Progressive Messages: Tracking Message Progress Through Events

Christopher C. Aycock

Christ Church

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2011

Abstract

This thesis introduces the Progressive Messages model of communication. It is an event-driven framework for building scalable parallel and distributed computing applications on modern networks. In particular, the paradigm provides notification of message termination. That is, when a message succeeds or fails, the user's application can capture an event (often through a callback) and perform a designated action.

The semantics of the Progressive Messages model are defined as an extension to the message-driven model, which is like an asynchronous RPC. Together, these models can be contrasted to the message-passing model (the basis of Sockets and MPI), which has no event notification.

Using Progressive Messages allows for a more scalable design than permitted by either the message-passing or message-driven model. In particular, Progressive Messages can handle communication concurrently with computation, which means that one process does not need to wait in order to service a request or response from another process. This overlap leads to more efficiency.

As part of the study of Progressive Messages, we create the MATE (Message Alerts Through Events) library, which is a prototype API that supports event notification in communication. This API was implemented in both MPI and InfiniBand *verbs* (OpenFabrics). "Unit tests" of network metrics shows that there is some latency in event-driven message handling, though it is difficult to determine if the source of the latency is hardware or software based.

The goal of the Progressive Messages model is that parallel and distributed computing applications will be easier to build and will be more scalable.

Acknowledgements

I would first like to thank my supervisor, Richard Brent. His insightful guidance during my studies is matched only by his infinite patience. He also secured a visiting fellowship for me at The Australian National University, which allowed me to complement my Oxford experience.

I'd also like to thank Bill McColl, who kept in touch with me even after leaving the university to pursue his startup. An hour of his feedback was equivalent to a week in the library.

From the Department of Computer Science, I'd like to thank Tom Melham, who took-over as my official supervisor when Richard and I left for ANU. His ability to juggle so many administrative tasks while maintaining his own research agenda will always keep me in awe. Furthermore, I'd like to thank the attic crowd (Bruno Oliveira, Daniel Goodman, Rui Zhang, Jolie de Miranda, Edward Smith and Penelope Economou) for their support over the years.

At ANU, I owe my gratitude to Peter Strazdins for his suggestions on my research. He welcomed me as a member of the department straight away. I also would like to thank the HPC group (Jin Wong, Pete Janes, Joseph Antony and Stuart Watson) for their friendship during my time there.

I can't ignore the generous support of the American Friends of Christ Church, whose scholarship provided funding for my studies. And finally, I must give a shout-out the Housemen (Charles Fox, Akshay Mangla, Jason Lotay, Dan Koch and Raphael Espinoza), who made it all worthwhile.

Contents

1 Introduction **1**

1.1 Contributions . . . 2

1.2 Organization . . . 2

2 Background and Related Work **4**

2.1 Definitions . . . 4

2.2 Networks . . . 5

2.2.1 Kernel-Bypass, Zero-Copy and Asynchronous Communication 6

2.2.2 Remote Direct Memory Access . . . 8

2.2.3 Special Features . . . 9

2.2.4 Shared-Memory Innovations . . . 10

2.3 Messages . . . 11

2.3.1 Message-Passing Platforms . . . 12

2.3.2 Message-Driven Platforms . . . 13

2.4 Applications: Parallel Computing . . . 14

2.4.1 Shared-Memory Abstractions . . . 15

2.4.2 Partitioned Global Address Spaces . . . 17

2.4.3 Other Languages . . . 18

2.5 Applications: Distributed Computing . . . 18

2.5.1 File Systems . . . 19

2.5.2 Databases 20

2.5.3 Web Services 20

2.6 Summary 21

3 Formalising Existing Messaging Systems **23**

3.1 Message-Driven Semantics 23

3.2 Message-Passing Semantics 25

3.3 Equivalence of Message-Passing and Message-Driven Semantics 27

3.4 Example: Scientific Computing 28

3.4.1 Message-Passing Approach 29

3.4.2 Message-Driven Approach 30

3.5 Example: Order Manager 30

3.6 Summary 31

4 Tracking Message Progress **33**

4.1 Progressive Messages 33

4.1.1 Syntax and Semantics 34

4.1.2 Axioms 36

4.2 Examples 37

4.3 Real-World Progressive Messages: AJAX 38

4.4 Summary 39

5 Design and Implementation **40**

5.1 MATE Specification 40

5.1.1 Events, Handles and Schedules 40

5.1.2 Process Management 42

5.1.3 Communication 43

5.1.4 Progress 45

5.1.5 Miscellaneous 45
5.2 MPI Implementation 46
5.2.1 One-sided Communication 47
5.2.2 A Note About Multiplexing 48
5.3 OpenFabrics Implementation 48
5.3.1 Receiving Data 49
5.3.2 Sending Data 50
5.3.3 Progress Thread 50
5.3.4 Timeout Thread 51
5.4 Summary 51

6 Performance Results 53
6.1 Simulating Scalability 53
6.1.1 Message-Passing Scalability 55
6.1.2 Event-Driven Scalability 56
6.1.3 Results 58
6.2 Network Metrics 59
6.2.1 Bandwidth 59
6.2.2 Latency 60
6.2.3 CPU Utilization 61
6.2.4 Ping Tests 63
6.2.5 Other Tests 63
6.2.6 Results 64
6.3 Summary 65

7 Conclusion 75
7.1 Future Work 76
7.2 Final Thoughts 76

A Dynamic Typing in Remote Memory Access **78**

A.1 The Ruby Bindings of MPI-2 78

A.1.1 Window Creation 79

A.1.2 One-sided Communication 79

A.1.3 Attribute Caching 82

A.1.4 Synchronization 83

A.2 Design and Implementation 84

A.3 Examples 86

A.4 Comparing RMA Speeds of Ruby and C 87

Bibliography **90**

Chapter 1

Introduction

Parcel services allow customers to track their packages and see when the transportation enters a new state of progress, such as completion of delivery or refusal by the recipient. Such facilities add peace of mind and allow carrier clients to better handle unforeseen events. Knowledge of message progress is essential to building robust communication frameworks.

This thesis is about Progressive Messages, a model of communication for computer networks. At the heart of this model is the ability to track a message's termination (success or failure). The user, like the shipping company's customer, relies on alerts to follow the message and to carry out contingency plans.

Associated with this model is a set of simple semantics to describe communication systems. As such, Progressive Messages is a tool for building communication software based on the premise that user knowledge of message progress leads to better scalability.

The work in this thesis arose from the need to address inadequacies in current communication tools. Users traditionally have had to "design around" libraries to handle unexpected messages. Because such issues are application-specific, a library that provides more flexible communication primitives is ideal. The techniques for such a library are captured here as Progressive Messages.

A couple of patterns are observable in communications software for high-end systems. First is that data transfer procedures are rarely coordinated and that the system's organization is only very loosely coupled. The second is that the system is mostly managed by the user, rather than by the computer itself. While these methods may degrade performance slightly or introduce extra complexity, they lead to better processing of unexpected events.

Within the Progressive Messages framework, events represent changes of progress for a message, such as submission or arrival. The user, upon request, is alerted once

an event occurs. The user may then handle this event in whatever manner is best suited for his application. The entire mechanism follows from the above observations. Unlike existing communications models, the Progressive Messages framework does not assume static message patterns.

1.1 Contributions

The aim of this thesis is to make communications programming more expressive and scalable. Towards this goal, we introduce the Progressive Messages model of communication. This is an event-driven framework where message termination[1] can be observed by the user application. The semantic definition for this framework is an extension to the message-driven model of communication. We can contrast event-driven communication to the message-passing model, but because the the precise meaning of both message-passing and message-driven is hazy in the literature, this thesis identifies its own semantics for each.

In comparing the relative performance of all three communications models, we show via simulation that Progressive Messages scales better than message-driven or message-passing simply because it can handle communication concurrently with computation. We also believe that Progressive Messages is more expressive for designing parallel and distributed computing applications.

The thesis further introduces a prototype API known as MATE, which supports Progressive Messages communication. MATE was implemented in both MPI and directly on InfiniBand *verbs*, and this thesis explains the design details for these implementations.

Lastly, this thesis covers a vast amount of the literature in communications frameworks, modern networks, and applications. The survey is significant and provides a holistic view of the problem space.

1.2 Organization

This thesis begins with Chapter 2, which outlines the preliminaries of modern networks and the current trends in message-based communication. One of the primary trends is towards asynchronous, event-driven communication. The chapter introduces the two most common communication frameworks, which are message-passing (which is used in Sockets) and message-driven (which acts as an asynchronous RPC). This

[1] The name "Progressive Messages" came from the goal of tracking message progress. In this thesis, the scope of progress has been limited to success or failure of the message, but we include some investigation of other state changes. The MATE API can track when a message has timed out or when the sending buffer has become reusable.

second framework is the springboard for our notion of Progressive Messages. Several features in modern networks also point towards event-driven communication, and we investigate these features in some detail. Lastly, the chapter covers some applications in parallel and distributed computing.

Chapter 3 covers the message-passing and message-driven frameworks in greater detail. Literature does not have a precise definition of what these are, so the chapter introduces its own semantics inherited from CSP and MPI. We then cover how a few computing applications are programmed with these two models. The sample application of dynamic load balancing will be referenced in later chapters when exploring issues with scalability and expressiveness.

Chapter 4 presents the Progressive Messages model. This model is an expansion of the message-driven paradigm to include progress notification for several state changes that a message may go through. This thesis contends that providing these event alerts to user applications makes communications programming more expressive and scalable. The chapter closes by covering AJAX, a programming methodology widely used in web development that is very similar to Progressive Messages. The concepts in this thesis were developed independently and around the same time that AJAX became known, and it is useful to see that there are real-world applications using this.

We then turn to Chapter 5, which presents an implementation of the Progressive Messages model as a prototype API called MATE. Unlike AJAX, Progressive Messages is intended for high-performance networks, and so MATE has been implemented using both MPI and InfiniBand *verbs* (OpenFabrics). The chapter explains the design used for the implementation and references literature for other high-performance messaging APIs.

Chapter 6 then investigates the performance of the Progressive Messages model. The chapter first presents scalability simulations of the three messaging paradigms (message-passing, message-driven and Progressive Messages) given a dynamic load-balancing application. Because a program designed with Progressive Messages can respond to work requests at anytime, communication is concurrent with computation, which leads to better scalability. The chapter also covers "unit test" performance metrics for MATE. There is added latency to using an event-driven model, which this chapter investigates.

