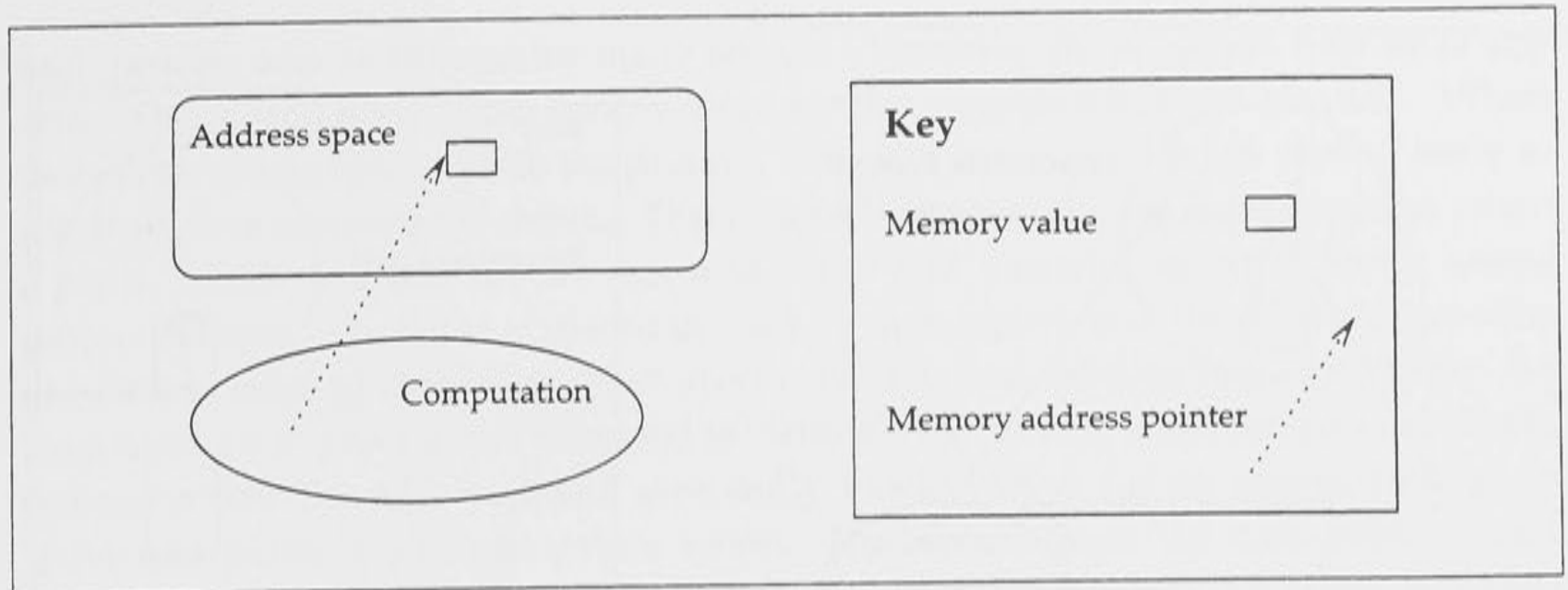


Figure 8.1 – Shared memory program execution.

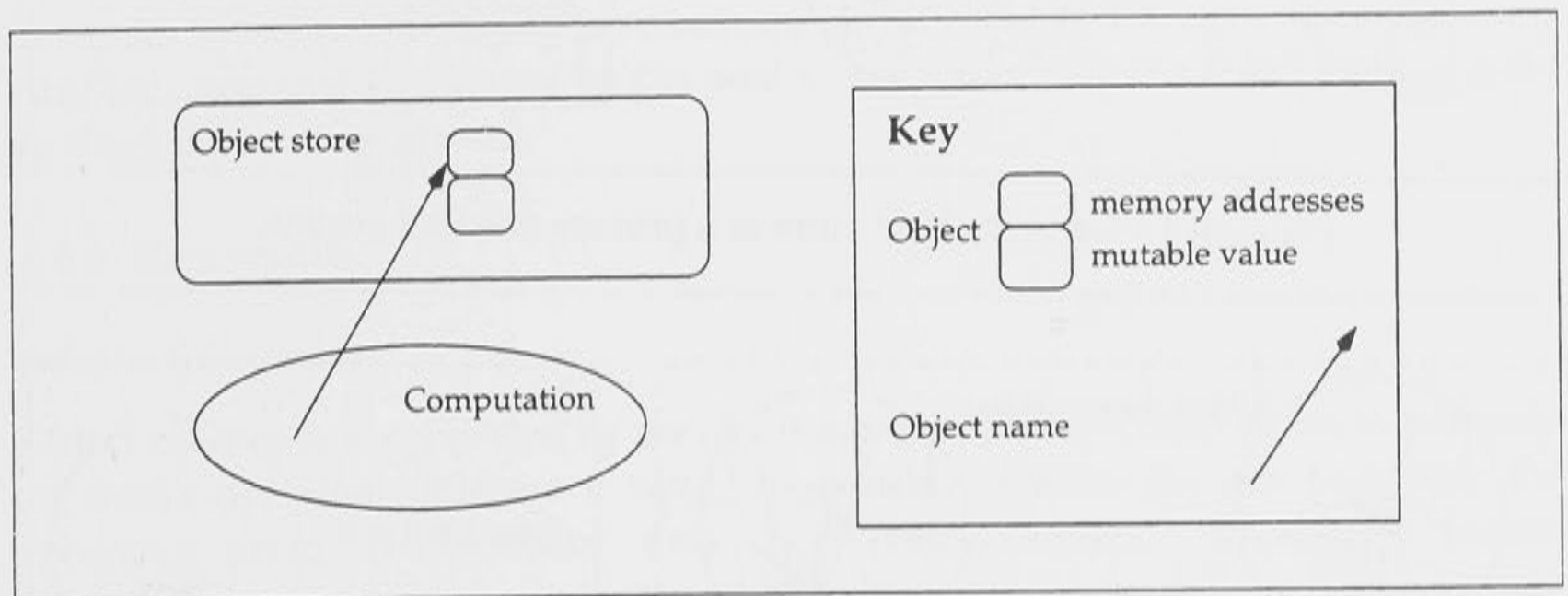


Figure 8.2 – Generic object store in shared memory.

8.3.3 Realisation in paraML

For the purposes of this experiment in paraML, an object is characterised by:

- a mutable data structure whose value is some first-class ML object;
- a set of access methods for the data structure;
- a protocol for interpreting access requests;
- a unique name that identifies the object.

Since paraML is a process-oriented language, object stores in paraML are also going to be process-oriented. The basic model of a single object store process remains; however there are a number of extensions in the realisation of the object store on the grounds of efficiency. Instead of a single object store process, there are a group of processes for storing objects. This alteration permits the distribution of objects across processes on every PE of the machine. Secondly, instead of a centralised object allocation strategy, the implementation of the object store provides a distributed object allocation strategy, which allows computation processes to contact processes in the object store directly. The object allocation strategy can be either a very simple round-robin one or it can take user advice on the desirability of object co-location. The

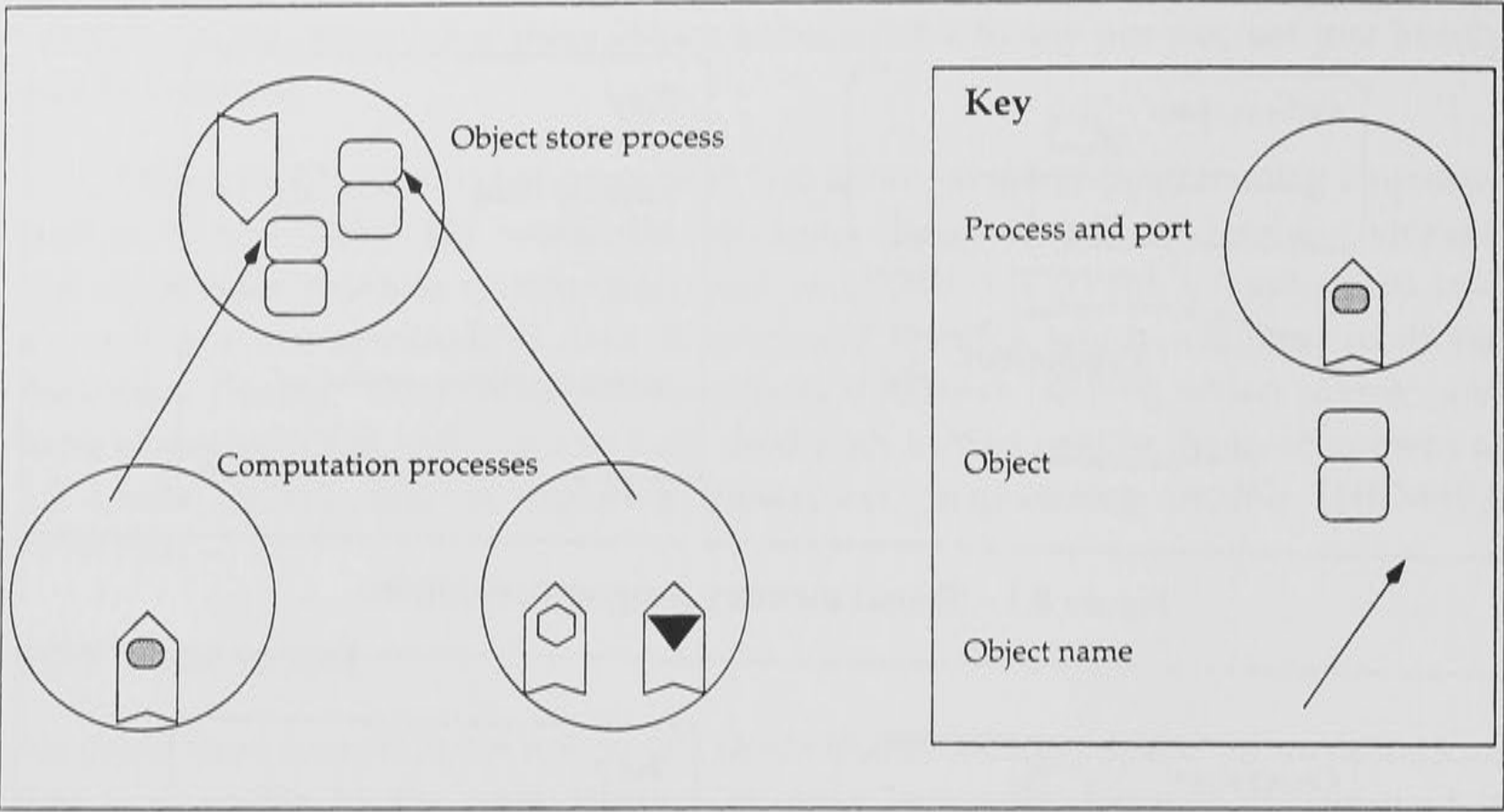


Figure 8.3 – Generic object store in a process-oriented system.

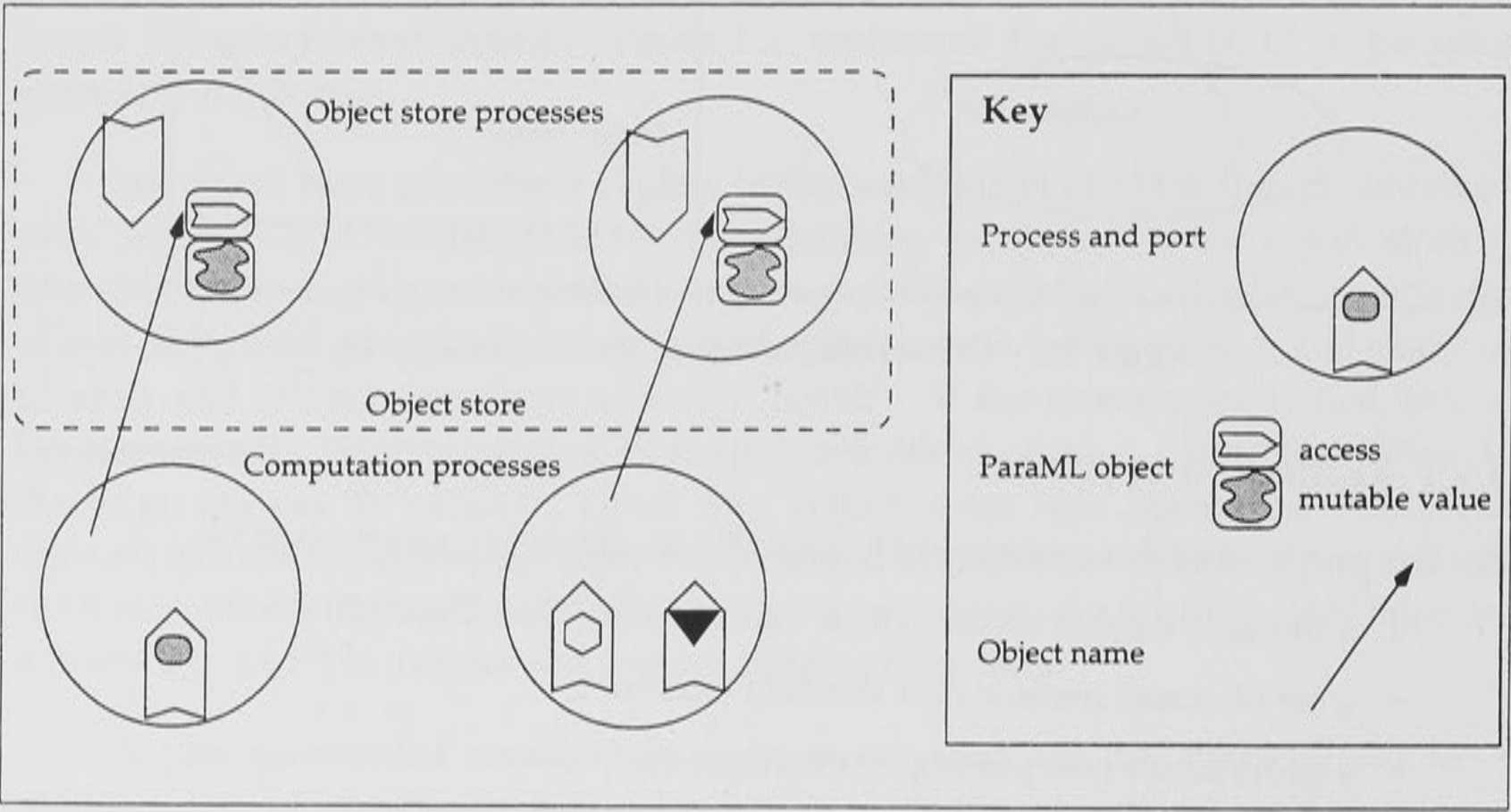


Figure 8.4 – Object store in paraML.

construction of the object store is represented in Figure 8.4. Note that there is nothing that has specified that only one object store can exist; indeed, a multiplicity of object stores can exist, each uniquely identified by a name. Individual objects are identified by a name; object names will actually be represented by port names. The use of port names carries the typing information essential with typed objects in ML. The scope of knowledge about how to contact object stores or objects within object stores is governed by the usual rules which determine scoping of port names in paraML.

The object store implementation provides an interesting demonstration of the impact of concurrency within processes. The key idea behind the object store is that

each process acts as storage for many objects. However, these objects may be of any type. The central body of the process must service requests for object creation. Where there is no concurrency within the process, it is also necessary for this central body to service access requests for objects. This is done by having the creation operation return a tuple, which will test for an access message and perform an appropriate access protocol action. The tuple is placed in a list of such tuples, and the derived operation `choose` is used to check for object access requests and creation requests. When the language within processes is extended to include concurrency, a creation request simply spawns a new thread which will repeatedly block waiting for an access message to arrive and perform the appropriate action. The advantage of the concurrency-based approach is that the process is only active when servicing requests, and does not need to continually probe for message arrival. The performance difference between the two is significant when object store processes are placed on every PE, since other processes are then unnecessarily delayed by the need to keep activating the object store process to check for message arrivals.

8.3.4 Key results

8.3.4.1 Coherency

Object coherency is supported by the object access protocol. Object access is currently an atomic operation, although it would be possible to allow the user to specify the coherency protocol at creation time of individual objects. Serialisable atomic transactions over collections of objects or compound objects could be programmed through the use of global locks using standard techniques, as has been done previously with ML [HKM+93].

8.3.4.2 Sharing

Sharing of data structures across processes can now be supported in a loose sense, in that objects can reside in the object store, and thus any process that knows the object's name also knows how to access the object and therefore "shares" it with the other processes.

8.3.4.3 Reduced copying

Use of an object store cuts down the amount of object copying required. Transmission of objects can now be done by reference alone – an object's name – which is relatively small in size compared to the potential size of the object. Another advantage brought about by the nature of ML is that computations may be sent to the object to perform locally. Thus, an entire function may be sent to the object store process hosting an object. The implementation of the access protocol passes the current value of the object to the function, which is used to perform some computation, and then the result is sent back to the accessing process. This facility promotes locality of data access, rather than requiring copying of data values backwards and forwards between store processes and computation processes.

8.3.5 Limitations in the object store model

There are a number of limitations with the current object store model. The reintroduction of sharing is particularly problematic since the memory remains distributed and is now under the control of individual processes, rather than a memory manager. Since object access is being managed through message passing to an object's access port, it is clearly not as efficient as direct memory manipulation, so there is a question of efficiency to take into consideration. Concomitant with efficiency, the issue of object granularity becomes important. Lastly, the implementation strategy makes strong demands on process scheduling in the paraML runtime system.

8.3.5.1 Garbage collection

Object names may be shared across many processes. Individually, a computation process may garbage collect an object name once it is identified as no longer reachable. However, since the garbage collection mechanism is not global, there is no mechanism by which the objects themselves can be garbage collected in the object store processes. This is a serious weakness since one of the main advantages of using high-level languages such as ML is that they do all the memory management for the programmer.

8.3.5.2 Granularity

An obvious factor in the search for efficiency will be object granularity. If global objects are too small (for instance, an integer), then the costs involved in accessing them will be very large indeed relative to the computation performed involving the object itself. Obviously, the bigger the object, the relative cost of object access will be smaller. These performance characteristics impose some natural limits on the granularity of shared objects in programs, which is very similar to the problems faced by DSM systems.

8.3.5.3 Process scheduling

The other requirement of the current implementation of the object store is that process scheduling is both efficient and frequent. Since individual objects are managed by object store processes (which may be co-located with computation processes) it is essential that mechanisms are available to service access requests promptly. The obvious solution to this problem is to use message-driven interrupts to drive process-rescheduling. The implementation of object store processes without concurrency still incurs unnecessary overhead due to the need to scan through the list (possibly quite large) of possible guards to find the one which is satisfied by a new access request message.

An alternative implementation which also does not require concurrency solves this particular problem. Instead of just identifying an object by a port name, functions that perform the access operations also become part of the object identity. However, for these to work correctly after being copied to a different process, they must have a unique way of identifying the object's state. This identity obviously cannot be the

object's reference variable address, since a copy will have been taken if sent to some other process. The solution here is to use the technique of encoding the object's state by sending to, and receiving from, a private port. This technique is that same as suggested by Reppy for shared references in CML [Rep92]. The port's message queue then buffers the object state. Since the port is local to the object store process, the object's state can be retrieved or set only by the access functions when executing on the object store process. While removing the overheads of checking lists of access requests, this solution leads to the object identifier becoming a much bigger data value than just a simple port name. It also requires bigger overheads in message transmission of access function closures to the store process.

The best solution is to permit concurrency within processes. The individual threads within the process are then blocked until a message arrives to access an object. Message-driven rescheduling for a process will result in the process being activated only when necessary to service an object access or creation request. Implementing threads within processes efficiently requires some additional support from the paraML runtime system, as discussed in §9.5.

8.3.6 Summary

The experiments with the object store system reveal that scheduling of processes in paraML is critical for efficient performance. The best approach appears to require pre-emption of a currently-executing process on message arrival at a PE in order that the sending process is not unnecessarily delayed. It also suggests that a priority system for process scheduling (for server processes) would be a useful addition to the paraML runtime system. Overall, the experiment illustrates that the paraML system can be used to construct object stores relatively successfully. They provide a useful abstraction which emphasises the distinction between local mutable values and globally accessible objects which may be shared across a number of application processes. The benefits of ML's data abstraction and information hiding features for constructing such alternative programming models are apparent in this experiment.

8.4 SIMPLE

The SIMPLE benchmark [CHL78] is a hydrodynamics simulation of a pressurised fluid within a spherical shell. It has been studied for a variety of different parallel programming languages, and there exists a sequential implementation in ML as part of the SML/NJ benchmark suite. Various choices in parallelisation strategies for SIMPLE are outlined by Lin and Snyder in their discussion of implementing SIMPLE using their portable parallel programming system Orca [LS91].

The simulation is typically encoded by projecting the discretized spherical domain onto a 2-dimensional Cartesian space, which is naturally represented as a 2D array. Attributes of the fluid, such as viscosity, specific internal energy, temperature, heat conductivity and so forth, are maintained for each discrete point in the space.

The progression of time is also broken into discrete steps, satisfying the Courant condition that the time for a signal travelling at the speed of sound across a grid cell is greater than a time step in the simulation. At each time step, a number of computations are performed to recalculate the state of the fluid attributes at each point in the discretized space. These calculations require some knowledge of neighbouring points' attribute values, with different sets of neighbour points required for different attributes. The calculation of the attributes typically iterates over the two dimensions of the attribute arrays using two for-loops.

In parallelising the sequential ML implementation of SIMPLE, there are two obvious approaches. The first approach uses a data parallel strategy, where the 2D array is partitioned among a number of processes, but the algorithm remains effectively sequential. The calculations involved at each step are sent to the processes to perform. The implementation of a 2D array data abstraction distributed among processes is relatively straightforward. Included in the array access operations is the ability to execute for-loop calculations over the local array elements. The critical requirement for efficient performance is that local array access operations remain fast. The sequential ML implementation represents each of the attributes with a separate 2D array. Since the array access operations are encoded through access to ports in the 2D distributed array data abstraction, local accesses are in danger of being unacceptably slow.

A better strategy is to encode a state attribute datatype, which is used as the type parameter of the distributed 2D array data abstraction. All local array accesses to any of the attribute values can then be performed without recourse to message passing. A side effect of this approach is that the array creation costs are only incurred once, rather than for each attribute. The disadvantage of this approach is that in certain of the calculations, knowledge of neighbouring elements is required. This non-local access requires sending messages to other processes involved in the 2D distributed array abstraction. Without concurrency in the ability to handle array access requests, such non-local access requests can lead to deadlock. The solution requires servicing any array access request that comes from a remote process with a new thread. The implementation of the interface to the for-loop calculations then needs to perform a barrier synchronisation on completion of the request to prevent the program progressing to a new step of the simulation prior to it being completed in all the distributed 2D array processes.

The second approach requires a more explicitly parallel strategy. As before, the 2D array is partitioned among a number of processes. However, the data movement requirements among neighbouring processes are now made explicit. Thus at each step of the simulation, sending and receiving data to and from other processes is performed explicitly for each of the attribute calculations requiring information from neighbouring points if these points exist on other processes. These changes avoid the deadlock problems associated in the data parallel implementation without having to use concurrency within processes. This simplification comes at the cost of making the overall parallelisation of the sequential code more complex. The choice of how to

partition the 2D array among the processes is also open. Lin and Snyder analysed the relative merits of partitioning the space in strip and block decompositions, and determined that the block decomposition always produces better results due to decreased overall message size [LS91]. This second approach of an explicitly parallel strategy is more naturally suited to paraML than the first. It also avoids the complexities of dealing with concurrency and synchronisation requirements in the construction of the 2D distributed array data abstraction.

8.5 Conclusion

In both of the alternative programming paradigms explored, there are a number of advantages provided by paraML. The flexibility gained from the first-class nature of process names and port names, and the dynamic creation of processes and ports are strong attributes of the language. In building the algorithmic skeletons and the object store system, the desirability of collective group-based operations is clear. An interesting avenue for exploration would be marrying collective and process-oriented computation models with particular emphasis on high performance. However, this support should not be at the cost of flexibility for individual processes.

The overall conclusion of the investigations into alternative high performance programming paradigms is that paraML's process-oriented model provides a useful basis on which to experiment. However it cannot be wholly relied upon to provide an efficient solution for any conceivable model of programming. Nor does the safety emphasis of paraML's design translate into equivalent levels of safety in implementations of other paradigms. For example, object store systems typically promote a transaction-based model for guaranteeing data coherency. While paraML may facilitate certain aspects of a transaction system implementation, the safety of transactions would depend on the quality of the implementation. ParaML is most suited to paradigms which adopt or are based on similar characteristics of distributed address spaces and message passing communication facilities among collections of computation entities.

More extensive application development is merited before conclusive statements can be made about paraML's efficiency for high performance programming. The experiments with SIMPLE illustrate that explicitly parallel solutions can be encoded straightforwardly, albeit with increased code complexity. Providing support for internal concurrency within processes would enable a greater variety of approaches to be taken when parallelising an existing sequential implementation. The next chapter goes on to describe the details of implementing paraML itself.

9. Implementation

9.1 Introduction

This chapter examines the implementation of the paraML system using current software technologies. The design of the runtime system for paraML is described, together with a discussion of how the sequential ML runtime system has been extended. The paraML runtime system is one approach to satisfying the requirements specified in the design and theoretical modelling described in Parts II and III. There is a particular concern for achieving portability, and the use of MPI as a communication platform is analysed with respect to this goal. Limitations of the implementation and future possibilities for development are also included.

9.2 ParaML runtime system design

9.2.1 Motivation

The process-oriented model for computation described in Part II gives a clear account of three layers of abstraction. The top layer consists of the paraML operations embodying processes and ports. The next layer is the virtual processors, and the bottom layer is the physical machine layer of collections of PEs, connected by an inter-PE connection network. These components of the programming model are represented in Figure 9.1. The abstract design description for these layers is realised through available software technologies. The goal of this implementation is to provide a safe and efficient system with which to execute paraML programs.

The fundamental requirement for implementing the programming model is a paraML runtime system. ML compilers use runtime systems to manage dynamic aspects of an ML program's execution such as garbage collection and interfacing to the operating system. Similarly, the paraML runtime system is responsible for managing dynamic aspects of a paraML program's execution. These include process scheduling, message delivery, and error handling.

9.2.2 Components of a paraML runtime system

Within each process, the language for describing computation is ML, extended with the paraML operations for processes and ports. Thus a minimal requirement in the paraML runtime system is support for executing ML programs. Naturally, using an existing ML compiler and runtime system to do this is desirable, and the SML/NJ compiler has been chosen for this purpose [AM91]. The reasons for this choice are explained in more detail in §9.2.5 and §9.4.1.

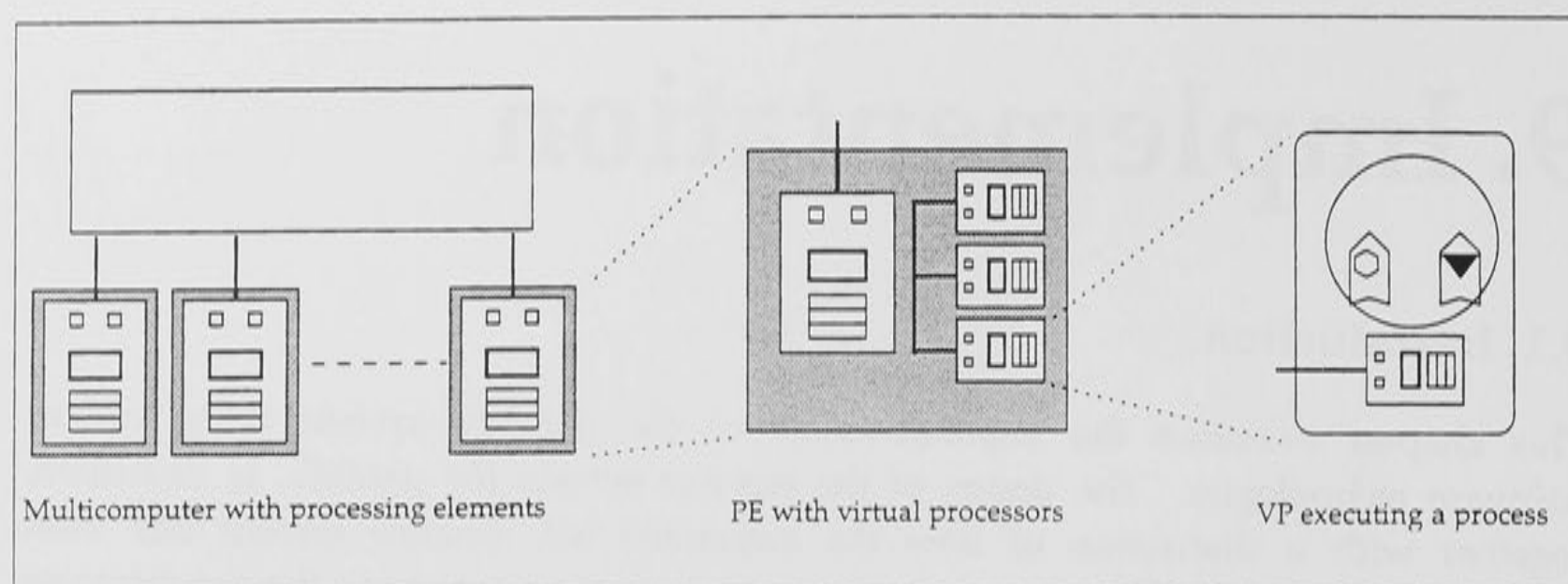


Figure 9.1 – Programming model components.

Support is required for executing programs which will act as the processing elements in the programming model. Additionally, inter-PE communications must be supported. The chosen system is MPI [MPI94], which provides an efficient and portable inter-PE communication environment. Interfaces to the MPI routines are available in C or Fortran. MPI systems such as the one for the Fujitsu AP1000+ [Sits95] adopt a SPMD model, where a single program executable is used for each MPI process. The MPI system discriminates each process by its rank within the global communicator `MPI_COMM_WORLD` which connects all processes. At this layer, there is no knowledge of ML or its type system. MPI does provide its own type representation for values communicated between processes, but this type system is simply used to describe the byte layout of values within a process's memory. There is no inter-process type checking, and the MPI standard remarks that if a process receives a value and declares it to be a different type to that used when sending the value, then there is no guarantee about successful execution of the program.

The paraML runtime system then is the program which executes as an MPI process. This program must interface with the MPI communications operations to support dynamic communication linkages between paraML processes. It must also support multiple virtual processors, each executing a paraML process which may require full ML functionality.

9.2.3 Embedding the ML compiler/runtime system

The SML/NJ compiler [AM91], version 0.93, provides either interpreted execution of code or batch compilation of modules. Executing an ML program minimally requires the runtime system and either a dynamic link loader (for separately compiled modules) or the interpreter. Together, these occupy between 2 and 5 MB of program executable size. It is clearly undesirable to provide support for several of these programs within the same paraML runtime system. Instead, the paraML runtime system provides a single ML runtime system and interpreter or link loader. Thus the paraML runtime system (which executes as each MPI process) is constructed as an extended SML/NJ

link loader/interpreter and runtime system, executing a program written in ML to support inter-PE communications and multiple VPs.

9.2.4 Inter-PE requirements

The paraML runtime system must deliver and accept communications from the paraML runtime systems on other PEs. The mechanisms by which the SML/NJ runtime system is extended to interface with MPI are described in §9.4. On receiving a request (which may be requests for VP/process creation, other process control requests, or communications for ports), it must be handled, and a response sent to the requester. These requirements suggest that the paraML runtime system adopt an event-driven core, where events are generated by message arrival.

The paraML runtime system is also responsible for establishing ML type information about newly-arrived messages. The MPI layer delivers messages with no type information, and nor does the SML/NJ system carry type information in the runtime representation of values. (Note that other ML compilers, such as the TIL compiler [TMC+96], preserve some or all type information in the runtime representation of values.) Thus mechanisms are required to guarantee that the ML type of a received message is identical to the ML type of the message in the sending paraML runtime system.

9.2.5 Intra-PE requirements

The paraML runtime system must provide support for multiple VPs. Each VP executes one paraML process, and is responsible for managing process state and the communication ports. Since there may be many VPs, the processing resources of the PE must be shared among them equitably. VP scheduling is thus an exercise in concurrency within the paraML runtime system. The SML/NJ compiler provides various programming facilities which may be used for supporting concurrency, which is another reason the SML/NJ compiler was chosen.

The paraML runtime system must also successfully deliver communications between any paraML processes for which it is responsible. The central requirement is that the semantics of communications between processes within the same paraML runtime system is identical to communications between processes from different paraML runtime systems.

9.2.6 ParaML operations

Just as the paraML runtime system is written in ML, constructing the paraML operations in ML is also advantageous. Each operation may require access to the VP data structures and some of the VP scheduling mechanisms. These requirements are far simpler if their implementation can manipulate these mechanisms within the language framework of ML. Choosing to build the paraML operations in ML also means that alterations to the underlying SML/NJ compiler system do not require

changes to the implementation of the paraML operations. This simplification is a significant consideration with limited resources for language development.

9.3 ParaML runtime system implementation

Some language runtime systems are heavily dependent on operating system support for such things as process/thread scheduling. Avoiding operating system dependencies is desirable in order to maximise portability. Since the paraML runtime system is written entirely in ML, with minimal extensions required for interfacing with the MPI communications layer, many of the operating system dependencies are eliminated.

9.3.1 Multi-threading support within the processor abstraction

The paraML runtime system provides support for both the processor and VP layers of the machine model. The processor layer is emulated by the paraML runtime system itself. An executing paraML system will consist of a collection of paraML runtime system programs. These are initiated as MPI processes, with the entire set connected by the `MPI_COMM_WORLD` communicator. On a multicomputer each MPI process executes on a separate node. Alternatively, each MPI process may simply be executed by a UNIX process on a workstation version of MPI. The design of the paraML runtime system must also permit the use of only one MPI process.

Each paraML runtime system uses a collection of threads to support multiple VP abstractions and the handling of messages. Multi-threading within the paraML runtime system is implemented using continuations, in a fashion similar to that described by Appel [App92]. In SML/NJ, there are two functions for manipulating continuations: `callcc` and `throw`. The `callcc` operation applies a function (of type `'la cont -> 'la`) to the current continuation of the program and returns a value (of type `'la`). The `throw` operation (of type `'a cont -> 'a -> 'b`) invokes a continuation with an appropriately-typed argument. This operation never returns.

The implementation of the continuation functions in SML/NJ is extremely efficient. The SML/NJ compiler is constructed using a continuation-passing style (CPS) semantics, originally due to Steele [Ste78] and described for SML/NJ by Appel [App92]. Reppy reports in detail on the advantages of using SML/NJ's support for continuations for efficient implementation of thread support [Rep92]. The code for implementing `callcc` and `throw` is very similar in performance to that for function calls, although restoration of the enclosing exception handler must be performed for `callcc`, thus breaking tail recursion.

9.3.2 Basic components of the runtime system

The message handling thread is the initial component common to all runtime systems. It must respond to any new message which arrives at the runtime system. The

Request	Information	Prerequisites	Action	Reply
process	requester	none	create new VP abstraction, with handlers for port and execute requests	new process name
port	process name; requester	process is not terminal	pass request to VP port handler; it creates a new port queue	new port name or failure
execute	process name, thunk; requester	process is not terminal or executing already	pass request to VP execute handler; it changes execute state and creates new thread for thunk to execute, moves VP to suspended VP queue	success or failure
port message	port name, value; requester	process is not terminal	pass message to port's handler; it queues message and moves a receive blocked VP to suspended VP queue	success or failure
reply	requester id; reply	none	pass reply to descheduled VP thread, move VP to suspended VP queue	none

Table 9-1 – Categories of message requests accepted by the message handling thread.

response to a message depends on what information is carried by the message. There are five main categories of message, which have differing prerequisites, information, actions and responses, as summarised in Table 9-1.

Receipt of a process request results in the creation of a data structure containing information for the VP abstraction's support of a process. This data structure is inactive, but contains various handlers for port, execute and port message requests. The initial process execution state in the VP is set to awaiting. As detailed in the table, the response to requests is dependent on the current state of the process: awaiting, evaluating, or terminal. The responses are prescribed by the semantics given for the parallel evaluation relation in Chapter 6. Once a process has received an execute request, a thread of evaluation is associated with the VP data structure. Execution state in the VP is set to evaluating until the computation completes; then it is set to terminal. If the evaluation must be suspended for any reason, the state of computation is captured by a continuation. Thus the basic components of the paraML runtime system consist of a collection of threads, including the dedicated message handling thread, a collection of VP data structures, and the single processor data structure. These interactions are coordinated by the scheduling mechanisms in the paraML runtime system.

9.3.3 Scheduling

Since continuations are first-class objects, an inactive thread's continuation may be queued for rescheduling in data structures. Rescheduling of threads usually requires only a small amount of data structure manipulation (enqueueing a continuation/identifier pair onto a descheduled queue, dequeuing a continuation/identifier pair, and updating information about the currently executing VP), before throwing to the scheduled thread's continuation.

The paraML runtime system always has one thread active, whether it be the message handling thread or one of the threads for an active process. The identifier of the current thread is maintained, together with the VP identifier if appropriate. The continuations of inactive threads are stored in various data structures, which facilitate the scheduling policies of the current implementation. These data structures are as follows:

- the dedicated message handler thread;
- a dedicated interrupted process thread;
- a FIFO queue of executable, but currently suspended, processes, together with the ids of their VPs;
- *recv*-blocked processes, associated with the port on which they hope to receive a message;
- a hash table of descheduled processes, awaiting a reply from a request message they have sent; the requests are uniquely keyed which allows the reply to find the requesting process in the table.

For single-threaded processes, no other data structures are required. However, we have argued previously that concurrent execution within processes is essential [BNS+94]. Concurrency within processes has also been shown to be useful in the implementation of systems such as the object store experiment discussed earlier. Modifications to these data structures in support of concurrent execution within processes is discussed later in §9.5.

A number of different events impel the rescheduling of threads managed by the paraML runtime system:

- message-arrival signals;
- interval timer signals;
- blocking paraML operations;
- request/reply paraML operations.

9.3.3.1 Message-arrival signals

The signal handler for message-arrivals attempts to suspend the currently executing process if there is one. If it is unable to do so due to the process executing code within

an atomic region, it sets a flag to indicate the message handler thread should be called on exit from the atomic section. When permitted, an executing process is interrupted and its continuation stored in a dedicated data structure, unless there is no currently executing process. The message handler thread is invoked and performs appropriate actions for any newly-arrived messages. When no more messages are available, the message handler thread checks to see whether there is an interrupted process thread to be rescheduled. If there is, the message handler thread suspends and invokes the interrupted process. If there is no interrupted process, the suspended process queue is checked instead. If the queue is not empty, the message handler dequeues the first suspended process thread, suspends itself, and invokes the process. If there are no suspended processes either, the message handler thread performs a blocking probe to await a new message.

9.3.3.2 Interval timer signals

The interval timer signal handler is similar to the message-arrival signal handler. However, instead of invoking the message handler thread, it simply results in the suspension of any currently executing process. The thread for this suspended process and the id of its VP are enqueued on the FIFO queue of suspended processes, and the first member dequeued and invoked. The interval timer is used to guarantee progress to all processes being executed on a paraML runtime system.

9.3.3.3 Blocking paraML operations

The only blocking paraML operation is `recv`. If this operation is called and there are no messages ready in the port's queue, the process's executing thread is suspended, and its continuation placed in the data structure associated with the port. The continuation of the next process on the suspended process queue is dequeued and invoked. If no process is currently suspended, the message handling thread is invoked instead. At some later time, if a message arrives for the port, the suspended process's continuation is moved from the port data structure to the suspended process queue.

9.3.3.4 Request/reply paraML operations

Any operation involving communications with other processes requires a request message to be sent, which is uniquely identified within the paraML runtime system. The requesting process's executing thread is suspended, and its continuation inserted into the hash table with the identifying key. The continuation of the next process on the suspended process queue is dequeued and invoked. If no process is currently suspended, the message handling thread is invoked instead. At some later time, when the reply arrives for the request, the suspended process's continuation is extracted from the hash table, and placed back on the suspended process queue.

Currently, there is no decision to alter priorities of executing processes based on whether a blocked or reply-suspended process has just been made executable again.

More detailed profiling of the performance of the current implementation would need to be done to determine whether this was a useful scheduling mechanism.

9.3.4 ParaML operations

Given the support from the processor and VP abstractions and the scheduling mechanisms in the paraML runtime system described above, implementation of the core paraML operations is straightforward.

9.3.4.1 `process` and `self_id`

Creating a new process requires selecting a paraML runtime system to host the virtual processor. A request message is sent to it, and the requesting thread is descheduled to await the reply. When the reply is received, either it will indicate that it was not possible to create the process (due to memory constraints for example), or it will report the new process's name, which is then provided to the requesting thread which is rescheduled. Determining the process's name with `self_id` just involves extracting the process name value stored by the VP abstraction.

9.3.4.2 `execute`

An execute request packages up the thunk and the key for the target process's execute request handler. The request is sent, and the requesting thread descheduled with a continuation which will act on the contents of the reply message. When the reply is received, either it will indicate success or it will report that the request could not be satisfied in which case an exception will be raised.

9.3.4.3 `port`

A port creation request simply sends a request message to the target process's port creation request handler. The requesting process is descheduled with a continuation to act upon the contents of the reply message. When the reply is received, either it will indicate success and carry the new port name or it will indicate the reason for failure in which case an exception will be raised. The handler is created at process creation time.

As explained earlier, the message containing the new port name simply arrives as an undifferentiated array of bytes. The message body component that is the port name must then be coerced to the correct (weakly polymorphic) type and returned to the requesting process computation. This mechanism is similar to the treatment of other messages, where the message content is treated as a generic object (the SML/NJ type given by `System.Unsafe.object`), and only coerced when the actual ML type can be identified from the other information contained in the message. Treating messages as generic objects is beneficial in maintaining the port queues. Since the VP data structures are implemented in ML, it is necessary to have collective data structures (such as hash tables) for maintaining the port queues. However, collective data

structures must be over uniform ML types, and thus the hash tables are for queues of `System.Unsafe.object` values.

Since there are no extensions to the fundamental datatypes in SML/NJ, port names must be constructed from existing ML values. The particular requirement is for port names to carry weakly polymorphic type information. Since port names are used to identify location information (the PE's id, the owning process's id, and a unique integer key), there is no intrinsic field to carry type information. Thus port names are actually constructed as records, which contain the identifying information and a `'la list` field initialised as a `nil` list. This field is used to carry the type of the port name, and since the field never contains anything other than `nil`, port names are always of uniform size.

9.3.4.4 **send, recv and probe**

A sending request attempts to deliver a message to a target port's queue. As usual, the requesting process is descheduled to await the reply. The reply message will either indicate that the message was successfully inserted into the queue or indicate the reason for failure. In the latter case, an exception is raised in the rescheduled requesting process.

The `recv` and `probe` operations have no need to generate request messages. Instead, they examine the state of the queue associated with the port given as argument. In the case of `probe`, a simple boolean value is returned to indicate the presence or absence of messages in the queue. In the case of `recv`, if a message is available in the queue, it is removed and returned. If no message is available, the thread is descheduled and placed into a continuation data structure awaiting message arrival on the port.

The type system is used to make sure that sending a message to a port only occurs when the type of the value sent is of the corresponding type to the port name's parameter type. Thus it is essential that ports are identified in a unique way throughout the entire collection of paraML runtime systems. Messages carry with them the unique id of the port for which they are destined. On arrival, all messages are stored in the queues as generic objects, which are then coerced to the appropriate type when removed from a port queue by the `recv` operation. All type coercion happens in the implementation of the paraML runtime system and the paraML operations. Coercion is necessary only due to the fact that no ML type information is carried with the runtime representation of values when received from a communications network as an array of bytes.

9.3.4.5 **paraml**

To commence execution of the root process in a paraML program, the `paraml` function is called with the thunk for that process. All paraML runtime systems execute the same basic program up to the point of calling this function. An initialisation

function is called to establish the paraML runtime system programs as a collection of MPI processes. All paraML runtime systems initialise their processor abstractions, and set off their message handling thread. The first paraML runtime system also initialises a virtual processor abstraction and starts execution of the root process.

9.4 ML compiler and runtime system extensions

High performance of a paraML system is not achievable without some support for execution on parallel computers. Since the paraML runtime system is an ML program in its own right, high performance execution of paraML programs requires multiple paraML runtime system programs to cooperate. Each runtime system will execute on a different node of the computer. Basic support for such multiple program execution comes from an operating system at the most primitive level and an MPI package at the layer above. The first set of extensions to an ML system is thus interfacing to a core set of routines from the MPI package. The second set of extensions are those to marshal and unmarshal data objects into contiguous byte arrays for transmission between the paraML runtime system programs.

9.4.1 ML compiler

The ML compiler chosen as the basis for extensions to support paraML was the SML/NJ compiler (version 0.93), from AT&T Bell Laboratories and Princeton University [AM91]. This system was selected for a number of reasons:

- it is freely available and complete source code accompanies the distribution;
- the generated code is of high quality, and it conforms almost totally to the formal definition of Standard ML;
- automatic *marshalling* and *unmarshalling* operations are provided for data objects to and from file storage;
- mechanisms exist for interfacing ML code with C routines.

The mechanisms for calling C language routines from ML have been vastly improved in new versions of the compiler (1.09 and later), but at the time of implementation, there was only partial support for marshalling of data objects. However, with version 0.93, adding interfaces to a small set of MPI routines was practicable. The routines are added to a library of C functions compiled into the ML runtime system. ML wrappers to these routines are made available through a module in the pervasive environment that is available on startup of the system. The primary disadvantage of the SML/NJ system is its large memory requirements, with the basic compiler/runtime system for SPARC architectures requiring over 4 MB of memory. Since the extended SML/NJ system is embedded as the core of each paraML runtime system, there is significant overhead in initialising each MPI process.

9.4.2 Basic ML-C interface operations

These are the set of operations necessary to support multiple paraML runtime system programs:

- invoking the MPI initialisation and finalisation operations;
- querying the MPI system on the number of MPI processes and the rank of an individual process;
- sending arbitrary ML objects to another MPI process;
- receiving arbitrary ML objects;
- probing for message arrival.

In total, there are seven ML-C operations required. All but the sending and receiving operations are straightforward to implement. Each requires conversion of any ML arguments to a C representation, call to the appropriate MPI routine, and return of any results after conversion back to an ML format. An additional operation was provided in the runtime system so that ML-format strings of arbitrary size could be allocated in the ML heap from the C runtime system. This operation is a minor modification of an existing string allocation routine.

The SML/NJ runtime system uses `brk` and `sbrk` to perform all of its memory allocation (explicitly setting available process memory), and typically `malloc/free` cannot be safely intermingled with these operations. However, an alternative implementation of `brk/sbrk` was created for the AP/Linux operating system [TMS+96] by Andrew Tridgell, using `mmap` on `/dev/zero`. This alteration was required to enable the MPI library and the other extensions to the ML runtime system to use standard `malloc/free` operations without potentially corrupting the SML/NJ-controlled memory.

9.4.3 Marshalling operations

The sending and receiving operations are the most complex, primarily because they must accept arbitrary ML values for transmission. The key problem here is that the representation of an ML value at runtime may have different components of the object in widely different areas of memory. The MPI communications package only transmits contiguous byte arrays as messages. Thus there needs to be a mechanism which maps an arbitrary ML value into a contiguous byte array. This mechanism is known as *marshalling*. Similarly, when a contiguous byte array is received, it may require some work to generate a recognisable ML value from the byte array representation. This reverse operation is known as *unmarshalling*. The formal description of `copy` in Chapter 6 is one model of the semantics of what happens during the marshal operation. It should be stressed that it is only a model, not the actual algorithm used in the implementation.

The SML/NJ system provides two operations to perform such operations, but the implementation in version 0.93 writes and reads these arrays to and from files. The implementation also produces a header of three integers, used to provide information about the location of the start of the object within the byte array. The operation is analogous to the problem of garbage collection of a single object using a copy algorithm, where objects are copied between “from” and “to” spaces. The operations are capable of marshalling any first-class ML value, including functions (by generating an appropriate closure which gives meaning to the free variables of the function expression) and reference values (where a deep copy of the object which is referenced is included).

Modifications for sending were required so that the header and byte array could be sent to a destination MPI process instead of being written to file. Ideally, the implementation would package up the header and the byte array into a single data object, thereby incurring only a single message overhead, rather than sending the header and byte array separately. However, the main problem with such an approach is the extra copying and buffer allocation involved and the interactions with memory allocation between the SML/NJ system and the MPI communications package. The likely future implementation of the marshalling operation in SML/NJ will make this alternative straightforward [Rep95], since the result of such an operation will be a `Word8Vector.vector` (effectively a byte array). Rather than writing the header and array to file directly, it will be up to the user to determine what to do with the byte array representation. In `paraML`’s case, this would be passed directly to the MPI sending operation. (In the current implementation, there are performance advantages to be gained from sending a header message, due to the complexities of receiving messages of unknown size in MPI. Further discussion of this point is found in §10.3.)

On receiving a message, first the header is decoded to determine the length of the following byte array. A byte array is allocated within the ML heap space (using the sized string allocation operation), and the subsequent byte array message transferred into this buffer. Some relocation of relative pointer offsets occurs, and then control is returned to the ML system, with the result being the offset into the buffer which identifies the start of the ML object.

The sending and receiving operations must be able to work for arbitrary ML objects. Thus the type signature of the ML interface to these C operations uses a strongly polymorphic type variable, respectively:

```
val mc_send : int * int * 'a -> unit
val mc_recv : int * int -> 'a
```

Avoiding type errors when receiving messages then becomes an issue for the `paraML` runtime system. The mechanisms for re-establishing type information about newly-arrived messages (which do not possess ML type information) was detailed earlier in §9.3.4.

Since the processor and virtual processor abstractions are just ML data values, the reader may wonder how marshalling of messages which contain uses of the paraML operations avoids copying all of these abstractions. One of the characteristics of the marshalling/unmarshalling operations in the SML/NJ compiler is that they assume the executable that unmarshals an object has the same locations for non-heap-allocated data structures and routines. This assumption means that when marshalling an object, any pointer to a data structure not in the heap can be shallow copied (the pointer is copied, not the object it references). The SML/NJ compiler provides an operation as part of the pervasive environment (not heap-allocated) which allows the caller to set and get an arbitrary variable (`setvar` and `getvar` respectively). The `getvar` operation allows the user to coerce the returned value to any type desired. These routines are used to set and get the processor abstraction, which in turn holds all the virtual processor abstractions. Thus all the paraML operations first retrieve the appropriate abstraction using the `getvar` operation, before performing data structure manipulation. When copying a message to send to another process, any mention in the paraML operations of these local processor and virtual processor abstractions will not involve copying them, since they are protected by access through the `getvar` operation.

9.5 Concurrency within processes

Since SML/NJ provides continuations, it is possible for any paraML program to exhibit concurrency within the code executed by a process. ParaML does not itself address the issue of providing concurrency primitives in addition to its own process-oriented primitives. However, for efficiency and safety reasons, it is sensible for paraML to incorporate support for multi-threading within processes. The level of support is only to provide hooks into the scheduling mechanisms. Thus someone who wishes to provide a concurrency package (say CML [Rep92] or the lower-level ML-threads system developed by Morrisett and Tolmach [MT93]) will be able to do so without significant alterations to the paraML runtime system.

The major potential problems to avoid are:

- Interrupting call outs to the MPI operations, since there are no current MPI systems which are re-entrant and thus they are not thread-safe.
- Interrupting parts of the paraML runtime system code where it is manipulating VP/process state.

Either of these situations could result in total failure of a paraML system. The basic alteration required is for atomic regions in the paraML runtime system code, which the scheduling mechanisms respect. The existing implementation already provides support for interrupts on message arrival and timer-based signals.

The other alterations are to provide queues for various of the data structures. Since there may be more than one thread within a process suspended waiting for a

message to arrive, it becomes necessary to guarantee fair access to these messages. For example, it makes sense to use queues of thread continuations for blocking port communications. The threads are inserted into the queue and will be satisfied in the order they made their blocking `recv` requests. Replies to control messages do not need to be queued in the same way, since control messages use uniquely-generated keys to discriminate between different VPs on the same paraML runtime system. These keys uniquely identify a continuation which represents the requesting computation, together with the VP identifier and thread identifier in the case of this additional concurrency support. Some additional changes are required to avoid an active thread receiving a message, whose arrival had caused the message handler thread of the runtime system to move a descheduled thread back onto the runnable queue in the expectation of it receiving this new message. Simple solutions exist to tag messages in a port's queue to particular descheduled (but now-runnable) threads, or alternatively adjust the scheduling priorities so that the descheduled thread becomes the next active thread within the process.

9.6 MPI as a communication platform

The earlier versions of paraML used communication primitives specific to the Fujitsu AP1000 for inter-PE message passing. One of the problems with using this approach was that the communications facilities in paraML were biased towards the characteristics of the architecture, rather than being designed in a principled manner. In the current version of paraML, the communications primitives have been redesigned without machine-specific considerations. The new design places few demands on the underlying communications platform.

9.6.1 Advantages

MPI has been chosen as the basis for the underlying communications platform for a number of reasons.

- High performance implementations of MPI exist for virtually every parallel computer commercially available.
- MPI implementations also exist for networks of workstations, offering the ability to develop paraML systems on readily available hardware.
- The programming model embodied by MPI mimics that of the physical machine model for paraML.
- The MPI standard abstracts the communications and machine-attribute characteristics of a broad class of high performance computers. Portability of paraML systems between different architectures is thus made simpler.

The current paraML system was first implemented using the workstation version of CHIMP/MPI developed at the Edinburgh Parallel Computing Centre [BMS94]. ParaML was then ported to the Fujitsu AP1000+ using the MPI system developed at

the Australian National University [Sits95]. The porting took approximately half a day, of which around half an hour was due to MPI-related aspects. The ease with which the porting was accomplished reflects the success of standardising communication and machine-attribute operations in MPI.

Another benefit of using MPI is that it will perform optimised delivery of messages when the destination PE is also the sending PE. Instead of involving network communications, a message sent between two processes on the same PE will just involve copying the message into the MPI system's message queues. An alternative would be to directly transfer the message within the paraML runtime system, once a marshalled copy of the original value had been made. As mentioned before, the future implementation of the marshalling operations will make this straightforward to perform.

9.6.2 Disadvantages

Despite the benefits of using MPI, it imposes certain restrictions. For paraML, there are three basic restrictions. The first is the lack in MPI of any form of notification on message arrival. When a request or port message is delivered to a paraML runtime system, it always requires a response of some form to indicate whether or not the message can be incorporated at some later point. Thus to prevent delay of the sending process, it is highly desirable to know when a message arrives and to send the reply message immediately. Without any indication that a message has arrived, it becomes necessary to wait until the paraML runtime system schedules the message handling thread before a reply can be sent.

A set of extension operations suggested for MPI [SDV+94] includes one to manage just this type of situation, and this `MPI_Hrecv` operation has been implemented for the MPI system on the AP1000+. The operation allows the programmer to install a C handler routine to be executed on receipt of a message. Since the paraML runtime system is written in ML however, it is necessary to interrupt its execution in some manner to advise it of the message arrival. The obvious mechanism to achieve this is by generating a signal, since SML/NJ includes support for asynchronous signal handling [Rep90].

An operating system for the AP1000+, AP/Linux [TMS+96], allows the current implementation to avoid such MPI extensions altogether, since it is possible to send a signal directly to a remote PE. This mechanism is used by the sending PE to generate a signal in the destination PE after an MPI message has been sent. A handler routine written in ML is installed for receipt of this signal (in the current implementation, `SIGUSR2`). When invoked, the signal handler in ML then causes a reschedule as soon as possible to the main message handling thread in the paraML runtime system.

The second restriction arises in trying to optimise some of the derived operations in paraML. Ideally, all the group-based communication operations would make use of the collective communication facilities in MPI. A specific example helps to describe the

situation. If a group of processes has been created, and a group of ports has been created for that process group, then a new MPI communicator should be created for the port group. MPI communicators allow efficient collective communication among members of the communicator. However, the problem is that for a new communicator to be created, all members of the parent communicator (in the first instance, `MPI_COMM_WORLD` which spans all the paraML runtime systems) must cooperate in the creation operation. Effectively then, this is a synchronising operation over all the paraML runtime systems, not just those which host the new paraML processes.

In fact, collective communication within a communicator is only feasible with a broadcast operation, `MPI_Bcast`. This operation requires that all members of the communicator call it, and that there is a fixed message size to be delivered. The synchronising nature of the broadcast operation means that all paraML runtime systems must be advised first by normal point-to-point messages that a broadcast operation is to be performed before the broadcast routine can be called. It is also impossible for collective operations to be performed from outside a communicator. Thus it would not be possible for a process to send a message using MPI collective communication support to a port group unless the process was also a member of the underlying MPI communicator.

The third restriction involves matching buffer sizes in `MPI_Send/MPI_Recv` operations. This restriction is discussed in more detail in relation to the performance of MPI communications when combined with paraML in §10.3.

9.7 Limitations of implementation

The main discrepancy between the formal description of paraML in Part III and the current implementation is with exceptions. The theoretical model for exceptions requires that exceptions are globally defined. However, the current implementation has minimised changes to the SML/NJ runtime system and pervasive environment, with no changes to the compiler itself. Implementing a global name space for exceptions would require that serious changes be made to the way exceptions are created within the internals of the compiler. Avoiding such types of changes is highly desirable; minimising difficulties in tracking compiler updates is one reason. Exceptions may still be safely transmitted in messages between paraML processes, with the following restrictions. If the process resides in a separate paraML runtime system, the exception must be declared prior to a call to the `paraml` operation which initiates the root process of the paraML program. Any exception declared dynamically during the course of execution of paraML processes may only be used locally within the declaring process.

The current implementation of paraML embodies the three layers of process model, abstract machine model and physical machine model in software. Ideally, these different layers would be cleanly separated by defined interfaces for the operations permitted across the layers. In order to provide efficient performance however, the

existing system exhibits a stronger level of coupling between these layers. This coupling is necessary to make use of knowledge about thread and virtual processor scheduling in the implementation of the paraML operations. It is also used in manipulating data structures associated with the virtual processor layer, particularly in message handling. Some of the changes outlined recently to the definition of Standard ML [SML96], particularly with respect to the inclusion of type definitions in signatures, could make stronger abstractions between layers more attainable.

9.8 Conclusion

The implementation of paraML reveals a number of strengths and weaknesses involved in using ML and MPI individually and in combination. The use of MPI goes a long way towards providing a degree of platform-independence essential in meeting the goal of portability, despite its other limitations. Similarly, using ML for building the paraML runtime system is effective at minimising operating system dependencies. The theoretical description of paraML provided an essential guide in building paraML to meet specifications. The resulting system is robust with respect to these specifications, with the exception of exceptions defined within process executable code. Unlike previous versions of paraML, discrepancies between the theoretical specification and the implementation are clearly identifiable. In the next chapter, the actual performance of various aspects of the current implementation of paraML is measured.

The first part of the history of the world is the history of the human race. It is a history of the progress of the human mind, of the growth of human knowledge, of the development of human civilization. It is a history of the human race, of the human mind, of the human knowledge, of the human civilization. It is a history of the human race, of the human mind, of the human knowledge, of the human civilization.

The second part of the history of the world is the history of the human race. It is a history of the progress of the human mind, of the growth of human knowledge, of the development of human civilization. It is a history of the human race, of the human mind, of the human knowledge, of the human civilization. It is a history of the human race, of the human mind, of the human knowledge, of the human civilization.

The third part of the history of the world is the history of the human race. It is a history of the progress of the human mind, of the growth of human knowledge, of the development of human civilization. It is a history of the human race, of the human mind, of the human knowledge, of the human civilization.

The fourth part of the history of the world is the history of the human race. It is a history of the progress of the human mind, of the growth of human knowledge, of the development of human civilization. It is a history of the human race, of the human mind, of the human knowledge, of the human civilization. It is a history of the human race, of the human mind, of the human knowledge, of the human civilization.

The fifth part of the history of the world is the history of the human race. It is a history of the progress of the human mind, of the growth of human knowledge, of the development of human civilization. It is a history of the human race, of the human mind, of the human knowledge, of the human civilization.

10. Performance

10.1 Introduction

This chapter analyses the costs of the various components of the implementation of paraML: the MPI communication system; the ML runtime system; the paraML runtime system; and the paraML operations themselves. Examining the performance of these components determines how well the implementation meets its goals of scalability and efficiency. It also provides a useful insight into avoidable performance penalties with the current implementation and hence avenues for future optimisations.

10.2 Multicomputer characteristics

The current implementation has been tested primarily on the Fujitsu AP1000+ [SKI+93], a 16-PE multicomputer running AP/Linux [TMS+96] installed at the Australian National University. Each PE is based around a 50 MHz SuperSPARC processor with 16 MB of memory. Attached to each PE is an additional option board connected to a 4 GB disk, which provides access to swap disk partitions and local filesystems. Custom-designed network chips interface the PE with the three communications networks. These networks are:

1. a wormhole-routed point-to-point torus network;
2. a broadcast network that allows one-to-all communications, and is also used as the interface to a host computer which connects to external networks;
3. a synchronisation network, which can be used for fast global synchronisation.

The networks provide low latency and high bandwidth communications between different PEs. The hardware also supports PUT and GET primitives, for remote PE memory access. These facilities are used in the implementation of the MPI system to optimise large message delivery.

The AP/Linux operating system is a multi-user, multi-process operating system for the AP1000+, extended from the SPARC Linux operating system. In most respects, it behaves just like a normal UNIX operating system for a workstation. However, it has additional support for executing processes in parallel. This facility establishes one process on each of a subset (possibly maximal) of the available PEs of the AP1000+. These processes share the same context number on each node, and are scheduled for execution according to a simple gang-scheduling algorithm.

10.3 MPI performance

The MPI implementation for the AP1000+ was developed at the Australian National University [Sits95]. Different message protocols are used to support efficient message passing performance across a range of message sizes. The primary distinction is between small and large messages.

1. Small messages are sent using standard mechanisms for communications on the torus network. The message is sent immediately, and stored in a system buffer at the destination.
2. Large messages require a small information message to be sent first. On receipt, the MPI system directly fetches the message contents from the sender with the remote memory GET operation.

The default crossover point between the two message sizes is at 600 bytes, but this can be altered in the initialisation phase of an MPI program execution. Latency measurements of the MPI system with AP/Linux are shown in Figure 10.1 and Figure 10.2. The ping-pong benchmark used in these results sends a message from one cell to another, which receives the message and sends it back. The latency measured is between the initiation time of a send operation on one cell and the time at which it has been received into a user buffer on the destination cell. The MPI sending mode used is standard mode.

One of the drawbacks of MPI's standard mode is the potential to introduce deadlocks without careful programming. Since a message sent with standard mode may not complete until the matching receive operation has been invoked (and in particular, a buffer for message receipt has been allocated), programs must ensure that blocking sends are not invoked until the corresponding receives have been invoked also.

There are two major constraints in using MPI for paraML's communications. The first constraint is that message sizes are not known in advance since they may be of arbitrary ML objects, and thus it is not possible to set up a receive operation prior to a send operation. The second constraint is that when control is returned to the paraML runtime system from a send operation in MPI, the message buffer must no longer be required by MPI. The reason for the second constraint is that the ML runtime system may at any time perform a garbage collection operation, which might invalidate any subsequent send from the message buffer.

The solution to the first constraint is to send a header message initially, which encodes the size of the object to be sent. The main message containing the actual ML object is sent subsequently. However, both these MPI send operations must be completed prior to any more paraML runtime system operations; other messages from the same PE with similar control information could cause confusion at the destination if this did not happen. The arrival of the header message will result in the paraML runtime system at the destination invoking its message handler code, which will allocate a buffer for the following message containing the object itself.

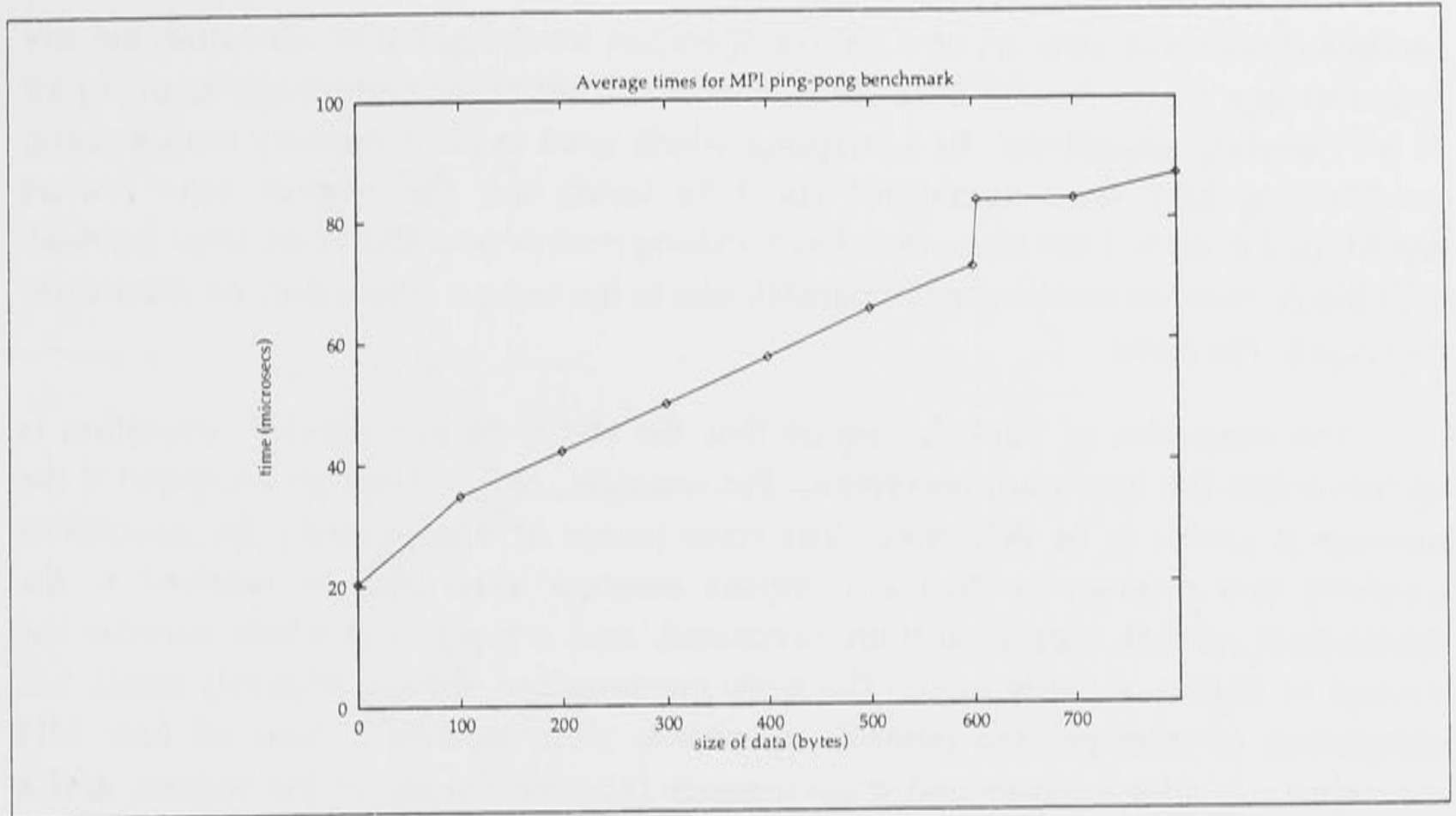


Figure 10.1 – Performance for small messages under MPI.

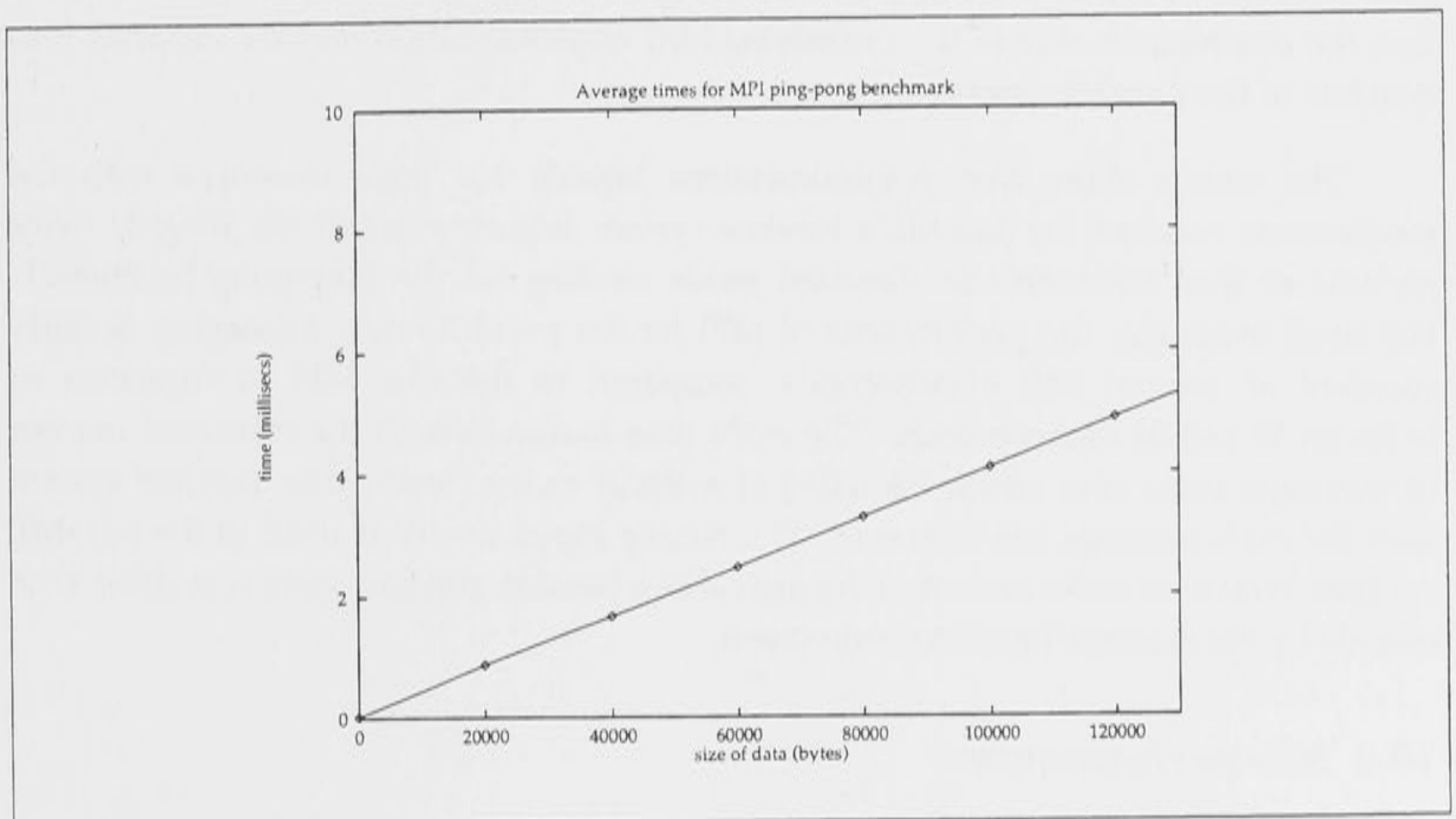


Figure 10.2 – Performance for large messages under MPI.

The solution to the second constraint is to use buffered mode sending of large messages. This mode involves an additional memory copy of the ML message buffer into an MPI system buffer. However, on completion of this copy, control can be returned to the paraML runtime system because the original message buffer is free to be reused or garbage collected. The matching receive operation at the destination does not have to be invoked before this send operation completes, though the contents of the copied message will not be transferred until the receive buffer has been allocated. Use of buffered mode sending also eliminates the potential for deadlock, since the paraML

runtime system may now set up a receive operation involving buffer allocation for any large message whose header message may have arrived while performing its own pair of MPI sending operations. In a language which used explicit memory management, non-blocking MPI send operations could be used, and the original send buffers deallocated at some later time when the matching receive operation had been invoked. This mechanism cannot be used in paraML due to the second constraint, on immediate message buffer reuse.