Finally, Chapter 7 summarizes the main points in this thesis. Progressive Messages was conceived as an extension to message-driven semantics to take better advantage of modern networks when scaling parallel and distributed applications. We conclude with some potential future work to further advance this model.

Chapter 2

Background and Related Work

This chapter introduces terms and concepts that will be used throughout the thesis. Section 2.1 briefly defines common vocabulary to remove background ambiguity. Section 2.2 presents features in the area of modern high-performance networks that have attracted notice among software developers. Chief among these features is user-level, zero-copy, asynchronous communication. Section 2.3 then describes existing message-based communication models with a view towards efficient use of modern network features. In particular, it introduces the message-passing and message-driven models, which will be detailed in later chapters. Finally, Sections 2.4 and 2.5 present two primary applications of networks: parallel and distributed computing.

This chapter is important as it discusses the core concepts that lead to the Progressive Messages model, which ultimately tries to bridge the gap between applications and modern commodity hardware.

2.1 Definitions

Today's high-performance computing systems usually employ multiple processors to perform different tasks simultaneously. These machines may be classified by the manner of communication that processors use among themselves. In a "shared-memory" model, memory is accessible by all processors via read / write primitives. This is different from a "distributed-memory" arrangement, in which processors have their own private memory and thus explicitly communicate with each other, usually through messages. The shared-memory vs. distributed-memory classification can describe both hardware (physical partitioning of memory) and software (partitioning of the address space). Thus it is possible for a machine to have distributed-memory hardware, and shared-memory software; this special case is usually referred to as "distributed shared memory" or occasionally "virtual shared memory".

Given that the above taxonomy only classifies computers using their inter-process communication, it may be useful to describe machines based on latency differences. In a "symmetric multiprocessor" (SMP), the amount of time required to load a word from memory is constant, regardless of where in memory that word is stored. The opposite is the "non-uniform memory access" (NUMA) architecture, in which the time to fetch may vary from one memory region to another. These classifications are entirely hardware-based because they rely on performance characteristics. Software may be identical on either machines as both SMPs and cache-coherent NUMAs (ccNUMAs) are shared memory.

Software shared-memory systems are programmed with "threads", which represent independent steps of execution within a single address space. The communications medium is usually a set of global variables. In contrast, as mentioned earlier, software distributed-memory systems have multiple address spaces and thus rely on messages. The message is an abstraction of communication (similar to a file's being an abstraction of IO) and contains meta-data about the communication. This thesis covers messages.

2.2 Networks

A network is the physical infrastructure that hosts communication among processors in a distributed-memory environment. The most common network in use today is Ethernet [20, 38, 93, 116], though there are many others that may be more appropriate for high-end applications.

The Virtual Interface Architecture (VIA) [27, 49] is an abstract model of user-level zero-copy communication (see Section 2.2.1) that may be used to enhance existing networks or design new ones. One VIA-inspired network is the InfiniBand Architecture (IBA) [111], which was devised to connect almost *everything* within a computing system and thus replace Ethernet, Fibre Channel, PCI, etc. While its no-legacy goal has not panned out [106], InfiniBand has gained ground within the high-end interconnect market. In addition to hardware, the IBA standard lists a set of "verbs", functions that must exist. Originally the syntax of these verbs was left to the vendor, but the OpenFabrics Alliance eventually created an open-source software stack known as the OpenFabrics Enterprise Distribution (OFED) [65]. This API will be covered in more detail in later chapters.

Another VIA-inspired network is the Internet Engineering Task Force (IETF) Remote Data Direct Placement (RDDP) standard, more commonly known as the Internet Wide Area RDMA Protocol (iWARP) [44, 92]. It is an update of the RDMA Consortium's RDMA over TCP standard [107], which specifies zero-copy transmission over legacy TCP (see next section). Because the kernel implementation of the TCP stack is a tremendous bottleneck, a few vendors now implement TCP in hardware. As simple data losses are rare in tightly coupled network environments, the

error-correction mechanisms of TCP may be performed by software while the more frequently performed communications are handled strictly by logic embedded on the network interconnect card (NIC). This additional hardware is known as the TCP offload engine (TOE) [10, 42].

TOE itself does not obviate copying on the receive side, and must be combined with RDMA hardware (see Section 2.2.2) for totally zero-copy results. The iWARP specification is a set of different wire protocols intended to be implemented in hardware (though it seems feasible to emulate them in software for compatibility without the performance benefits). The main component is the Data Direct Protocol (DDP), which permits the actual zero-copy transmission. DDP itself does not perform the transmission; TCP does. However, TCP does not respect message boundaries; it sends data as a sequence of bytes without regard to protocol data units (PDU). In this regard, DDP itself may be better suited for SCTP. To run DDP over TCP requires a tweak known as marker PDU aligned (MPA) framing so as to guarantee boundaries of messages.

Furthermore, DDP is not intended to be accessed directly. Instead, a separate RDMA protocol (RDMAP) provides the services to read and write data. Therefore, the entire RDMA-over-TCP specification is really RDMAP over DDP over MPA over TCP. All of these protocols are intended to be implemented in hardware. In sum however, iWARP may be thought of as the principles of InfiniBand applied to Ethernet.

While InfiniBand and iWARP are gaining acceptance in the community, the VIA model of communication hardly represents the only means of obtaining higher performance in a network. One of the more popular interconnects is Myrinet [17], which has been around for far longer than the Virtual Interface Architecture. Another mature network standard is the Scalable Coherent Interface (SCI) [64], which is unique in that it specifies its topology as a ring instead of the more common switch (see Section 2.2.3). A third, newer, interconnect is QsNet [102].

The following subsections explain many of the concepts present in modern networks. Not all networks have every feature described below, nor do they all agree on a best approach. This thesis will advocate some methods over others because of their impact on the messaging techniques presented in Chapter 4.

2.2.1 Kernel-Bypass, Zero-Copy and Asynchronous Communication

As mentioned above, kernel implementations of transmission protocols such as TCP are a performance bottleneck in communication. However, traditional networks require the kernel for protection because the network is a shared resource among the processes. For user-level NIC control, the network must somehow be privatised for each process.

One of the classic developments in computer science is virtual memory, a combination of hardware and software that creates the illusion of private memory for each process, among other benefits. In the same school of thought, a virtual network interface protected across process boundaries could be accessed at the user level. With this technology, the “consumer” manages his own buffers and communication schedule while the “provider” handles the protection.

Thus, the NIC provides a “private network” for a process, and a process is usually allowed to have multiple such networks. The virtual interface (VI) of VIA refers to this network and is merely the destination of the user’s communication requests. Communication takes place over a pair of VIs, one on each of the processing nodes involved in the transmission. Other networks have different terminology but essentially the same concept: the user manages his own buffers in “kernel-bypass” communication.

In addition to user-level communication, all of the high-end networks have “zero-copy” communication. In traditional networks, arriving data is placed in a pre-allocated buffer and then copied to the user-specified final destination. Copying large messages can take a long time, and so eliminating this step is beneficial. Another classic development in computer science is direct memory access (DMA), in which a device can access main memory directly. Regarding networks, a DMA-enabled NIC can fetch outgoing data directly from the user’s memory buffer without waiting for the OS to first copy the data to a kernel-level pool. Thus, zero-copy and kernel-bypass features are entwined.

An additional benefit of DMA is that the CPU is free to perform other tasks while the NIC accesses the memory. Because the speed of light is constant and because the internode connector (the network cable) is longer than the intranode connector (the system bus), *all* remote memory accesses will take longer than local memory accesses. Thus regardless of any tweaks to the NIC or driver, there will always be communication latency. Therefore, some applications overlap communication with computation to “hide” message latency.

Yet another classic development in computer science is the cache, and particularly prefetching to the cache [100]. With prefetching, a processor may request a word from main memory before it is needed; the processor is able to continue working while the data is retrieved in the background. Similarly, high-end networks allow calls for communication before the effects are required.

The user normally submits his request for remote data to a queue on the NIC. VIA-inspired networks have separate queues for send and receive requests that form “queue pairs”; other networks make no distinction between requests to send instead of receive. However the request is lodged, the user must alert the NIC—VIA refers to this action as pressing the “doorbell”—ordinarily by setting a memory-mapped register. (Most programming interfaces actually combine the actions of enqueueing and alerting so that the sequence is accomplished in one stroke.) The processor may continue with its workload once the request has been acknowledged.

The user then has three options to determine when the request has been fulfilled. The VIA networks have a completion queue that the user may investigate; other networks have a similar mechanism. For these kinds of checks, the user may either poll the NIC for status, or block until the completion of the request. The third option is to associate a callback function to be invoked upon completion. Depending on the network, the callback may be implemented as a thread or as an interrupt. Regardless of the method used to ensure completion, once the request has been fulfilled, the results of the communication will be valid.

Kernel-bypass, zero-copy, asynchronous communication is usually presented as a single package in modern networks because of the overlap in requirements for hardware support. There are, however, some tradeoffs among these features because of the way they are implemented [71]. Given such diversity in network capability (see Section 2.2.3), it is useful to measure the performance of the three characteristics. Latency measures the benefits of user-level messages, bandwidth demonstrates zero-copy results, and CPU overhead shows how much communication and computation can overlap. A fuller explanation of these metrics is available in Section 6.2.

2.2.2 Remote Direct Memory Access

The above capabilities are represented via two-sided communication: one side explicitly sends while another side explicitly receives. Two-sided communication does have a drawback in that only the sending node experiences zero-copy transmission; the receiving node must intermittently copy the data through a buffer for the simple reason that the receiving NIC does not know where the data is ultimately to be placed. That is, the application must give the NIC the address of the destination buffer.

With “remote direct memory access” (RDMA), the sending node provides the receiving node with its destination memory address; at no point does the receiving node explicitly (from the application’s perspective) handle the message. This one-sided communication is modeled after classic load / store instructions and thus permits zero-copy benefits through DMA on the receiving node.

RDMA is not a panacea [59]. For a NIC to access data through DMA, the user’s buffer must actually be in memory and not paged-out to the disk. The VIA-inspired networks require that the page be “pinned-down” to physical memory through a memory registration, which invokes the kernel. That is, the operating system is actually required for RDMA in VIA, at least for initial communication.

Furthermore, to ensure that only the process that owns the registered memory may access it, the VIA NICs require permission keys known as “protection tags” during communication. The sending node must have this protection tag before communication can even begin, which means that the RDMA network must rely on two-sided communication as a bootstrap. Of course the protection tag would ideally be cached on the sending node. For numerous protection tags (if there are many communication

partners over the network, and each has a large number of buffers), the cache to store these tags may become quite large.

There is a more subtle problem as well [6]. The Ohio State University's series of papers on InfiniBand programming [73, 86, 83, 84, 90] represent a set of design patterns for VIA. Design patterns are best-practice architecture that permit reuse of a solution to a common programming problem [56]. Design patterns are also a sign that the underlying language is incomplete [94]. After all, a pattern implies automation, and automation implies a machine. That is, VIA does not provide explicit support for many common communication cases.

QsNet provides RDMA without VIA's troubles. All remote memory is directly accessible by the remote process's virtual address. Protection for this network occurs through swapping process ids during job startup. Thus, protection is for the whole process, not just regions of memory. Also, there is no memory pinning with QsNet; the NIC will simply hold received data until the page is swapped into memory.

QsNet is able to provide these features by means of a kernel patch. These modules are developed for very specific versions of the kernel based on assumptions regarding the Linux API. Given this level of required specificity, administering a cluster that involves kernel patches can be quite tedious.

Whether through kernel patches or user intervention, there is currently no commodity RDMA network that does not possess serious downsides for the end user. Therefore, this thesis does not rely on RDMA.

2.2.3 Special Features

Some high-performance networks include unique features that may be of interest to software developers. While the Progressive Messages model does not require any of these characteristics, a few interesting properties will be presented here as they did inspire some subtle aspects of the model.

One very important mechanism for communication reliability is the acknowledgment, in which a recipient confirms the arrival of data. If the sender does not receive an acknowledgment within a certain amount of time, it may try to resend the message (possibly through another path [68]). Networks that are more resilient—able to recover quickly in the event of a packet loss—have higher bandwidth, especially in error-prone environments [76]. All of the networks described here provide acknowledgments directly in hardware rather than require them from software. What distinguishes some of these networks is the necessity of connections (in the TCP sense) to obtain reliability.

For example, VIA is connection-oriented; the user must have linked two VIs—one local and one remote—prior to any communication. A VI is exclusive to a connection, and

a connection is exclusive to a pair of NICs. (The same is true for QsNet, though the connection setup is not as explicit.) That is, a node must have an established connection for each concurrently open communication partner in a network, and an established VI for either endpoint of each connection. All of this information is stored in memory, either on the NIC or with the host node.

It would appear that this potentially vast amount of data—the information required for the connections between all pairs—would actually lead to less reliability [101]. If a remote node is reset, then the local records must be reconfigured. And given the storage requirements of this set-up, it would appear connection-oriented communication is less scalable as well.

For this reason, the InfiniBand standard offers a "reliable datagram" service that features acknowledgments but does not require connections. While not widely used, the ability to communicate reliably without the overhead of connections appears to be very beneficial. Note that the iWARP standard offers no such datagram service as TCP is inherently connection-oriented.

Regarding physical connections, most modern networks link nodes together through a switch, a central hub that routes data to its final destination. The use of a switch allows for easy scalability as new nodes are added to the network merely by plugging them in. QsNet's switch, in addition to point-to-point routing, offers hardware support for collective communication, thereby giving greater scalability for operations such as one-to-many broadcasts and many-to-one reductions [39, 40, 110]. InfiniBand also has multicasting through its "unreliable datagram" service, but only for data that can fit within a single message transmission unit (up to 4KB).

The use of a switch, however, presents a single point of failure, though this risk may be reduced through the inclusion of redundant switches. A greater limiting factor in the use of switches is the length of the cable when connecting a distant node. A solution to these problems is to not use a switch at all, which is precisely what the SCI standard dictates. All nodes in this network are placed in a ring in which only neighbouring nodes are connected together. SCI allows for rings of rings, such as two-dimensional or three-dimensional torus topologies. This set-up is intended for better reliability and scalability in very large computing systems.

2.2.4 Shared-Memory Innovations

Though this thesis is dedicated to message-based communication, it may be helpful to review the state of shared-memory systems. Some trends here are growing rapidly and will undoubtedly have an impact on distributed-memory computing. Consider, for example, traditional CPU architecture.

Decreasing transistor sizes have lead to a diminishing Moore's Law. Chips must consume ever more energy to yield slightly better performance. This increased con-

sumption further leads to increased heat generation. Simply decreasing the transistor size is no longer viable for future growth.

A solution to this issue is the multicore processor, in which many CPUs fit onto one die [58, 60]. This arrangement allows for multiprocessing with a single chip, thereby making parallel hardware more accessible in the market. Given the recent widespread adoption of multicore processors, it is likely that even consumer desktop software will be written as multithreaded applications [48].

Of course, massively multicore chips can lead to the potential for bus saturation. The solution to this problem is not to have a bus at all, but rather to embrace a direct-connect architecture through a link like HyperTransport (HT) [3], in which components are connected point-to-point. HT has been the subject of some other supercomputing innovations, such as the InfiniPath network [47].

Despite the name, InfiniPath is not InfiniBand (the two are merely "compatible"). The NIC features high-speed memory that is mapped into a user-level process. The NIC notices when an application accesses this mapped memory and then performs the communication. Thus, it relies on the host node's CPU to copy the data and perform any additional set-up.

This architecture of course violates the observations in Section 2.2.1. Indeed, while InfiniPath can move many small messages in a short time period (by relying on the sheer speed of the CPU), it permits no overlap of communication and computation, and therefore provides no performance benefits for large messages [21]. The InfiniPath vendor's business plan relied heavily on the adoption of multicore CPUs in the hopes of making this issue moot.

2.3 Messages

All of the networks discussed here have their own lower-level programming interfaces. Myrinet has both MX (Myrinet Express) and GM (Glenn's Messages). QsNet has Elan and the associated Tports. SCI has SISCI (Software Infrastructure for SCI). VIA has VIPL (VI Provider Library). Even the Open Group has defined its own API: RDMA-Capable NIC Programming Interface (RNIC-PI) [108].

Obviously, software written directly for a lower-level interface is not portable. Therefore, it is beneficial to have an upper-layer protocol (ULP) to support applications. An early transport-independent API is the user Direct Access Programming Library (uDAPL) [45], defined by the DAT Collaborative. This ULP provides much of the same functionality of VIPL, but substitutes notions like queue pairs and virtual interfaces with the more generic "endpoints" and "asynchronous communication". Additionally, uDAPL provides some quality-of-service controls as this feature is natively present in InfiniBand [30]. Borrowing heavily from uDAPL is the Open Group's

Interconnect Transport API (IT-API) [70], which includes unreliable datagram communication, another InfiniBand feature.

While these new APIs may be suitable for new applications, Sockets still have a major place in application development given the vast amount of existing software based on the legacy interface. (Also of note is that the new APIs require an external mechanism for bootstrap, which usually ends up being Sockets anyway.) To maintain backwards-compatibility with legacy applications, both InfiniBand and iWARP define the Sockets Direct Protocol (SDP). SDP is a byte-stream oriented transport protocol (`SOCK_STREAM`) similar to TCP, but allows for exploitation of RDMA devices. Each socket in SDP corresponds to a single queue pair.

In some SDP implementations, the software may only require a re-link to use the new interconnects. For better exploitation though, the Open Group has defined the Extended Sockets API (ES-API) [115] with functions that handle asynchronous communication on RDMA networks. This API merely adds a few new subroutines to traditional Sockets.

What is important to note here is that in all of the variety of networks and their APIs, one abstraction is consistent: messages. Messages are the dominant abstraction of communication, regardless of the platform or application. The following subsections detail two communications paradigms involving messages. Chapter 4 will propose an alternative model that captures many of the benefits of modern networks.

2.3.1 Message-Passing Platforms

The most straightforward means of communicating with messages is to simply pass data from one node to another. The basic procedure involves both nodes, in which one actively sends while the other receives. Because this two-sided model is represented in the Sockets API and is available on many networks, the message-passing paradigm has become ubiquitous and is commonly used for its portability. A standard API available for parallel computing using this model is the Message-Passing Interface (MPI) [62, 113].

MPI also offers one-sided communication in which the user requests data to be placed in or retrieved from a remote node's address space. These semantics (which mimic get / set accessor methods) map directly to RDMA network capabilities as described earlier. Thus, in theory, an RDMA-enabled NIC can offload the communication and thereby have higher performance. To achieve the full potential, the user must overlap his communication and computation by taking advantage of MPI's asynchronous functions; this assumes that the MPI implementation supports independent progress, whether through the use of threads or interrupts [23].

While MPI has many features, it does not encompass all of message passing. For example, the ARMCI library [95] can perform one-sided communication of non-

contiguous data, similar to the POSIX functions `readv()` and `writev()`. Furthermore, MPI requires synchronization across processing nodes for seemingly trivial actions, such as memory registration [19]. The GASnet [18] library can perform one-sided communication without such restrictions.

One other remote memory access library worth mentioning is Sandia Portals [22]. In this system, each process maintains a queue of events. An event is generated whenever a message enters a new state of progress, such as submission, completion or failure. The events are entered into the queue, which the user may check to determine message progress. The goal is that Portals allows for independent progress and thus better overlap of communication and computation. The Progressive Messages model presented in Chapter 4 similarly uses event-driven progress monitoring, albeit through callbacks rather than a queue that must be checked explicitly.

As well as point-to-point communication, MPI defines functions for collective communication, useful for bulk synchronous operations. This mode of transmission is usually implemented as coordinated point-to-point message transfers, though it can be mapped to special network hardware for better performance. This thesis does not require collective communication.

2.3.2 Message-Driven Platforms

Even the least restrictive message-passing systems lack the functionality many applications require [24, 25]. Ordinarily the remote node is not alerted (at the user level) when a message arrives. Thus, the user must probe for incoming messages, which violates the principle of independent progress. A better approach is for the user to preordain a function to handle the arrival of a message. In the message-driven model, an incoming message specifies which handler to execute and may include any parameters for the handler [63].