The semantics of paraML require that the status of any paraML operation is known before the operation completes. For example, `send` raises an exception if the message is unable to be delivered. The consequence of this property for operations involving two processes is that any request message must both be received at the destination paraML runtime system, processed, and a reply sent which encodes the success or failure of the request. The reply messages are always relatively small, but completion of inter-process paraML operations thus requires a total of four MPI messages: a header message and main message (possibly large) for the request, and a header message and main message (always small) for the reply. The elapsed times for such patterns of message sending in MPI are given in Figure 10.3 and Figure 10.4, so that the components due to the underlying MPI implementation may be factored into analysis of the paraML operations.

The results show that communications latency for large messages with the mechanisms required for paraML's runtime system implementation are roughly twice as slow as that achievable in standard mode sending for the ping-pong benchmark. For small messages, the performance of MPI for the paraML-style messaging is fairly constant at around 340 microseconds, compared to the raw MPI performance of between 35 and 84 microseconds. The extra time is due to both the increased number of messages being sent and the sending of a signal to the destination runtime system after the main message has been sent. The remote signal device is used in the paraML runtime system to make sure message arrival at a paraML runtime system is dealt with quickly by the message handling component.

10.4 ML performance

The main aspect of the SML/NJ system performance which it is useful to measure involves the marshalling operations. These figures report the time taken to marshal various objects into contiguous byte arrays and to unmarshal such arrays back into ML objects. When unmarshalling an object, it is sometimes necessary to perform an operation to increase the heap if there is not currently enough free space available to allocate a buffer for the object. All times are elapsed times in microseconds, recorded for the performance of the extended SML/NJ runtime system executing on one PE of the AP1000+. Table 10-1 reports the times for a marshalling a variety of different objects, some of which are used in the performance testing of the paraML operations. The sizes of the objects in their marshalled format are also given; each marshalled object includes a 12-byte header.

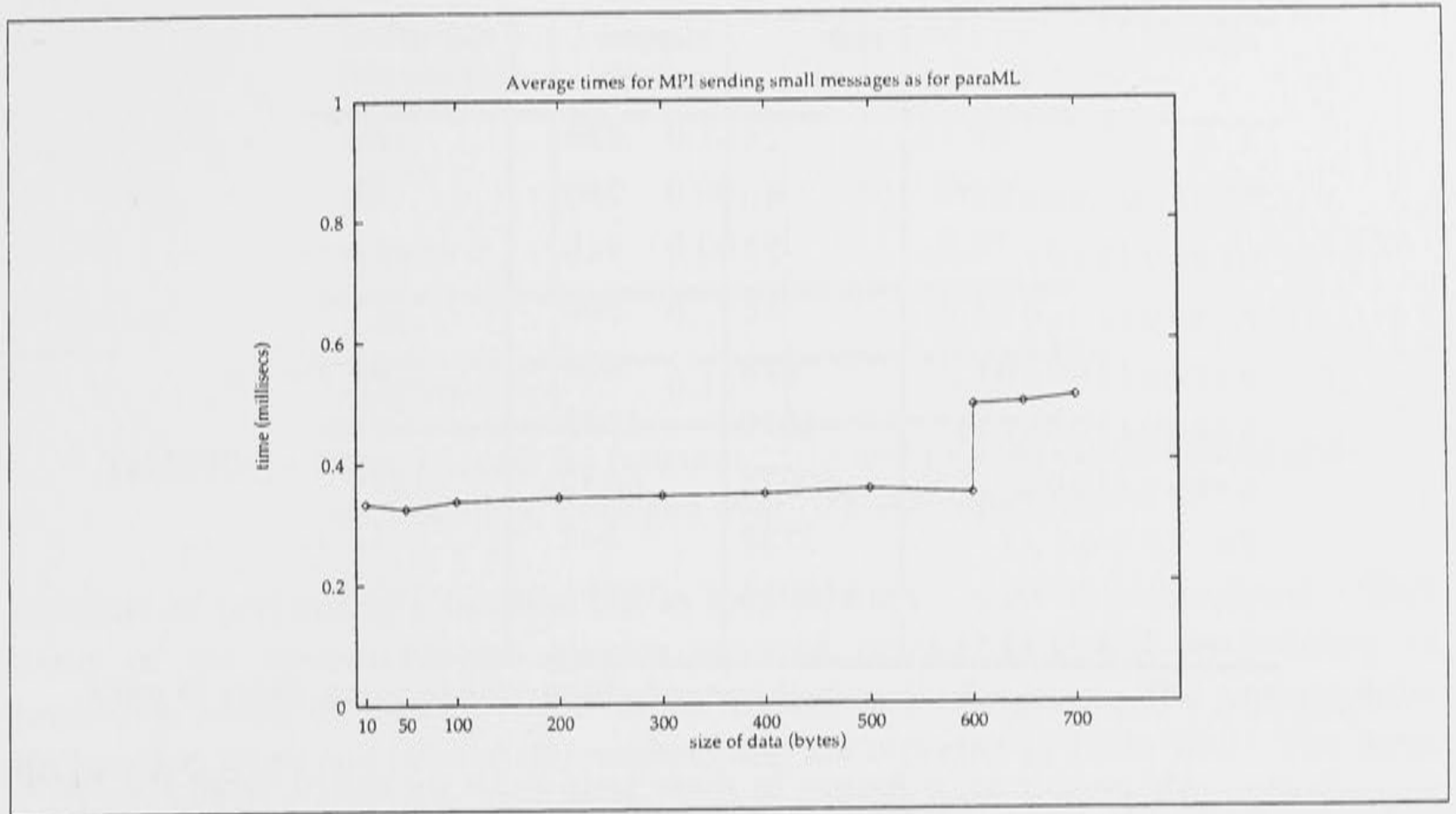


Figure 10.3 – Performance for small messages under MPI for paraML.

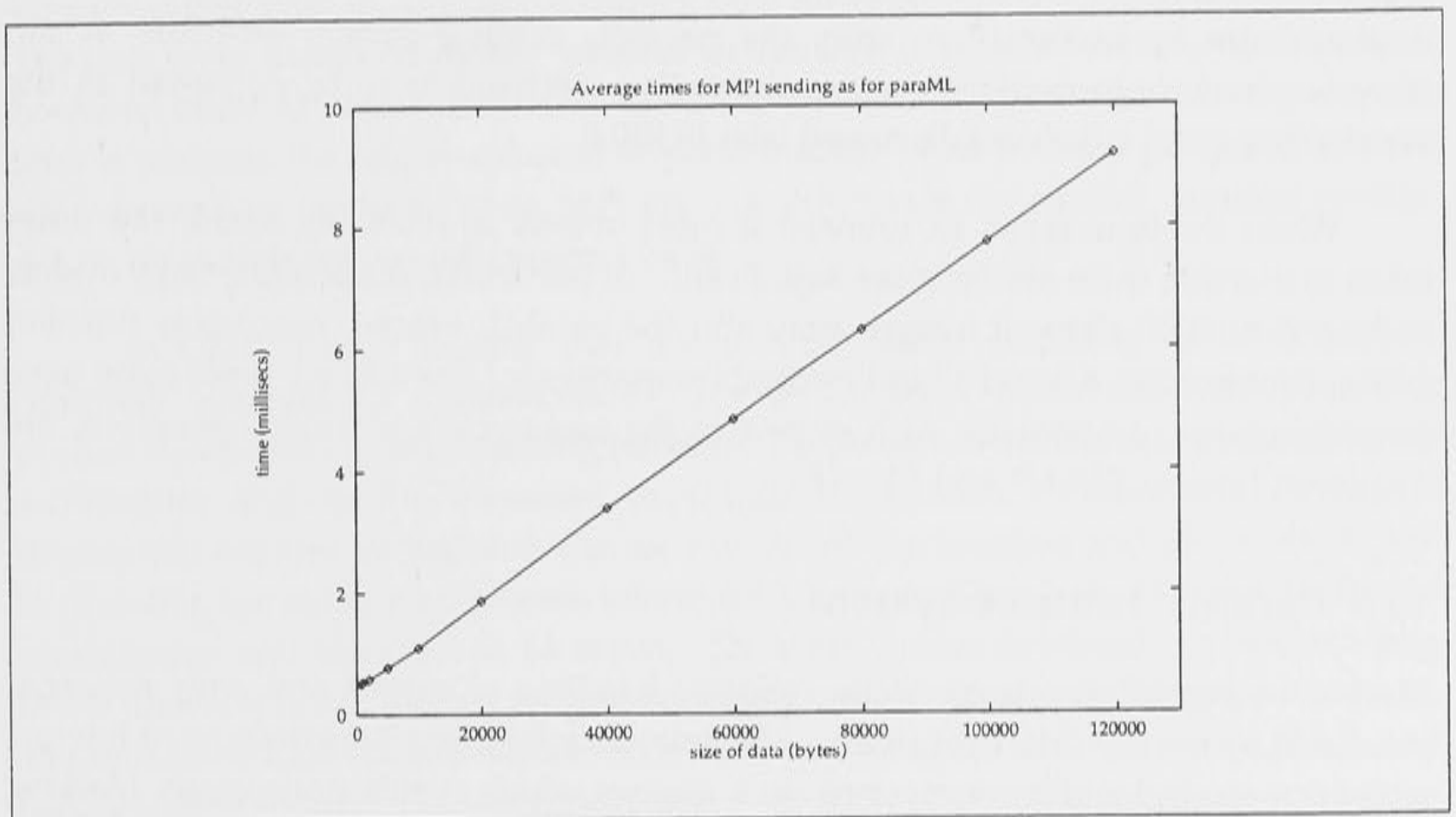


Figure 10.4 – Performance for large messages under MPI for paraML.

The two functions are chosen deliberately to illustrate characteristics of the marshalling operations in the SML/NJ runtime system. The first function, `fn () => ()`, is the most minimal function possible. The second function, `fn () => (self_id (); ())`, represents a minimal function that a paraML process might execute. The enormous difference in sizes between the paraML process executable function and the null function is due to copying of the entire paraML runtime system structure. The sizes for these same marshalled functions using the interpreted system, rather than the batch compiler, were 620 bytes and 689604 bytes

<i>object</i>	<i>size</i>	<i>elapsed marshal</i>	<i>elapsed unmarshal</i>
1	24	436	10
"hello world"	40	260	11
array(1,0)	32	340	11
array(10,0)	68	500	11
array(100,0)	428	525	21
array(1000,0)	4028	2078	148
array(10000,0)	40028	6832	1420
fn () => ()	3732	442	13
fn () => (self id();())	112916	16951	1057

Table 10-1 – Object marshalling and unmarshalling; size in bytes, times in μ secs.

respectively. All performance figures in these tests were measured using the batch compiler, because the costs of marshalling and sending objects 680KB in size are prohibitively large. Since each paraML runtime system contains exactly the same code, it should not be necessary to copy the paraML runtime system structure at all. However, such optimisations require alternative mechanisms to be employed in the marshalling code, which are discussed later in §10.8.

While the time taken to unmarshal most objects is relatively small, the times taken to marshal them are far more significant. In particular, marshalling large objects such as the 10000-element integer array and the paraML process executable function take approximately 6.8 and 17 milliseconds respectively. The impact these costs have on performance, particularly with respect to the `execute` and `send` operations, are discussed later in §10.6.7 and §10.6.8.

10.5 ParaML runtime system

Most of the paraML runtime system concerns handling of request and reply messages generated by the paraML operations. The process scheduling behaviour must interact with the message handling component in a manner which avoids unnecessary blocking of runnable processes. Similarly the support for potential multi-threading within processes requires some careful manipulation within processes to avoid blocking other runnable threads. Hence there are various mechanisms for descheduling of processes/threads to match the different prerequisite data requirements for continued execution of the current operation.

The measurement of these alternative mechanisms is implicitly bound up with the measurements of the various paraML core operations, and thus is not separated out here. All of the measurements are performed by executing the operations a number of times, so that the probe effects of the timing routines are minimised by taking an average. The tests are also repeated over a number of runs. It is useful to know the

<i>time</i>	<code>loop ()</code>	<code>deschedule / reschedule</code>
user	0.12	31.89
gc	0.00	0.11
system	0.00	3.57
sum	0.12	35.57
elapsed	0.13	35.58

Table 10-2 – Times in μsecs for function `loop` and process descheduling and rescheduling, averaged over 1000 000 repetitions.

cost just of performing a function call to loop through a number of iterations. Since many of the operations also involve repeated descheduling and rescheduling of processes, timing of a generic deschedule/reschedule is also given. The average times for function loops and process de/rescheduling are reported in Table 10-2. The times are broken down into user, system (sys), and garbage collection (gc) times measured using the `System.Timer` facilities in SML/NJ. The sum of these three times is also given, together with an elapsed wallclock time measured using `gettimeofday ()`. The tests were measured mostly without interference from other users of the machine; however, other AP/Linux processes are occasionally scheduled, and thus the elapsed time is perhaps the fairest measure of performance. Note that the per-process times are measured on the basis of an MPI process, effectively the paraML runtime system, not on an individual paraML process basis.

In CML, Reppy reports that the cost of thread switching on a SPARC 2 computer averages 22 microseconds. The performance difference with paraML's process descheduling/rescheduling (effectively a process switch) is due to the different architecture and to the increased overheads in paraML. Performing a process deschedule requires saving both the current thread continuation and identifier as well as updating the suspended process information, whereas CML only requires the thread continuation and identifier to be saved. There are similar overheads on rescheduling. In the times reported, the test program does not need to do any checking for messages, and thus these times do not include times for performing the message arrival check and handling operations. The additional costs for such operations are those for setting atomic regions around a block of code and performing an MPI probe. These are of the order of 50 microseconds.

10.6 ParaML operations

The most interesting things to measure are the paraML operations themselves. Some of the operations, such as `probe`, require no communication with remote paraML runtime systems. The results for these operations are thus reported in table form, because they are independent of the number of processes in the system. Other operations, such as `execute`, are dependent on various aspects of the current system state. For instance,

<i>time</i>	<i>self_id</i>	<i>probe</i>	<i>recv</i>	<i>self_port</i>
user	2.00 ± 4.00	14.00 ± 8.00	52.00 ± 8.72	113.00 ± 13.45
gc	2.00 ± 4.31	11.00 ± 8.31	5.00 ± 6.71	12.00 ± 09.80
system	0.00 ± 0.00	0.00 ± 0.00	4.00 ± 4.90	43.00 ± 11.87
sum	4.00 ± 4.90	25.00 ± 5.00	61.00 ± 3.00	168.00 ± 04.00
elapsed	10.80 ± 7.74	30.80 ± 2.56	66.20 ± 2.96	172.10 ± 02.55

Table 10-3 – Times and (\pm) standard deviations in μ secs for *self_id*, *probe*, *recv* and *self_port* operations, averaged over 1000 repetitions, 10 runs.

if a remote PE is heavily loaded with executing processes, it can delay the time taken to satisfy requests which involve communications with that paraML runtime system. The results for these operations are reported as graphs, where the number of processes in the system is the independent variable.

10.6.1 *self_id*

The *self_id* operation is very low-cost, since it involves only finding the current VP's data structure and retrieving the process name object. The times for the operation are given in Table 10-3. Considerable variation in the times between runs is observable, and there is a big relative difference between the elapsed time and the sum of process time, which may be due to other users of the computer at the time the test was being performed. No interactions with other processes are necessary.

10.6.2 *probe*

The times for *probe* are given in Table 10-3, and are relatively low-cost again because there is no inter-process communication required. The operation needs to examine the state of the queue associated with the port name, which requires slightly more data structure manipulation than for *self_id*. Note that this operation does not call the *MPI_Probe* operation, since it is the internal paraML data structures that must be checked, not the MPI message queues which are manipulated only by the message handling component of the paraML runtime system. The operation must also do some error checking, since a process may attempt to probe a port that does not belong to it, and this error condition returns an exception.

10.6.3 *recv*

The *recv* operation is roughly twice as slow as the *probe* operation, as shown in Table 10-3. It too is a low-cost operation as measured in this test, because no inter-process communication is required. The test program initially sends 1000 messages to the port, and then sends a start message to a different port on the same process. Thus the times being measured do not require the *recv* operation to block at all, since it blocks on the start message, and then there are 1000 messages to be received on the

<i>time</i>	<i>naive self port</i>
user	1200.00 ± 071.49
gc	514.44 ± 052.30
system	887.78 ± 062.50
sum	2602.22 ± 084.82
elapsed	7060.44 ± 228.64

Table 10-4 – Times and (\pm) standard deviations in μsecs for naive translation of `self_port` operation, averaged over 1000 repetitions, 10 runs.

other port. It would be meaningless to measure blocking times, since they may be arbitrarily long. The performance of `recv` is not affected by the size of the message.

The reason that `recv` takes twice as long as `probe` is that it must do roughly twice as much work. Firstly, it has to check to see whether the port queue is non-empty. If the queue is empty, thread suspension mechanisms are put in place to reschedule the thread when a message has arrived that may be received by the thread (there may be other threads currently suspended as well). If the queue is not empty, then the first message may be dequeued and returned. Similar error checking to the `probe` operation must be performed to prevent processes attempting to receive messages from ports not owned by them.

10.6.4 `self_port`

An interesting contrast to the times for the `port` operation discussed later in §10.6.6 are the times for a derived operation, `self_port`, given in Table 10-3. The semantics of `self_port` are given simplistically as `port (self_id ())`. The operation is very useful for a process to create its own ports, which is often done in order to advertise to other processes a willingness to receive data. The operation is also used in constructing various of the derived communications operations, which involve creating a local port (usually used only once), sending its name to another process, and then receiving the data sent by it. The `place` and `retrieve` operations are implemented in this fashion.

Since this operation is used regularly, it is sensible to optimise its implementation. The act of creating a new local port is much the same as that for creating a new remote port, but there are no inter-process messages required. This dramatically reduces the elapsed time, and there are also no process rescheduling overheads involved. The operation is still around three times as expensive as the `recv` operation, primarily due to the need to create new data structures and insert them into existing ones, rather than just looking up existing values.

To demonstrate the clear advantages of creating such optimised implementations, the times for a naive translation of the semantics, albeit with only a

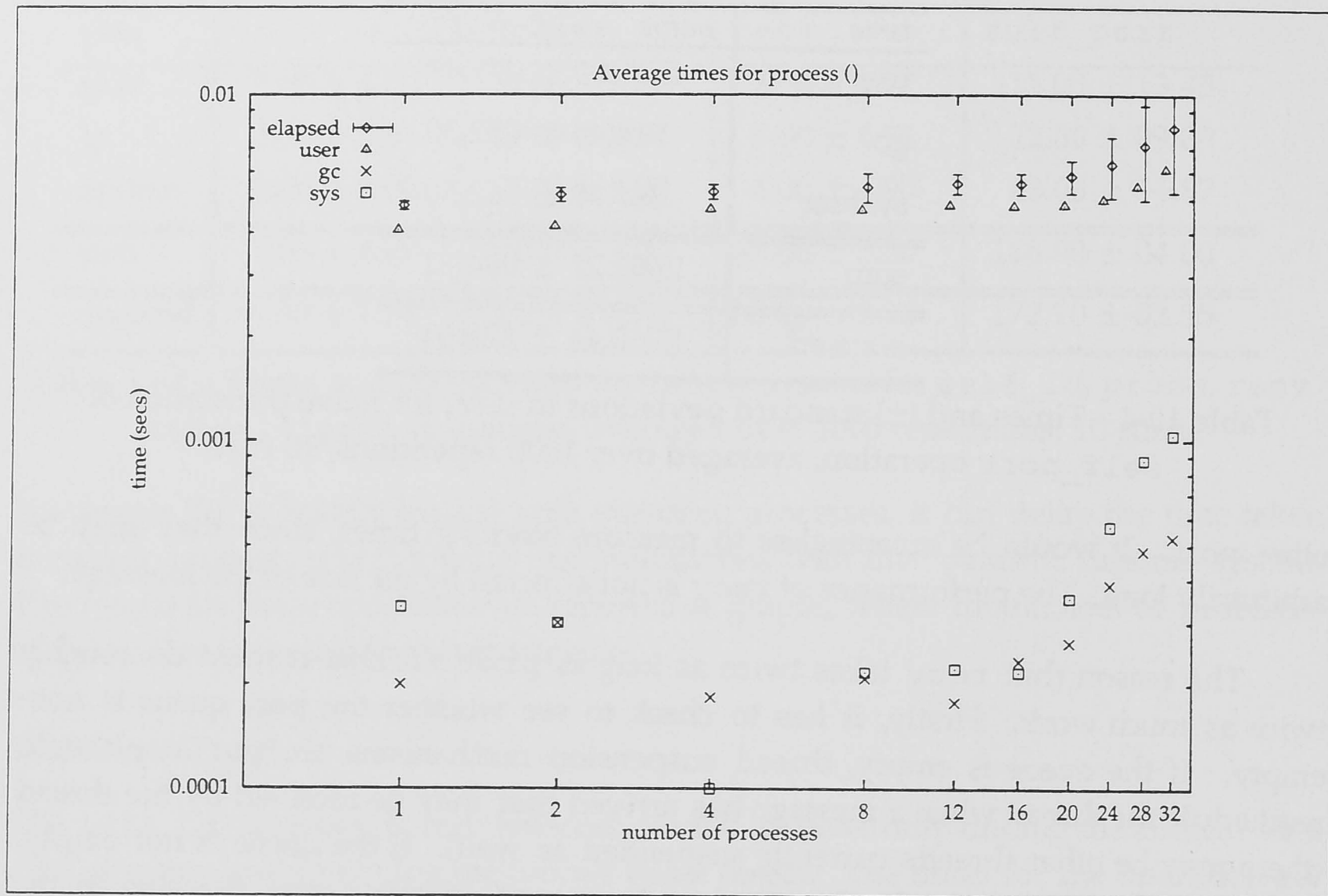


Figure 10.5 – Times for process operation with differing numbers of processes.

single use of the `self_id` operation, are shown in Table 10-4. The code used for the test was as follows:

```
let val my_id = self_id ()
in
  port my_id (* 1000 repeats of this line timed *)
end
```

The massive increase in elapsed time is due to the extra overheads of sending messages to the process via the MPI communications library, and the additional process rescheduling and message handling required. For reasons of simplicity, the current paraML runtime system implementation does not try to avoid MPI send operations when the destination is on the same PE. Significant overheads could be avoided clearly, but major changes to the mechanisms for marshalling objects would be required also. These changes are foreshadowed for newer versions of the SML/NJ compiler.

10.6.5 process

The test program for the `process` operation is the first requiring communications between different paraML runtime systems. Since the times will be partially dependent on the overall load among the collection of runtime systems, the times were measured with increasing numbers of processes involved. In each case, the creating processes performs 15 `process` operations, and average the times to give a result for just one operation. The number of processes involved in doing these tests ranged from

1 to 32, with 10 runs of the program performed for each new process number. Thus, for one creating process, 10 runs were performed, and the times are an average of these 10 runs. For 32 creating processes, 10 runs were performed again, but the times are an average of the results reported by each of the processes across these runs. Thus the standard deviation shown for the elapsed time reports the range of times for a normal distribution across all processes in a system. This seemed a more valid measurement to make, since what is interesting is the variation among processes in any particular program, not just between runs.

The results are shown in Figure 10.5, plotted on log-log scales, with the number of processes performing the operation on the x-axis. Standard deviation is shown only for the elapsed times, since it is fairly negligible, and the log scale distorts minor deviations for the garbage collection and system times. The elapsed and user times are almost the same, with relatively minor contributions played by the garbage collection and system times. Consequently, the sum of the user, gc and system times is not shown in the results for this operation and subsequent tests. The elapsed times vary from 4.8 to 8.0 milliseconds across the differing numbers of processes. This result indicates that the message handling component of the paraML runtime system does not cause unnecessary delays in dealing with new process creation requests. It should be noted that where there are 32 processes involved, there will be 2 creating processes on each of the paraML runtime systems, since it is only a 16-processor AP1000+. Examining the graph of MPI performance for small messages in Figure 10.3 illustrates that only around 0.35 milliseconds of the overall time of the operation is due to MPI. Another 1 millisecond will be taken up by the marshalling and unmarshalling of messages in ML. The bulk of the remaining time is due to the creation of new VP data structures for storing information about the new process, setting up handlers for `port` and `execute` requests, and the usual process rescheduling.

10.6.6 `port`

The times for performing the `port` operation are measured in identical ways to that for the `process` operation. Again, the times are relatively constant across the range of processes measured, gradually increasing from 1.8 to 2.6 milliseconds. The increase in time arises due to the increased load across the paraML runtime systems. The greater the load, the greater the probability that a paraML runtime system will be involved in processing some other request when a new request arrives, and thus the new request will be blocked until completion of the earlier one. The MPI messaging costs remain the same, at roughly 0.35 milliseconds. Since there is virtually no processing to be performed at the destination paraML runtime system, the marshalling overheads of roughly 1 millisecond account for the remaining bulk of this operation's cost.

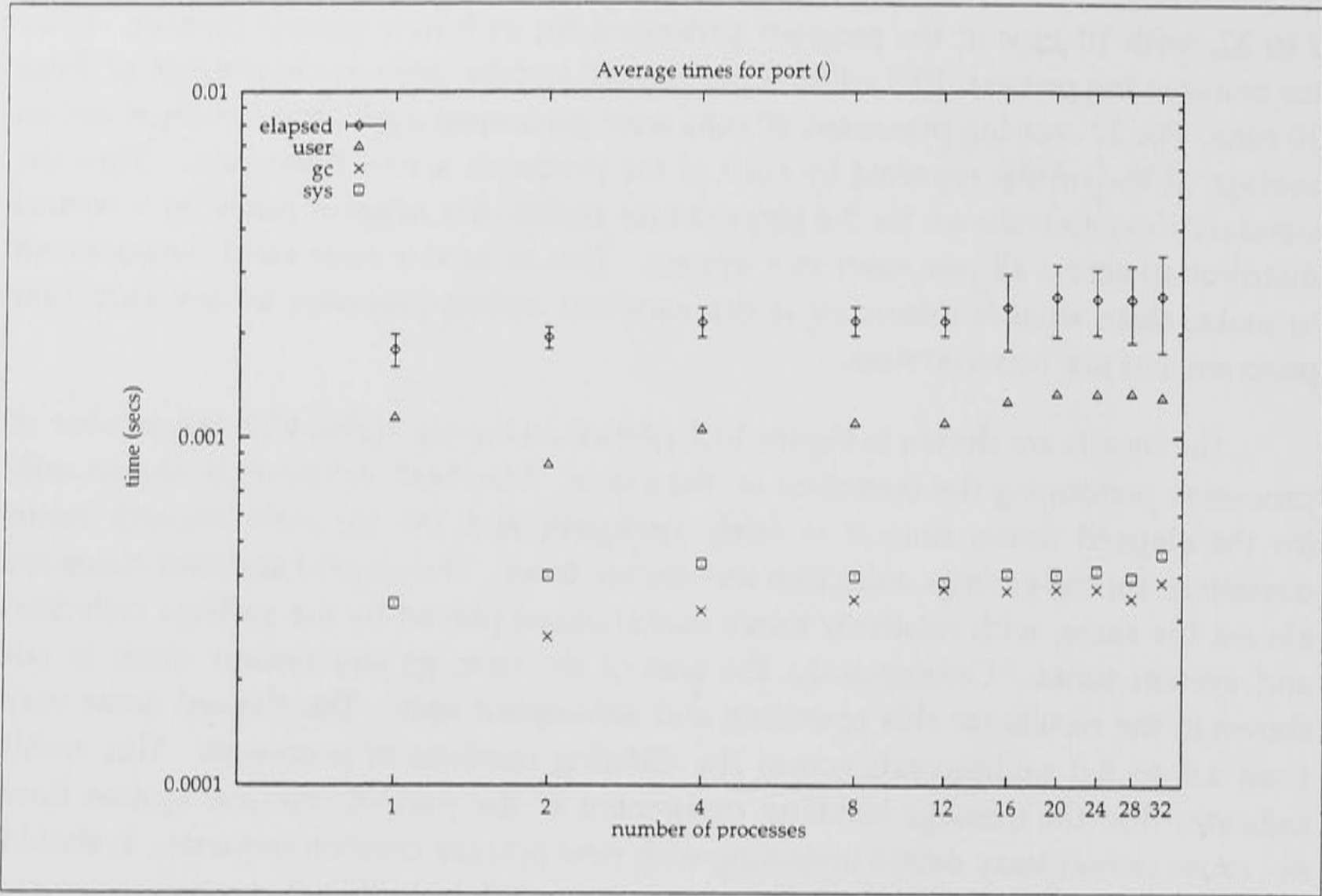


Figure 10.6 – Times for port operation with differing numbers of processes.

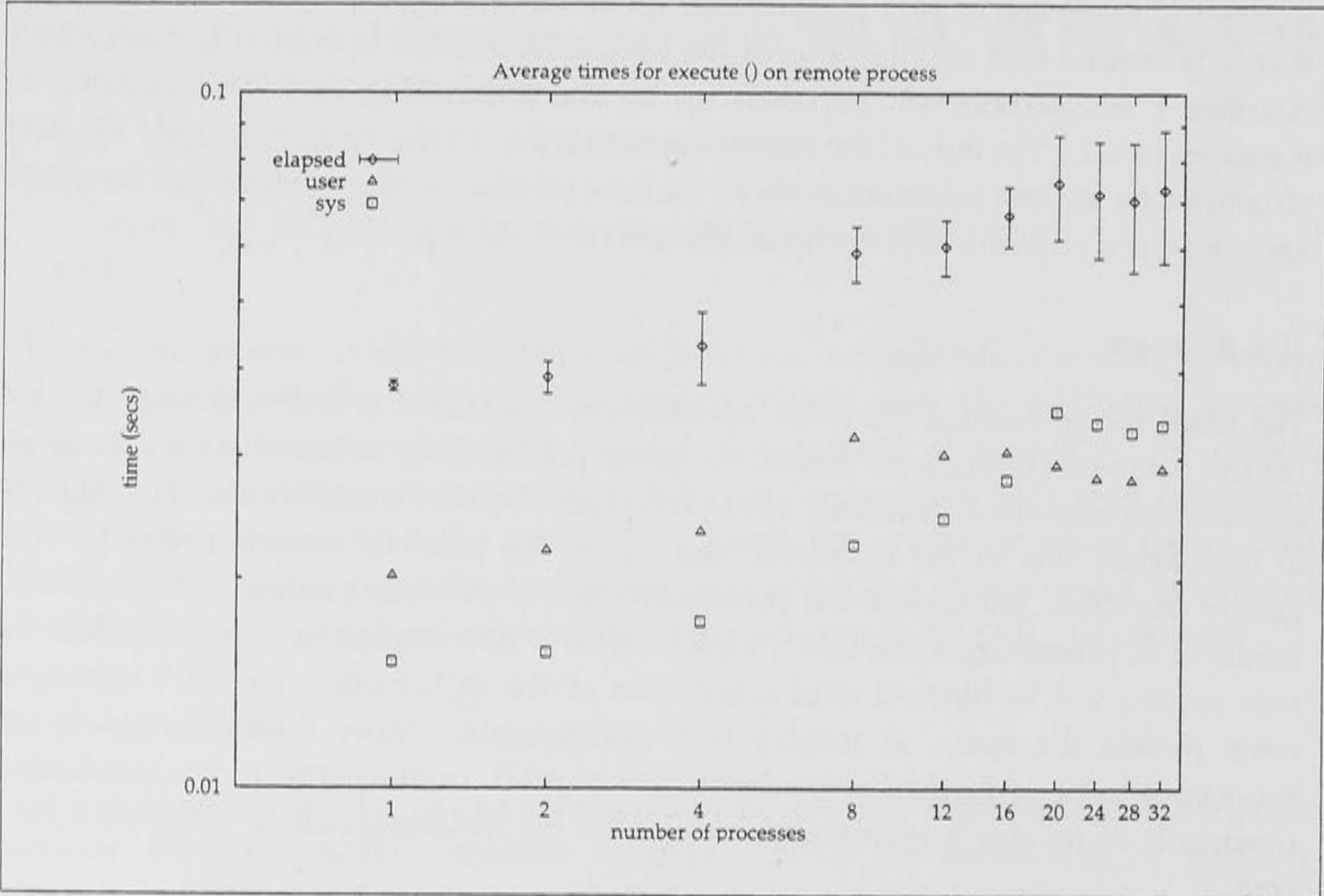


Figure 10.7 – Times for execute operation with differing numbers of processes.

10.6.7 execute

The times for the `execute` operation are reported in Figure 10.7. The garbage collection times are so negligible relative to the other times that they are not shown. Again, the way these times were measured was basically identical to that for the `process` and `port` operation results. However, since the cost of `execute` is so much larger than these others, only 5 operations were performed, rather than 15. Thus each process performing the test would create 5 processes. Once all the processes had been created, then all processes started to perform `execute` operations on the 5 processes it had created. The average of the times for these operations is reported. The code to be executed by each new process was the function given in Table 10-1, `fn () => (self_id (); ())`. Note that the size of this function is approximately 110 Kb, due to the manner in which SML/NJ marshals objects. The large size is due to inclusion of the `self_id` operation in the function, which thus requires the entire paraML runtime system structure to be copied.

The average elapsed times increase progressively from approximately 38 to 72 milliseconds. The time required to send 110 Kb messages in MPI is given in the graph for large messages in Figure 10.3, and is approximately 9.2 milliseconds. The overheads of marshalling and unmarshalling such large messages plus the replies account for an additional 19 milliseconds. Network contention is probably the major factor in the increase in times as the number of processes increases. Since the physical network cannot carry two messages simultaneously through the same piece of wire, the communications for an `execute` operation may be blocked for greater periods of time while other communications are being completed. Another factor that comes into play more frequently with messages of this size is the occasional need to perform heap re-sizing prior to message buffer allocation. For large messages, there is an increased probability that the current available space in the heap will be insufficient. Increasing the heap size also involves a garbage collection operation.

The cost of the `execute` operation is the dominant factor in the overheads of using processes in paraML. If the marshalling mechanism could be altered to avoid copying the paraML runtime system structure (or indeed other structures that do not alter between paraML runtime systems), the time for the operation would be much reduced. The same `execute` test program run with the null function instead of the function which performs `self_id`, reports only 4.6 milliseconds for the operation. The reason for the order of magnitude improvement in times is due to the decreased copying overheads. Unfortunately, such reduced copying is not currently available for useful process executable functions.

Although it is relatively difficult to provide meaningful comparisons with alternative systems, it is interesting to note the times for process creation in AP/Linux. Executing a trivial program (such as `/bin/hostname`) with the `prun` operation (which establishes a set of processes, in this case 1, running the given program on a subset of the PEs of the AP1000+) takes 160 milliseconds. Various sockets and inter-PE communications must be performed for a `prun` operation. In contrast, the fork of a

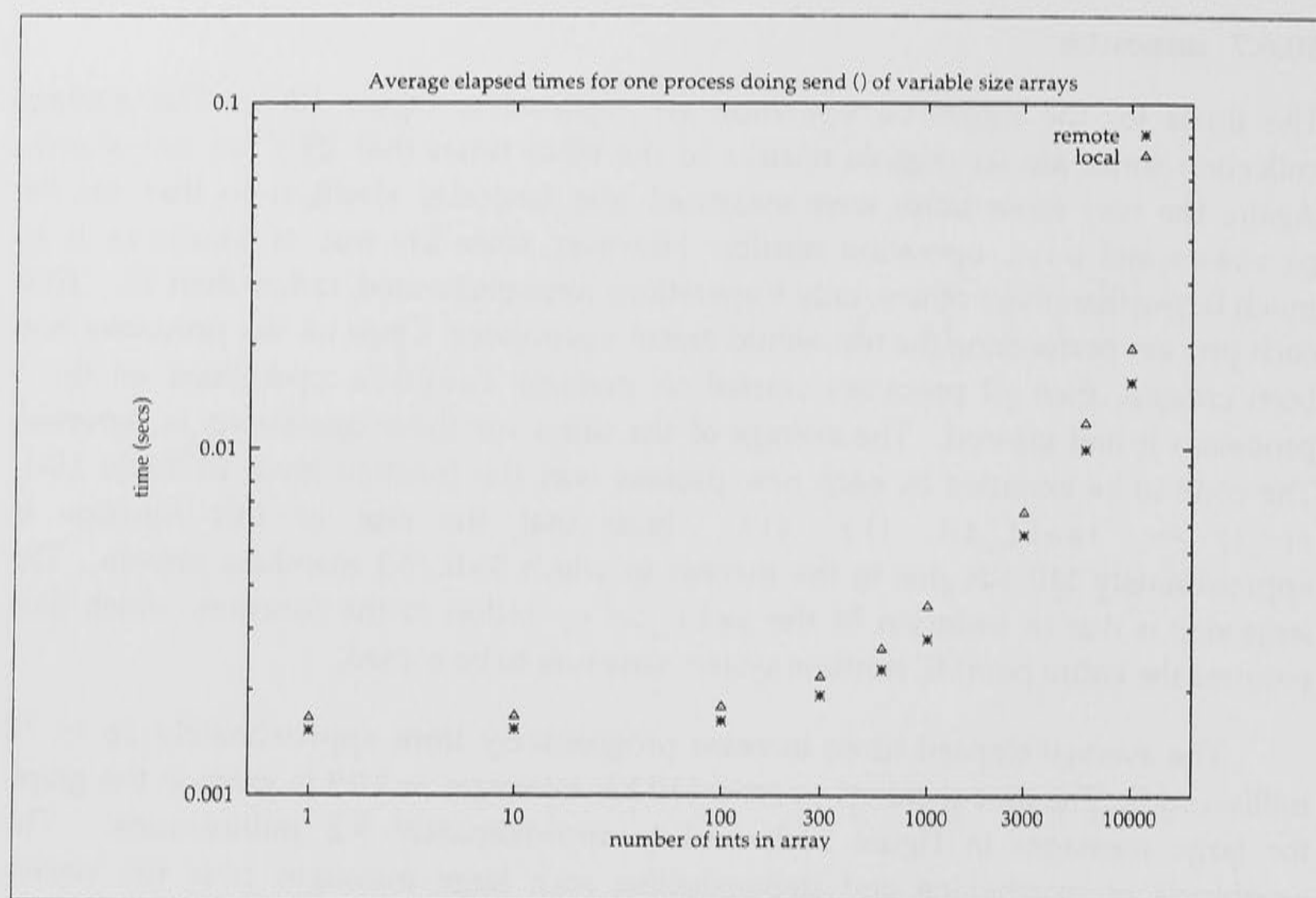


Figure 10.8 – Elapsed sending times for one process and different message sizes.

null process in the Linux benchmark suite for AP/Linux takes approximately 4.3 milliseconds.

10.6.8 send

A number of tests were performed for the `send` operation, since performance varies according to two factors: first, the amount of data being sent, and second, the overall load across the paraML runtime systems.

The first set of results, Figure 10.8, report the elapsed times to perform sending of messages of varying sizes. These times are averaged as usual over a number of `send` operations, and over 10 runs. The standard deviation represents the difference between runs, since there is only one process involved. Mostly this standard deviation is negligible. The messages being sent are integer arrays, ranging in size from 1 to 10000. The overall size in bytes of a selection of these different arrays is given in Table 10-1. Effectively, there is a 12 byte overhead for the header, plus a 16 byte overhead for the array, plus 4 bytes for each integer. The times are given for sending to a port on a remote process and a port on the local process. The local sending is actually slightly more expensive than the remote sending, which is probably due to the additional memory copying involved in the MPI implementation for local sending. Again, this demonstrates a potential for optimising performance by avoiding any interaction with the MPI system for operations that involve only the local paraML runtime system. The times are roughly four to six times as slow as the equivalent MPI times reported in Figure 10.3 and Figure 10.4. The difference is explained by the increased times for

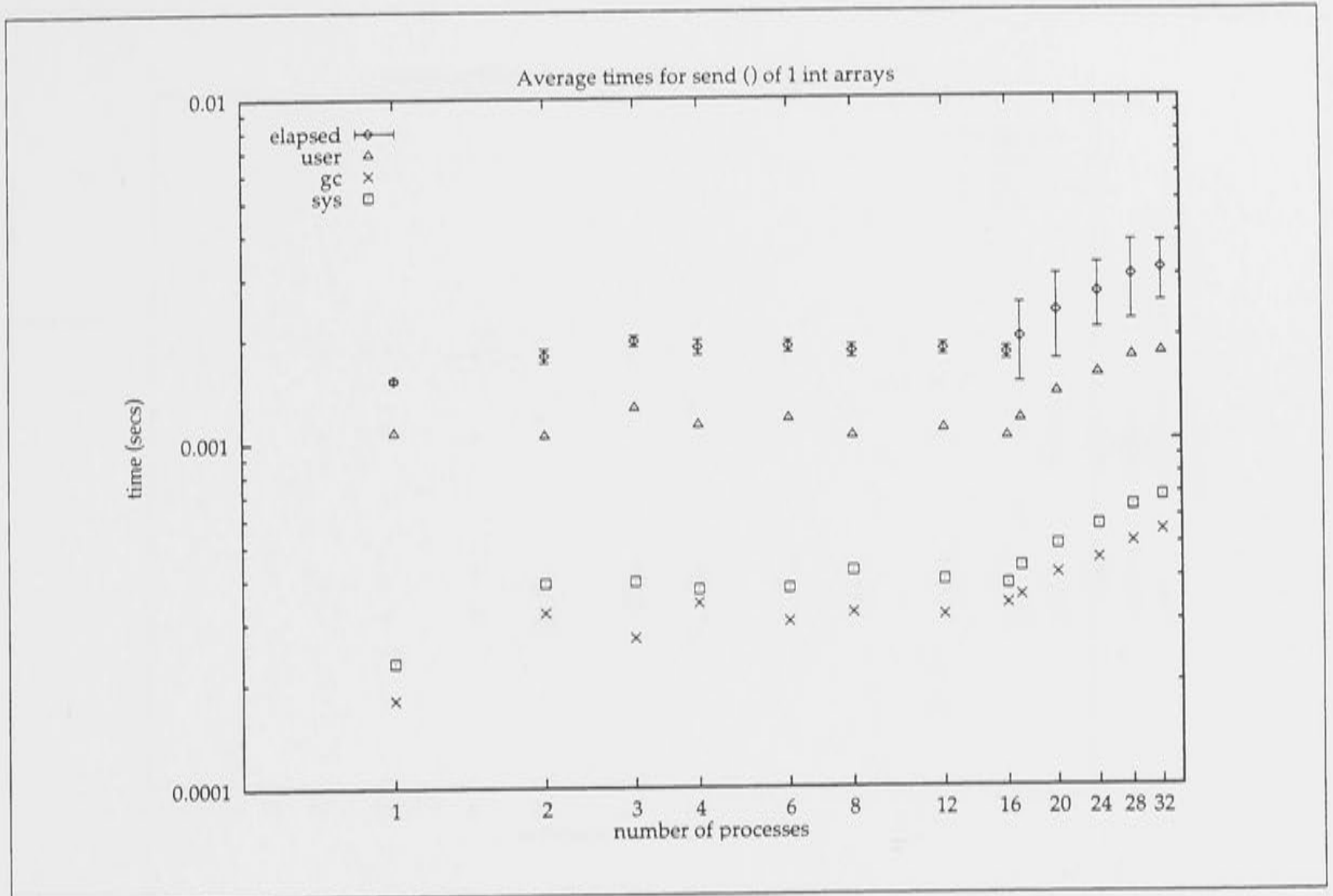


Figure 10.9 – Sending times for differing numbers of processes.

marshalling and unmarshalling messages and the message handling times to queue messages into the destination ports. The times for ML-style messaging in MPI plus the marshalling/unmarshalling overheads account for roughly 1.1 of the 1.5 millisecond total elapsed time for arrays less than 100 integers long. At the 10000 integer array point, the MPI and marshalling/unmarshalling costs account for approximately 12 of the 15.5 milliseconds elapsed time.

The next set of results, sending 1 integer arrays to ports on remote processes, is presented in four different ways. The times are averaged over 100 sends and 10 runs, with the number of processes involved in sending ranging from 1 to 32. While there are 16 or fewer processes, only one process is executing on each paraML runtime system. After this, every paraML runtime system has at least one process executing, and sometimes two; at 32 processes there are two processes on each paraML runtime system.

The first graph in this series, Figure 10.9, measures standard deviation in the same way as for the process, port and execute graphs, being the variation in times any individual process may expect when the total number of processes is set. The times when the number of sending processes is equal to or less than 16 are all fairly uniform in the range 1.5 to 2.0 milliseconds, with little variation at all in the elapsed times. Immediately the number of processes rises to 17, with one paraML runtime system now executing two processes, there is both a rise in the times, and a noticeable jump in the standard deviation. The times gradually increase as the number of processes increases, from around 2.0 to 3.3 milliseconds.

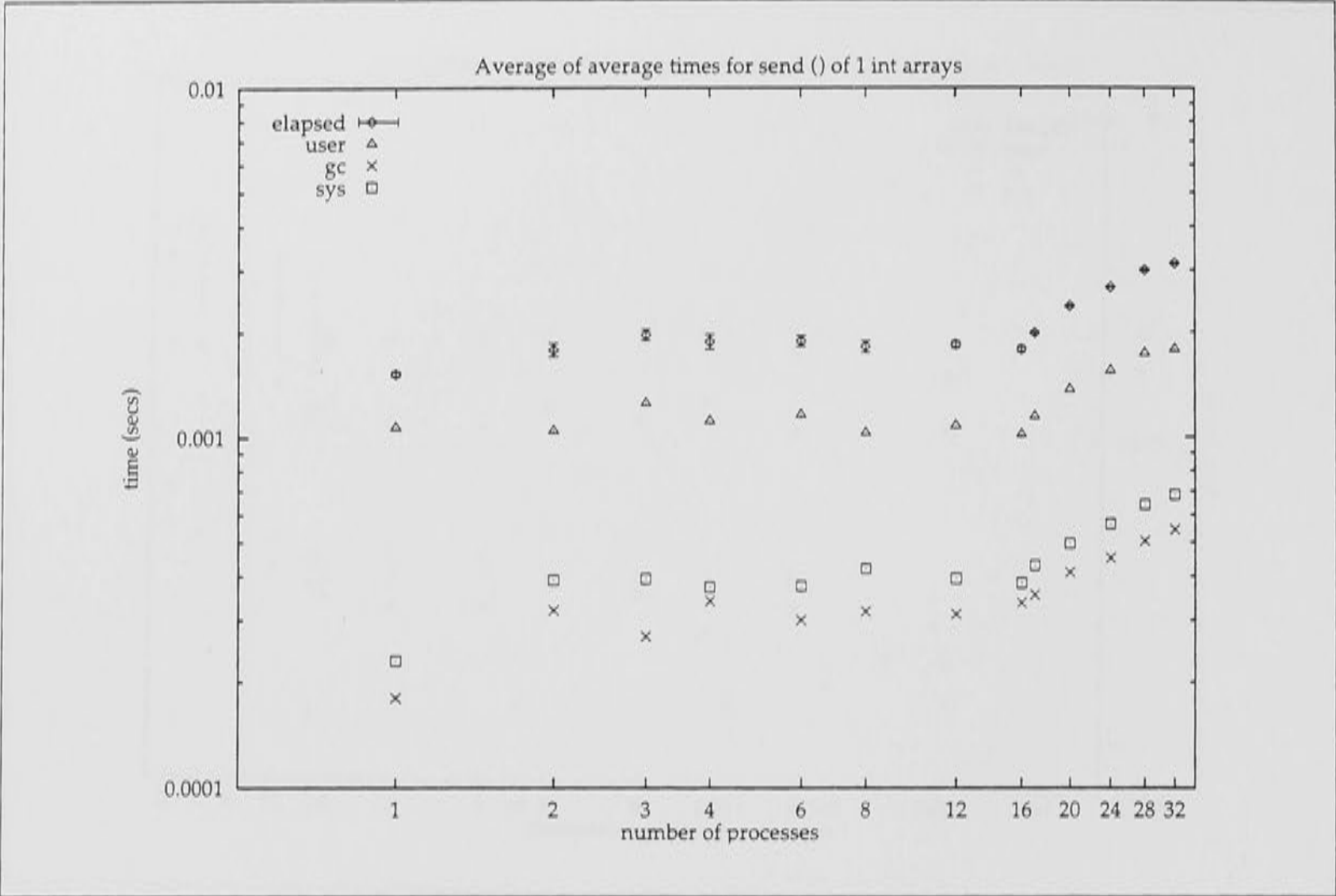


Figure 10.10 – Sending times with standard deviation between runs.

The next graph plots the same results, but instead of being the average from each of the processes across all the runs for any particular number of processes, it is now an average across the average from each run. Standard deviation is thus measuring the differences between runs. Unlike the previous graph, almost no deviation is observed, which probably arises due to an absence of other users of the machine when the tests were run.

The final two graphs in the series examine the sending times from a per-process basis. In Figure 10.11, the average elapsed times from each process are plotted as a scatter graph. In the range 1 to 16 processes, the finishing times are all clumped together reasonably densely. Again, as soon as 17 processes are involved, differences emerge with three distinct clumps, which coalesces into two main clumps for the results for 20 and 24 processes. For the final two data points of 28 and 32 processes, it splits again into 3 distinct clumps of finishing times. In the range 17 to 32 processes, there is still a clump of processes whose elapsed time for completion is the same as that in the first range of 1 to 16, suggesting that the first process on each paraML runtime system takes much the same time as before.

The second scatter plot examines the average user times for each of the processes involved, as shown in Figure 10.12. These results show almost uniform scattering of times, without distinct clumping at all. These results may reflect the iterated starting times, since each process must receive a message to commence their timing, and thus each later process will commence timing when there is slightly increased load across the paraML runtime systems, thus delaying the time to complete its own test.

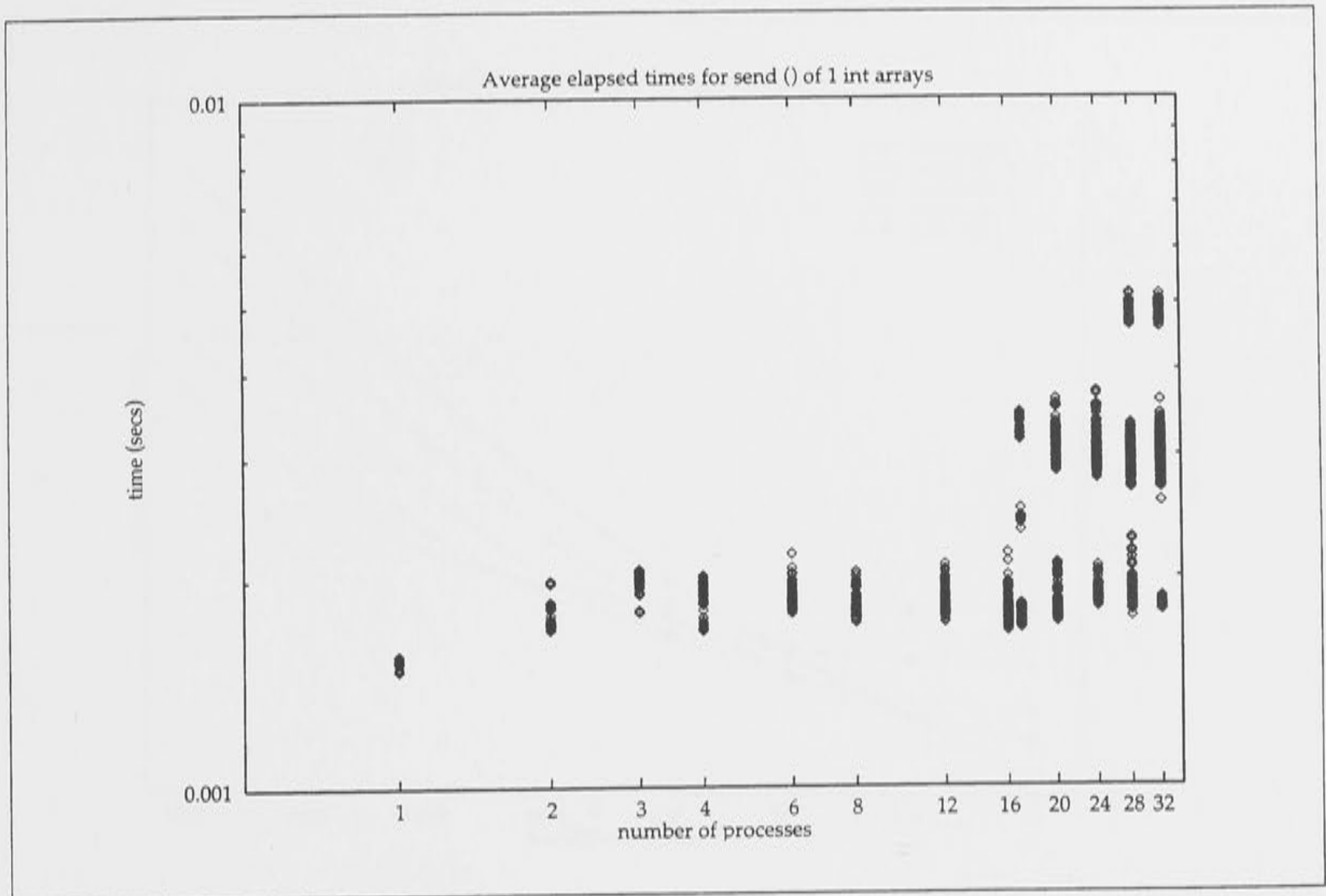


Figure 10.11 – Scatter plot of elapsed sending times.

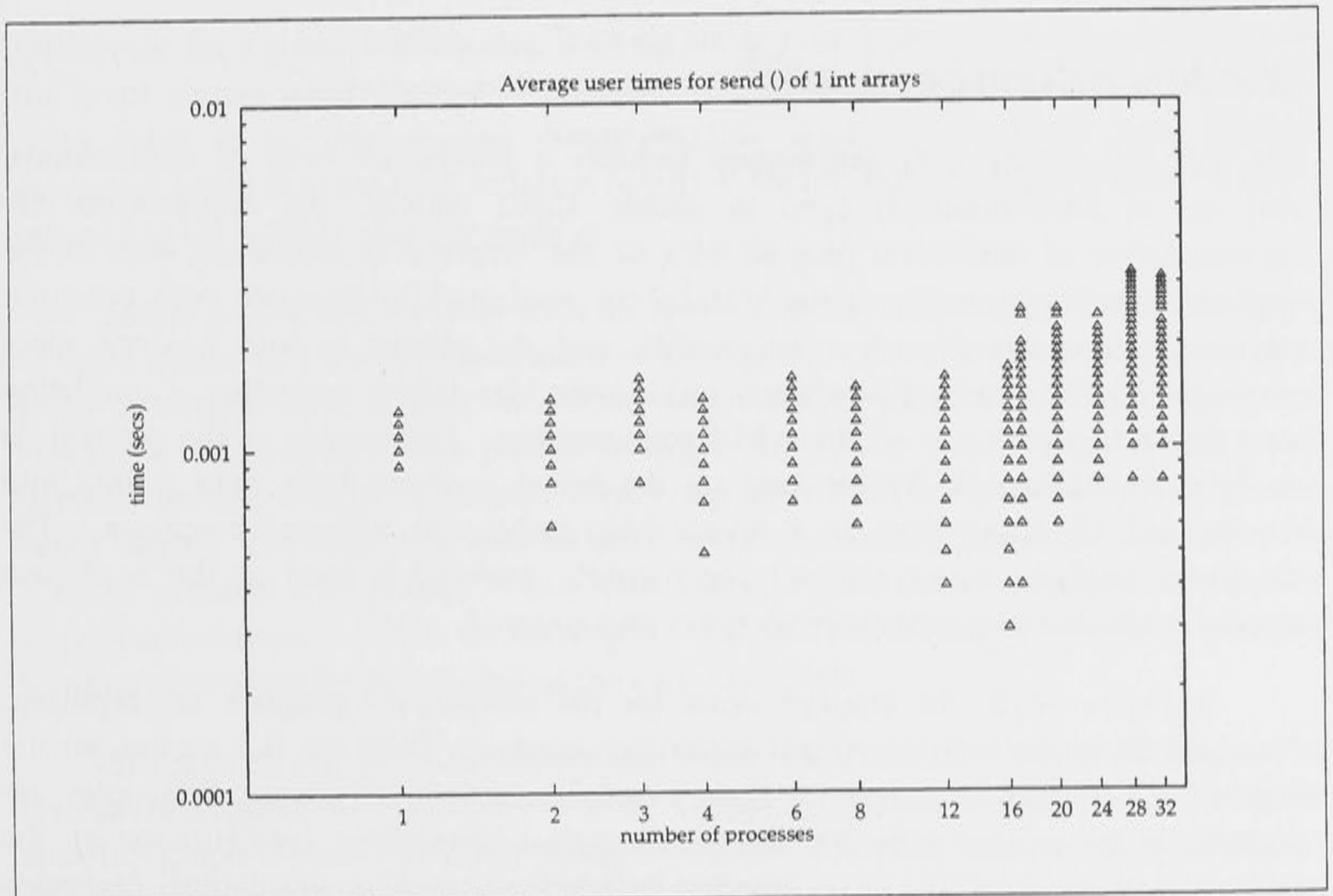


Figure 10.12 – Scatter plot of user sending times.

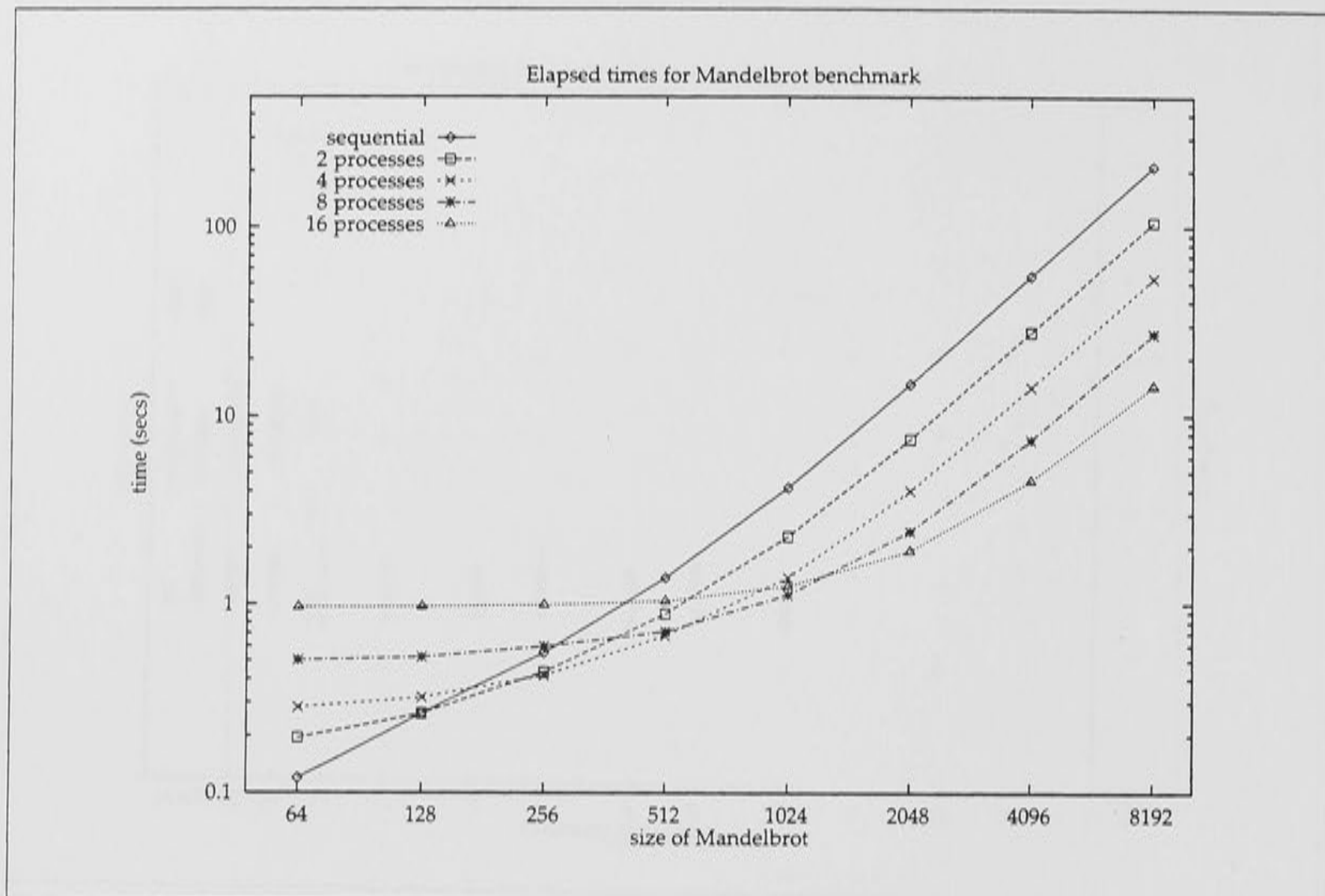


Figure 10.13 – Times for Mandelbrot with differing numbers of processes.

10.7 Mandelbrot benchmark

The SML/NJ version 0.93 distribution includes a benchmark suite of applications. Any set of benchmarks is open to debate about whether the applications are representative of real-world programming or not. Instead of examining each of the applications, I have used only the Mandelbrot benchmark to illustrate both potential advantages and disadvantages of paraML and the pitfalls in such analysis more generally. The Mandelbrot benchmark implements Mandelbrot's equation. Completion time depends on the size of the initial configuration. Partitioning of the problem is straightforward, simply by breaking up the initial configuration's data points into chunks and allocating them in a round robin fashion to different processes. The sequential implementation without any paraML overhead is used as the base case against which the process-based solutions are measured.

In Figure 10.13, the elapsed times for the Mandelbrot program are reported. Standard deviation between runs is almost non-existent. There are five curves: for the sequential program, and then for 2, 4, 8 and 16 process solutions. Both axes are plotted on log scales, with the size of the initial Mandelbrot configuration as the independent variable. The times reported include the time taken to establish processes and commence their execution. As can be seen from the results, the sequential solution is fastest for a problem of size 64. Subsequently, doubling the problem size requires a doubling in the number of processes to achieve the optimal completion time, up to size 2048 and larger, where 16 processes provides the fastest time to completion.

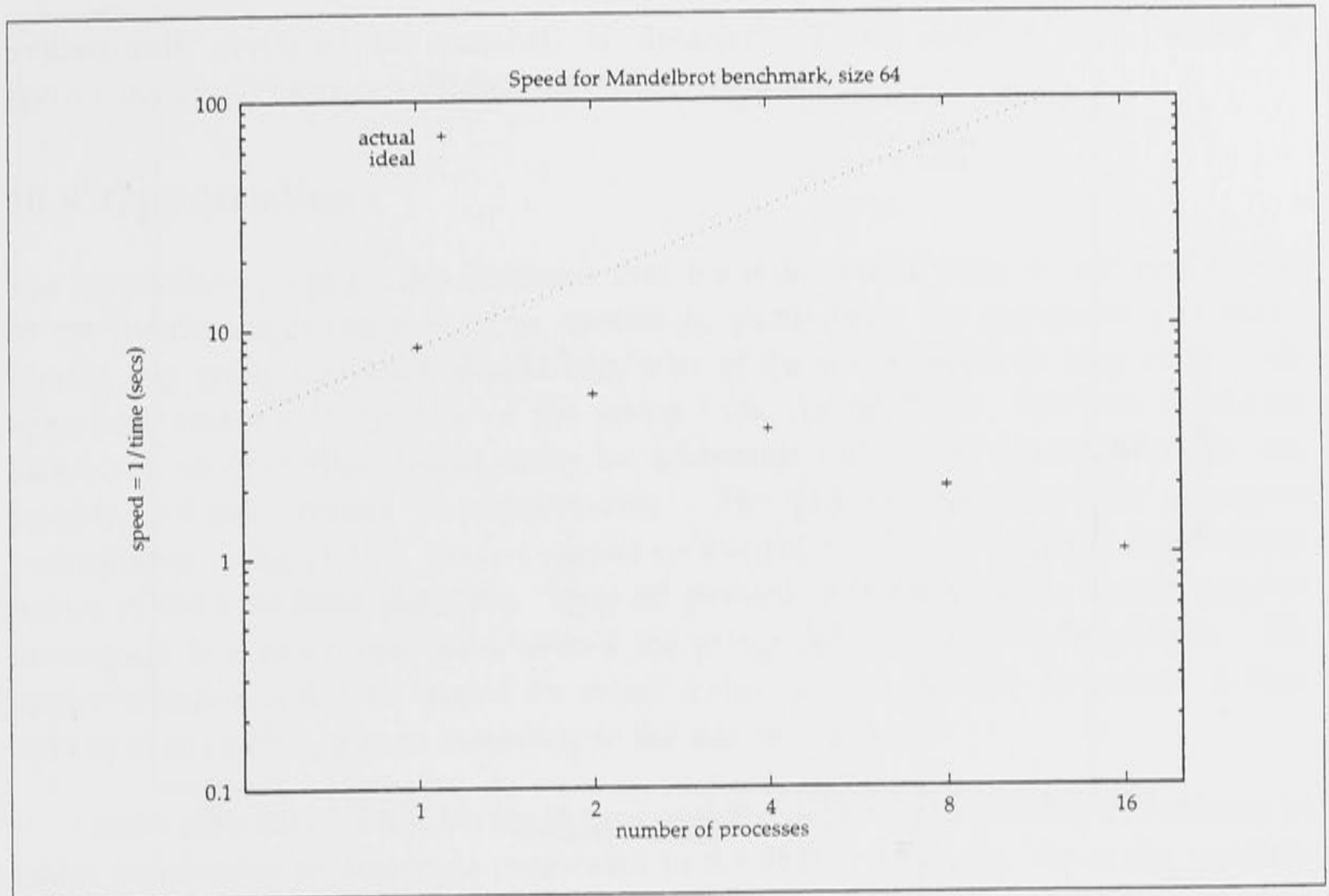


Figure 10.14 – Speed for Mandelbrot at size 64.

Effectively, for a given problem size, walking along a curve joining the lowest points on the graph shows which number of processes will be required for the fastest solution. Since there is no inter-process communication once all processes have started executing, other than the final collation of partial results, the algorithm is (what is often referred to as) “embarrassingly parallel.”

Before getting carried away with the apparent utility of having multiple processes to solve problems faster, consideration of the costs involved is also essential. Figure 10.14 illustrates that at size 64 for the Mandelbrot problem, more processes are actually a disadvantage. The reason for this is that in the current implementation of paraML, process creation/execution costs are proportional to the number of processes being created. The sequential time for the solution negates any benefit of dividing such a small problem into pieces. Note that the one process solution is actually the sequential solution, and that the ideal speed line plotted is the speed for the sequential solution multiplied by the number of processes available.

In contrast, Figure 10.15 shows that when the problem size is 8192, process creation/execution overheads have been almost completely amortised. Thus the speedup graph is almost perfectly linear. With 16 processes, it is just detectable that the speed is not quite 16 times the sequential speed. This kind of variation demonstrates why benchmarks for any parallel program must be treated carefully.

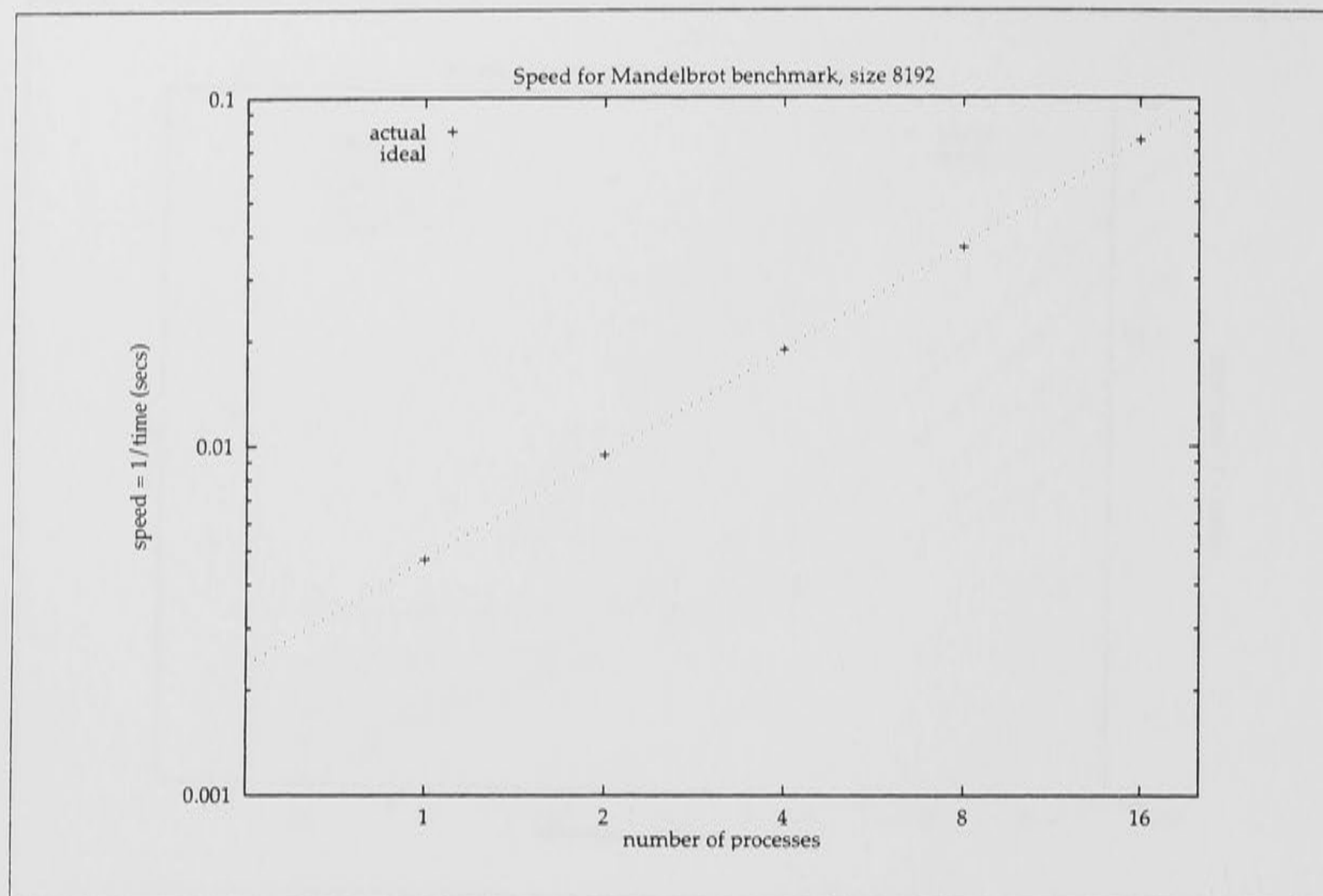


Figure 10.15 – Speed for Mandelbrot at size 8192.

<i>operation</i>	<i>time</i>	<i>percentage</i>
self_port	0.17	0.38%
process	4.8	10.78%
send	1.5	3.37%
execute	38	85.32%
recv	0.066	0.15%
total	44.536	100%

Table 10-5 – Costs of paraML operations per process in Mandelbrot program; times in milliseconds

The paraML operations involved for each process in the Mandelbrot program are a single one each of `self_port`, `process`, `execute`, `send` and `recv`. From the times reported earlier, the costs of each operation can be approximated. These are given in Table 10-5, illustrating yet again that the `execute` operation dominates overall overheads in the current implementation, with 85% of the time due to paraML operations. Since the sequential solution at size 64 takes approximately 120 milliseconds to complete, it is not surprising that a two-process solution takes longer, as each process requires 44.5 milliseconds of paraML overhead, plus at least half the sequential solution time (adding another 60 milliseconds). The elapsed time for this version will be minimally the time at which the last process commences execution, plus half the solution time. When the size of the Mandelbrot is 8192 however, the 16×44.5

milliseconds overhead for paraML is dwarfed by the solution time, which is approximately 212 seconds for the sequential version.