The arrival of a message triggers an interrupt or alerts a thread, thus invoking the handler (for this reason, a precursor to the message-driven paradigm was known as the "Actor model"). The handler is a small function that runs to completion; it is usually atomic and may not be interrupted. In many implementations, it may not block or busy-wait (to prevent deadlock).

There are a number of message-driven implementations, the most common being the Remote Procedure Call (RPC) [41]. RPCs encapsulate the communications infrastructure so that the user believes he is making local function calls. The remote handler returns a value back to the calling process by means of a message. Both messages are blocking, which prevents overlap of computation and communication.

The Active Messages library [122] is asynchronous. The handler incorporates the message's data and may respond with a result, but the sending node does not block while waiting for a response. Active Messages was originally created to reduce packet

processing overhead, and thus allow applications to take full advantage of a high-performance network. It was designed for library writers and was never intended for users.

The Charm++ [75] extensions to C++ allow for parallel object-oriented programming. Here a "chare" object represents a basic unit of parallelism. The public methods of a chare object may be invoked remotely and asynchronously through a pointer-like structure known as a "proxy". The Charm++ runtime system will, for each processor, nondeterministically choose a pending message and invoke its associated method.

Regardless of idiosyncrasies, message-driven frameworks are better able to handle unexpected communication. Indeed, a variant of Active Messages serves as the basis for both GASNet and ARMCI when the underlying network hardware does not natively support RDMA. The reason is that get / set accessors for abstract data types are usually implemented as functions anyway, and these accessors act over a network.

This thesis advocates event notification for handling unexpected messages. The Progressive Messages model extends this concept to all facets of communication, not just message receipt.

2.4 Applications: Parallel Computing

Parallel computing refers to running a similar job on multiple processors simultaneously. The software is often written in a single-program, multiple-data (SPMD) format, in which the same process executes on different processors with different input. Usually these processes must communicate or synchronize during execution to ensure correctness or improve efficiency. Many times the communication operations are performed collectively in that all processes are involved in the bulk exchange of messages.

As of this writing, the most popular means of parallel programming involves explicit communication because user-directed messages are portable and flexible, and thus may be used in many situations [35, 61]. Furthermore, distributed-memory programming solves one of the difficulties of concurrent software development: nondeterminacy [79]. Shared-memory programs may require mutex locks or condition variables, which present their own troubles as indiscriminate use can lead to deadlock.[1]

However, explicitly writing parallel software is difficult for reasons that include the manual division of tasks or data, the job initialization and startup, synchronization, and the exchange of data [121]. Furthermore, distributed-memory software lacks the data structures to have a continuously updated shared state. That is, tasks

[1] There is some active research in the area of lock-free synchronization, particularly with "transactional memory", but this work has yet to lead to production-level systems.

that are heavily dependent on each other would do well to have a central location for globally required information, which implies that they would benefit more from shared-memory models. And to be fair, nondeterminism is not always a requirement for concurrency, as witnessed by VLIW processors.

This section covers a few tools that automate some parts of parallel programming. These compilers and languages manage the common cases, thus abstracting out the network and freeing the user to develop higher-level software. While a few special-purpose languages feature radically different syntax and semantics, most are extensions of C or Fortran because of the volume of legacy technical code written in those two languages. This allows to user to parallelise his sequential code in incremental, easy-to-debug steps [74].

Parallelising compilers and languages do have downsides though. In abstracting out the network, they remove direct control of communication, thereby preventing optimisations of certain special cases. Today's parallelising compilers also introduce their own overhead. The user must weigh these shortcomings against the benefits of higher-level software development.

2.4.1 Shared-Memory Abstractions

Most high-performance compilers aim to parallelise loops to speed-up technical codes. Automatic parallelisation is possible but extremely difficult because the semantics of the sequential program may be changed. Therefore, most users provide some advice to the compiler. A very common method is to use a standard set of directives known as OpenMP [31], in which the user expresses which sections of the code are to be `parallel` via a "pragma".

Ordinarily with OpenMP, the software is executed serially by a single master thread. Upon reaching a `parallel` loop, slave threads are spawned to perform some iterations. When the loop completes, the threads are rejoined to the master, which continues executing alone and sequentially. An important trait of this master / slave model is that the program's structure is independent of the system resources. There are a few rules about what can be parallelised though. For example, the loops must have a deterministically countable number of iterations. That is, the exact number of iterations must be determined before the loop executes.

Within the loop, the user specifies the scope of the variables. Variables may be `shared` across all threads (typically global variables), or may have `private` allocations within each thread (normally local variables). The only communication between two threads then is through one of these shared variables. By default, OpenMP will privatise the index of the outermost loop and leave all other variables as shared. Correctness is left to the user.

To handle race conditions, the user expresses where `critical` sections are. These

sections are abstractions for mutex locks and are usually sufficient for synchronization. Dealing with dependencies, however, is left to the user.

OpenMP has a few mechanisms for handling load imbalance. By default, the scheduling of loop iterations among the threads is performed statically. However, the user may request that the `schedule` be `dynamic`, in which case a fixed-size chunk of iterations is assigned at runtime. A `guided` self-scheduling using a varying chunk size is also available.

While these features make OpenMP a great tool for SMPs, because there is no means for directing data placement—either statically or dynamically—OpenMP has poor performance on NUMAs [33]. Some vendors have added language extensions that resemble High Performance Fortran's directives for data distribution (see below). Others have experimented with sophisticated page migration routines that move data closer to the needing thread. None of these schemes are standard as yet.

There have also been recommendations to privatise as much data as possible and only use shared data to communicate between threads. But one has to wonder whether this simply degenerates to programming like MPI. For now, there is no clear solution to using OpenMP beyond SMPs.

In contrast to OpenMP's task-driven model, in which the user specifies the distribution of iterations among processors, "data-parallel" programming allows users to specify the distribution of arrays among processors. Only those processors owning the data will perform the computation. In OpenMP's master / slave approach, all code is executed sequentially on one processor by default; in data-parallel programming, all code is executed on every processor in parallel by default.

The most widely used standard set of extensions for data-parallel programming are those of High Performance Fortran (HPF) [77]. With HPF, a user declares how to `DISTRIBUTE` data among abstract processors, usually in a `BLOCK` or `CYCLIC` fashion, the former intended for applications with nearest-neighbour communication, and the latter for load-balancing purposes. Additionally, the user may also `ALIGN` data elements with each other. The elements of arrays that have thus been mapped will be assigned to exactly one processor; all other (non-mapped) data is copied to each of the processors.

To parallelise a loop, the user declares iterations as being `INDEPENDENT`. Data within the loop that has been given the `NEW` attribute will remain private in scope; all other data is copied to each processor. Correctness is left to the user.

While HPF has been designed for NUMAs, its performance in practice has been unpredictable [104]. The reason does not appear to be the communication, but rather the data manipulation that occurs before and after the communication. It is important here to observe that the performance impact of modern high-performance networks is actually less of an issue compared to the software's overhead.

Another key observation to make at this time is that of the "optimisation envy" of the parallelising-compiler groups. Just as some OpenMP users seek to exploit NUMAs, there are a few HPF users who hope to achieve better performance on SMPs [14]. One approach has been to include `DYNAMIC` and `GUIDED` attributes for the `SHARE` clause associated with `INDEPENDENT`. The compiler then produces multithreaded object code, rather than message-passing code. An important difference with this approach from the HPF standard is that non-mapped data is owned by a single master thread and is globally accessible by all other threads. The performance benefit of this approach is that there is no software translation of the global address (a major source of overhead during runtime) as the hardware already provides this support.

2.4.2 Partitioned Global Address Spaces

To reconcile the parallelism in both tasks and data, a different programming model has emerged that claims to have the best of both HPF and OpenMP. It is the partitioned global address space (PGAS) model, and has been implemented in a variety of languages, the most widely used being Unified Parallel C (UPC) [29].

In UPC, by default, all variables are private and every instruction is executed by each process. However, the user may declare that some data be `shared` and may specify its distribution in either block or cyclic form. Furthermore, the user may divide the iterations of a parallel loop either based on data ownership (like HPF) or in a thread-specific manner (as in OpenMP). UPC also has synchronization mechanisms and allows the user to choose between `strict` or `relaxed` memory consistency.

The benefit of PGAS is that communication with remote memory resembles a simple assignment statement within the programming language. Furthermore, the PGAS compilers are simpler than those for data-parallel languages because the level of sophistication is lower. (To that end, there is even a portable PGAS library, known as Global Arrays (GA) [96], that requires no special compiler.)

While UPC's constructs appear to be well-suited for parallel programming, its performance has not lived up to expectations as of yet [15, 28, 34]. Again, a major cause of concern is the address translation. Ordinary pointer manipulation requires integer arithmetic; manipulating the pointers in UPC require much more computation. The runtime system will not know where a `shared` array's elements lie until after the translation has been made, even if the element in question resides in local memory. The performance studies indicate that the time required to access a locally stored element through a `shared` array is close to the time required to retrieve a remotely stored element! To access the locally stored element in a smaller amount of time, the user must cast the element as `private`.

Given these results, one might wonder if an SMP or a ccNUMA might be a better target platform for UPC. A study on the matter did demonstrate substantially better access times when forgoing translation in favour of simple loads and stores [12].

Therefore, PGAS languages may be best suited for multicore CPUs [7].

2.4.3 Other Languages

There are many other programming environments and paradigms for parallel computing. This section gives a brief overview of some newer or lesser known languages. No independent analysis of these have been found in the literature, thus performance results will not be listed.

ZPL [114] is an array-based parallel language in which the regions of an array, rather than the array's individual elements, are the target of operation. ZPL's higher level of abstraction allows for a special optimisation where the compiler generates its own loop iteration order. The constructs are also easier to parallelise because the communication requirements are more deterministic.

Because the primary use of parallel computers is to speed-up calculations, it makes sense to provide a parallel mathematical environment. Star-P [37] is such a platform. Originally part of an attempt to parallelise MATLAB, Star-P is now an independent middleware that has very high-level numerical computing constructs. Again, this level of abstraction aids the compiler in parallelising the user's desires.