10.8 Optimisations

The Mandelbrot program demonstrates that the most crucial thing to optimise would be the overheads in the `execute` operation, particularly for groups of processes. Ideally, any group operation should only take of the order $\log(n)$ as long as a single operation, where n is the size of the group. On the AP1000+, with its hardware broadcast network this should easily be achievable with some modifications to the paraML runtime system implementation. The primary restriction to improved performance using an `MPI_Bcast` operation instead of `MPI_Send` is the synchronous nature of the broadcast primitive. Thus all paraML runtime systems would need to participate in a group operation, even if the group did not span all the nodes. The implementation could be tuned to select either a `MPI_Bcast` or create a tree broadcast from `MPI_Sends` according to the size of the group.

More generally, a huge saving in time could be achieved by altering the manner in which marshalling of objects is performed in the SML/NJ system. Since the paraML runtime system code, represented in a single structure, never changes, there is no need to copy the structure when marshalling a function that uses paraML operations. All of the data structures in the paraML runtime system are protected from copying by isolating them with the `setvar/getvar` primitives. A substantial modification to the marshalling code could have the desired effect, thereby reducing the size of `execute` requests from 110Kb to only a few kilobytes at most. The overheads in process creation/execution would be dramatically reduced, thus reducing the point at which process overheads would be amortised by the potential speedup. This observation is borne out by the comparison of `execute` times with the null function versus the paraML executable function involving `self_id`, which gave an order of magnitude improvement.

Two other less major contributors to optimisation would be multi-port creation operations and use-once ports. Since every port operation requires a request/reply communication with a remote paraML runtime system, if these could be bundled together into a single request, large savings would be made. The marginally increased size of data would be completely offset by the reduction in repeated message latencies. Use-once ports occur quite frequently in the implementation of several of the derived operations. In these operations, a port is created solely for the purpose of receiving an answer, and it is thrown away at the end of the operation. Instead of paying the price of the `self_port` operation each time, use-once ports could be allocated as needed, and then their data structures placed on a free list for reuse once the operation has finished with them. The concept is somewhat limited, since they could not be made available to the user due to their generic `System.Unsafe.object` typing, and the associated cast operations required to preserve the type-safe communications in paraML.

10.9 Conclusion

The degree to which the paraML system meets the efficiency aspects of its stated goal of supporting safe and efficient high performance programming is moot. Ultimately, high performance is judged by the ability to deliver better performance than a sequential system and also by an ability to deliver scalable performance in the face of increasing system resources for a broad range of applications. There remains considerable scope for optimisation of aspects of the current implementation to achieve better performance. Nevertheless, the existing system demonstrates that process-oriented computing for distributed address spaces in an ML language framework is practicable and need not be inefficient. The basic construction of the system illustrates acceptable scalability for the paraML operations as the number of processes increases. For problems that do not require large amounts of inter-process communication during their execution, the existing implementation allows process-based solutions to provide appreciably faster performance once sequential execution time grows beyond a few seconds.

Part V

CONCLUSION

Part V

CONCLUSION

11. Future research

11.1 Introduction

This thesis has explored in depth the design, theoretical specification, and practical application and implementation issues for a process-oriented programming language. The same framework is used in discussing avenues for future research.

11.2 Design

The design of paraML provides a close-to-minimal set of extensions for process-oriented programming. A characteristic of these operations is that they involve at most two processes and/or one port. The benefit of this characteristic is that theoretical specifications of the extensions are relatively straightforward. Two obvious avenues for future research are collective operations and the replacement of a choice primitive for the `probe` operation. Both would break the simplicity of having at most two interacting processes to an operation.

It is clear that for practical parallel programming, the ability to specify collective operations over a set of processes or communication ports is highly desirable. The difficulty arises in finding an appropriately minimal set of such collective operations and specifying them theoretically. Krumvieda's work with Distributed ML and its multiports provides an example of one possible option [Kru93]. The approach taken in this thesis of specifying collective operations as derivations from the core paraML operations, while utilitarian, may lead to some undesirable performance characteristics. Finding collective operations that are as flexible and powerful for building abstractions as the paraML core operations, yet as straightforward to model in the context of dynamic process and port creation, would be an interesting challenge.

The decision to use `probe` rather than a choice primitive was made primarily because of the ML type system. Had the primitives been intended for an untyped language such as Lisp, or a language with dynamic typing at runtime such as Scheme, a choice primitive might have been selected instead. In either case, it would be possible to construct lists of communication ports where the type of objects to be received on each port might differ, and then choose one port non-deterministically. It would be an intriguing exercise to apply the paraML extensions to a language providing similar high level facilities to ML, and analyse which choices would be different.

The other obvious future work would be to incorporate shared address space concurrency extensions along the lines of CML into the language. The desirability of

such a two-level approach to the provision of concurrency (that is, those which operate in a shared address space environment and those which operate in the context of a distributed address space) has been argued earlier. The practical implementation of such a system is relatively straightforward, but to achieve a clean integration of this at the design level and in the theoretical specification is more challenging.

11.3 Theory

The theoretical specification of the semantics of the paraML language is valuable for building an implementation of paraML. One of the other advantages of this style of semantics was the inclusion of references and exceptions in the sequential calculus. The major contribution of the theoretical work has been to show how values in the calculus, including references and exceptions, are transmitted between distributed address space evaluation environments. However, the style of semantics adopted makes no provision for reasoning about equivalence between different programs. ParaML shares this limitation with CML since I have adopted the style used by Reppy in his dissertation. Program fragment equivalence can be particularly useful in determining possible classes of optimisations. The work on providing an alternative theoretical model for CML by Ferreira *et al* [FHJ95], equivalent to Reppy's semantics, provides an example approach to developing such a semantics. Alternatively, the LTS style adopted for Facile [TLG92] could be used. The inclusion of locality modelling in the Facile semantics is particularly attractive.

11.4 Practice

Although earlier versions of paraML had users other than the author involved in developing applications, the current system has had less testing. Various improvements were made to the earlier versions as a result of user-feedback, which ultimately led to the complete redesign of the language. Additional feedback would assist in critical analysis of the efficacy of the current system's design and implementation choices.

It would be desirable also to complete any future implementation on top of the latest versions of the SML/NJ compiler (version 1.09 and later), since they support the newly-standardised library modules and provide better foreign language interfaces. Various other changes in the SML/NJ compiler would assist in making the implementation of paraML more portable and robust. For instance, the standardisation of interfaces to POSIX definitions would make the provision of a low-level concurrent I/O library more portable. Alternatively, the Objective ML system could be used [RV96]; its smaller memory footprint might optimise aspects of the performance. Similarly, some of the techniques for making use of type information at runtime demonstrated in the TIL compiler [Mor95, TMC+96] hold some intriguing possibilities for optimising communications.

An interesting experiment involves exploring alternative scheduling policies and implementations of the different layers in the programming model. Such an experiment would require providing clearer abstractions between the different layers in the paraML runtime system. The STING system uses a similar approach to experiment with virtual processor scheduling [JP92]. Alternatives such as gang scheduling for process groups might be particularly appropriate if stronger support for the derived operations in paraML were thought desirable.

Another experiment would be to explore garbage collection of data structures for uncontactable processes and ports. At present, a process either executes forever (at least until the root process terminates) or terminates. Since the type system guarantees that the result from a process is the unit value, any data structures created by the process can be garbage collected on termination. There is also some state managed by the paraML runtime system for the process and its ports, which can be garbage collected on process termination. However, if the process is still executing, it would be elegant to garbage collect the entire process if it is uncontactable by other processes and it is incapable of contacting processes in its own right. In CML, this is readily achievable since garbage collection is performed over the entire set of threads and channels. Any thread which becomes disconnected from the active part of the system will be identified as garbage (since it will be attempting to perform blocking communication on a channel that no other thread knows about). In paraML, this situation is significantly more complex. The only case where it would be acceptable to garbage collect a process would be when all threads within the process were attempting to perform blocking communications on ports, and none of the associated port names existed in the remainder of the system. Since the system is distributed, identification of the non-existence of these port names becomes a global garbage collection problem. Significant changes to the existing garbage collection algorithm would be required in order to identify both the execution state of processes and the distributed knowledge of port names.

Load balancing of paraML systems might be a more useful experiment in the context of achieving high performance. The evaluation state of a process can be captured with continuations, and simple data structures represent the state associated with the process and its ports. If the system detected cases of serious load imbalance, these data structures and the evaluation continuation could be shifted to a less heavily loaded paraML runtime system. Forwarding information would need to be left at the old runtime system to make sure that any messages were correctly delivered to the new location. The data structures used to represent process names and port names could be updated on information that a process had been moved from one paraML runtime system process to another.

As with all systems constructed from processes, debugging facilities are inadequate for paraML. As yet, no breakthrough has been achieved in making such parallel debugging systems as easy to use as their sequential counterparts. The

challenges faced in a system like paraML with dynamic process and port communication make this problem doubly hard.

Lastly, it would be valuable to explore implementation of paraML on other multicomputer platforms and also to include shared address space computers such as the SGI PowerChallenge. One of the dangers of developing a language such as paraML on a single platform is that the language can start to reflect particular biases in the underlying architecture and operating system. The major bias that implementation on the Fujitsu AP1000+ is likely to have caused is an assumption of well-balanced communication performance (with low-latency and high-bandwidth characteristics) relative to computation performance. A conscious effort has been made to avoid additional biases due to the hardware. This effort stands in contrast to the earlier versions of paraML, which optimised the implementation using machine-specific hardware characteristics such as a separate broadcast network. Another way of avoiding such biases has been to use MPI as the underlying communication platform, and to restrict use of the MPI primitives to a fairly minimal set that are efficient on any platform.

12. Conclusion

Computing environments that encompass distributed address spaces are ever more prevalent. Programming languages for distributed address space computation are emerging rapidly. Similarly, researchers are placing increasing emphasis on finding effective and high-level solutions to the problems of partitioning programs. Research into systems which combine these interests with language mechanisms that are both safe and efficient has been relatively rare. In this thesis, I have proposed that the concept of process-oriented programming facilities can be used to provide an effective solution to some of the problems. The developments outlined in the thesis address the areas of interest in novel ways, particularly with respect to the transmission of mutable values between processes in distributed address spaces.

The context for these developments is the language paraML, an extension of ML, which provides a process-oriented programming framework. The design of paraML, described in Part II, captures the key concerns of distributed address space computation, and mechanisms for partitioning and coordination of program components. The extensions to ML carry the safe aspects of programming in that language into a process-oriented model. The paraML operations are also used in developing various alternative forms of communication and synchronisation.

The ability to formally model the ideas embodied by paraML is critical. In Part III, an operational semantics is developed for an untyped λ -calculus, called λ_{pv} . This calculus models the evaluation of programs with processes and ports in the context of a distributed address space. The sequential aspects of the calculus closely resemble ML, including exceptions and references. The formal modelling of transmission of such values among different processes is new. An ability to formally describe evaluation with transmitted values will be critical in emerging languages which permit computations that may be globally distributed. A polymorphic type discipline for λ_{pv} is also presented, which is again similar to the type system of ML. The typing of λ_{pv} is proved sound with respect to its operational semantics. To my knowledge, this is the first proof of type soundness for a polymorphic process-oriented language with communications involving non-shared mutable values across a distributed address space.

The paraML language has been implemented, and used for various alternative programming paradigms and applications, as described in Part IV. The implementation of the language and the requirements for building similar systems in the framework of multicomputer architectures are described in detail. Performance measurements of the system-level aspects of the implementation are presented and discussed. These experiments characterise the overheads in using process-oriented constructs and demonstrate that paraML is a useful and scalable system for safe and efficient high performance programming on distributed address space computers.

12. Conclusion

The purpose of this study was to investigate the effects of the proposed system on the performance of the participants. The results of the study show that the proposed system significantly improved the performance of the participants compared to the control group. The improvement was observed in all the measured variables, including the number of correct answers, the time taken to complete the task, and the number of errors. The results also show that the proposed system was effective in reducing the cognitive load of the participants, as indicated by the lower scores on the NASA-TLX questionnaire. The findings of this study suggest that the proposed system is a promising tool for improving the performance of participants in tasks that require complex decision-making and problem-solving. Further research is needed to investigate the long-term effects of the proposed system and to determine the optimal parameters for its use.

The study was conducted using a between-subjects design, with participants assigned to either the proposed system group or the control group. The participants were recruited from a pool of volunteers who were screened for eligibility based on their age, education level, and experience with the task. The study was approved by the Institutional Review Board (IRB) of the university. The data were analyzed using a two-tailed t-test to compare the performance of the two groups. The results of the t-test show that the proposed system group performed significantly better than the control group in all the measured variables. The effect sizes for the comparisons were medium to large, indicating that the proposed system has a meaningful impact on the performance of participants.

The results of this study have several implications for the design of interactive systems. First, the proposed system is a promising tool for improving the performance of participants in tasks that require complex decision-making and problem-solving. Second, the proposed system is effective in reducing the cognitive load of participants, which is an important consideration for the design of interactive systems. Third, the proposed system is easy to use and requires minimal training, which makes it a practical tool for use in a variety of settings. Finally, the results of this study suggest that the proposed system is a promising tool for improving the performance of participants in tasks that require complex decision-making and problem-solving.

APPENDICES
GLOSSARY
REFERENCES

APPENDICES

GLOSSARY

REFERENCES

Appendix A -1

Proofs from Chapter 5

This appendix contains proofs of the lemmas in Chapter 5. Many of the proofs utilise partial results from the work of Reppy [Rep92] and Wright and Felleisen [WF91].

Proof of Lemma 5-6

Lemma 5-6 If $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$ and $\sigma' \succ \sigma$, then $TE \pm \{x \mapsto \sigma'\} \vdash e : \tau$.

Proof. The proof of this lemma is by induction on the height of the typing deduction of $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$, and case analysis on the shape of e for the last step. Remember that x is not bound in e by the variable convention.

The cases which are interesting are those which possibly affect the typing environment TE 's variable component. These cases are: $e = x'$, $e = \lambda x'(e')$, $e = \text{let } x' = v \text{ in } e'$, $e = \text{let } x' = e_1 \text{ in } e_2$, $e = := x'$, $e = \text{exception } x' \text{ in } e'$, and $e = \rho\theta.e'$. All but the last three are proven by Reppy for λ_{cv} (Lemma A.2 proof, pp.179-180 [Rep92]), whose sequential components are effectively identical to λ_{cv} . Thus the cases for $:= x'$, exception terms and ρ -bound expressions are the only ones given here. The case for the term $:= x'$ follows from the rules for (τ -app) and (τ -assign). In this proof, the case is shown, but for future lemmas which follow similar strategies using these rules, the details will be ignored.

Case $e = := x'$.

Rule (τ -assign) applies:

$$TE \pm \{x \mapsto \sigma\} \vdash := : \psi \text{ ref} \rightarrow \psi \rightarrow \psi \text{ and } TE \pm \{x \mapsto \sigma\} \vdash := x' : \tau$$

such that $\tau = \psi \rightarrow \psi$, then by rules (τ -app) and (τ -var):

$$TE \pm \{x \mapsto \sigma\} \vdash x' : \psi \text{ ref}$$

By the induction hypothesis:

$$TE \pm \{x \mapsto \sigma'\} \vdash := : \psi \text{ ref} \rightarrow \psi \rightarrow \psi \text{ and } TE \pm \{x \mapsto \sigma'\} \vdash x' : \psi \text{ ref}$$

And by (τ -app) again,

$$TE \pm \{x \mapsto \sigma'\} \vdash := x' : \tau$$

Case $e = \text{exception } x' \text{ in } e'$.

Rule (τ -exn) applies:

$$\frac{\text{TE} \pm \{x \mapsto \sigma, x' \mapsto \psi \text{ exn}\} \vdash e' : \tau}{\text{TE} \pm \{x \mapsto \sigma\} \vdash \text{exception } x' \text{ in } e' : \tau}$$

By the induction hypothesis:

$$\text{TE} \pm \{x \mapsto \sigma', x' \mapsto \psi \text{ exn}\} \vdash e' : \tau$$

And applying (τ -**exn**):

$$\text{TE} \pm \{x \mapsto \sigma'\} \vdash \text{exception } x' \text{ in } e' : \tau$$

Case $e = \rho\theta.e'$.

Rule (τ -**rho**) applies:

$$\frac{\begin{array}{l} \text{TE} \pm \{x \mapsto \sigma, x_1 \mapsto \psi_1 \text{ ref}, \dots, x_n \mapsto \psi_n \text{ ref}\} \vdash e' : \tau \\ \text{TE} \pm \{x \mapsto \sigma, x_1 \mapsto \psi_1 \text{ ref}, \dots, x_n \mapsto \psi_n \text{ ref}\} \vdash v_i : \psi_i \end{array}}{\text{TE} \pm \{x \mapsto \sigma\} \vdash \rho\langle x_1, v_1 \rangle \dots \langle x_n, v_n \rangle.e' : \tau}$$

By the induction hypothesis:

$$\text{TE} \pm \{x \mapsto \sigma', x_1 \mapsto \psi_1 \text{ ref}, \dots, x_n \mapsto \psi_n \text{ ref}\} \vdash e' : \tau$$

If $x \notin \text{FV}(v_i)$, **Lemma 5-5** means $\text{TE} \pm \{x_1 \mapsto \psi_1 \text{ ref}, \dots, x_n \mapsto \psi_n \text{ ref}\} \vdash v_i : \psi_i$

and again means $\text{TE} \pm \{x \mapsto \sigma', x_1 \mapsto \psi_1 \text{ ref}, \dots, x_n \mapsto \psi_n \text{ ref}\} \vdash v_i : \psi_i$

Otherwise, if $x \in \text{FV}(v_i)$, by the induction hypothesis:

$$\text{TE} \pm \{x \mapsto \sigma', x_1 \mapsto \psi_1 \text{ ref}, \dots, x_n \mapsto \psi_n \text{ ref}\} \vdash v_i : \psi_i$$

And by (τ -**rho**):

$$\frac{\begin{array}{l} \text{TE} \pm \{x \mapsto \sigma', x_1 \mapsto \psi_1 \text{ ref}, \dots, x_n \mapsto \psi_n \text{ ref}\} \vdash e' : \tau \\ \text{TE} \pm \{x \mapsto \sigma', x_1 \mapsto \psi_1 \text{ ref}, \dots, x_n \mapsto \psi_n \text{ ref}\} \vdash v_i : \psi_i \end{array}}{\text{TE} \pm \{x \mapsto \sigma'\} \vdash \rho\theta.e' : \tau}$$

Thus the lemma is proved. ■

Proof of Lemma 5-7

Lemma 5-7 (Substitution) If $x \notin \text{FV}(v)$, $\text{TE} \vdash v : \tau$, and $\text{TE} \pm \{x \mapsto \forall \alpha_1 \dots \forall \alpha_n. \sigma\} \vdash e : \tau'$, with $\{\alpha_1, \dots, \alpha_n\} \cap \text{FTV}(\text{TE}) = \emptyset$, then $\text{TE} \vdash \{v/x\}e : \tau'$.

Proof. Just as in the previous lemma, the proof of the substitution lemma is by induction on the height of the typing deduction, and case analysis on the shape of e for the last step. Let $\text{TE} = (\text{VT}, \text{ET})$, $\text{VT}' = \text{VT} \pm \{x \mapsto \forall \alpha_1 \dots \forall \alpha_n. \tau\}$, and $\text{TE}' = (\text{VT}', \text{ET})$ in the following cases. As in **Lemma 5-6**, many of the cases are proven by Reppy for λ_{cv} (Lemma 8.5 proof, pp.181-184 [Rep92]). Reppy remarks that Wright and Felleisen also provide proofs for many of the cases. This situation holds true here also, and various of the cases involving references will refer the reader to either their work (Lemma 5.3 proof, pp. 28-29 [WF91]) or Reppy's proof.

Cases $e = c, x', \lambda x'(e'), e_1 e_2, (e_1 . e_2)$, and $\text{let } x' = v' \text{ in } e'$. See [Rep92].

Cases $e = Y, \text{ref}, !, :=, \text{let } x' = e_1 \text{ in } e_2$, and $\rho\theta.e'$. See [WF91].

Case $e = := x'$. Follows from the cases for $:=$ and x' by the rules for $(\tau\text{-app})$, $(\tau\text{-assign})$ and $(\tau\text{-var})$.

Case $e = ex$.

Rule $(\tau\text{-ex})$ applies, thus $\text{ET}(ex) = \tau'$.

Since $\{v/x\}ex = ex$, $(\tau\text{-ex})$ can be applied to get $\text{TE} \vdash \{v/x\}ex : \tau'$.

Case $e = \text{exception } x' \text{ in } e'$.

Rule $(\tau\text{-exn})$ applies:

$$\frac{\text{TE}' \vdash \{x' \mapsto \psi \text{ exn}\} \vdash e' : \tau'}{\text{TE}' \vdash \text{exception } x' \text{ in } e' : \tau'}$$

By the variable convention, $x' \notin \text{FV}(v)$, so Lemma 5-3 gives:

$$\text{TE} \vdash \{x' \mapsto \psi \text{ exn}\} \vdash v : \tau.$$

Thus, by the induction hypothesis and rule $(\tau\text{-exn})$:

$$\frac{\text{TE} \vdash \{x' \mapsto \psi \text{ exn}\} \vdash \{v/x\}e' : \tau'}{\text{TE} \vdash \text{exception } x' \text{ in } \{v/x\}e' : \tau'}$$

and therefore, $\text{TE} \vdash \{v/x\}(\text{exception } x' \text{ in } e') : \tau'$.

Case $e = \text{raise}$.

Rule $(\tau\text{-raise})$ applies, thus $\text{TE}' \vdash \text{raise} : \tau'$ where $\tau' = \psi \text{ exn} \rightarrow \psi \rightarrow \tau''$, for any TE' .

Hence $\text{TE} \vdash \text{raise} : \tau'$. Also, since $\{v/x\}\text{raise} = \text{raise}$, $\text{TE} \vdash \{v/x\}\text{raise} : \tau'$.

Case $e = e' \text{ handle}$.

Rule $(\tau\text{-handle})$ applies, so:

$$\frac{\text{TE}' \vdash e' : \tau''}{\text{TE}' \vdash e' \text{ handle} : \tau'} \quad \text{such that } \tau' = \psi \text{ exn} \rightarrow (\psi \rightarrow \tau'') \rightarrow \tau'.$$

By the induction hypothesis and rule $(\tau\text{-handle})$:

$$\frac{\text{TE} \vdash \{v/x\}e' : \tau''}{\text{TE} \vdash \{v/x\}e' \text{ handle} : \tau'}$$

Therefore, $\text{TE} \vdash \{v/x\}(e \text{ handle}) : \tau'$.

Case $e = [ex, v']$.

The typing rules for raised exceptions gives $\text{TE}' \vdash ex : \psi \text{ exn}$ and $\text{TE}' \vdash v' : \psi$, where $\text{TE}' \vdash [ex, v'] : \tau'$. By the induction hypothesis, $\text{TE} \vdash \{v/x\}ex : \psi \text{ exn}$ and $\text{TE} \vdash \{v/x\}v' : \psi$. Hence, $\text{TE} \vdash \{v/x\}[ex, v'] : \tau'$.

Thus the lemma is proved. ■

Proof of Lemma 5-8

Lemma 5-8 (Type preservation) For any type environment TE, expression e_1 , and type τ , such that $TE \vdash e_1 : \tau$, if $e_1 \xrightarrow{vr} e_2$ then $TE \vdash e_2 : \tau$.

Proof. Let $E[e] = e_1$ and $E[e'] = e_2$, and assume that $TE' \vdash e : \tau'$ with $TE' = (VT', ET')$. Then by **Replacement (Lemma 5-4)**, it is sufficient to show that $TE' \vdash e' : \tau'$. This is done by case analysis of the definition of \xrightarrow{vr} .

As in **Lemma 5-7**, many of the cases are proven by either Reppy for λ_{cv} (Theorem 8.7 proof, pp.181-184 [Rep92]) or by Wright and Felleisen (Lemma 4.3 proof, p. 15, and Lemma 5.2 proof, pp. 26-28 [WF91]).

Cases $E[c \ v] \xrightarrow{vr} E[\delta(c, v)]$, $E[\lambda x(e) \ v] \xrightarrow{vr} E[\{v/x\}e]$,
 $E[\text{let } x = v \text{ in } e] \xrightarrow{vr} E[\{v/x\}e]$. See [Rep92].

Case $E[Y \ v] \xrightarrow{vr} E[v \ (\lambda x(Y \ v) \ x)]$. See [WF91].

Case $\chi, E[\text{exception } x \text{ in } e] \xrightarrow{vr} \chi + ex, E[\{ex/x\}e]$ $ex \notin \chi$.

Then there is a $TE = (VT, ET)$ and types τ' and ψ , such that:

$$\frac{TE \vdash \{x \mapsto \psi \text{ exn}\} \vdash \chi, e : \tau'}{TE \vdash \chi, \text{exception } x \text{ in } e : \tau'}$$

Let ex be a new exception name (hence $ex \notin \chi$) and define $ET' = ET \pm \{ex \mapsto \psi \text{ exn}\}$; obviously $ET \subseteq ET'$. Then by **Lemma 5-3**:

$$TE' \vdash \chi, \text{exception } x \text{ in } e : \tau'.$$

By **Replacement (Lemma 5-4)** and **Substitution (Lemma 5-7)**, $TE' \vdash \chi + ex, \{ex/x\}e : \tau'$.

Case $\chi + ex, E[\text{raise } ex \ v] \xrightarrow{vr} \chi + ex, E[\text{[ex, v]}]$.

By rules **(τ -raise)**, **(τ -ex)** and **(τ -app)**:

$$\frac{\frac{TE' \vdash \text{raise} : (\psi \text{ exn} \rightarrow \psi \rightarrow \tau') \quad TE' \vdash ex : \psi \text{ exn}}{TE' \vdash \text{raise } ex : \psi \rightarrow \tau'} \quad TE' \vdash v : \psi}{TE' \vdash \text{raise } ex \ v : \tau'}$$

And thus $TE' \vdash \chi + ex, [\text{ex}, v] : \tau'$.

Case $\chi + ex, E[v \ \text{handle } ex \ v'] \xrightarrow{vr} \chi + ex, E[v]$.

Rules **(τ -handle)** and **(τ -app)** apply:

$$\frac{TE' \vdash v : \tau'}{TE' \vdash v \ \text{handle} : \psi \text{ exn} \rightarrow (\psi \rightarrow \tau') \rightarrow \tau'}$$

And thus $TE' \vdash \chi + ex, v : \tau'$.

Case $\chi + ex, E[\text{[ex, v] handle } ex \ v'] \xrightarrow{vr} \chi + ex, E[v' \ v]$.

Rules **(τ -handle)**, **(τ -ex)** and **(τ -app)** apply:

$$\frac{\frac{\frac{\text{TE}' \vdash [ex, v] : \tau'}{\text{TE}' \vdash [ex, v] \text{ handle } e : \psi \text{ exn} \rightarrow (\psi \rightarrow \tau') \rightarrow \tau'} \quad \text{TE}' \vdash ex : \psi \text{ exn} \quad \text{TE}' \vdash v : \psi}{\text{TE}' \vdash [ex, v] \text{ handle } ex : (\psi \rightarrow \tau') \rightarrow \tau' \quad \text{TE}' \vdash v' : \psi \rightarrow \tau'}}{\text{TE}' \vdash [ex, v] \text{ handle } ex \ v' : \tau'}$$

Thus $\frac{\text{TE}' \vdash v' : (\psi \rightarrow \tau') \quad \text{TE}' \vdash v : \psi}{\text{TE}' \vdash v' v : \tau'}$

Case $\chi + ex_1 + ex_2, E[[ex_1, v] \text{ handle } ex_2 \ v'] \xrightarrow{\text{vr}} \chi + ex_1 + ex_2, E[[ex_1, v]] \quad ex_1 \neq ex_2.$

Rules (τ -**handle**), (τ -**ex**) and (τ -**app**) apply:

$$\frac{\frac{\frac{\text{TE}' \vdash [ex_1, v] : \tau'}{\text{TE}' \vdash [ex_1, v] \text{ handle } e : \psi' \text{ exn} \rightarrow (\psi' \rightarrow \tau') \rightarrow \tau'} \quad \text{TE}' \vdash ex_2 : \psi' \text{ exn}}{\text{TE}' \vdash [ex_1, v] \text{ handle } ex_2 : (\psi' \rightarrow \tau') \rightarrow \tau' \quad \text{TE}' \vdash v' : \psi' \rightarrow \tau'}}{\text{TE}' \vdash [ex_1, v] \text{ handle } ex_2 \ v' : \tau'}$$

Thus $\text{TE}' \vdash \chi + ex_1 + ex_2, [ex_1, v] : \tau'.$

Case $E[exn \ e] \xrightarrow{\text{vr}} E[exn].$

Rule (τ -**app**) applies, but due to the way exceptions propagate, the type of the expression $exn \ e$ is also the type of the exception packet exn .

$$\frac{\text{TE}' \vdash exn : \tau' \quad \text{TE}' \vdash e : \tau}{\text{TE}' \vdash exn \ e : \tau'}$$

Cases $E[v \ exn], E[(exn \ . \ e)], E[(v \ . \ exn)], E[\text{let } x = exn \text{ in } e], E[\text{raise } exn \ e], E[\text{raise } ex \ exn], E[e \text{ handle } ex' \ exn] \xrightarrow{\text{vr}} E[exn].$

These cases all follow by similar arguments to that used in the previous case.

Cases $R[\text{ref } v] \xrightarrow{\text{vr}} R[\rho(x, v).x], \rho\theta(x, v).R[!x] \xrightarrow{\text{vr}} \rho\theta(x, v).R[v], \rho\theta(x, v_1).R[:= x \ v_2] \xrightarrow{\text{vr}} \rho\theta(x, v_2).R[v_2], R[\rho\theta_1.\rho\theta_2.e] \xrightarrow{\text{vr}} R[\rho\theta_1.\rho\theta_2.e].$ See [WF91].

Case $R[\rho\theta.e] \xrightarrow{\text{vr}} \rho\theta.R[e].$

The assumption gives $\text{TE}' \vdash R[\rho\theta.e] : \tau'$, so the way to proceed is by induction on the structure of R to show $\text{TE}' \vdash \rho\theta.R[e] : \tau'.$

Cases $R = [], (R' \ e''), (v \ R'), \text{let } x = R' \text{ in } e''.$ See [WF91].

Case $(R' \ . \ e'').$

Then $\text{TE}' \vdash (R'[\rho\theta.e] \ . \ e'') : \tau',$

and by rule (τ -**pair**), where $\tau' = \tau_1 \times \tau_2,$

$\text{TE}' \vdash \rho\theta.R'[e] : \tau_1$ and $\text{TE}' \vdash e'' : \tau_2.$

By the induction hypothesis, $\text{TE}' \vdash \rho\theta.R'[e] : \tau_1,$ and thus $\text{TE}' \vdash (\rho\theta.R'[e] \ . \ e'') : \tau'.$

Case $(v \ . \ R').$

Similar to the previous case.

Case (raise $R' e''$).

Then $TE' \vdash \text{raise } R'[\rho\theta.e] e'' : \tau$,

and by rules (τ -raise) and (τ -app),

(1) $TE' \vdash R'[\rho\theta.e] : \psi \text{ exn}$

and $TE' \vdash e'' : \psi$.

By the induction hypothesis on (1),

$TE' \vdash \rho\theta.R'[e] : \psi \text{ exn}$

And by rules (τ -raise) and (τ -app) again,

$TE' \vdash \text{raise } \rho\theta.R'[e] e'' : \tau$.

Case (raise $ex R'$).

Similar to previous case.

Case (R' handle $ex v$).

Then $TE' \vdash R'[\rho\theta.e] \text{ handle } ex v : \tau$,

and by rules (τ -handle) and (τ -app) twice,

(2) $TE' \vdash R'[\rho\theta.e] : \tau$

$TE' \vdash ex : \psi \text{ exn}$

$TE' \vdash v : \psi \rightarrow \tau$

By the induction hypothesis on (2),

$TE' \vdash \rho\theta.R'[e] : \tau$

And by rules (τ -handle) and (τ -app) again,

$TE' \vdash \rho\theta.R'[e] \text{ handle } ex v : \tau$.

Case (e'' handle $ex R'$).

Similar to previous case.

Thus the lemma is proved. ■

Proof of Lemma 5-9

Lemma 5-9 (Uniform evaluation) For closed expressions e , if no e' exists such that $e \xrightarrow{\text{vr}} e'$ and e' is faulty, then either $e \uparrow$ or $e \xrightarrow{\text{vr}}^* a$ where the answer $a = \{\rho\theta.\} v \mid \{\rho\theta.\} [ex, v]$ with $ex \in \chi$.

Proof. By induction on the length of the reduction sequence. The proof requires that one of the following holds for any expression e : e is faulty, $e \xrightarrow{\text{vr}} e'$ and e' is closed, or e is an answer a .

As in the previous lemmas, many of the cases have been established in the work of Wright and Felleisen (Lemma 4-10 proof, p. 19 [WF91]). However, several of the cases require further induction on the structure of e , and thus all cases are given in full.

Case $e = c, Y, \lambda x(e)$. Then e is an answer a .

Case $e = x$ or ex . Since e is closed, these cases cannot occur.

Case $e = \text{let } x = e_1 \text{ in } e_2$. Considering the possible cases for e_1 , the inductive hypothesis gives three situations. If e_1 is faulty, then $\text{let } x = e_1 \text{ in } e_2$ is faulty. If $e_1 = E_1[e']$ and $e' \xrightarrow{\text{vr}} e''$, then $e = E[e']$ with $E = \text{let } x = E_1 \text{ in } e_2$; hence $e \xrightarrow{\text{vr}} E[e'']$. Lastly, if e_1 is a value, then $\text{let } x = e_1 \text{ in } e_2 \xrightarrow{\text{vr}} \{e_1/x\}e_2$.

Case $e = (e_1 \ e_2)$. The inductive hypothesis gives three possible situations, depending on the nature of e_1 . If e_1 is faulty, then $(e_1 \ e_2)$ is faulty. If $e_1 = E_1[e']$ and $e' \xrightarrow{\text{vr}} e''$, then $e = E[e']$ with $E = (E_1 \ e_2)$; thus $e \xrightarrow{\text{vr}} E[e'']$. Otherwise, e_1 is a value v_1 , and then there are three subcases to consider with e_2 . If e_2 is faulty, then $(e_1 \ e_2)$ is faulty. If $e_2 = E_2[e']$ and $e' \xrightarrow{\text{vr}} e'''$, then $e = E[e']$ with $E = (v_1 \ E_2)$; thus $e \xrightarrow{\text{vr}} E[e''']$. Otherwise, e_2 is also a value, v_2 . In this instance, the possibilities can be summarised by the following:

v_2	c	$\lambda x(e)$	Y	$(v_1.v_2)$	x	ex	ref	!	$:=$	$:= x$
v_1										
c	faulty or $\xrightarrow{\text{vr}}$	faulty or $\xrightarrow{\text{vr}}$	faulty or $\xrightarrow{\text{vr}}$	faulty or $\xrightarrow{\text{vr}}$	\times	\times	faulty or $\xrightarrow{\text{vr}}$	faulty or $\xrightarrow{\text{vr}}$	faulty or $\xrightarrow{\text{vr}}$	\times
$\lambda x(e)$	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	faulty or $\xrightarrow{\text{vr}}$	\times	\times	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	\times
Y	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	faulty	\times	\times	faulty	faulty	faulty	\times
$(v_1.v_2)$	faulty	faulty	faulty	faulty	\times	\times	faulty	faulty	faulty	\times
x	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
ex	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times
ref	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	\times	\times	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	$\xrightarrow{\text{vr}}$	\times
!	faulty	faulty	faulty	faulty	faulty or $\xrightarrow{\text{vr}}$	\times	faulty	faulty	faulty	\times
$:=$	faulty	faulty	faulty	faulty	\times	\times	faulty	faulty	faulty	\times
$:= x$	\times	\times	\times	\times	\times	\times	\times	\times	\times	\times

Entries marked with a \times cannot occur because the expression e is not closed. Entries with $\xrightarrow{\text{vr}}$ indicate that the expression reduces with $E = []$.