An approach to bring parallel array-based programming to MATLAB has resulted in Hierarchically Tiled Arrays (HTAs) [55]. An HTA is an array whose elements are grouped together in tiles. Tiles themselves may be grouped together in tiles. Tiles do not need to be the same size; the user defines how the array is partitioned and how the partitions may be partitioned and so on. The compiler may then parallelise the program by holding the inner tile locally and accessing the outer tiles remotely. This partitioning scheme resembles data-parallel programming and maintains the ability to operate on whole chunks of the array.

2.5 Applications: Distributed Computing

Distributed computing ensures that access to resources across a network remains transparent. Most often the objective is to improve reliability or provide a means for retrieving shared data, though increased performance may also be an aim. Given the abstraction requirements, the lowest-level software device tends to be the RPC, mentioned in Section 2.3.2.

An RPC appears to the user much like a normal function call. However, the function may reside in the address space of a process different from the caller. The RPC library handles the data encoding and the message passing, hiding the details of these actions from the user. An existing procedure need only be registered as a service to be remotely invoked.

The RPC system relies on the client / server model, in which the caller consumes the resources that the server provides. In this manner, distributed software is most often written in multiple-program, multiple-data (MPMD) form, which simplifies the handling of unexpectedly arriving messages.

Communication within a distributed system does tend to be slower than in parallel systems [32]. The MPMD model relies on threads (at least logically), which leads to overhead from creation, synchronization and switching. Also, a message is not considered complete until it has been serviced, though this is a concern with two-sided message passing as well. For some applications, MPMD execution may actually be faster overall as the threads will hide latency even if the communication is slower. Furthermore, distributed systems may be more responsive by providing a mechanism for immediate callbacks, rather than relying on the user's polling. These are some trade-offs that must be evaluated for each application.

The following sections cover a few of the more common applications in distributed computing.

2.5.1 File Systems

A distributed file system presents users with a single logical file system even if storage resources are physically located across a network [81]. Remote directories may be mounted into a machine's local name space. By using a consistent naming scheme, it is possible to have the same user directories present on any workstation in a network, thereby providing mobility for the end user.

A distributed file system must be provided by a server, though this machine may in fact be a client of another file system. Clients may access remote files just as they do local ones; the server performs all file operations independently. The file system is implemented as software, either in the kernel or as a user-level service, by means of RPCs to handle *create*, *read*, *write* and *delete* operations.

For performance, the file is often cached on the client, similar to how open local files are buffered in a conventional file system. Of course, if two clients modify the same file, then cache coherence becomes an issue. A server in a "stateful" file system maintains information about clients, which allows the server to alert the clients that a change has been made and that the buffers must be flushed. A server in a "stateless" file system maintains no information; each action from the client is self-contained. The benefit here is that the client can crash with no side-effects to the server. Likewise, the server can even reboot with minimal impact on the client.

The most common file system is Sun's Network File System (NFS) [117], which usually serves as the baseline when considering other architectures. In addition to its use as a workgroup filesystem, NFS has been extended to be a back-end for application servers. By mapping RPCs to RDMA transports like the Virtual Interface

Architecture, NFS can have high performance [26]. Furthermore, the Direct Access File System (DAFS) [46] extended the NFS protocol with RDMA to include richer semantics, such as shared file locks that can recover if the node that owns the lock has crashed. The RDMA of course requires that the memory be registered, but the server may be able to cache commonly accessed regions [51].

There are filesystems that exist beyond NFS. Newer architectures may use more recent communications infrastructures; Lustre [124] aims for greater scalability in clustered systems by using Sandia Portals.

2.5.2 Databases

A distributed database stores and processes information on a number of nodes [13]. The architecture for these systems can be distinguished by how data responsibilities are shared among the processors.

In a "shared-nothing" architecture, the partitioning is such that each compute node owns a subset of the data. That is, a node will operate exclusively on a particular subset. Scalability of course depends on wise partitioning. The physical partitioning may be that a compute node places its data portion on its local disk, and that all nodes are connected via a high-speed network. Such a scheme—which resembles a storage area network—reduces fault tolerance as a node crash makes a portion of the database inaccessible. Therefore, the physical partitioning may actually share the disks so that one node can "take over" in the event that another node crashes. This is approach that IBM's DB2 favours [16].

In a "shared-everything" (or "shared-disk") architecture, any compute node can operate on any portion of the database. The physical layout usually involves network-attached storage, in which all nodes communicate with a separate, shared disk via a high-speed interconnect. Again the logical partitioning is key for scalability. The obvious downside to a shared-everything architecture is that there may be enormous contention when numerous nodes attempt to communicate with the disk simultaneously. Oracle RAC builds a shared cache on each of the compute nodes by using the network to maintain coherency [98].

Both DB2 and Oracle RAC use uDAPL to communicate among nodes in the server, and the Sockets Direct Protocol to communicate with the client node.

2.5.3 Web Services

Most PCs connected to the Internet today are underused, which means that their computing powers could be shared and combined to form a supercomputer on-the-fly. SETI@home [2, 78] uses many volunteer computers around the world to process

observations made in the sky. The program works over HTTP to get around firewalls and supports two types of communications from clients: requests for more work, and solutions to previous work requests. There is no communication between clients. Because observations in the sky are independent from one another, the work is easy to parallelise. This model of computing is generally known as "grid" computing [53] and is available through software packages such as Globus [54] and Condor [82].

Given the restrictions on inter-client communication, this model of computing is ill-suited for applications that cannot be divided into individual units. However, the "Web services" used in grid computing and other Internet applications do provide an interesting case study. SOAP[2] [80] is an RPC protocol that communicates with XML over HTTP. While flexible, it has a number of performance drawbacks: the XML encoding / decoding and the need to reestablish the TCP connection for each SOAP message [103].

A competing model of Web services programming is Representational State Transfer (REST) [50], in which HTTP's own facilities are used to specify a service. For example, the URL identifies the desired resource while the method (`POST`, `PUT`, `GET`, or `DELETE`) specifies the desired action. In analogy to remote memory access as described earlier, SOAP is like relying on GASNet or ARMCI, whereas REST is like having RDMA capabilities directly. Of course, neither of these systems are expected to have nearly the performance required in a parallel computing environment, and thus their respective overheads are usually acceptable.

One final Web communications paradigm worth noting is AJAX (Asynchronous JavaScript and XML) [57], which relies on the XMLHttpRequest API. The client machine can initiate an HTTP message with a server (usually through REST) and request that a callback be invoked when the status of communication changes. This indeed is the essence of the Progressive Messages model. The material in this thesis was developed independently of Web services and is aimed at high-performance computing applications. However, AJAX's success in production-level software already provides strong evidence for event-driven message tracking.

2.6 Summary

This chapter introduced the starting concepts for this thesis. We focused on messages since that is the most portable and low-level communication primitive. Parallel computing applications rely on messages in distributed-memory systems. While there are higher-level frameworks such the partitioned global address space model, the best performance outside multicore chips has been recorded from explicit communication. Likewise, distributed computing applications like file systems and databases hide

[2]The W3C has considered the definition "Service Oriented Access Protocol", but officially SOAP stands for nothing.

resource partitioning from the user, but still require robust messaging for implementation.

A few of the modern network features presented in this chapter will form the assumed platform for this thesis. User-level communication handles messaging without the need of the kernel. Zero-copy communication further provides communication without intermediate copying, and often accompanies user-level messaging. Finally, asynchronous communication permits independent message progress, which overlaps communication handling with the user's computation needs. Together, these features form the messaging infrastructure that this thesis will assume is present.

The messaging paradigms presented in this chapter bridge the gap between applications and networks. Message-passing primitives like Socket's `send()` and `recv()` are ubiquitous and easily usable for single-threaded applications. Message-driven semantics, like remote procedure calls, are available for applications that require concurrency. This thesis takes the view that neither of these models fully capture how parallel or distributed applications are developed, nor do these models map well to modern network features. This belief will be explained in Chapter 3.

Chapter 3

Formalising Existing Messaging Systems

The previous chapter discussed the tangible systems available to us that use and transmit messages. This chapter identifies some common theoretical constructs in messaging models through formal semantics. We will need these semantics in the next chapter to describe Progressive Messages and its relationship to existing platforms. Because there is no standard formal definition of message-passing or message-driven models, we must make our own.

Section 3.1 presents the message-driven semantics and Section 3.2 presents message-passing semantics. The message-driven model is presented first simply because its syntax is better defined in the literature. Section 3.3 will be shown that message-driven semantics can be defined in terms of message-passing and vice-versa. Sections 3.4 and 3.5 will present a few common software examples and how existing semantics are used to encode these examples. This chapter should ideally give motivation for a different way of handling communication, which will be revealed in the next chapter.

3.1 Message-Driven Semantics

As mentioned in Section 2.3.2, message-driven frameworks operate by invoking a handler whenever a message arrives. Many of these frameworks, like Active Messages and Charm++, are represented by asynchronous RPCs. The user specifies what a process is supposed to do when a particular message arrives.

To inject some formalism into our message-driven semantics, we can look towards process algebras. A process algebra [8] is a framework for reasoning about concurrency. It is a labeled transition system that describes the behaviours that occur as

a result of interaction in concurrent systems. A process is represented as a directed graph where the nodes are states and the edges are transitions. In this respect, the transition system is similar to the state machines of automata theory, but it does not necessarily have finitely many states or transitions.

There are many process algebras. Unlike MPI's ubiquity in message-passing, there is no one algebra that has risen above the others. So for our purposes, we may as well just pick one and carry on. We will identify the process algebra Communicating Sequential Processes (CSP) [67, 109] as the representative of message-driven semantics.[1]

The simplest CSP process is $STOP$. There is no progress beyond a $STOP$ and the process does not communicate.

Now consider a process P that triggers when event a occurs:

$$a \rightarrow P$$

This is read "a then P" and is known as "prefixing". Thus, it is possible to define a complete process that halts when an action occurs:

$$P = a \rightarrow STOP$$

In the context of a service, an application defined this way could only ever handle one event before shutting down. CSP allows recursion in process definitions:

$$P = a \rightarrow P$$

Thus, a server that responds to one request at a time can be defined as:

$$SERVER = request \rightarrow response \rightarrow SERVER$$

CSP has a choice operation to define alternative event possibilities:

$$(a \rightarrow P \mid b \rightarrow Q)$$

So we can extend the server definition to explicitly state what happens during an unrecoverable error:

$$SERVER = (request \rightarrow response \rightarrow SERVER \mid error \rightarrow STOP)$$

[1] In particular, CSP has gained in popularity recently as the basis for the concurrency model in the Go programming language.