Case $e = (e_1 \ . \ e_2)$. The inductive hypothesis gives three cases to consider. If e_1 is faulty, then $(e_1 \ . \ e_2)$ is faulty. If $e_1 = E_1[e']$ and $e' \xrightarrow{\text{vr}} e''$, then $e = E[e']$ with $E = (E_1 \ . \ e_2)$; thus $e \xrightarrow{\text{vr}} E[e'']$. Otherwise, if e_1 is a value, then there are three subcases to consider with e_2 . If e_2 is faulty, then $(e_1 \ . \ e_2)$ is faulty. If $e_2 = E_2[e']$ and $e' \xrightarrow{\text{vr}} e'''$, then $e = E[e']$ with $E = (v_1 \ . \ E_2)$; thus $e \xrightarrow{\text{vr}} E[e''']$. Otherwise, if e_2 is also a value, then $(e_1 \ . \ e_2)$ is an answer $(v_1 \ . \ v_2)$.

Case $e = \text{exception } x \text{ in } e_1$. The inductive hypothesis gives three cases to consider. If e_1 is faulty, then $\text{exception } x \text{ in } e_1$ is faulty. If $e_1 = E_1[e']$ and $e' \xrightarrow{\text{vr}} e''$, then $e = E[e']$ with $E = \text{exception } x \text{ in } E_1$; thus $e \xrightarrow{\text{vr}} E[e'']$. Otherwise, if e_1 is a value, then $\chi, \text{exception } x \text{ in } e_1 \xrightarrow{\text{vr}} \chi + ex, \{ex/x\}e_1$.

Case $e = \text{raise } e_1 e_2$. The inductive hypothesis gives three cases to consider. If e_1 is faulty, then $\text{raise } e_1 e_2$ is faulty. If $e_1 = E_1[e']$ and $e' \xrightarrow{\text{vr}} e''$, then $e = E[e']$ with $E = \text{raise } E_1 e_2$; thus $e \xrightarrow{\text{vr}} E[e'']$. Otherwise, if e_1 is a value (an exception name), then there are three subcases to consider with e_2 . If e_2 is faulty, then $\text{raise } e_1 e_2$ is faulty. If $e_2 = E_2[e']$ and $e' \xrightarrow{\text{vr}} e'''$, then $e = E[e']$ with $E = \text{raise } ex E_2$; thus $e \xrightarrow{\text{vr}} E[e''']$. Otherwise, if e_2 is also a value, then $\text{raise } e_1 e_2 \xrightarrow{\text{vr}} [e_1, e_2]$.

Case $e = e_1 \text{ handle } x e_2$. Similar to previous case.

Case $e = \text{exn}$. Then e is an unhandled exception packet, which is an answer a .

Case $e = \rho\theta.e_1$. By the induction hypothesis, there are three cases to consider. If e_1 is faulty, then $\rho\theta.e_1$ is faulty. If $e_1 = E_1[e']$ and $e' \xrightarrow{\text{vr}} e''$, then $e = E[e']$ with $E = \rho\theta.E_1$; hence $e \xrightarrow{\text{vr}} E[e'']$. Lastly, if e_1 is a value, then $\rho\theta.e_1$ is an answer a .

Thus the lemma is proved. ■

Proof of Lemma 5-10

Lemma 5-10 (Faulty expressions are untypable) If e is faulty, then there are no TE and τ such that $\text{TE} \vdash e : \tau$.

Proof. It is sufficient to show that the subexpressions e' of e that cause e to be faulty are untypable. The proof proceeds by case analysis on the form of the subexpression e' .

As in the previous lemmas, many of the cases have been established in the work of Wright and Felleisen (Lemma 4-11 proof, p. 20 and Lemma 5-6 proof, p. 30 [WF91]). These cases are indicated, and the new cases are given in full.

Case $(c \ v)$ where $\delta(c, v)$ is undefined. See [WF91].

Case $((v_1 \cdot v_2) \ v)$. Assume that $\text{TE} \vdash ((v_1 \cdot v_2) \ v) : \tau$. Then by rule $(\tau\text{-app})$, the type judgement $\text{TE} \vdash (v_1 \cdot v_2) : \tau' \rightarrow \tau$ is obtained. However, $\text{TE} \vdash v_1 : \tau_1$ and $\text{TE} \vdash v_2 : \tau_2$, which together with rule $(\tau\text{-pair})$ gives $\text{TE} \vdash (v_1 \cdot v_2) : \tau_1 \times \tau_2$, which is a contradiction.

Case $(! \ v)$ where $v \notin \text{VAR}$. See [WF91].

Case $(:= \ v)$ where $v \notin \text{VAR}$. See [WF91].

Case $\rho\theta\langle x, v_2 \rangle.C[x \ v_1]$. See [WF91].

Case exception x in $C[! \ x]$. Assume that $\text{TE} \vdash \text{exception } x \text{ in } C[! \ x] : \tau$. Then $\text{TE}' \vdash C[! \ x] : \tau$ by rule $(\tau\text{-exn})$, where $\text{TE}'(x) = \psi \text{ exn}$. Since $\text{TE}'' \vdash ! \ x : \psi'$ where $\text{TE}'(x) = \text{TE}''(x)$, $\text{TE}'' \vdash x : \psi' \text{ ref}$ by rules $(\tau\text{-app})$ and $(\tau\text{-deref})$. But this contradicts $\text{TE}''(x) = \psi \text{ exn}$.

Case exception x in $C[:= \ x]$. Similar to previous case, but uses rule $(\tau\text{-assign})$.

Case exception x in $C[x \ v]$. Similar to previous case, but uses rule $(\tau\text{-app})$.

Case $(\text{raise } v_1 v_2)$ where $v_1 \notin \text{VAR} \cup \text{EXNNAME}$. Assume that $\text{TE} \vdash \text{raise } v_1 v_2 : \tau$. Then $\text{TE} \vdash v_2 : \psi$ and $\text{TE} \vdash v_1 : \psi \text{ exn}$ by rules $(\tau\text{-app})$ twice and $(\tau\text{-raise})$. However, this implies that v_1 is either an exception name or a variable, contrary to the assumption.

Case $(e' \text{ handle } v_1 v_2)$ where $v_1 \notin \text{VAR} \cup \text{EXNNAME}$. Similar to previous case, but uses rule $(\tau\text{-handle})$.

Case $\rho\theta\langle x, v \rangle.C[\text{raise } x v']$. Assume that $\text{TE} \vdash \rho\theta\langle x, v \rangle.C[\text{raise } x v'] : \tau$. Then $\text{TE}' \vdash C[\text{raise } x v'] : \tau$ by rule $(\tau\text{-rho})$, where $\text{TE}'(x) = \psi \text{ ref}$. Since $\text{TE}'' \vdash \text{raise } x v' : \tau'$ then by rules $(\tau\text{-app})$ and $(\tau\text{-raise})$, $\text{TE}'' \vdash \text{raise } x : \psi' \rightarrow \tau'$ and $\text{TE}'' \vdash x : \psi' \text{ exn}$ where $\text{TE}'(x) = \text{TE}''(x)$. But this contradicts $\text{TE}''(x) = \psi' \text{ ref}$.

Case $\rho\theta\langle x, v \rangle.C[e \text{ handle } x v']$. Similar to previous case, but uses rule $(\tau\text{-handle})$.

Thus the lemma is proved. ■

Appendix A -2

Proofs from Chapter 7

This appendix contains proofs of the lemmas in Chapter 7 for λ_{pv} . The proofs follow the style developed in the work of Reppy [Rep92] and Wright and Felleisen [WF91]. Several of the lemmas are extensions of the equivalent proofs given in Appendix A-1.

Proof of Lemma 7-4

Lemma 7-4 If $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$ and $\sigma' \succ \sigma$, then $TE \pm \{x \mapsto \sigma'\} \vdash e : \tau$.

Proof. The proof of this lemma is by induction on the height of the typing deduction of $TE \pm \{x \mapsto \sigma\} \vdash e : \tau$, and case analysis on the shape of e for the last step. Remember that x is not bound in e by the variable convention.

As in **Lemma 5-6**, the cases which are interesting are those which possibly affect the typing environment TE 's variable component. The two new cases which must be considered are: $e = \text{proc } x' \text{ in } e'$ and $e = \text{prt } x' \text{ on } \pi \text{ in } e'$.

Case $e = \text{proc } x' \text{ in } e'$.

Rule (τ -proc) applies:

$$\frac{TE \pm \{x \mapsto \sigma, x' \mapsto \text{ProcessName}\} \vdash e' : \tau}{TE \pm \{x \mapsto \sigma\} \vdash \text{proc } x' \text{ in } e' : \tau}$$

By the induction hypothesis:

$$TE \pm \{x \mapsto \sigma', x' \mapsto \text{ProcessName}\} \vdash e' : \tau$$

And applying (τ -proc):

$$TE \pm \{x \mapsto \sigma'\} \vdash \text{proc } x' \text{ in } e' : \tau.$$

Case $e = \text{prt } x' \text{ on } \pi \text{ in } e'$.

Rule (τ -prt) applies:

$$\frac{TE \pm \{x \mapsto \sigma\} \vdash \pi : \text{ProcessName} \quad TE \pm \{x \mapsto \sigma, x' \mapsto \psi \text{ PortName}\} \vdash e' : \tau}{TE \pm \{x \mapsto \sigma\} \vdash \text{prt } x' \text{ on } \pi \text{ in } e' : \tau}$$

By the induction hypothesis:

$$TE \pm \{x \mapsto \sigma', x' \mapsto \psi \text{ PortName}\} \vdash e' : \tau \text{ and } TE \pm \{x \mapsto \sigma'\} \vdash \pi : \text{ProcessName}.$$

Thus applying ($\tau\text{-prt}$) gives:

$$\text{TE} \pm \{x \mapsto \sigma\} \vdash \text{prt } x' \text{ on } \pi \text{ in } e' : \tau.$$

Thus the lemma is proved. ■

Proof of Lemma 7-5

Lemma 7-5 (Substitution) If $x \notin \text{FV}(v)$, $\text{TE} \vdash v : \tau$, and $\text{TE} \pm \{x \mapsto \forall \alpha_1 \dots \forall \alpha_n. \sigma\} \vdash e : \tau'$, with $\{\alpha_1, \dots, \alpha_n\} \cap \text{FTV}(\text{TE}) = \emptyset$, then $\text{TE} \vdash \{v/x\}e : \tau'$.

Proof. Just as in the previous lemma, the proof of the substitution lemma is by induction on the height of the typing deduction, and case analysis on the shape of e for the last step. Let $\text{TE} = (\text{VT}, \text{ET}, \text{PT})$, $\text{VT}' = \text{VT} \pm \{x \mapsto \forall \alpha_1 \dots \forall \alpha_n. \tau\}$, and $\text{TE}' = (\text{VT}', \text{ET}, \text{PT})$ in the following cases. The new cases to consider are those for the new expressions in λ_{pv} .

Case $e = \pi$.

Rule ($\tau\text{-procvar}$) applies, thus $\text{TE} \vdash \pi : \tau'$, where $\tau' = \text{ProcessName}$. Since $\{v/x\}\pi = \pi$, rule ($\tau\text{-procvar}$) can be applied to get $\text{TE} \vdash \{v/x\}\pi : \tau'$.

Case $e = \phi$.

Rule ($\tau\text{-prtvar}$) applies, thus $\text{PT}(\phi) = \tau'$. Since $\{v/x\}\phi = \phi$, rule ($\tau\text{-prtvar}$) can be applied to get $\text{TE} \vdash \{v/x\}\phi : \tau'$.

Case $e = \text{proc } x' \text{ in } e'$.

Rule ($\tau\text{-proc}$) applies:

$$\frac{\text{TE}' \pm \{x' \mapsto \text{ProcessName}\} \vdash e' : \tau'}{\text{TE}' \vdash \text{proc } x' \text{ in } e' : \tau'}$$

By the variable convention, $x' \notin \text{FV}(v)$, so **Lemma 7-1** gives:

$$\text{TE} \pm \{x' \mapsto \text{ProcessName}\} \vdash v : \tau.$$

Thus, by the induction hypothesis and rule ($\tau\text{-proc}$):

$$\frac{\text{TE} \pm \{x' \mapsto \text{ProcessName}\} \vdash \{v/x\}e' : \tau'}{\text{TE} \vdash \text{proc } x' \text{ in } \{v/x\}e' : \tau'}$$

and therefore, $\text{TE} \vdash \{v/x\}(\text{proc } x' \text{ in } e') : \tau'$.

Case $e = \text{prt } x' \text{ on } \pi \text{ in } e'$.

Rule ($\tau\text{-prt}$) applies:

$$\frac{\text{TE}' \vdash \pi : \text{ProcessName} \quad \text{TE}' \pm \{x' \mapsto \psi \text{ PortName}\} \vdash e' : \tau'}{\text{TE}' \vdash \text{prt } x' \text{ on } \pi \text{ in } e' : \tau'}$$

By the variable convention, $x' \notin \text{FV}(v)$, so **Lemma 7-1** gives:

$$\text{TE} \vdash \{x' \mapsto \psi \text{ PortName}\} \vdash v : \tau.$$

Thus, by the induction hypothesis and rule (τ -proc):

$$\frac{\text{TE} \vdash \pi : \text{ProcessName} \quad \text{TE} \vdash \{x' \mapsto \psi \text{ PortName}\} \vdash \{v/x\}e' : \tau'}{\text{TE} \vdash \text{prt } x' \text{ on } \pi \text{ in } \{v/x\}e' : \tau'}$$

and therefore, $\text{TE} \vdash \{v/x\}(\text{prt } x' \text{ on } \pi \text{ in } e) : \tau$.

Case $e = \text{execute } e'$.

Rule (τ -execute) applies:

$$\frac{\text{TE}' \vdash \pi : \text{ProcessName} \quad \text{TE}' \vdash v' : \text{unit} \rightarrow \text{unit}}{\text{TE}' \vdash \text{execute } (\pi . v') : \text{unit}}$$

By the induction hypothesis and rule (τ -execute):

$$\frac{\text{TE} \vdash \{v/x\}\pi : \text{ProcessName} \quad \text{TE} \vdash \{v/x\}v' : \text{unit} \rightarrow \text{unit}}{\text{TE} \vdash \text{execute } (\{v/x\}\pi . \{v/x\}v') : \text{unit}}$$

and therefore, $\text{TE} \vdash \{v/x\}(\text{execute } (\pi . v')) : \tau$.

Case $e = \text{self_id } e'$.

Rule (τ -self_id) and rule (τ -app) apply, so $\text{TE}' \vdash \text{self_id } e' : \tau'$, where $\tau' = \text{ProcessName}$, for any TE' . In particular, $\text{TE} \vdash \text{self_id } e' : \tau'$, and thus $\text{TE} \vdash e' : \text{unit}$. By the induction hypothesis, $\text{TE} \vdash \{v/x\}e' : \text{unit}$. Since $\{v/x\}\text{self_id} = \text{self_id}$, $\text{TE} \vdash \{v/x\}(\text{self_id } e') : \tau$.

Case $e = \text{send } e'$.

Rule (τ -send) applies:

$$\frac{\text{TE}' \vdash \phi : \psi \text{ PortName} \quad \text{TE}' \vdash v' : \psi}{\text{TE}' \vdash \text{send } (\phi . v') : \text{unit}} \text{ with } e' = (\phi . v') \text{ and } \tau' = \text{unit}.$$

By the induction hypothesis and rule (τ -send):

$$\frac{\text{TE} \vdash \{v/x\}\phi : \psi \text{ PortName} \quad \text{TE} \vdash \{v/x\}v' : \psi}{\text{TE} \vdash \text{send } (\{v/x\}\phi . \{v/x\}v') : \text{unit}}$$

and therefore, $\text{TE} \vdash \{v/x\}(\text{send } (\phi . v')) : \tau$.

Case $e = \text{recv } e'$.

Rule (τ -recv) applies:

$$\frac{\text{TE}' \vdash \phi : \psi \text{ PortName}}{\text{TE}' \vdash \text{recv } \phi : \psi} \text{ with } e' = \phi \text{ and } \tau' = \psi.$$

By the induction hypothesis and rule (τ -recv):

$$\frac{TE \vdash \{v/x\}\phi : \psi \text{ PortName}}{TE \vdash \text{recv } \{v/x\}\phi : \psi}$$

and therefore, $TE \vdash \{v/x\}(\text{recv } \phi) : \tau$.

Case $e = \text{probe } e'$. Similar to previous case.

Thus the lemma is proved. ■

Proof of Theorem 7-6

Theorem 7-6 (Sequential type preservation) For any type environment TE , expression e_1 , and type τ , such that $TE \vdash e_1 : \tau$, if $e_1 \longrightarrow e_2$ then $TE \vdash e_2 : \tau$.

Proof. Let $E[e] = e_1$ and $E[e'] = e_2$, and assume that $TE' \vdash e : \tau'$ with $TE' = (VT', ET', PT')$. Then by the **Replacement Lemma** (Lemma 7-2), it is sufficient to show that $TE' \vdash e' : \tau'$. This is done by case analysis of the definition of \longrightarrow , effectively on the structure of e .

In fact, since the sequential aspects of λ_{pv} are almost identical to those of λ_{ev} , there is very little work to do. The case for the exception term reduction is no longer required since this is now part of the parallel evaluation relation. All cases other than that for the reduction $R[\rho\theta.e] \longrightarrow \rho\theta.R[e]$ are identical to those given in **Lemma 5-8**. The proof for the λ_{pv} - ρ_{lift} case is dependent on the structure of R . As R has been extended, this requires some new subcases.

Case $R[\rho\theta.e] \longrightarrow \rho\theta.R[e]$.

The assumption gives $TE' \vdash R[\rho\theta.e] : \tau'$, so the way to proceed is by induction on the structure of R to show $TE' \vdash \rho\theta.R[e] : \tau'$.

Cases $R = []$, $(R' \ e'')$, $(v \ R')$, $\text{let } x = R' \text{ in } e''$, $(R' . e'')$, $(v . R')$, $(\text{raise } R' \ e'')$, $(\text{raise } ex \ R')$, $(R' \ \text{handle } ex \ v)$, and $(e'' \ \text{handle } ex \ R')$. See proof of **Lemma 5-8**.

Case **execute** R' .

Then $TE' \vdash \text{execute } R'[\rho\theta.e] : \tau'$, where $\tau' = \text{unit}$

and by rule (τ -execute) and (τ -pair),

(1) $TE' \vdash R'[\rho\theta.e] : \text{ProcessName} \times \text{unit} \rightarrow \text{unit}$.

By the induction hypothesis on (1),

$TE' \vdash \rho\theta.R'[e] : \text{ProcessName} \times \text{unit} \rightarrow \text{unit}$.

And by rule (τ -execute) and (τ -pair) again,

$TE' \vdash \text{execute } \rho\theta.R'[e] : \tau'$.

Case **self_id** R' .

Then $TE' \vdash \text{self_id } R'[\rho\theta.e] : \tau'$, where $\tau' = \text{ProcessName}$

and by rule (τ -self_id) and (τ -app),

$$(2) \quad TE' \vdash R'[\rho\theta.e] : unit.$$

By the induction hypothesis on (2),

$$TE' \vdash \rho\theta.R'[e] : unit.$$

And by rule (τ -self_id) and (τ -app) again,

$$TE' \vdash self_id \rho\theta.R'[e] : \tau'.$$

Case send R' .

$$\text{Then } TE' \vdash send R'[\rho\theta.e] : \tau', \text{ where } \tau' = unit$$

and by rule (τ -send) and (τ -pair),

$$(3) \quad TE' \vdash R'[\rho\theta.e] : \psi PortName \times \psi.$$

By the induction hypothesis on (3),

$$TE' \vdash \rho\theta.R'[e] : \psi PortName \times \psi.$$

And by rule (τ -send) and (τ -pair) again,

$$TE' \vdash send \rho\theta.R'[e] : \tau'.$$

Case recv R' .

$$\text{Then } TE' \vdash recv R'[\rho\theta.e] : \tau'$$

and by rule (τ -recv) and (τ -app),

$$(4) \quad TE' \vdash R'[\rho\theta.e] : \tau' PortName.$$

By the induction hypothesis on (4),

$$TE' \vdash \rho\theta.R'[e] : \tau' PortName.$$

And by rule (τ -recv) and (τ -app) again,

$$TE' \vdash recv \rho\theta.R'[e] : \tau'.$$

Case probe R' . Similar to previous case.

Thus the theorem is proved. ■

Proof of Theorem 7-7

The second type preservation theorem states that parallel evaluation preserves configuration typing. Before setting out to prove this theorem, two subsidiary lemmas are required. These lemmas are required in the cases involving copying of values between different processes.

The first lemma states that if there exists a closed expression consisting of a context E and a value v , the expression formed by the **copy** operation is also closed. This is effectively a refinement of **Lemma 6-1**. Recall that **copy** is defined in terms of **mem**, which produces a set of memory cells.

The formal definition of **copy** is $\text{copy}(E, v) \equiv \rho(\text{mem}(E, v)).v$

Lemma A-2-1. If $E[v]$ is closed then $\text{copy}(E, v)$ is also closed.

Proof. The proof proceeds by induction on the structure of v and E , according to the definitions of **copy** and **mem**.

Case $v = c, Y, ex, \pi, \phi, \text{ref}, !, :=$. In each case, v is closed and $\text{mem}(E, v) = \emptyset$, hence $\text{copy}(E, v)$ is also closed.

Case $v = x$. By **Lemma 6-1**, if x is free, then either x is also free in $E[x]$, which contradicts the assumption, or $E = \rho\theta\langle x, v' \rangle.E'$.

Then $\text{copy}(E, x) = \rho((\text{mem}(\rho\theta.E', v')) \cup \{\langle x, v' \rangle\}).x$.

By the induction hypothesis, if $\rho\theta.E'[v']$ is closed, then $\rho(\text{mem}(\rho\theta.E', v')).v'$ is also closed. Hence $\rho((\text{mem}(\rho\theta.E', v')) \cup \{\langle x, v' \rangle\}).x = \text{copy}(E, x)$ is closed.

If $\rho\theta.E'[v']$ is not closed, then it must be because there exists a $v_i \in \text{rng}(\theta)$ with x part of v_i (e.g. $v_i = x, v_i = (x . v_i)$) such that $\rho\theta\langle x, v' \rangle.E'[v']$ is closed as otherwise **Lemma 6-1** would not hold. (The situation is dealt with in the recursive definition of **mem** by the case where the evaluation context E does not contain a ρ -bound memory cell $\langle x, v' \rangle$.) Hence $\rho((\text{mem}(\rho\theta.E', v')) \cup \{\langle x, v' \rangle\}).v$ is closed.

Thus **copy** (E, v) is closed when $v = x$.

Case $v = := x$. Follows by similar arguments to the previous case.

Case $v = \lambda x(e)$. By **Lemma 6-1**, if x' is free in $\lambda x(e)$, then either x' is also free in $E[\lambda x(e)]$, which contradicts the assumption, or $E = \rho\theta\langle x', v' \rangle.E'$. Using similar arguments to the previous cases, **copy** (E, x') is closed.

Since $\lambda x(e)$ may contain several free variables x'_i , such that i ranges from $1 \dots |\text{FV}(\lambda x(e))|$, there may be several sets of memory cells θ_i produced by the **mem** operation. The generalised union operation in the definition of **mem_gc** ensures that duplicates are eliminated when combining these sets together to provide a single set of memory cells as the result of $\text{mem}(E, \lambda x(e))$. The consequence is that

$$\rho\left(\bigcup_{x' \in \text{FV}(\lambda x(e))} \text{mem}(E, x')\right).\lambda x(e) = \text{copy}(E, \lambda x(e))$$

is closed, since any free variables from the $\lambda x(e)$ term are now bound in the ρ -expression.

Case $v = (v_1 . v_2)$. By the induction hypothesis, if $E[v_1]$ is closed then $\rho(\text{mem}(E, v_1)).v_1$ is closed. Similarly for v_2 .

Hence, since $\rho(\text{mem}(E, v_1)).v_1$ and $\rho(\text{mem}(E, v_2)).v_2$ are closed, $\text{copy}(E, (v_1 . v_2)) = \rho((\text{mem}(E, v_1)) \cup (\text{mem}(E, v_2))).(v_1 . v_2) =$ is also closed.

Thus the lemma is proved. ■

The next lemma states that the type of the ρ -bound expression produced by the **copy** operation from a context and value is the same as the type of the value alone.

Lemma A-2-2. If $TE' \vdash E[v] : \tau$ and $E[v]$ is closed, then if there is some TE such that $TE \vdash v : \tau$, then $TE \vdash \text{copy}(E, v) : \tau$.

Proof. The proof proceeds by induction on the structure of v using the rule (τ -rho).

Case $v = c, Y, ex, \pi, \phi, \text{ref}, !, :=$. Since in each case v is closed, $\text{mem}(E, v) = \emptyset$, and thus if $TE \vdash v : \tau$, then $TE \vdash \rho\emptyset.v : \tau$ by rule (τ -rho).

Case $v = x, \lambda x(e), (v_1 . v_2), := x$. By Lemma A-2-1, since $E[v]$ is closed, $\rho(\text{mem}(E, v)).v$ is closed also. But Lemma 7-2 states that variables in the domain of the type environment TE which are not free in v can be ignored when typing v .

Thus, since $TE \vdash v : \tau$, and $\rho(\text{mem}(E, v)).v$ is closed, $TE \vdash \rho(\text{mem}(E, v)).v : \tau$ by rule (τ -rho); in other words, $TE \vdash \text{copy}(E, v) : \tau$.

Thus the lemma is proved. ■

The Parallel type preservation theorem can now be proved.

Theorem 7-7 (Parallel type preservation) If a configuration χ, \mathcal{P} is well-formed with

$$\chi, \mathcal{P} \Rightarrow \chi', \mathcal{P}'$$

and, for some exception name typing ET and port name typing PT,

$$ET, PT \vdash \chi, \mathcal{P} : CT$$

Then there is an exception name typing ET', port name typing PT', and configuration typing CT', such that the following hold:

- $ET \subseteq ET'$,
- $PT \subseteq PT'$,
- $CT \subseteq CT'$,
- $ET', PT' \vdash \chi', \mathcal{P}' : CT'$, and
- $ET', PT' \vdash \chi, \mathcal{P} : CT'$

Proof. The final property follows from the others. The proof of the first four properties proceeds by case analysis of the left hand side of the parallel evaluation relation \Rightarrow .

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; e \rangle : CT$

If $e \longrightarrow e'$, then, by Sequential type preservation (Theorem 7-7),

$$(\{\}, ET, PT) \vdash e' : CT(\pi)$$

and hence

$$ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; e' \rangle : CT$$

Letting $ET' = ET$, $PT' = PT$, and $CT' = CT$ satisfies the theorem.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{exception } x \text{ in } e] \rangle : CT$

Then there is a type environment $TE = (VT, ET, PT)$ and types τ and ψ , such that

$$\frac{\begin{array}{c} \vdots \\ TE \pm \{x \mapsto \psi \text{ exn}\} \vdash e : \tau \\ \hline TE \vdash \text{exception } x \text{ in } e : \tau \\ \vdots \end{array}}{(\{\}, ET, PT) \vdash E[\text{exception } x \text{ in } e] : CT(\pi)}$$

Let ex be the name of the new exception (hence $ex \notin \chi$) and define $ET' = ET \pm \{ex \mapsto \psi \text{ exn}\}$ (obviously $ET \subseteq ET'$). Then by **Lemma 7-1**,

$$(\{\}, ET', PT) \vdash E[\text{exception } x \text{ in } e] : CT(\pi)$$

Thus, by the **Replacement (7-2)** and **Substitution (7-5)** lemmas,

$$(\{\}, ET', PT) \vdash E[\{ex/x\}e] : CT(\pi)$$

and therefore, $ET', PT \vdash \chi + ex, \mathcal{P} + \langle \pi; \Phi; E[\{ex/x\}e] \rangle : CT$. Letting $PT' = PT$, and $CT' = CT$ satisfies the theorem.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{proc } x \text{ in } e] \rangle : CT$

Then there is a type environment $TE = (VT, ET, PT)$ and type τ , such that

$$\frac{\begin{array}{c} \vdots \\ TE \pm \{x \mapsto \text{ProcessName}\} \vdash e : \tau \\ \hline TE \vdash \text{proc } x \text{ in } e : \tau \\ \vdots \end{array}}{(\{\}, ET, PT) \vdash E[\text{proc } x \text{ in } e] : CT(\pi)}$$

Let π' be the identifying name of the new process (hence $\pi' \notin \text{PROCNAME}(\mathcal{P})$). Trivially by rule $(\tau\text{-procvar})$, $ET, PT \vdash \pi' : \text{ProcessName}$. Then by **Lemma 7-1**,

$$(\{\}, ET, PT) \vdash E[\text{proc } x \text{ in } e] : CT(\pi)$$

Thus, by the **Replacement (7-2)** and **Substitution (7-5)** lemmas,

$$(\{\}, ET, PT) \vdash E[\{\pi'/x\}e] : CT(\pi)$$

and therefore, $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\{\pi'/x\}e] \rangle + \langle \pi'; \emptyset; \varepsilon \rangle : CT \pm \{\pi' \mapsto \text{unit}\}$, (recall that the type of ε is defined to be *unit*). Letting $ET' = ET$, $PT' = PT$, and $CT' = CT \pm \{\pi' \mapsto \text{unit}\}$ satisfies the theorem.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{self_id } ()] \rangle : CT$

Then there is a type environment $TE = (VT, ET, PT)$ such that

$$\begin{array}{c}
\vdots \\
\hline
TE \vdash \pi : ProcessName \\
\vdots \\
TE \vdash self_id : unit \rightarrow ProcessName \quad TE \vdash () : unit \\
\hline
TE \vdash self_id () : ProcessName \\
\vdots \\
(\{\}, ET, PT) \vdash E[self_id ()] : CT(\pi)
\end{array}$$

Thus by the **Replacement lemma** (7-2),

$$(\{\}, ET, PT) \vdash E[\pi] : CT(\pi)$$

Letting $ET' = ET$, $PT' = PT$, and $CT' = CT$ satisfies the theorem.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{prt } x \text{ on } \pi \text{ in } e] \rangle : CT$

Then there is a type environment $TE = (VT, ET, PT)$ and types τ and ψ , such that

$$\begin{array}{c}
\vdots \\
\hline
TE \vdash \pi : ProcessName \\
\vdots \\
TE \pm \{x \mapsto \psi PortName\} \vdash e : \tau \\
\hline
TE \vdash \text{prt } x \text{ on } \pi \text{ in } e : \tau \\
\vdots \\
(\{\}, ET, PT) \vdash E[\text{prt } x \text{ on } \pi \text{ in } e] : CT(\pi)
\end{array}$$

Let ϕ' be the identifying name of the new port (hence $\phi' \notin \text{PORTN}(\mathcal{P}) \cup \text{dom}(\Phi)$) and define $PT' = PT \pm \{\phi' \mapsto \psi PortName\}$ (obviously $PT \subseteq PT'$). Then by **Lemma 7-1**,

$$(\{\}, ET, PT') \vdash E[\text{prt } x \text{ on } \pi \text{ in } e] : CT(\pi)$$

Thus, by the **Replacement** (7-2) and **Substitution** (7-5) lemmas,

$$(\{\}, ET, PT') \vdash E[\{\phi' / x\}e] : CT(\pi)$$

and therefore, $ET, PT' \vdash \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto \perp); E[\{\phi' / x\}e] \rangle : CT$. Letting $ET' = ET$ and $CT' = CT$ satisfies the theorem.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{prt } x \text{ on } \pi' \text{ in } e] \rangle + \langle \pi'; \Phi'; es \rangle : CT$

Then there is a type environment $TE = (VT, ET, PT)$ and types τ and ψ , such that

$$\begin{array}{c}
\vdots \\
\hline
TE \vdash \pi : ProcessName \\
\vdots \\
\hline
TE \vdash \pi' : ProcessName
\end{array}$$

$$\begin{array}{c}
\vdots \\
\frac{\text{TE} \pm \{x \mapsto \psi \text{ PortName}\} \vdash e : \tau}{\text{TE} \vdash \text{prt } x \text{ on } \pi' \text{ in } e : \tau} \\
\vdots \\
(\{\}, \text{ET}, \text{PT}) \vdash E[\text{prt } x \text{ on } \pi \text{ in } e] : \text{CT}(\pi)
\end{array}$$

Let ϕ'' be the identifying name of the new port (hence $\phi'' \notin \text{PORTN}(\mathcal{P}) \cup \text{dom}(\Phi \cup \Phi')$) and define $\text{PT}' = \text{PT} \pm \{\phi'' \mapsto \psi \text{ PortName}\}$ (obviously $\text{PT} \subseteq \text{PT}'$). Then by **Lemma 7-1**,

$$(\{\}, \text{ET}, \text{PT}') \vdash E[\text{prt } x \text{ on } \pi' \text{ in } e] : \text{CT}(\pi)$$

Thus, by the **Replacement (7-2)** and **Substitution (7-5)** lemmas,

$$(\{\}, \text{ET}, \text{PT}') \vdash E[\{\phi'' / x\}e] : \text{CT}(\pi)$$

and hence, $\text{ET}, \text{PT}' \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\{\phi'' / x\}e] \rangle + \langle \pi'; \Phi' + (\phi'' \mapsto \perp); es \rangle : \text{CT}$. Letting $\text{ET}' = \text{ET}$ and $\text{CT}' = \text{CT}$ satisfies the theorem.

Case $\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{execute } (\pi'. v)] \rangle + \langle \pi'; \Phi'; \varepsilon \rangle : \text{CT}$

Then there is a type environment $\text{TE} = (\text{VT}, \text{ET}, \text{PT})$, such that

$$\begin{array}{c}
\vdots \\
\frac{\text{TE} \vdash \pi' : \text{ProcessName}}{\vdots} \\
\frac{\text{TE} \vdash v : \text{unit} \rightarrow \text{unit}}{\text{TE} \vdash \text{execute } (\pi'. v) : \text{unit}} \\
\vdots \\
(\{\}, \text{ET}, \text{PT}) \vdash E[\text{execute } (\pi'. v)] : \text{CT}(\pi)
\end{array}$$

Then by the **Replacement lemma**,

$$(2) \quad (\{\}, \text{ET}, \text{PT}) \vdash E[()] : \text{CT}(\pi)$$

By **Lemma A-2-2** from (1),

$$(\text{VT}, \text{ET}, \text{PT}) \vdash \text{copy } (E, v) : \text{unit} \rightarrow \text{unit}$$

and by **Lemma A-2-1**, $\text{copy } (E, v)$ is closed, which means by **Lemma 6-1**

$$(\{\}, \text{ET}, \text{PT}) \vdash \text{copy } (E, v) : \text{unit} \rightarrow \text{unit}$$

Since $\text{CT}(\pi') = \text{unit}$ (by virtue of the execution state ε of π'), by applying rule ($\tau\text{-app}$),

$$(3) \quad (\{\}, \text{ET}, \text{PT}) \vdash [(\text{copy } (E, v)) ()] : \text{CT}(\pi')$$

Hence, from (2) and (3),

$$\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[()] \rangle + \langle \pi'; \Phi'; [(\text{copy } (E, v)) ()] \rangle : \text{CT}$$

Letting $\text{ET}' = \text{ET}$, $\text{PT}' = \text{PT}$, and $\text{CT}' = \text{CT}$ satisfies the theorem.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{execute } (\pi'.v)] \rangle : CT$

This case deals with the failure conditions, where it is not possible to request a process to execute. Then there is a type environment $TE = (VT, ET, PT)$, such that

$$\begin{array}{c}
 \vdots \\
 \hline
 TE \vdash \pi' : \text{ProcessName} \\
 \vdots \\
 TE \vdash v : \text{unit} \rightarrow \text{unit} \\
 \hline
 TE \vdash \text{execute } (\pi'.v) : \text{unit} \\
 \vdots \\
 (4) \quad (\{\}, ET, PT) \vdash E[\text{execute } (\pi'.v)] : CT(\pi)
 \end{array}$$

Both exception cases are considered to avoid repetition of the earlier part. The embedding of the exception names used in the surrounding context for a λ_{pv} program means that both the following exist:

$$\begin{array}{c}
 \vdots \\
 ET(\text{NoSuchProcess}) = \text{ProcessName } \text{exn} \\
 \hline
 TE \vdash \text{NoSuchProcess} : \text{ProcessName } \text{exn}
 \end{array}$$

and

$$\begin{array}{c}
 \vdots \\
 ET(\text{ProcessExecuting}) = \text{ProcessName } \text{exn} \\
 \hline
 TE \vdash \text{ProcessExecuting} : \text{ProcessName } \text{exn}
 \end{array}$$

Then by rules (τ -raise) and (τ -app) twice,

$$(5) \quad (\{\}, ET, PT) \vdash \text{raise } \text{NoSuchProcess } \pi' : \text{unit}$$

$$(6) \quad (\{\}, ET, PT) \vdash \text{raise } \text{ProcessExecuting } \pi' : \text{unit}$$

By the **Replacement lemma** on (4) with (5) and (6)

$$(\{\}, ET, PT) \vdash E[\text{raise } \text{NoSuchProcess } \pi'] : CT(\pi)$$

$$(\{\}, ET, PT) \vdash E[\text{raise } \text{ProcessExecuting } \pi'] : CT(\pi)$$

and thus either

$$ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise } \text{NoSuchProcess } \pi'] \rangle : CT \quad \text{or}$$

$$ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise } \text{ProcessExecuting } \pi'] \rangle : CT$$

Letting $ET' = ET$, $PT' = PT$, and $CT' = CT$ satisfies the theorem.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto q'); E[\text{send } (\phi'.v)] \rangle : CT$

Then there is a type environment $TE = (VT, ET, PT)$ and type ψ , such that

$$\begin{array}{c}
\vdots \\
\frac{PT(\phi') = \psi \text{ PortName}}{TE \vdash \phi' : \psi \text{ PortName}} \\
\vdots \\
\frac{TE \vdash v : \psi}{TE \vdash \text{send}(\phi' . v) : \text{unit}} \\
\vdots \\
(\{\}, ET, PT) \vdash E[\text{send}(\phi' . v)] : CT(\pi)
\end{array}$$

Thus by the **Replacement lemma**,

$$(\{\}, ET, PT) \vdash E[()] : CT(\pi)$$

and therefore $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto q' @ \text{copy}(E, v); E[()]) \rangle : CT$. Letting $ET' = ET$, $PT' = PT$, and $CT' = CT$ satisfies the theorem.