CSP has no global variables; each process has local variables and and can perform boolean and arithmetic operations. Processes must communicate with each other, and it is CSP's communication primitives that are the primary interest of this section. Sending a value v over a channel c and continuing as process P is represented by:

$$c!v \rightarrow P$$

Likewise, receiving a value v over a channel c and continuing as process P is represented by:

$$c?v \rightarrow P(v)$$

Suppose our server from above implements a function f and communicates via a bidirectional socket. Then a perpetual request/response is indicated by:

$$\mathit{SERVER} = \mathit{socket}?x \rightarrow \mathit{socket}!f(x) \rightarrow \mathit{SERVER}$$

3.2 Message-Passing Semantics

Message-passing frameworks are often not as sophisticated as message-driven semantics, but they are important to cover since almost all communication is implemented via message passing at the lowest level of the stack. As described in Section 2.3.1, message-passing frameworks simply have one node send data while another node receives it.

For the purposes of this thesis, we define message-passing semantics as the Cypher-Leu semantics [43], which is the formal basis of the Message-Passing Interface (MPI). Cypher-Leu does not have algebraic syntax, so we define our own to match the syntax from the message-driven semantics.

Given a value v and a channel c, the following non-blocking communication primitives are available in Cypher-Leu:

Post Send	$r := c \Uparrow v$		
Post Recv	$r := c \Downarrow v$		
Wait	$\triangleright(r)$	$\equiv$	while $\neg \triangleleft (r)$ do SKIP end
Poll	$\triangleleft(r)$	$\equiv$	$\begin{cases} \mathit{true} & \text{if request has completed} \\ \mathit{false} & \text{otherwise} \end{cases}$

The r above is a "request object", which uniquely identifies the communication attempt. *SKIP* is a process that terminates immediately and successfully. Unlike with *STOP*, further progress is allowed following a *SKIP*.

We will add the sequential composition operator from CSP. This process behaves like P initially; when P terminates, the process behaves like Q:

$$(P; Q)$$

There are two properties that come directly from the above definitions ($\equiv$ means indistinguishable):

$$(SKIP; P) \equiv P \equiv (P; SKIP)$$

$$(STOP; P) \equiv STOP$$

Consider this example process, which begins a communication, completes a series of steps, and then finishes the communication:[2]

$$P = r := c \Uparrow v; P_1; P_2; \cdots; P_n; \triangleright(r)$$

Message-passing semantics allow for wild cards in the receive channel, which means that a message may be received on any channel. The following is perfectly reasonable (the specific channel with the message is returned by the wait primitive):

$$r := \Downarrow v; \cdots; c := \triangleright(r)$$

For the sake of clarity, message-passing semantics include blocking communication:

Send	$c \uparrow v$	$\equiv$	$r := c \Uparrow v; \triangleright(r)$
Recv	$c \downarrow v$	$\equiv$	$r := c \Downarrow v; \triangleright(r)$

Combining the above two receive operators, we have blocking communication with a wild-card channel (the specific channel is returned when the message is received):[3]

$$c := \downarrow v$$

[2] This is similar to a "superstep" in the Bulk-Synchronous Parallel model of computation.

[3] This is the mechanism for multiplexing, like `select()` in the Sockets API.

3.3 Equivalence of Message-Passing and Message-Driven Semantics

The models introduced in the previous sections present very different mechanisms for receiving data. (It should be clear that sending data is equivalent when using message-passing's blocking semantics.) The message-driven paradigm triggers a handler when a message arrives, whereas the message-passing paradigm requires the user to explicitly receive the message in the thread that needs it. Interestingly though, either model can be implemented in terms of the other.

The primary use case for message-driven programming is a service handler:

$$(c_1?v_1 \rightarrow P_1(v_1) \mid c_2?v_2 \rightarrow P_2(v_2) \mid \cdots \mid c_n?v_n \rightarrow P_n(v_n))$$

Each channel represents a separate process that will be invoked upon message receipt. This fits naturally within the framework of a process algebra, which as mentioned above is simply a labeled transition system. The state transitions are easily represented by the receiving channel.

To perform service handling within the context of a procedural message-passing model, we will need a look-up table to map a given channel to the desired process.

$$\mathit{table.register}(c_1, P_1); \mathit{table.register}(c_2, P_2); \cdots ; \mathit{table.register}(c_n, P_n)$$

With the table ready, we can perpetually invoke the correct process when a message arrives. We will use the blocking receive with a wild card:

$$c :=\downarrow v; P := \mathit{table.lookup}(c); P(v)$$

This method of implementing message-driven in terms of message-passing is straightforward. The converse, however, is much more difficult because message-driven handlers are assumed to be non-blocking. In a single-threaded dispatch like above, blocking on one handler will cause the entire system to become unresponsive. (The dispatch could be multi-threaded, though that is unlikely in practice because the number of threads is unbounded.)

The technique for absorbing the blocking process is via stack ripping [1]. The user must establish a continuation to store his state before invoking an asynchronous wrapper to an otherwise blocking routine. Upon completion of the routine, the user's state will continue. Thus, the handler is split into two or more sub-handlers that must reference a global state.

Given that either model can be implemented in terms of the other, the message-passing and message-driven models are equivalent in capability. Expressiveness within a programming language is a different issue.

3.4 Example: Scientific Computing

Given the two existing semantics for messaging systems, we will now investigate how common coding problems are expressed in each model. Consider the example of parallelising a loop in a scientific program. Many technical codes feature a loop that iterates over a load of work. The load may be a matrix whose elements are needed for arithmetic, or a graph whose edges are traversed. In general, such a loop resembles:

```
main loop:
    perform unit of work from load
```

Here we assume that there is a "main loop" for the program, and that the load contains divisible units of work that each require the same amount of time to compute. All units are independent and may be executed in any order.

If an even division of the work load is possible, then this loop may be run on multiple processors simply by using static distributions of the load as inputs. This straightforward parallelisation technique—using different units from the load as inputs for different processors—is quite effective for classes of regularly divisible problems.

On the other hand, if the problem to be executed is "irregular", meaning that the work load cannot be evenly divided, then the software must feature dynamic load balancing. When one processor's load runs low, it must ask another processor for a portion of its load [123]. (This assumes that all tasks are independent and may be executed on any processor in any sequence.) The intended result is that over the course of the software's execution, the loads will eventually become equal. This scheme is entirely decentralized and does not depend on the topology of the network; it is portable, scalable and fault-tolerant, not to mention simple.

Beyond load balancing, a second and more serious issue is failure within the system. A quick running program (where the result is available before the mean-time between component failure) might run as-is. But if the software is to run for much longer, then there must be some coordinating communication among the processors. In a fault-tolerant system, there may be no central server; thus it is reasonable to assume that all processing nodes are peers. Therefore, the coordinating communication must take place between two processors [36].

In summary, dynamic load balancing and fault-tolerance are required to achieve scalability and reliability. The remainder of this chapter considers how message-passing and message-driven approaches attempt to achieve these goals.

3.4.1 Message-Passing Approach

With a message-passing library like MPI, the user can simply send a request for more work to a randomly selected peer. A processor then needs to poll the network to check for such a request from other nodes. If such a request has come through, then the processor must respond with a portion of its own work load. And finally, upon receiving a response with more work, a node must incorporate the data into its own load. A sketch of the algorithm appears as follows:

```
main loop:
    perform unit of work from load (if any)                    2

    if load size is below a threshold:                         4
        loop until request ack has been received:
            randomly select a node                             6
            send a request for more work to node
                                                               8
    if there is a request for more work on network:
        send a response with portion from load                 10
        if response ack has been received:
            discard portion                                    12

    if there is a response with more work on network:          14
        incorporate work into load
```

When a processor is ready to request more work, it must make sure that the randomly selected peer is still alive. Hence, the processor must send requests to different peers until one replies with an acknowledgment (line number 5 above). The acknowledgment is different from the response with more work; it is merely a receipt that the message had been received by the remote NIC. Many modern networks with independent progress are able to send the acknowledgment automatically. Because the user need not worry about sending an acknowledgment, that step is not in the algorithm.

An acknowledgment is also needed before a portion of work may be discarded (line number 11). When a node is ready to respond with a portion from its load, it must make sure that the originally requesting node has not crashed. Thus, the portion will only be deleted if an acknowledgment comes through.

It is also worth noting that a requested node may respond with an empty portion of work. Because incorporating an empty portion does nothing to change the load (line number 15), the load size will still be under threshold and thus the processor will continue to make requests. Also, because a processor must incorporate any portion received, an erroneous response—an unrequested portion of work—will not lead to a loss of work.

While the message-passing approach is able to provide fault tolerance, it lacks scalability. The delay in waiting for acknowledgments prevents a processor from carrying out other tasks. Furthermore, the processor will be unable to handle arriving messages when it is performing its own work (line number 2). The lack of "as-needed" responsiveness greatly increases the latency between recognizing that the load is too small, and obtaining a portion to incorporate into the load.

3.4.2 Message-Driven Approach

In message-driven systems like Charm++, the user specifies handling of messages prior to their arrival. Rather than polling the network for messages explicitly, a processor expects to be alerted so that it may respond to communication "as needed". The algorithm follows:

```
upon receipt of request message:                          1
    send a response with portion from load
    if response ack has been received:                    3
        discard portion
                                                          5
upon receipt of response message:
    incorporate work into load                            7

main loop:                                                9
    perform unit of work from load (if any)
                                                          11
    if load size is below a threshold:
        loop until request ack has been received:         13
            randomly select a node
            send a request for more work to node          15
```

Here, the processor will perform its own work and only handle requests or responses when needed. The result is greater scalability than the message-passing approach. However, it still suffers from the delay in waiting for acknowledgments, and thus does not provide optimum performance.