Note that since $TE \vdash \phi' : \psi \text{ PortName}$, q' is of type $\psi \text{ queue}$. By **Lemma A-2-2**, from $TE \vdash v : \psi$,

$$(VT, ET, PT) \vdash \text{copy}(E, v) : \psi$$

and by **Lemma A-2-1**, $\text{copy}(E, v)$ is closed, which means by **Lemma 6-1** that

$$(\{\}, ET, PT) \vdash \text{copy}(E, v) : \psi$$

Thus the arguments q' and $\text{copy}(E, v)$ are of the correct type ($\psi \text{ queue}$ and ψ) for the queue append operation $@$.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{send}(\phi'' . v)] \rangle + \langle \pi'; \Phi' + (\phi'' \mapsto q'') @ \text{copy}(E, v); es \rangle : CT$

Then there is a type environment $TE = (VT, ET, PT)$ and type ψ , such that

$$\begin{array}{c}
\vdots \\
\frac{PT(\phi'') = \psi \text{ PortName}}{TE \vdash \phi'' : \psi \text{ PortName}} \\
\vdots \\
\frac{TE \vdash v : \psi}{TE \vdash \text{send}(\phi'' . v) : \text{unit}} \\
\vdots \\
(\{\}, ET, PT) \vdash E[\text{send}(\phi'' . v)] : CT(\pi)
\end{array}$$

Thus by the **Replacement lemma**,

$$(\{\}, ET, PT) \vdash E[()] : CT(\pi)$$

and thus $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[()] \rangle + \langle \pi'; \Phi' + (\phi'' \mapsto q'') @ \text{copy}(E, v); es \rangle : CT$. Letting $ET' = ET$, $PT' = PT$, and $CT' = CT$ satisfies the theorem.

Note that since $TE \vdash \phi'' : \psi \text{ PortName}$, q'' is of type $\psi \text{ queue}$. By **Lemma A-2-2**, from $TE \vdash v : \psi$,

$$(VT, ET, PT) \vdash \text{copy}(E, v) : \psi$$

and by **Lemma A-2-1**, $\text{copy}(E, v)$ is closed, which means by **Lemma 6-1** that

$$(\{\}, \text{ET}, \text{PT}) \vdash \text{copy}(E, v) : \psi$$

Thus the arguments q'' and $\text{copy}(E, v)$ are of the correct type ($\psi \text{ queue}$ and ψ) for the queue append operation $@$.

Case $\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{send}(\phi'.v)] \rangle : \text{CT}$

This case deals with the failure situation, when the owning process has terminated, as given in λ_{pv} -**send3**. Then there is a type environment $\text{TE} = (\text{VT}, \text{ET}, \text{PT})$ and type ψ , such that

$$(7) \quad \begin{array}{c} \vdots \\ \text{PT}(\phi') = \psi \text{ PortName} \\ \hline \text{TE} \vdash \phi' : \psi \text{ PortName} \\ \vdots \\ \text{TE} \vdash v : \psi \\ \hline \text{TE} \vdash \text{send}(\phi'.v) : \text{unit} \\ \vdots \\ (\{\}, \text{ET}, \text{PT}) \vdash E[\text{send}(\phi'.v)] : \text{CT}(\pi) \end{array}$$

The embedding of the failure exception names used in the surrounding context for a λ_{pv} program means that the following exists:

$$\begin{array}{c} \vdots \\ \text{ET}(\text{NoSuchPort}) = \text{unit exn} \\ \hline \text{TE} \vdash \text{NoSuchPort} : \text{unit exn} \end{array}$$

Then by rules (τ -**raise**) and (τ -**app**) twice,

$$(\{\}, \text{ET}, \text{PT}) \vdash \text{raise NoSuchPort}() : \text{unit}$$

By the **Replacement lemma** on (6)

$$(\{\}, \text{ET}, \text{PT}) \vdash E[\text{raise NoSuchPort}()] : \text{CT}(\pi)$$

and thus $\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise NoSuchPort}()] \rangle : \text{CT}$. Letting $\text{ET}' = \text{ET}$, $\text{PT}' = \text{PT}$, and $\text{CT}' = \text{CT}$ satisfies the theorem.

Case $\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto [m_1, m_2, \dots, m_M]); E[\text{recv} \phi'] \rangle : \text{CT}$

Then there is a type environment $\text{TE} = (\text{VT}, \text{ET}, \text{PT})$ and type ψ , such that

$$\begin{array}{c} \vdots \\ \text{PT}(\phi') = \psi \text{ PortName} \\ \hline \text{TE} \vdash \phi' : \psi \text{ PortName} \\ \vdots \\ \text{TE} \vdash \text{recv} \phi' : \psi \\ \vdots \\ (\{\}, \text{ET}, \text{PT}) \vdash E[\text{recv} \phi'] : \text{CT}(\pi) \end{array}$$

As established in the cases for `send`, if ϕ' has type $\psi \text{ PortName}$ then $\lceil m_1, m_2, \dots, m_M \rceil$ is a queue of closed expressions of the form $\rho\theta.v$ with type ψ . Thus

$$(\{\}, \text{ET}, \text{PT}) \vdash m_1 : \psi$$

By the **Replacement lemma**,

$$(\{\}, \text{ET}, \text{PT}) \vdash E[m_1] : \text{CT}(\pi)$$

and thus $\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[m_1] \rangle : \text{CT}$. Letting $\text{ET}' = \text{ET}$, $\text{PT}' = \text{PT}$, and $\text{CT}' = \text{CT}$ satisfies the theorem.

Case $\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{recv } \phi'] \rangle : \text{CT}$

This case deals with the failure situation, when the port is not owned by the process, as given in $\lambda_{pv}\text{-recv2}$. Then there is a type environment $\text{TE} = (\text{VT}, \text{ET}, \text{PT})$ and type ψ , such that

$$\frac{\begin{array}{c} \vdots \\ \text{PT}(\phi') = \psi \text{ PortName} \end{array}}{\text{TE} \vdash \phi' : \psi \text{ PortName}}$$

$$\frac{\begin{array}{c} \vdots \\ \text{TE} \vdash \text{recv } \phi' : \psi \end{array}}{\vdots}$$

$$(8) \quad (\{\}, \text{ET}, \text{PT}) \vdash E[\text{recv } \phi'] : \text{CT}(\pi)$$

The embedding of the failure exception names used in the surrounding context for a λ_{pv} program means that the following exists:

$$\frac{\begin{array}{c} \vdots \\ \text{ET}(\text{PortNotOwned}) = \text{unit exn} \end{array}}{\text{TE} \vdash \text{PortNotOwned} : \text{unit exn}}$$

Then by rules $(\tau\text{-raise})$ and $(\tau\text{-app})$ twice,

$$(\{\}, \text{ET}, \text{PT}) \vdash \text{raise PortNotOwned } () : \psi$$

By the **Replacement lemma** on (8)

$$(\{\}, \text{ET}, \text{PT}) \vdash E[\text{raise PortNotOwned } ()] : \text{CT}(\pi)$$

and thus $\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{raise PortNotOwned } ()] \rangle : \text{CT}$. Letting $\text{ET}' = \text{ET}$, $\text{PT}' = \text{PT}$, and $\text{CT}' = \text{CT}$ satisfies the theorem.

Case $\text{ET}, \text{PT} \vdash \chi, \mathcal{P} + \langle \pi; \Phi + (\phi' \mapsto q'); E[\text{probe } \phi'] \rangle : \text{CT}$

This case deals with the situation where the probe operation returns either `true` or `false`, depending on the contents of the port's queue. These correspond to the parallel evaluation rules $\lambda_{pv}\text{-probe1}$ and $\lambda_{pv}\text{-probe2}$. So there is a type environment $\text{TE} = (\text{VT}, \text{ET}, \text{PT})$, a type ψ , and applying $(\tau\text{-app})$, such that

$$\begin{array}{c}
\vdots \\
\frac{PT(\phi') = \psi \text{ PortName}}{TE \vdash \phi' : \psi \text{ PortName}} \\
\vdots \\
TE \vdash \text{probe } \phi' : \text{bool} \\
\vdots \\
(\{\}, ET, PT) \vdash E[\text{probe } \phi'] : CT(\pi)
\end{array}$$

Since by definition of the constants `true` and `false`, their typing according to rule ($\tau\text{-const}$) is: $TE \vdash \text{true} : \text{bool}$ and $TE \vdash \text{false} : \text{bool}$.

Thus by the **Replacement lemma**, either

$$(\{\}, ET, PT) \vdash E[\text{true}] : CT(\pi) \text{ or}$$

$$(\{\}, ET, PT) \vdash E[\text{false}] : CT(\pi)$$

Hence,

$$ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{true}] \rangle : CT \text{ and}$$

$$ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{false}] \rangle : CT$$

Letting $ET' = ET$, $PT' = PT$, and $CT' = CT$ satisfies the theorem.

Case $ET, PT \vdash \chi, \mathcal{P} + \langle \pi; \Phi; E[\text{probe } \phi'] \rangle : CT$

This follows by similar reasoning to that for $\lambda_{pv}\text{-recv2}$.

Thus the theorem is proved. ■

Proof of Lemma 7-10

Lemma 7-10 A process π with process state $\langle \pi; \Phi; e \rangle$ and e closed, is stuck iff e has one of the following forms:

- | | | |
|-----|-----------------------------------|---|
| 1. | $E[c \ v]$ | where $\delta(c, v)$ is undefined |
| 2. | $E[v \ v']$ | where v has the form $(v_1 . v_2)$, ex , π , or ϕ |
| 3. | $E[! \ v]$ | where $v \notin \text{VAR}$ |
| 4. | $E[: = v]$ | where $v \notin \text{VAR}$ |
| 5. | $\rho\theta(x, v_2).E[x \ v_1]$ | |
| 6. | exception x in $E[! \ x]$ | |
| 7. | exception x in $E[: = x]$ | |
| 8. | exception x in $E[x \ v]$ | |
| 9. | $E[\text{raise } v_1]$ | where $v_1 \notin \text{VAR} \cup \text{EXNNAME}$ |
| 10. | $E[e \ \text{handle } v_1 \ v_2]$ | where $v_1 \notin \text{VAR} \cup \text{EXNNAME}$ |

- | | | |
|-----|--|--|
| 11. | $\rho\theta\langle x, v \rangle. E[\text{raise } x \ v']$ | |
| 12. | $\rho\theta\langle x, v \rangle. E[e' \text{ handle } x \ v']$ | |
| 13. | $E[\text{prt } x \text{ on } v \text{ in } e']$ | where $v \notin \text{VAR} \cup \text{PROCESSNAME}$ |
| 14. | $E[\text{execute } (v_1 . v_2)]$ | where $v_1 \notin \text{VAR} \cup \text{PROCESSNAME}$ or v_2 has the form $Y, (v_1 . v_2), \text{ref}, !, :=, := x, ex, \pi$, or ϕ |
| 15. | $E[\text{send } (v_1 . v_2)]$ | where $v_1 \notin \text{VAR} \cup \text{PORTNAME}$ |
| 16. | $E[\text{recv } v]$ | where $v \notin \text{VAR} \cup \text{PORTNAME}$ |
| 17. | $E[\text{probe } v]$ | where $v \notin \text{VAR} \cup \text{PORTNAME}$ |
| 18. | $\rho\theta\langle x, v \rangle. E[\text{proc } x \text{ in } e']$ | |
| 19. | $\rho\theta\langle x, v' \rangle. E[\text{prt } x \text{ on } v \text{ in } e']$ | |
| 20. | $\rho\theta\langle x, v \rangle. E[\text{execute } (x . v_2)]$ | |
| 21. | $\rho\theta\langle x, v \rangle. E[\text{send } (x . v_2)]$ | |
| 22. | $\rho\theta\langle x, v' \rangle. E[\text{recv } x]$ | |
| 23. | $\rho\theta\langle x, v' \rangle. E[\text{probe } x]$ | |

Proof.

(if) Let $E[e'] = e$, then *if* component of the *iff* is proved by case analysis on the possible forms of e' .

Case $e' = v$.

Then $E[e'] = E[v]$, and thus π is not stuck.

Case $e' = v \ v'$.

This proceeds by analysis of the form of v :

Case $v = c$.

If $\delta(c \ v)$ is defined, then π is not stuck, otherwise it is stuck and has form 1.

Case $v = x$.

By Lemma 6-1, either e is not closed, which is a contradiction, or E is of the form $\rho\theta\langle x, v' \rangle. E'$, and π is stuck and has form 5.

Case $v = Y, \text{ref}, !, :=, := x$

In these cases, π is not stuck.

Case $v = (v_1 . v_2), ex, \pi$, or ϕ

In these cases, π is stuck and has form 2.

Case $e' = \text{let } x = v \text{ in } e''$.

In this case, π is not stuck.

Case $e' = \text{exception } x \text{ in } e''$.

Letting $e'' = E[e''']$, this case proceeds by case analysis on the form of e''' .

Case $e''' = !x, :=x$, or xv .

In these cases, π is stuck and has form 6, 7 or 8 respectively.

Otherwise.

e''' is stuck precisely in those cases for which e' is stuck.

Case $e' = \text{raise } v$.

This case proceeds by analysis of the form of v .

Case $v = x$.

In this case, if E is not of the form $\rho\theta\langle x, v' \rangle.E'$, then π is not stuck. Otherwise, it is stuck and has form 11.

Case $v = ex$.

In this case, π is not stuck.

Otherwise.

In the other cases, π is stuck and has form 9.

Case $e' = e'' \text{ handle } v$.

This case proceeds by analysis of the form of v .

Case $v = x$.

In this case, if E is not of the form $\rho\theta\langle x, v' \rangle.E'$, then π is not stuck. Otherwise, it is stuck and has form 12.

Case $v = ex$.

In this case, π is not stuck.

Otherwise.

In the other cases, π is stuck and has form 10.

Case $e' = \text{exn}$.

Then $E[e'] = E[\text{exn}]$, and thus π is not stuck.

Case $e' = \rho\theta.e''$.

Then $E[e'] = E[\rho\theta.e'']$, and thus π is stuck precisely when e'' is stuck.

Case $e' = \text{proc } x \text{ in } e''$.

Then $E[e'] = E[\text{proc } x \text{ in } e'']$. If E does not have the form $\rho\theta\langle x, v' \rangle.E$, then π is stuck precisely when e'' is stuck. Otherwise π is stuck and has form 18.

Case $e' = \text{self_id } e''$.

Then π is stuck precisely when e'' is stuck.

Case $e' = \text{prt } x \text{ on } v \text{ in } e''$.

This case proceeds by analysis of the form of v .

Case $v = \pi'$.

Then $E[e'] = E[\text{prt } x \text{ on } \pi' \text{ in } e'']$. If E does not have the form $\rho\theta\langle x, v' \rangle.E'$, then π is stuck precisely when e'' is stuck. Otherwise π is stuck and has form 19.

Case $v = x'$.

Then $E[e'] = E[\text{prt } x \text{ on } x' \text{ in } e'']$. If E does not have the form $\rho\theta\langle x, v' \rangle.E'$ or $\rho\theta\langle x', v' \rangle.E'$, then π is stuck precisely when e'' is stuck. Otherwise π is stuck and has form 19.

Otherwise.

Then π is stuck and has form 13.

Case $e' = \text{execute } (v_1 . v_2)$.

This case proceeds by analysis of the form of v_1 .

Case $v_1 = \pi'$.

Then $E[e'] = E[\text{execute } (\pi' . v_2)]$. If v_2 has form $Y, (v_1 . v_2), \text{ref}, !, :=, := x, ex, \pi$, or ϕ , then π is stuck and has form 14. Otherwise π is not stuck.

Case $v_1 = x'$.

Then $E[e'] = E[\text{execute } (x' . v_2)]$. If E has the form $\rho\theta\langle x', v' \rangle.E'$, then π is stuck and has form 20.

Otherwise.

Then π is stuck and has form 14.

Case $e' = \text{send } (v_1 . v_2)$.

This case proceeds by analysis of the form of v_1 .

Case $v = \phi$.

Then $E[e'] = E[\text{send } (\phi . v_2)]$ and π is not stuck.

Case $v = x'$.

Then $E[e'] = E[\text{send } (x' . v_2)]$. If E has the form $\rho\theta\langle x', v' \rangle.E'$, then π is stuck and has form 21. Otherwise π is not stuck.

Otherwise.

Then π is stuck and has form 15.

Case $e' = \text{recv } v$.

This case proceeds by analysis of the form of v .

Case $v = \phi$.

Then $E[e'] = E[\text{recv } \phi]$ and π is not stuck.

Case $v = x'$.

Then $E[e'] = E[\text{recv } x']$. If E has the form $\rho\theta\langle x', v' \rangle.E'$, then π is stuck and has form 22. Otherwise π is not stuck.

Otherwise.

Then π is stuck and has form 16.

Case $e' = \text{probe } v$.

This case proceeds by analysis of the form of v .

Case $v = \phi$.

Then $E[e'] = E[\text{probe } \phi]$ and π is not stuck.

Case $v = x'$.

Then $E[e'] = E[\text{probe } x']$. If E has the form $\rho\theta\langle x', v' \rangle.E'$, then π is stuck and has form 22. Otherwise π is not stuck.

Otherwise.

Then π is stuck and has form 16.

(only if) This direction of the *iff* follows directly from the definitions.

Thus the lemma is proved. ■

Proof of Lemma 7-11

The final lemma to prove establishes that stuck configurations are untypable.

Lemma 7-11 (Stuck configurations are untypable) If π is stuck in a well-formed configuration χ, \mathcal{P} , with $\# = \langle \pi; \Phi; e \rangle$ then there do not exist $ET \in \text{EXNNAMETY}$, $PT \in \text{PORTNAMETY}$ and $CT \in \text{CONFIGTY}$, such that

$$(\{\}, ET, PT) \vdash e : CT(\pi)$$

In other words, χ, \mathcal{P} is untypable.

Proof. Let π be stuck in χ, \mathcal{P} , with $\langle \pi; \Phi; E[e'] \rangle \in \mathcal{P}$, and assume that there exist $ET \in \text{EXNNAMETY}$, $PT \in \text{PORTNAMETY}$ and $CT \in \text{CONFIGTY}$, such that $(\{\}, ET, PT) \vdash E[e'] : CT(\pi)$. It suffices to show that e' is untypable, which is a contradiction. Let τ be the type of e' ; that is, $TE' \vdash e' : \tau$, for some TE' . Note that since χ, \mathcal{P} is well-formed, $E[e']$ is closed; and thus **Lemma 7-10** gives the possible forms of e' . The proof proceeds by case analysis on the form of the subexpression e' , showing that e' is untypable in each case.

Case $e' = v \ v'$. Rule ($\tau\text{-app}$) applies:

$$(1) \quad \frac{TE' \vdash v : (\tau' \rightarrow \tau) \quad TE' \vdash v' : \tau'}{TE' \vdash v \ v' : \tau}$$

There are six subcases, depending on the structure of v :

Case $v = c$, with $\delta(c, v')$ undefined.

By the δ -typability restriction, $\delta(c, v')$ is defined, which contradicts π being stuck.

Case $v = x$.

By **Lemma 6-1**, since $E[e']$ is closed, it must have the form $\rho\theta(x, v').E'[e']$. Then there must exist a TE'' , such that by rules (τ -app) and (τ -ref),

$$\frac{TE'' \vdash \text{ref} : \psi \rightarrow \psi \text{ ref} \quad TE'' \vdash v' : \psi}{TE'' \vdash \text{ref } v' : \psi \text{ ref}}$$

And by **Sequential type preservation (Theorem 7.6)**

$$TE'' \vdash \rho\theta(x, v').x : \psi \text{ ref}$$

Rule (τ -rho) requires that $TE'' \vdash \{x \mapsto \psi \text{ ref}\} \vdash x : \psi \text{ ref}$, but this contradicts the first premise of (1), and thus e' is untypable.

Case $v = (v_1 . v_2)$.

Rule (τ -pair) requires that $TE' \vdash (v_1 . v_2) : (\tau_1 \times \tau_2)$, where $TE' \vdash v_i : \tau_i$, which contradicts the first premise of (1), thus e' is untypable.

Case $v = ex$.

Rule (τ -ex) requires that ex have the type $\psi \text{ exn}$, for some ψ , but this contradicts the first premise of (1), thus e' is untypable.

Case $v = \pi$.

Rule (τ -procvar) requires that π have the type *ProcessName*, but this contradicts the first premise of (1), thus e' is untypable.

Case $v = \phi$.

Rule (τ -prtvar) requires that ϕ have the type $\psi \text{ PortName}$, for some ψ , but this contradicts the first premise of (1), thus e' is untypable.

Case $e' = \text{exception } x \text{ in } e''$. Rule (τ -exn) applies:

$$(2) \quad \frac{TE' \vdash \{x \mapsto \psi \text{ exn}\} \vdash e'' : \tau}{TE' \vdash \text{exception } x \text{ in } e'' : \tau}$$

Letting $e'' = E'[e''']$, this case proceeds by case analysis on the form of e''' , where e''' has type τ' .

Case $e''' = ! x$.

Rule (τ -app) and rule (τ -deref) require that $TE' \vdash x : \tau' \text{ ref}$, but this contradicts the requirement in (2) that $TE' \vdash \{x \mapsto \psi \text{ exn}\}$, which gives $TE' \vdash \{x \mapsto \psi \text{ exn}\} \vdash x : \psi \text{ exn}$. Thus e''' is untypable.

Case $e''' = := x$.

Similar to previous case, but using rule (τ -assign).

Case $e''' = x v$.

Rule (τ -app) requires that $TE' \vdash x : \tau \rightarrow \tau'$, but this contradicts the requirement in (2) that $TE' \vdash \{x \mapsto \psi \text{ exn}\}$, which gives $TE' \vdash \{x \mapsto \psi \text{ exn}\} \vdash x : \psi \text{ exn}$. Thus e''' is untypable.

Otherwise.

e''' is untypable precisely in those cases for which e' is untypable.

Case $e' = \text{raise } v$. Rules (τ -app) and (τ -raise) apply:

$$(3) \quad \frac{TE' \vdash \text{raise} : (\psi \text{ exn} \rightarrow \psi \rightarrow \tau') \quad TE' \vdash v : \psi \text{ exn}}{TE' \vdash \text{raise } v : \psi \rightarrow \tau'} \quad \text{with } \tau = \psi \rightarrow \tau'.$$

This case proceeds by analysis on the form of v .

Case $v = c$.

Rule (τ -const) and the second premise of (3) require that c has type $\psi \text{ exn}$, but there are no constant exceptions, and thus e' is untypable.

Case $v = x$.

If $E[e']$ has the form $\rho\theta\langle x, v' \rangle.E'[e']$, then there must exist a TE'' , such that by rules (τ -app) and (τ -ref),

$$\frac{\begin{array}{c} \vdots \\ TE'' \vdash \text{ref} : \psi \rightarrow \psi \text{ ref} \quad TE'' \vdash v' : \psi \end{array}}{TE'' \vdash \text{ref } v' : \psi \text{ ref}}$$

And by **Sequential type preservation (Theorem 7.6)**

$$TE'' \vdash \rho\theta\langle x, v' \rangle.x : \psi \text{ ref}$$

Rule (τ -rho) requires that $TE'' \vdash \{x \mapsto \psi \text{ ref}\} \vdash x : \psi \text{ ref}$, but this contradicts the second premise of (3), and thus e' is untypable.

Cases $v = Y, (v_1 . v_2), \lambda x(e), \text{ref}, !, :=, := x, \pi$, and ϕ .

In each of these cases, the rules (τ -Y), (τ -pair), (τ -abs), (τ -ref), (τ -deref), (τ -assign), (τ -assign) with (τ -app), (τ -procvar), and (τ -prtvar) apply respectively to contradict the second premise of (3), and thus e' is untypable.

Case $e' = e'' \text{ handle } v$.

This case proceeds by analysis of the form of v , in a similar fashion to the previous case, but with rules (τ -app) and (τ -handle) applying instead.

Case $e' = \rho\theta.e''$.

Then $E[e'] = E[\rho\theta.e'']$, and thus e' is untypable precisely when e'' is untypable.

Case $e' = \text{proc } x \text{ in } e''$. Rule ($\tau\text{-app}$) applies:

$$(4) \quad \frac{\text{TE}' \pm \{x \mapsto \text{ProcessName}\} \vdash e'' : \tau}{\text{TE}' \vdash \text{proc } x \text{ in } e'' : \tau}$$

If E has the form $\rho\theta\langle x, v' \rangle.E'$, then there must exist a TE'' , such that by rules ($\tau\text{-app}$) and ($\tau\text{-ref}$),

$$\frac{\begin{array}{c} \vdots \\ \text{TE}'' \vdash \text{ref} : \psi \rightarrow \psi \text{ ref} \quad \text{TE}'' \vdash v' : \psi \end{array}}{\text{TE}'' \vdash \text{ref } v' : \psi \text{ ref}}$$

And by **Sequential type preservation (Theorem 7.6)**

$$\text{TE}'' \vdash \rho\theta\langle x, v' \rangle.x : \psi \text{ ref}$$

Rule ($\tau\text{-rho}$) requires that $\text{TE}'' \pm \{x \mapsto \psi \text{ ref}\} \vdash x : \psi \text{ ref}$, but this contradicts the premise of (4), and thus e' is untypable.

Otherwise e' is untypable when e'' is untypable.

Case $e' = \text{self_id } e''$.

Then e' is untypable precisely when e'' is untypable.

Case $e' = \text{prt } x \text{ on } v \text{ in } e''$. Rule ($\tau\text{-prt}$) applies:

$$(5) \quad \frac{\text{TE}' \vdash \pi' : \text{ProcessName} \quad \text{TE}' \pm \{x \mapsto \psi \text{ PortName}\} \vdash e'' : \tau}{\text{TE}' \vdash \text{prt } x \text{ on } \pi' \text{ in } e'' : \tau}$$

This case proceeds by analysis of the form of v .

Case $v = \pi'$.

If E has the form $\rho\theta\langle x, v' \rangle.E'$, then there must exist a TE'' , such that by rules ($\tau\text{-app}$) and ($\tau\text{-ref}$),

$$\frac{\begin{array}{c} \vdots \\ \text{TE}'' \vdash \text{ref} : \psi \rightarrow \psi \text{ ref} \quad \text{TE}'' \vdash v' : \psi \end{array}}{\text{TE}'' \vdash \text{ref } v' : \psi \text{ ref}}$$

And by **Sequential type preservation (Theorem 7.6)**

$$\text{TE}'' \vdash \rho\theta\langle x, v' \rangle.x : \psi \text{ ref}$$

Rule ($\tau\text{-rho}$) requires that $\text{TE}'' \pm \{x \mapsto \psi \text{ ref}\} \vdash x : \psi \text{ ref}$, but this contradicts the second premise of (5), and thus e' is untypable.

Otherwise e' is untypable precisely when e'' is untypable.

Case $v = x'$.

Similar to previous case.

Cases $v = Y, (v_1 . v_2), \lambda x(e), ex, \text{ref}, !, :=, := x$, and ϕ .

In each of these cases, the rules ($\tau\text{-Y}$), ($\tau\text{-pair}$), ($\tau\text{-abs}$), ($\tau\text{-ex}$), ($\tau\text{-ref}$), ($\tau\text{-deref}$), ($\tau\text{-assign}$), ($\tau\text{-assign}$) with ($\tau\text{-app}$), and ($\tau\text{-prtvar}$) apply respectively to contradict the second premise of (5), and thus e' is untypable.

Case $e' = \text{execute } (v_1 . v_2)$. Rule ($\tau\text{-execute}$) applies:

$$(6) \quad \frac{\text{TE}' \vdash \pi' : \text{ProcessName} \quad \text{TE}' \vdash v_2 : \text{unit} \rightarrow \text{unit}}{\text{TE}' \vdash \text{execute } (\pi' . v_2) : \text{unit}}$$

This case proceeds by analysis of the form of v_1 .

Case $v_1 = \pi'$.

Then there are various subcases to consider on the form of v_2 .

Case $v_2 = c$.

If $\text{TypeOf}(c) \neq \text{unit} \rightarrow \text{unit}$, then this contradicts the second premise of (6) and e' is untypable.

Case $v_2 = x$.

If $\text{TE}' \vdash v_2 : \tau$, such that $\tau \neq \text{unit} \rightarrow \text{unit}$, then this contradicts the second premise of (6) and e' is untypable.

Case $v_2 = := x$.

If $\text{TE}' \vdash v_2 : \tau$, such that $\tau \neq \text{unit} \rightarrow \text{unit}$, then this contradicts the second premise of (6) and e' is untypable.

Case $v_2 = \lambda x(e'')$.

If $\text{TE}' \vdash v_2 : \tau$, such that $\tau \neq \text{unit} \rightarrow \text{unit}$, then this contradicts the second premise of (6) and e' is untypable.

Cases $v_2 = Y, (v_3 . v_4), \text{ex}, \text{ref}, !, :=, \text{and } \phi$.

In each of these cases, the rules ($\tau\text{-Y}$), ($\tau\text{-pair}$), ($\tau\text{-ex}$), ($\tau\text{-ref}$), ($\tau\text{-deref}$), ($\tau\text{-assign}$), and ($\tau\text{-prtvvar}$) apply respectively to contradict the second premise of (6), and thus e' is untypable.

Case $v_1 = x'$.

If E has the form $\rho\theta\langle x', v' \rangle.E'$, then there must exist a TE'' , such that by rules ($\tau\text{-app}$) and ($\tau\text{-ref}$),

$$\frac{\begin{array}{c} \vdots \\ \text{TE}'' \vdash \text{ref} : \psi \rightarrow \psi \text{ ref} \quad \text{TE}'' \vdash v' : \psi \end{array}}{\text{TE}'' \vdash \text{ref } v' : \psi \text{ ref}}$$

And by **Sequential type preservation** (Theorem 7.6)

$$\text{TE}'' \vdash \rho\theta\langle x', v' \rangle.x' : \psi \text{ ref}$$

Rule ($\tau\text{-rho}$) requires that $\text{TE}'' \vdash \{x' \mapsto \psi \text{ ref}\} \vdash x' : \psi \text{ ref}$, but this contradicts the first premise of (6), and thus e' is untypable.

Otherwise e' is untypable in the same situations as in the previous subcase of $v_1 = \pi'$.

Cases $v_1 = Y, (v_1 . v_2), \lambda x(e), \text{ex}, \text{ref}, !, :=, := x$.

In each of these cases, the rules ($\tau\text{-Y}$), ($\tau\text{-pair}$), ($\tau\text{-abs}$), ($\tau\text{-ex}$), ($\tau\text{-ref}$), ($\tau\text{-deref}$), ($\tau\text{-assign}$), and ($\tau\text{-assign}$) with ($\tau\text{-app}$), apply respectively to contradict the first premise of (6), and thus e' is untypable.

Case $e' = \text{send}(v_1 . v_2)$. Rule ($\tau\text{-send}$) applies:

$$(7) \quad \frac{\text{TE}' \vdash \phi : \psi \text{ PortName} \quad \text{TE}' \vdash v_2 : \psi}{\text{TE}' \vdash \text{send}(\phi . v) : \text{unit}}$$

This case proceeds by analysis of the forms of v_1 which cause π to be stuck.

Case $v_1 = x'$.

If E has the form $\rho\theta\langle x', v' \rangle . E'$, then there must exist a TE'' , such that by rules ($\tau\text{-app}$) and ($\tau\text{-ref}$),

$$\frac{\text{TE}'' \vdash \text{ref} : \psi \rightarrow \psi \text{ ref} \quad \text{TE}'' \vdash v' : \psi}{\text{TE}'' \vdash \text{ref } v' : \psi \text{ ref}}$$

And by **Sequential type preservation** (7.6)

$$\text{TE}'' \vdash \rho\theta\langle x', v' \rangle . x' : \psi \text{ ref}$$

Rule ($\tau\text{-rho}$) requires that $\text{TE}'' \vdash \{x' \mapsto \psi \text{ ref}\} \vdash x' : \psi \text{ ref}$, but this contradicts the first premise of (7), and thus e' is untypable.

Cases $v_1 = Y, (v_1 . v_2), \lambda x(e), ex, \text{ref}, !, :=, := x$, and π .

In each of these cases, the rules ($\tau\text{-Y}$), ($\tau\text{-pair}$), ($\tau\text{-abs}$), ($\tau\text{-ex}$), ($\tau\text{-ref}$), ($\tau\text{-deref}$), ($\tau\text{-assign}$), ($\tau\text{-assign}$) with ($\tau\text{-app}$), and ($\tau\text{-procvar}$) apply respectively to contradict the first premise of (7), and thus e' is untypable.

Case $e' = \text{recv } v$. Rule ($\tau\text{-recv}$) applies:

$$(8) \quad \frac{\text{TE}' \vdash \phi : \tau \text{ PortName}}{\text{TE}' \vdash \text{recv } \phi : \tau}$$

This case proceeds by analysis of the form of v .

Case $v = c$.

But there are no constants of type $\tau \text{ PortName}$, so this contradicts the premise of (8) and thus e' is untypable.

Case $v = x'$.

If E has the form $\rho\theta\langle x', v' \rangle . E'$, then there must exist a TE'' , such that by rules ($\tau\text{-app}$) and ($\tau\text{-ref}$),

$$\frac{\text{TE}'' \vdash \text{ref} : \psi \rightarrow \psi \text{ ref} \quad \text{TE}'' \vdash v' : \psi}{\text{TE}'' \vdash \text{ref } v' : \psi \text{ ref}}$$

And by **Sequential type preservation** (7.6)

$$\text{TE}'' \vdash \rho\theta\langle x', v' \rangle . x' : \psi \text{ ref}$$

Rule ($\tau\text{-rho}$) requires that $\text{TE}'' \vdash \{x' \mapsto \psi \text{ ref}\} \vdash x' : \psi \text{ ref}$, but this contradicts the premise of (8), and thus e' is untypable.