3.5 Example: Order Manager

Financial services companies such as stock-brokerage firms require an order manager to route customer purchases to the stock exchange. This software must be able to send orders and receive acknowledgments or execution reports. It also must have a mechanism for noticing that an outstanding order message has not been acknowledged so that the trader can take necessary action. All of this order management must take

place concurrently with other data that might be coming in. For example, the order manager might have to keep track of stock prices that are streaming alongside the execution reports.[4]

Consider this message-passing implementation:

```
main loop:                                                        1
    if there is a new price on the network:
        incorporate the new datum                                 3

    if ready to send new order:                                   5
        send order
                                                                  7
    if there is an acknowledgment or execution report:
        clear outstanding order                                   9

    if an order is pending for too long:                          11
        cancel the order and handle error
```

This involves a lot of polling, so a message-driven approach may be more attractive:

```
upon acknowledgment or execution report:
    clear outstanding order                                       2

main loop:                                                        4
    if there is a new price on the network:
        incorporate the new datum                                 6

    if ready to send new order:                                   8
        send order
                                                                  10
    if an order is pending for too long:
        cancel the order and handle error                         12
```

While this absolves explicitly checking for the order report, the user must still check for the latent (unacknowledged) orders.

3.6 Summary

This chapter described the two existing messaging frameworks in greater detail. The message-passing model uses explicit send and receive operations to handle commu-

[4]This is not merely theoretical; in practice, traditional order managers receive streaming market updates via UDP and simultaneously communicate their orders with the exchange via the FIX Protocol [52], which is based on TCP.

nication, whereas the message-driven model relies on callbacks to handle message receipt. This chapter presented message-driven semantics as a subset of CSP and message-passing semantics as an algebra for the Cypher-Leu model.

This chapter also presented two example applications, namely work-stealing in scientific loops and a financial order manager. The message-driven approach presented here improved upon the message-passing model when computing irregular problems in an unreliable environment. Both however suffered a delay when handling acknowledgments. Some users have circumvented this latency by employing threads, but such practice is non-standard as general-purpose intraprocess concurrency is not provided by these models.

The ideal communication primitives would therefore provide explicit support for concurrency. This implies that the send-side semantics must be non-blocking. (The receiving side is already non-blocking in the message-driven paradigm since the action is triggered by message arrival.) Further, we can mirror the message-driven model by stating that the sending side is alerted when the message successfully arrives at its destination. But we can take this a step forward still to say that the sending side is also alerted in the event of an unrecoverable failure. In fact, the sending and receiving sides both can be alerted anytime the progress of the message has changed. This notion is the basis of the Progressive Messages model, which will be presented in the next chapter.

Chapter 4

Tracking Message Progress

From the previous chapter, we saw the opportunity for a new communication paradigm that relies on event-driven progress monitoring. The rationale behind this philosophy stems from many motivating factors. The first is performance gained through entirely asynchronous communication, which makes it possible to completely overlap communication and computation. Another benefit is portability, since user-managed buffers are essential to many modern high-performance networks. A third is ease of use, which improves programmer productivity for large-scale applications.

Event-driven programming has been available in GUI APIs for a long time for similar reasons.[1] Indeed, this thesis takes the stance that increasing application complexity in the high-performance computing domain mirrors the rise of GUIs. If the message-passing paradigm is analogous to command-line interfaces (linear train of though), then an event-driven model of communication is similar to graphical user interfaces.

This chapter presents the Progressive Messages model of communication, which alerts the user's application of changes in message progress. Section 4.1 provides the algebra as an extension of the message-driven model with status notification. The benefits of this deceptively simple construct are demonstrated in Section 4.2, which recodes the examples from the previous chapter in a more evented notification scheme. Section 4.3 presents the AJAX web programming paradigm, which is a widely used model that is very similar to Progress Messages.

4.1 Progressive Messages

We begin with two simple primitives:

send (*message, destination, function*)

[1]Example: Update the menu when a particular button is pressed.

recv (*buffer, source, function*)

The *message* is the actual data to send; the *buffer* is where to store this data upon receipt. The *destination* / *source* references the remote process, such as IP address and port, or MPI node and tag. The *function* is the action to invoke when there is a change in progress, and can be a callback handle or a lambda definition.

These primitives are non-blocking; they merely schedule the communication on the underlying infrastructure. The function is invoked when the communication's state changes, particularly success, failure, "ready"[2] and timeout.

At first glance, the event-driven nature of this paradigm has some similarities with the message-driven model. Recall the algebra for receiving data in the message-driven semantics:

$$c?v \rightarrow P(v)$$

This makes the implicit assumption that the message was received correctly; it has no expression for error handling. Instead, the Progressive Messages model generalizes the state transition to include message progress:

$$(c?v \xrightarrow{success} P(v) \mid c?v \xrightarrow{failure} Q)$$

Likewise, sending data has progress monitoring as well. For example:

$$(c!v \xrightarrow{ready} P \mid c!v \xrightarrow{timeout} Q)$$

Thus, the CSP prefix notation is extended to include auxiliary state-change information that is specific to the desired event.

$$a \xrightarrow{\sigma} P$$

Conceivably any sequence of state change can be captured, though this thesis is only concerned with communication routines.

4.1.1 Syntax and Semantics

The terms of Progressive Messages in Backus-Naur Form is:

[2]Within the context of send-side messaging, "ready" occurs when the buffer is safe to reuse. This state is always complimentary with success, failure and timeout; however, it is an independent state on networks that copy the user-level data to another location before actually sending.

$$P := X \mid P; P \mid P' \mid \mathit{SKIP} \mid \mathit{STOP}$$

$$P' := a \xrightarrow{\sigma} P \mid P'|P'$$

We are only concerned with communication, and not any features that may be present in other process algebra. We now present operational semantics [105].

Sequential Composition

In a composite process $P; Q$, P is run to completion, at which point Q is run to completion. Note that P can still succeed even if its trigger represents failure in communication; this is intentional since P may be the failure handler!

$$\frac{P \Rightarrow P'}{P; Q \Rightarrow Q}[P \text{ completes}]$$

$$\frac{P \Rightarrow P'}{P; Q \Rightarrow P'; Q}[P \text{ does not complete}]$$

Prefixing

The process $a \xrightarrow{\sigma} P$ will maintain its state if there is no change in message progress.

$$\frac{}{a \xrightarrow{\sigma} P \Rightarrow a \xrightarrow{\sigma} P}[\sigma \text{ has not triggered}]$$

Once the communication event has triggered, the process will behave as P.

$$\frac{}{a \xrightarrow{\sigma} P \Rightarrow P}[\sigma \text{ has triggered}]$$

An event is triggered by communication progress.

Choice

Unlike full CSP, Progressive Messages only considers alternative ("guarded") commands. There are no other choice operators.

$$\frac{}{(a \xrightarrow{\sigma} P) \mid (a \xrightarrow{\sigma'} Q) \Rightarrow P}[\sigma \text{ has triggered}]$$

$$\frac{}{(a \xrightarrow{\sigma} P) \mid (a \xrightarrow{\sigma'} Q) \Rightarrow Q}[\sigma' \text{ has triggered}]$$

Given a choice of processes following communication progress, the triggered event determines which process will proceed.

Successful Termination

SKIP is effectively a no-op; its process terminates immediately.

$$\overline{SKIP; P \Rightarrow P}$$

Deadlock

STOP is a process that will not transition to any other state (ie, it does not terminate).

$$\overline{STOP \Rightarrow STOP}$$

4.1.2 Axioms

Choice is commutative.

$$P \mid Q \equiv Q \mid P$$

Choice is associative.

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

Choice is idempotent.

$$P \mid P \equiv P$$

Choice is right-distributive over sequential composition.

$$(P \mid Q); R \equiv P; R \mid Q; R$$

Sequential composition is associative.

$$(P; Q); R \equiv P; (Q; R)$$

4.2 Examples

Formally comparing expressiveness between Progressive Messages and message-driven semantics is almost impossible because the set of observable events is different [99]. Intuitively though, Progressive Messages provides extra capabilities because it can natively encode unsuccessful communication. Consider the examples from Chapter 3. Recall that the first goal was to build a fault-tolerant load-balancing loop that is free of delayed responses.

As in the message-driven approach (Section 3.4.2), the processor should respond immediately to a request for more work. The improvement that the Progressive Messages approach provides is that it does not wait for an acknowledgment. Instead, a completion event for the response triggers the portion to be discarded.

```
upon receipt of request message:
    send a response with portion from load                2

upon receipt of response acknowledgment:                  4
    discard portion
                                                          6
upon receipt of response message:
    incorporate work into load                            8

main loop:                                                10
    perform unit of work from load (if any)
                                                          12
    if load size is below a threshold:
        randomly select a node                            14
        send a request for more work to node
                                                          16
upon receipt of request failure:
    randomly select a node                                18
    send a request for more work to node
```

The Progressive Messages approach also takes advantage of the failure event to move secondary work requests out of the main loop (line 17). This means that communication and computation can completely overlap, while providing the fault tolerance and load balancing that had been our original goal.

The second example from Chapter 3 was an order manager (Section 3.5). We will modify the message-driven approach to add a handler for latent orders:

```
upon acknowledgment or execution report:                  1
    clear outstanding order
                                                          3
main loop:
```

```
    if there is a new price on the network:              5
        incorporate the new datum
                                                          7
    if ready to send new order:
        send order                                        9

upon timeout of outstanding order:                        11
    cancel the order and handle error
```

Again, by moving the latency check out of the main loop, we provide fault tolerance without the need for a CPU poll.

4.3 Real-World Progressive Messages: AJAX

As mentioned in Section 2.5.3, AJAX is a programming model based on the XMLHttpRequest API. It is often used to create a dynamic look-and-feel to a website, such as autocomplete forms or page refreshes without full reloading. The concepts in this thesis were developed independently and for a different domain (low latency applications), but it useful to see a widely used implementation of event-driven message handling.

Consider this JavaScript snippet, which presents an alert box containing any contents from a requested URL:

```
function loadDoc (url)
{
  // set callback to alertXml()
  xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = alertXml;

  // GET the url asynchronously
  xmlhttp.open ("GET", url, true);
  xmlhttp.send (null);

  // handler
  function alertXml ()
  {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200)
    {
      alert (xmlhttp.responseText);
    }
  }
}
```

The "status" is merely the HTTP return status, like 404 for file not found, or 200 for OK. The possible values for the "readyState" are

0	Uninitialized
1	Set-up
2	Sent
3	Receiving
4	Complete

This asynchronous communication prevents the client from locking up should the server not respond. It is also possible to build ad-hoc fault tolerance for the particular application. XMLHttpRequest has too high latency to be useful for high-performance computing applications because each communication in HTTP requires a disconnect. That means *every* message requires a new TCP handshake.