Cases $v_1 = Y, (v_1 . v_2), \lambda x(e), ex, \text{ref}, !, :=, := x$, and π .

In each of these cases, the rules $(\tau\text{-}Y)$, $(\tau\text{-pair})$, $(\tau\text{-abs})$, $(\tau\text{-ex})$, $(\tau\text{-ref})$, $(\tau\text{-deref})$, $(\tau\text{-assign})$, $(\tau\text{-assign})$ with $(\tau\text{-app})$, and $(\tau\text{-procvar})$ apply respectively to contradict the premise of (8), and thus e' is untypable.

Case $e' = \text{probe } v$.

This is essentially identical to the last case, with rule $(\tau\text{-probe})$ applying instead.

Thus the lemma is proved. ■

There is a great deal of interest in the study of the history of the United States, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

The study of the history of the United States is one of the most popular in the country, and it is not surprising that the study of the history of the United States is one of the most popular in the country.

Glossary

acceptable	a trace of process configurations that results in all processes being complete or awaiting; Def. 6-12, p. 117
address space	an integer range; individual memory locations are identified by values within the range
applicative	applying or capable of being applied, as in functions
algorithmic skeletons	system to abstract details of parallelism in an algorithm from a user; see §8.2
answer	either a closed value or an unhandled exception; Def. 6-1, p.104
asynchronous	actions happen independently of others; thus in message passing, a sender continues immediately once the message has been sent
awaiting	status of a paraML process if it is yet to execute any code; Def. 6-7, p. 116
bandwidth	the quantity of data which may be transferred between different locations in a fixed period of time, often measured in MB/s (Megabytes per second)
blocked	status of a paraML process if it is trying to receive a message which is yet to be sent; Def. 6-7, p. 116
blocking	actions halt until data is available with which to proceed
broadcast	a message passing form where all entities must participate
buffer	some memory which is used to store values
central processing unit (CPU)	the core unit of a computer which executes machine code instructions; also referred to as a processor or chip
channel	a message buffer, usually allowing any entity to send or receive from it
client-server	model of programming where one or more processes are specialised to act as server resources for computation or data, passing results to clients which request the service
closed	no free variables are present in the expression or value
closure	an expression (usually for a function) which carries bindings for all the free variables of the expression
code	the programming language text of a process or program, in executable form
coherency	mechanisms for guaranteeing the integrity of data when multiple entities can access and alter the data
communicator	a communications layer private to a group of processes, and distinct from other communicators (which may span the same group of processes); defined in MPI

complete	a trace of a process configurations where all processes have either terminated or are awaiting a request to execute code; Def. 6-8, p. 116
computation	a trace of process configurations that is either infinite or finite with the final configuration complete; Def. 6-9, p. 116
concurrent	the description of a problem as a set of cooperating entities, whose execution overlaps each entity
configuration	a set of unique exception names and a set of unique process states; Def. 6-4, p. 114
consistency	the ordering and outcome of a set of operations on some data that may be shared among different computation components is agreed to by each component
constant	a value that does not change and is defined independently of any name binding environment
context	mechanism for describing the current expression evaluation, by splitting the expression into a context and redex; typically used to control order of evaluation; see §5.4.2
converge	an evaluation that eventually produces an answer; Def. 6-11, p. 116
coordination	mechanisms to achieve relative ordering of actions among a number of processes
coordination system	a system to achieve coordination, usually among distinct programs in a distributed computer environment
critical region	a part of a program that must not be executed by two processes concurrently
CSP	Communicating Sequential Processes; a formalised model of processes with synchronous communications between them
deadlock	situation in which two or more processes cannot proceed due to interdependencies which cannot be resolved (e.g. both processes waiting to receive a message before sending one)
disjoint memory	multiple memory modules, each with its own address space
distributed address space	a computing environment with multiple disjoint address spaces
distributed memory	multiple memory modules
distributed shared memory (DSM)	a software mechanism for constructing a shared address space from distributed memory
distribution	the issues that arise when computing resources are distributed physically, including failure of communication links for example; also used to describe how data or processes are assigned to a processor topology
diverge	an evaluation that never produces an answer (e.g. infinite loop); Def. 5-8, p. 98 and Def. 6-11, p. 116
dynamic	happens during a program's execution

efficiency	the inverse of speedup; measures the percentage of a program that has been successfully parallelised; see §3.3.2
efficient	can support programming without undue performance penalties
enabled	a paraML process that is able to perform a transition according to the parallel evaluation rules; Def. 6-8, p. 116
evaluation	execution of a program
evaluation state	a component of the state of a process which captures where in the evaluation the process is and what will execute next; see §6.3.2
exception	a language mechanism typically used to indicate an error condition
explicit	requiring direct user control of an action
fairness	condition of the progression from configuration to configuration in a program trace such that no process is unnecessarily prevented from execution; Def. 6.12, p. 117
faulty	evaluation state which approximates a attempts to perform an illegal action; Def. 5-7, p. 97
function	similar to procedures or subroutines, but with a strong mathematical basis; accepts a single argument, evaluates, and returns a single result
handle	a memory pointer or other language mechanism for identifying some data object
hole	marks the split between a context and redex partitioning of an expression
imperative	permits computation with mutable data; reference variables in ML are an example
implicit	some aspect of program control or action that happens without user intervention; typically inserted by the compiler or by the runtime system
irreducible	unable to reduce any further according to the evaluation rules, usually because the expression is now a closed value
isoefficiency	a model for calculating relative performance of an algorithm on different parallel computer systems; see §3.3.2
λ (lambda) calculus	a calculus, based on mathematical functions, which encodes computation by expression reductions from applying functions to arguments
latency	the time taken between initiating an event (often a communication) and the first data from the event arriving at the destination
locality	measurement of the latency in accessing values or memory
lock	protection mechanism for indicating some computation is accessing a data value
mailbox	communication mechanism equivalent to a port, with multiple senders but only one receiver

marshal	mechanism for representing arbitrary data values as a contiguous array of bytes for communication
MIMD	Multiple Instruction, Multiple Data; part of Flynn's taxonomy of parallel computers, referring to the number of instruction streams and data access streams that may be managed simultaneously
message passing	a mechanism for communicating data by sending messages between processes
ML	the mostly functional programming language, based on the call-by-value λ calculus
monitor	a programming mechanism that associates shared variable access operations with procedures to prevent more than one process accessing a variable concurrently
multicast	message passing construction which has a single sender and multiple receivers
multicomputer	a generic form of parallel computer, consisting of multiple processing elements connected by a communications network
multiprocessor	typically refers to a parallel computer with multiple CPUs but only a single shared memory
multi-threading	the situation of having multiple threads executing or used for programming a system
name	a variable or other value that uniquely identifies another other value
network	physical communication hardware for connecting different computers or processing elements; and associated software protocols
node	<i>processing element</i>
object store	a shared address space model of programming with objects
parallel	simultaneous execution of concurrent entities; also used to refer to the superset of concurrent/parallel/distributed computing systems; also used for hardware which permits simultaneous execution
parallel evaluation relation	a set of rules used to model the evaluation in paraML; Def. 6-5, p. 114
partitioning	breaking up a program into various distinct actions
π (pi) calculus	a calculus of concurrency consisting of processes, channels and messages, where computation is expressed by message passing
pipeline	description of data movement through processes, so that at any one time, multiple data values are in the pipe
polymorphic	possessing or capable of acting on more than one type of value
port	a communications buffer, which any entity may send to but only one entity may receive from

portability	the ability to execute some program on more than one form of computer; porting is the act of converting a program to run on another computer
process	a generic term for a computing entity; used in this thesis to indicate an entity with a self-contained evaluation environment and memory, and capable of communications through ports
processing element	(PE) the building block of a multicomputer, comprising the essence of a computer: a CPU, memory, and interfaces to the interconnection communication network. Sometimes referred to as <i>node</i> , <i>processor</i> , <i>cell</i> .
program	programming language code which describes some algorithm, and when executed performs the algorithm
ready	a process which is capable of applying one of the parallel evaluation relation rules to continue execution; Def. 6-7, p. 116
receive	action of accepting a message communicated by another process
redex	an expression which is the term currently being evaluated in a context/redex pair according to some reduction rule; see §5.4.2
reduction	a formalism that models evaluation by means of syntactically rewriting an expression, ultimately producing an answer
reference	the address of a memory cell whose value may be changed
rule	specification of an evaluation step, with some preconditions and before and after evaluation states
safe	a property of a programming language that does not allow statically detectable or avoidable errors, and provides ways of recovering from errors that occur at runtime
shared memory	a single memory module (or multiple clustered memory modules) with a single address space
SIMD	Single Instruction, Multiple Data; part of Flynn's taxonomy of parallel computers, referring to the number of instruction streams and data access streams that may be managed simultaneously
scalability	the ability for a program or system's performance to increase as the number of PEs increases
semaphore	a two-state data structure used to prevent access to some other data structure
send	action of communicating a message from one process to another
sequential evaluation relation	a set of rules used to model sequential evaluation in paraML; Def. 5-2, p. 89
shared address space	a single address space, regardless of whether the underlying memory is distributed or not
speedup	an increase in performance, specifically an increase as the number of PEs increases

SPMD	Single Program, Multiple Data; a single program executable is executed on all PEs, but with different data values for each
static	determined prior to or at the commencement of execution
status	a property of a process with relation to other processes in a configuration; Def. 6-7, p. 116
stuck	a process whose evaluation state results in the process being unable to proceed with evaluation due to a syntactic error; Def. 7-2, p. 125
substitution	replacement of a variable with an expression throughout some other expression
synchronous	actions that happen only when two (or more) entities collaborate; in message passing, a send only completes when the matching receive has also completed
task	alternative name for a process
terminated	paraML process that has produced an answer; Def. 6-7, p. 116
text	the written form (in some programming language) of an algorithm for a program
thread	an alternative name for a process; usually denotes a lighter weight component, often executing in a shared address space
trace	a series of transitions between process configurations according to the rules of the parallel evaluation relation; Def. 6-9, p. 116
transaction	a programming mechanism for a series of accesses to an object(s)'s data, which must either succeed or fail in entirety
typability	the ability of the type system to infer a type for an expression; in particular, δ -typability which abstracts details of the result of applying function constants to arguments; Def. 5-5, p. 93
type	a language mechanism for discriminating between different kinds of data values or objects
type environment	mapping from variable names to types; see §5.4.3.1 and §7.1.1
type judgement	sentences which can be inferred from a type system according to the rules and with respect to some type environment
type scheme	an abstract form of type, used for polymorphic type systems where one type may have several instantiations
unit	the null value in ML, written $()$; also the type of this value
unmarshal	accepts a contiguous array of bytes and assembles an ML value
variable	a name which may be bound to some value
virtual memory	a system for providing a logically contiguous address space, independent of the physical availability or location in memory
virtual processor	(VP) an abstract form of a physical processing element
well-formed	having certain defined properties, which are required in order to make other statements about evaluation; Def. 6-4, p. 114
well-typed	possesses certain defined properties with respect to typing

References

- [AAL95] Anderson, J.M., Amarasinghe, S.P. & Lam, M.S. "Data and Computation Transformations for Multiprocessors". In *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing* (1995).
- [ACG86] Ahuja, S., Carriero, N. & Gelernter, D. "Linda and Friends". *IEEE Computer* 19, 8 (1986) pp. 26-34.
- [Ada83] USA Department of Defence **Reference Manual for the Ada Programming Language**. Springer-Verlag, New York (1983).
- [AH87] Agha, G. & Hewitt, C. "Concurrent Programming Using Actors". In **Object-Oriented Concurrent Programming**, Yonezawa, A. & Tokoro, M. (ed), MIT Press (1987).
- [AM91] Appel, A.W. & MacQueen, D.B. "Standard ML of New Jersey". In *Proc. Third International Symposium on Programming Language Implementation and Logic Programming* (1991) pp. 1-13.
- [Ama94] Amadio, R.M. "Translating Core Facile". European Computer-Industry Research Centre, ECRC-1994-3 (1994).
- [Amd67] Amdahl, G. "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". In *Proc. AFIPS Conference* (1967) pp. 485-487.
- [ANP89] Arvind, Nikhil, R.S. & Pingali, K. "I-structures: Data structures for parallel computing". *ACM Transactions on Programming Languages and Systems* 11, 4 (1989) pp. 598-632.
- [App92] Appel, A.W. **Compiling with Continuations**. Cambridge University Press (1992).
- [App93] Appel, A.W. "A Critique of Standard ML". *Journal of Functional Programming* 3, 4 (1993) pp. 391-430.
- [Bai94] Bailey, P. "paraML Programming Manual" CAP Research Program, Australian National University available at <http://cap.anu.edu.au/projects/paraml> (January 1994).
- [Bai96] Bailey, P. "Towards a formal semantics for paraML". In *Proc. CATS'96 (Computing: The Australasian Symposium)*, Melbourne, Australia (1996) pp. 1-10.

- [Bar84] Barendregt, H.P. **The Lambda Calculus: Its Syntax and Semantics**. Studies in Logic and the Foundations of Mathematics 103, (1984).
- [BB92] Berry, G. & Boudol, G. "The chemical abstract machine". *Theoretical Computer Science* 96 (1992) pp. 217-248.
- [BCH+93] Blelloch, G., Chatterjee, S., Hardwick, J.C., Sipelstein, J. & Zagha, M. "Implementation of a Portable Nested Data-Parallel Language". In *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California (1993) pp. 102-111.
- [BDO+95] Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S. & Vanneschi, M. "P³L: a Structured High-level Parallel Language, and its Structured Support". *Concurrency: Practice and Experience* 7, 3 (1995) pp. 225-255.
- [BJ87] Birman, K. & Joseph, T. "Reliable Communication in the Presence of Failures". *ACM Transactions on Computer Systems* 5, 1 (1987) pp. 47-76.
- [BIS94] Bertomieu, B. & le Sergent, T. "Programming with behaviours in an ML framework: the syntax and semantics of LCS". In **LNCS 788**, Springer Verlag (1994) pp. 89-104.
- [BM92] Brown, A.L. & Morrison, R. "A Generic Persistent Object Store". *Software Engineering Journal* 7, 2 (1992) pp. 161-168.
- [BMS94] Bruce, R.A.A., Mills, J.G. & Smith, A.G. "CHIMP/MPI User Guide". Key Technology Project, Edinburgh Parallel Computing Centre, EPCC-KTP-CHIMP-V2-USER 1.2 (1994).
- [BMT92] Berry, D., Milner, R. & Turner, D.N. "A semantics for (ML) concurrency primitives". In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico (1992) pp. 119-129.
- [BN93] Bailey, P. & Newey, M. "An Extension of ML for Distributed Memory Multicomputers". In *Proc. Sixteenth Australian Computer Science Conference*, Brisbane, Australia (1993) pp. 387-396.
- [BNA91] Barth, P.S., Nikhil, R.S. & Arvind "M-structures: extending a parallel, non-strict functional language with state". In *Proc. Functional Programming Languages and Computer Architecture*, Boston, USA (1991) pp. 538-568.

- [BNS+94] Bailey, P., Newey, M., Sitsky, D. & Stanton, R. "Supporting Coarse and Fine Grain Parallelism in an Extension of ML". In *Proc. CONPAR'94 - VAPP VI*, Linz, Austria (1994) pp. 593-604.
- [Bor86] Bornat, R. "A protocol for generalized occam". *Software – Practice and Experience* 16, 9 (1986) pp. 783-799.
- [Bra94] Bratvold, T.A. "Skeleton-Based Parallelisation of Functional Programs". Ph.D. Thesis, Heriot-Watt University (1994).
- [Bri73] Brinch Hansen, P. "Concurrent Programming Concepts". *ACM Computing Surveys* 5, 4 (1973) pp. 223-245.
- [Bri75] Brinch Hansen, P. "The Programming Language Concurrent Pascal". *IEEE Transactions on Software Engineering* SE-1, 2 (1975) pp. 199-207.
- [Car83] Cardelli, L. "The Functional Abstract Machine". AT&T Bell Laboratories, TR-107 (1983).
- [Car90] Caromel, D. "Concurrency: An Object-Oriented Approach". In *Proc. TOOLS'90*, Paris, France (1990) pp. 183-197.
- [Car95] Cardelli, L. "A Language with Distributed Scope". In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, New York, USA (1995) pp. 286-298.
- [CDV88] Carey, M., DeWitt, D. & Vandenberg, S. "A Data Model and Query Language for EXODUS". In *Proc. 1988 ACM SIGMOD Conference*, Chicago, USA (1988) pp. 413-423.
- [CF93] Chandy, K.M. & Foster, I.T. "A Deterministic Notation for Cooperating Processes". Argonne National Laboratory, MCS-P346-0193 (1993).
- [CFT+94] Clarke, L.J., Fletcher, R.A., Trewin, S., Bruce, R.A.A., Smith, A.G. & Chapple, S.R. "Reuse, Portability and Parallel Libraries". Edinburgh Parallel Computing Centre, TR9413 (1994).
- [CGH93] Chandra, R., Gupta, A. & Hennessey, J.L. "Data Locality and Load Balancing in COOL". In *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, USA (1993) pp. 249-259.
- [CHL78] Crowley, W., Hendrickson, C.P. & Luby, T.I. "The SIMPLE Code". Lawrence Livermore Laboratory, Technical Report UCID-17715 (1978).

- [CJK95] Cetjin, H., Jagannathan, S. & Kelsey, R. "Higher-Order Distributed Objects". *ACM Transactions on Programming Languages and Systems* 17, 5 (1995) pp. 704-739.
- [CKK95] Carter, J.B., Khandekar, D. & Kamb, L. "Distributed Shared Memory: Where We Are and Where We Should Be Headed". In *Proc. Fifth Workshop on Hot Topics in Operating Systems* (1995) pp. 119-122.
- [CKP+93] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R. & von Eicken, T. "LogP: Towards a Realistic Model of Parallel Computation". In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, USA (1993) pp. 1-12.
- [Col89] Cole, M. **Algorithmic Skeletons: Structured Management of Parallel Computation**. Research Monographs in Parallel and Distributed Computing MIT Press (1989).
- [DD68] Daley, R.C. & Dennis, J.B. "Virtual Memory, Processes, and Sharing in MULTICS". *Communications of the ACM* 11, 5 (1968) pp. 306-312.
- [DGT+95] Darlington, J., Guo, Y.K., To, H.W. & Jing, Y. "Skeletons for Structured Parallel Composition". In *Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1995).
- [Dij68] Dijkstra, E.W. "Cooperating Sequential Processes". In **Programming Languages**, Genuys, F. (ed), Academic Press (1968) pp. 43-112.
- [Dij75] Dijkstra, E.W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". *Communications of the ACM* 18, 8 (1975) pp. 453-457.
- [DM82] Damas, L. & Milner, R. "Principal types for functional programs". In *Proc. 9th Annual Symposium on Principles of Programming Languages* (1982) pp. 207-212.
- [DP93] Danelutto, M. & Pellagatti, S. "Parallel implementation of FP using a Template-based approach". In *Proc. 5th International Workshop on the Implementation of Functional Languages* (1993) pp. 7-21.
- [DVH66] Dennis, J.B. & Van Horn, E.C. "Programming Semantics for Multiprogrammed Computations". *Communications of the ACM* 9, 3 (1966) pp. 143-155.

- [FC94] Foster, I. & Chandy, K.M. "Fortran M: A Language for Modular Parallel Programming". *Journal of Parallel and Distributed Computing* 25, 1 (1995).
- [FF86] Felleisen, M. & Friedman, D.P. "Control operators, the SECD-machine, and the λ -calculus". In **Formal Description of Programming Concepts - III**, Wirsing, M. (ed), Elsevier Science Publishers B.V. (1986) pp. 193-217.
- [FH92] Felleisen, M. & Hieb, R. "The revised report on the syntactic theories of sequential control and state". *Theoretical Computer Science* 103 (1992) pp. 235-271.
- [FHJ95] Ferreira, W., Hennessey, M. & Jeffrey, A. "A Theory of Weak Bisimulation for Core CML". School of Cognitive and Computing Sciences, University of Sussex, Report 05/95 (1995).
- [Fly66] Flynn, M.J. "Very High-Speed Computing Systems". *Proceedings of the IEEE* 54, 12 (1966) pp. 1901-1909.
- [Fos94] Foster, I. **Designing and Building Parallel Programs**. Addison-Wesley (1994).
- [GBD94] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. & Sunderam, V. **PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing**. Scientific and Engineering Computation MIT Press (1994).
- [GG88] Goldman, R. & Gabriel, R.P. "Preliminary Results with the Initial Implementation of Qlisp". In *Proc. ACM Lisp and Functional Programming* (1988) pp. 143-152.
- [GLS95] Gropp, W., Lusk, E. & Skjellum, A. **Using MPI**. MIT Press (1995).
- [GMB88] Gustafson, J.L., Montry, G.R. & Benner, R.E. "Development of Parallel Methods for 1024-Processor Hypercube". *SIAM Journal on Scientific and Statistical Computing* 9, 4 (1988) pp. 532-533.
- [GMc84] Gabriel, R. & McCarthy, J. "Queue-based Multi-processing Lisp". In *Proc. ACM Symposium on Lisp and Functional Programming* (1984) pp. 25-43.
- [Gor94] Gordon, A.D. **Functional Programming and Input/Output**. Cambridge University Press (1994).
- [Gra68] Graham, R.M. "Protection in an Information Processing Utility". *Communications of the ACM* 11, 5 (1968) pp. 365-369.

- [Hal85] Halstead, R. "Multilisp: A Language for Concurrent Symbolic Computation". *ACM Transactions on Programming Languages and Systems* 7, 4 (1985) pp. 501-538.
- [HF93] Hains, G. & Foisy, C. "The Data-Parallel Categorical Abstract Machine". In *Proc. PARLE'93*, Berlin, Germany (1993) pp. 56-67.
- [HKM+93] Haines, N., Kindred, D., Morrisett, J.G., Nettles, S.M. & Wing, J.M. "Tinkertoy® Transactions". School of Comp. Sci, Carnegie Mellon University, CMU-CS-93-202 (1993).
- [Hoa74] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept". *Communications of the ACM* 17, 10 (1974) pp. 549-557.
- [Hoa78] Hoare, C.A.R. "Communicating Sequential Processes". *Communications of the ACM* 21, 8 (1978) pp. 666-677.
- [HP94] Hennessy, J.L. & Patterson, D.A. **Computer Organisation and Design**. Morgan Kaufmann (1994).
- [HPF94] High Performance Fortran Forum "High Performance Fortran Language Specification v1.1". Center for Research on Parallel Computation, Rice University, available at <http://www.crpc.rice.edu/HPFF/hpf1/hpf-v11.ps.Z> (1994).
- [HQ91] Hatcher, P.J. & Quinn, M.J. **Data-Parallel Programming on MIMD Computers**. MIT Press (1991).
- [HS86] Hindley, R.J. & Seldin, J.P. **Introduction to Combinators and λ -Calculus**. Cambridge University Press (1986).
- [HT91] Honda, K. & Tokoro, M. "An object calculus for asynchronous communication". In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Geneva, Switzerland (1991) pp. 133-147.
- [JP92] Jagannathan, S. & Philbin, J. "A Foundation for an Efficient Multi-Threaded Scheme System". In *Proc. ACM Conference on Lisp and Functional Programming*, San Francisco, California (1992) pp. 345-357.
- [JP94] Jagannathan, S. & Philbin, J. "High-Level Abstractions for Efficient Concurrent Systems". In *Proc. International Conference on Programming Languages and System Architecture* (1994) pp. 171-190.
- [JW95] Jagannathan, S. & Wright, A. "Effective flow-analysis for avoiding runtime checks". In *Proc. International Static Analysis Symposium* (1995) pp. 207-224.

- [KG91] Kumar, V. & Gupta, A. "Analysis of Scalability of Parallel Algorithms and Architectures". Dept. Comp. Sci., University of Minnesota, TR 91-18 (1991).
- [KGG+93] Kumar, V., Grama, A., Gupta, A. & Karypis, G. **Introduction to Parallel Computing: Design and Analysis of Algorithms**. Benjamin/Cummings (1993).
- [Kna95] Knabe, F.C. "Language Support for Mobile Agents". Ph.D. Thesis, Carnegie Mellon University (1995).
- [KOH+94] Kuskina, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M. & Hennessy, J. "The Stanford FLASH multiprocessor". In *Proc. 21st International Symposium on Computer Architecture* (1994) pp. 302-313.
- [KR78] Kernighan, B.W. & Ritchie, D.M. **The C programming language**. Prentice-Hall (1978).
- [KR94] Kelsey, R. & Rees, J. "A Tractable Scheme Implementation". *Lisp and Symbolic Computing* 7, 2 (1994) pp. 315-335.
- [Kru93] Krumvieda, C.D. "Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming". Ph.D. Thesis, Cornell University (1993).
- [KS79] Kieburtz, R.B. & Silberschatz, A. "Comments on 'Communicating sequential processes'". *ACM Transactions on Programming Languages and Systems* 1, 2 (1979) pp. 218-225.
- [Kwi89] Kwiatkowska, M.Z. "Survey of fairness notions". *Information and Software Technology* 31, 7 (1989) pp. 371-386.
- [Lev75] Levin, D.Y. "Experimental Implementation of SETL". In **Methods of algorithmic language implementation**, Springer-Verlag, LNCS 47 (1975) pp. 268-276.
- [LG91] Lee, J.K. & Gannon, D. "Object Oriented Parallel Programming: Experiments and Results". In *Proc. Supercomputing '91*, Albuquerque, New Mexico (1991) pp. 273-282.
- [LH89] Li, K. & Hudak, P. "Memory Coherence in Shared Virtual Memory Systems". *ACM Transactions on Computing Systems* 7, 4 (1989) pp. 321-359.

- [LLG+92] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M. & Lam, M.S. "The Stanford Dash Multiprocessor". IEEE Computer 25, 3 (1992) pp. 63-79.
- [LRV92] Larus, J.R., Richards, B. & Viswanathan, G. "C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language". Computer Sciences Department, University of Wisconsin-Madison, (1992).
- [LS83] Liskov, B. & Scheifler, R. "Guardians and actions: Linguistic support for robust, distributed programs". ACM Transactions on Programming Languages and Systems 5, 3 (1983) pp. 382-404.
- [LS91] Lin, C. & Snyder, L. "A Portable Implementation of SIMPLE". International Journal of Parallel Programming 20, 5 (1991) pp. 363-401.
- [Mac92] MacQueen, D.B. "Reflections on Standard ML". In **Programming, Concurrency, Simulation and Automated Reasoning**, Springer-Verlag (1992) pp. 32-46.
- [Mat91] Matthews, D. "A Distributed Concurrent Implementation of Standard ML". Laboratory for Foundations of Computer Science, University of Edinburgh, ECS-LFCS-91-174 (1991).
- [MBC+89] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. "The Napier88 Reference Manual". Universities of Glasgow and St Andrews, PPRR-77-89 (1989).
- [Mil80] Milner, R. **A Calculus of Communicating Systems**. LNCS 92, Springer-Verlag (1980).
- [Mil93] Milner, R. "Elements of Interaction". Communications of the ACM 36, 1 (1993) pp. 78-89.
- [MLS95] Matthews, D.C.J. & Le Sergeant, T. "LEMMA: A Distributed Shared Memory with Global and Local Garbage Collection". In *Proc. International Workshop in Memory Management* (1995) pp. 297-311.
- [MMS79] Mitchell, J.G., Maybury, W. & Sweet, R. "Mesa Language Manual". Xerox Research Center, (1979).
- [Mor95] Morrisett, G. "Compiling with Types". Ph.D. Thesis, Carnegie Mellon University (1995).
- [Mos85] Moss, J.E.B. **Nested Transactions : An Approach to Distributed Computing**. MIT Press, Cambridge, Mass (1985).

- [MPI94] Message Passing Interface Forum "MPI: A Message Passing Interface Standard". University of Tennessee, Knoxville, CS-94-230 (1994).
- [MPI96] Message Passing Interface Forum "MPI-2: Extensions to the Message-Passing Interface". University of Tennessee, Knoxville, available at <ftp://ftp.mcs.anl.gov/pub/mpi/misc> (1996).
- [MPW92] Milner, R., Parrow, J. & Walker, D. "A Calculus of Mobile Processes I & II". *Information and Computation* 100, 1 (1992) pp. 1-77.
- [MT91] Milner, R. & Tofte, M. **Commentary on Standard ML**. MIT Press, Cambridge, Massachusetts (1991).
- [MT93] Morrisett, J.G. & Tolmach, A. "Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey". In *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California (1993) pp. 198-207.
- [MTH90] Milner, R., Tofte, M. & Harper, R. **The Definition of Standard ML**. MIT Press, Cambridge, Massachusetts (1990).
- [Nel91] Nelson, G. **Systems Programming with Modula-3**. Prentice-Hall (1991).
- [Net92] Nettles, S. "A LARCH Specification of Copying Garbage Collection". School of Comp. Sci., Carnegie Mellon University, CMU-CS-92-219 (1992).
- [NS93] Ngo, T.A. & Snyder, L. "Data Locality On Shared Memory Computers Under Two Programming Models". Dept. of Computer Science and Engineering, University of Washington, UW-CSE-93-06-08 (1993).
- [Org72] Organick, E.I. **The Multics System: An Examination of its Structure**. MIT Press, Cambridge, Mass. (1972).
- [Pau91] Paulson, L.C. **ML for the Working Programmer**. Cambridge University Press (1991).
- [PFG96] Peyton Jones, S.L., Gordon, A. & Finne, S. "Concurrent Haskell". In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, Florida, USA (1996) pp. 295-308.
- [Plo75] Plotkin, G.D. "Call-by-name, Call-by-value and the λ -Calculus". *Theoretical Computer Science* 1 (1975) pp. 125-159.
- [PRT93] Pierce, B.C., Rémy, D. & Turner, D.N. "A Typed Higher-Order Programming Language Based on the Pi-Calculus". In *Proc. Workshop*

on Type Theory and its Applications to Computer Science, Kyoto University, Japan (1993).

- [PT94] Pierce, B.C. & Turner, D.N. "PICT user manual" available at <http://www.cs.indiana.edu/hyplan/pierce/ftp/pict> (1994).
- [Rep90] Reppy, J.H. "Asynchronous Signals in Standard ML". Dept. Computer Science, Cornell University, TR 90-1144 (1990).
- [Rep91] Reppy, J.H. "An Operational Semantics of First-class Synchronous Operations". Cornell University, TR 91-1232 (1991).
- [Rep92] Reppy, J.H. "Higher-Order Concurrency". Ph.D. Thesis, Cornell University (1992).
- [Rep94] Reppy, J.H. "First-class Synchronous Operations". In *Proc. Theory and Practice of Parallel Programming*, Japan (1994) pp. 235-252.
- [Rep95] Reppy, J.H. "Semantics and blasting". Personal communication by (September 15, 1995).
- [RR96] Reppy, J.H. & Riecke, J.G. "Classes in Object ML via Modules". In *Proc. Third International Workshop on Foundations of Object-Oriented Languages*, New Jersey, USA (1996).
- [RV96] Rémy, D. & Vouillon, J. "Objective ML: A simple object-oriented extension to ML". In *Proc. Third International Workshop on Foundations of Object-Oriented Languages*, New Jersey, USA (1996).
- [San92] Sangiorgi, D. "Expressing Mobility in Process Algebras". Ph.D. Thesis, University of Edinburgh (1992).
- [SC91] Siegel, E.H. & Cooper, E.C. "Implementing Distributed Linda in Standard ML". School of Comp. Sci., Carnegie-Mellon University, CMU-CS-91-151 (1991).
- [SDV+94] Skjellum, A., Doss, N.E., Viswanathan, K., Chowdappa, A. & Bangalore, P.V. "Extending the Message Passing Interface (MPI)". In *Proc. 1994 Scalable Parallel Libraries Conference* (1994) pp. 106-118.
- [Sits95] Sitsky, D. "Implementing MPI Using Interrupts and Remote Copying on the AP1000/AP1000+". In *Proc. Fourth Fujitsu Parallel Computing Workshop*, London, UK (1995) pp. 323-333.
- [SKI+93] Shiraki, O., Koyanagi, Y., Imamura, N., Hayashi, K., Shimizu, O., Horie, T. & Ishihata, H. "Architecture of Highly Parallel Computer AP1000+". In *Proc. Third Parallel Computing Workshop*, Fujitsu PCRF, Kawasaki, Japan (1993) pp. P1-G1-8.

- [SML96] Lucent Technologies and Bell Laboratories. "Standard ML'96" <http://cm.bell-labs.com/cm/cs/what/smlnj/sml96.html> (1996).
- [Ste78] Steele, G. "Rabbit: a compiler for Scheme". MIT, Technical Report AI-TR-474 (1978).
- [Ste96] Steckler, P. "Detecting Local Channels in Distributed Poly/ML". Laboratory for Foundations of Computer Science, University of Edinburgh, ECS-LFCS-96-340 (1996).
- [Sto94] Stonebraker, M. **Readings in Database Systems**. Morgan Kaufman Publishers (1994).
- [Sun95] Sun Microsystems Inc. "The Java Language Environment White Paper". Sun Microsystems Inc., available at http://www.javasoft.com/doc/language_environment (1995).
- [Tho95] Thomsen, B. "A theory of higher order communicating systems". *Information and Computation* 116, 1 (1995) pp. 38-57.
- [TLG92] Thomsen, B., Leth, L. & Giacalone, A. "Some Issues in the Semantics of Facile Distributed Programming". European Computer-Industry Research Centre, ECRC-92-32 (1992).
- [TLP+93] Thomsen, B., Leth, L., Prasad, S., Kuo, T.-M., Kramer, A., Knabe, F. & Giacalone, A. "Facile Antigua Release Programming Guide". European Computer-Industry Research Centre, ECRC-93-20 (1993).
- [TMC+96] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R. & Lee, P. "TIL: A Type-Directed Optimizing Compiler for ML". In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, USA (1996) pp. 181-192.
- [TMS+96] Tridgell, A., Mackerras, P., Sitsky, D. & Walsh, D. "AP/Linux - initial implementation". Dept. Comp. Sci., Australian National University, TR-CS-96-07 (1996).
- [Tof88] Tofte, M. "Operational Semantics and Polymorphic Type Inference". Ph.D. Thesis, Edinburgh (1988).
- [Tof90] Tofte, M. "Type inference for polymorphic references". *Information and Computation* 89 (1990) pp. 1-34.
- [Ull93] Ullman, J.D. **Elements of ML Programming**. Prentice-Hall (1993).
- [Und95] Underhill, A.H. "Implementing the Hough Transform on a Distributed Memory Multiprocessor using a Parallel Functional Language". M.Comp.Sci. Thesis, La Trobe University (1995).

- [vEC92] von Eicken, T., Culler, D.E., Goldstein, S.C. & Schauser, K.E. "Active Messages: a Mechanism for Integrated Communication and Computation". In *Proc. 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia (1992) pp. 256-267.
- [WF91] Wright, A.K. & Felleisen, M. "A Syntactic Approach to Type Soundness". Dept. Computer Science, Rice University, TR91-160 (1991).
- [WFH+93] Wing, J.M., Faehndrich, M., Haines, N., Kietzke, K., Kindred, D., Morrisett, J.G. & Nettles, S. "Venari/ML Interfaces and Examples". School of Comp. Sci., Carnegie Mellon University, CMU-CS-93-123 (1993).
- [Wil95] Wilson, G.V. **Practical Parallel Programming**. MIT Press (1995).
- [Wir77] Wirth, N. "Modula: a language for modular multiprogramming". *Software – Practice and Experience* 7 (1977) pp. 3-35.
- [Wir83] Wirth, N. **Programming in Modula-2**. Springer-Verlag (1983).
- [WWW+94] Waldo, J., Wyant, G., Wollrath, A. & Kendall, S. "A Note on Distributed Computing". Sun Microsystems Laboratories, SMLI TR-94-29 (1994).
- [ZM90] Zdonik, S.B. & Maier, D. **Readings in Object-Oriented Database Systems**. Morgan Kaufmann (1990).