4.4 Summary

This chapter presented the Progressive Messages model, which uses a callback for progress monitoring. User applications can declare how to handle message state changes, which in theory leads to more expressiveness. The algebra presented here is an extension to message-driven semantics.

This chapter then provided solutions to some examples that message-passing and message-driven paradigms had trouble with. By associating a callback for timeouts or failures, the user can declare error-handling routines asynchronously. This should help with unreliable environments and irregular applications. This chapter closed with an overview of AJAX, a web programming paradigm that is very similar to Progressive Messages. The popularity of AJAX demonstrates a growing movement towards event-driven communication handling.

Just as event-driven programming has made intricate GUIs feasible, progress monitoring in Progressive Messages provides support for more sophisticated communication applications. User productivity and communication / computation overlap are enhanced directly by the simple concept of event notification. The remainder of this thesis explores the implementation of a library that supports Progressive Messages.

Chapter 5

Design and Implementation

The Progressive Messages model presented in the previous chapter provides a new framework for reasoning about communication. Because existing programming tools do not support this paradigm directly on high-performance networks, this thesis introduces MATE (Message Alerts Through Events), a library based directly on Progressive Messages. MATE was implemented twice, once using MPI and once using InfiniBand's OpenFabrics API directly. The implementations detailed in this chapter serve as a proof-of-concept for event-driven message notification.

Section 5.1 presents MATE's specification. Because this library was programmed more than once, it was helpful to know exactly what each implementation would need to do. Section 5.2 explains MATE's MPI implementation while Section 5.3 details the OpenFabrics implementation. The MPI version was much easier because of the higher-level routines. Indeed, the OpenFabrics version ran into a few difficulties, as detailed in the next chapter.

5.1 MATE Specification

The specification is primarily broken down into event handling, process management, communication routines, and message progress. The types, functions and constants are listed in Table 5.1.

5.1.1 Events, Handles and Schedules

MATE, intended for distributed-memory multiprocessor machines, generates an event whenever a message enters a new state of progress[1]. The user may choose to observe

[1]While Progressive Messages was conceived for monitoring termination, MATE was implemented with the view more progress notification is possible.

`mate_schedule_t`	`mate_startup()`	`MATE_ANYTAG`
`mate_receipt_t`	`mate_shutdown()`	`MATE_ANYSRC`
`mate_errcode_t`	`mate_mypid()`	`MATE_PROCMAX`
`mate_pid_t`	`mate_numprocs()`	`MATE_TAGMAX`
`mate_tag_t`	`mate_put()`	`MATE_SUCCESS`
`mate_callback_t`	`mate_get()`	`MATE_ERROR`
	`mate_send()`	
	`mate_recv()`	
	`mate_progress()`	

Table 5.1: Types, functions and constants for library

the events by whatever means are best suited for his application. Events within MATE represent completion, submission, timeout and failure.

Message "completion" occurs when the acknowledgment is received by the origin process; the user is guaranteed that delivery has occurred with this event. Specifically it means that the message has arrived at the remote node; it does not necessarily mean that the data is visible at the user level on the remote node. (There is also no further event if the remote node suddenly crashes after acknowledging receipt.)

A "submission" event indicates that the buffer on the origin process may be reused. As such, it is only valid for sending data; this event does not occur when receiving data. Submission does not indicate that the message has been received by the remote node. MATE guarantees delivery of a message if such is possible, so the underlying system must check for acknowledgments even after alerting the user that the buffer is reusable.

A "timeout" event means that an acknowledgment has not been received within a fixed amount of time after a message has been submitted (usually one second); it does not necessarily mean though that the message has failed to arrive at its destination. A timeout will cause the communication request to cancel on the local node. Even if the data subsequently reaches the remote node following a timeout alert, the completion alert will not be activated. Note that if the data has already been submitted, then there will be no way to stop it; the local node may record a timeout even if the remote node has received the data. If the user chooses *not* to handle a timeout, then the communication will only expire if it completes or fails. As with the submission event, the underlying system must check for acknowledgments even if the user chooses to ignore timeouts.

A "failure" occurs when it is known that the message will *never* reach its destination. Examples include inability to allocate resources during transmission and unconditional refusal to accept the message by the remote node.

These four events are matched against a callback function and a data buffer via a "schedule". Each communication requires a schedule, though any or all of the fields

may be blank if the user chooses to ignore a particular event. A schedule may be unique for a given instance of communication, or it may be shared among multiple communications. The schedule is of type `mate_schedule_t` and the callback function is of type `mate_callback_t`:

```
typedef void (mate_callback_t) ( mate_receipt_t *receipt,
                                 void *data );

typedef struct { mate_callback_t *completion_handle;
                 void *completion_data;
                 mate_callback_t *submission_handle;
                 void *submission_data;
                 mate_callback_t *timeout_handle;
                 void *timeout_data;
                 mate_callback_t *failure_handle;
                 void *failure_data; } mate_schedule_t;
```

The callback function will be invoked when the associated event is triggered for that particular message. The listed data buffer will be passed to this function, along with a receipt that contains information about the message. This receipt may also hold implementation-specific data. These receipts are of type `mate_receipt_t`:

```
typedef struct { mate_tag_t tag;
                 size_t length;
                 mate_pid_t local_pid;
                 mate_pid_t remote_pid;
                 void *local_address;
                 void *remote_address;
                 ...} mate_receipt_t;
```

The user may directly read the fields of a receipt to determine the message's tag, length, local and remote process id, and the local and remote buffer address. The user may not, however, overwrite the values in these fields.

5.1.2 Process Management

There are necessarily a few features within MATE to manage the computing resources. To launch the communications environment, the user must call `mate_startup()`. The processing nodes are specified via a separate interface such as an environment variable, system file, or command line option (the details of this interface can be found in the implementation's documentation). The `mate_startup()` function must be called exactly once and from the main thread before any other MATE functions may be invoked.

```
mate_errcode_t mate_startup ( int *argc, char ***argv );
```

To gracefully exit the communications environment, the user must call `mate_shutdown()` from the main thread exactly once. This function acts immediately; all outstanding communication is cancelled and any pending handlers may be lost. No other MATE functions may be invoked after this.

```
mate_errcode_t mate_shutdown ();
```

Both `mate_startup()` and `mate_shutdown()`, as with most functions in MATE, return an error code. The user may check this code to debug his program. The code, of type `mate_errcode_t`, represents incorrect parameters; system faults during communication usually trigger the failure event rather than pass back error codes.

```
typedef enum {MATE_SUCCESS,
              MATE_ERROR} mate_errcode_t;
```

The only functions that do not return an error code are for querying system state. For example, to determine the size of the computing environment user may request the number of processes with `mate_numprocs()`. The total number of processes will not be greater than `MATE_PROCMAX`. The type `size_t` is defined in `<stddef.h>`.

```
size_t mate_numprocs ();

#define MATE_PROCMAX ((size_t) ...)
```

Individual processes within the system are logically represented by a unique integer in the range

$$[0, mate_numprocs() - 1].$$

This representative integer is of type `mate_pid_t`. The user may query a process for its id via `mate_mypid()`.

```
typedef unsigned long mate_pid_t;

mate_pid_t mate_mypid ();
```

5.1.3 Communication

Beyond MATE's use of schedules, the message interface is fairly standard. All communication is point-to-point between two peers. The routines transfer contiguous data between two processes (which are allowed to be the same). A message is sent even if the specified buffer length is zero. A schedule must be specified, though any of the handlers may in fact be blank. The functions return "immediately" after the

user-specified receipt has been filled. It is not safe to reuse the buffers until an event has occurred.

The one-sided message functions transfer data without explicit involvement of the remote process. The user needs to only specify the virtual address on the remote node; the implementation will handle pinning and permissions independently of user involvement if such direction is required. Within the C language, the address may be represented in the array-offset format `&base[index]`.

The `mate_put()` function places data from a local buffer into a remote address space.

```
mate_errcode_t mate_put ( void *buffer,
                          size_t length,
                          mate_pid_t destination,
                          void *address,
                          mate_schedule_t *schedule );
```

The `mate_get()` function extracts data from a remote address space.

```
mate_errcode_t mate_get ( void *buffer,
                          size_t length,
                          mate_pid_t source,
                          void *address,
                          mate_schedule_t *schedule );
```

The two-sided message functions require some degree of synchronization between two processes. Rather than specify the remote address, the user lists a unique tag that is matched between both nodes. The tag is an integer of type `mate_tag_t` and may be any non-negative value not greater than `MATE_TAGMAX`.

```
typedef unsigned long mate_tag_t;

#define MATE_TAGMAX ((mate_tag_t) ...)
```

The `mate_send()` function submits data to a remote node.

```
mate_errcode_t mate_send ( void *buffer,
                           size_t length,
                           mate_pid_t destination,
                           mate_tag_t tag,
                           mate_schedule_t *schedule );
```

The `mate_recv()` function matches the tag against a submission from the remote node. Note however that the user may specify a wildcard by listing `MATE_ANYSRC` or `MATE_ANYTAG`. Upon completion, the wildcards will be filled with their true values, accessible in the receipt.

```
#define MATE_ANYSRC ((mate_pid_t) ...)
#define MATE_ANYTAG ((mate_tag_t) ...)

mate_errcode_t mate_recv ( void *buffer,
                           size_t maxlength,
                           mate_pid_t source,
                           mate_tag_t tag,
                           mate_schedule_t *schedule );
```

5.1.4 Progress

Communication functions in MATE initiate a message and then return without waiting for completion (or any other event). Thus, message progress occurs concurrently with the user's application. The order of progress among multiple messages is not guaranteed, and thus communication to and from overlapping buffers without some form of synchronization may produce unintended results.

Handlers are executed "eventually" after an event, not necessarily "immediately". The reason is that handlers are atomic with respect to each other and thus must be queued if more than one exists at a given time. The order of execution is not guaranteed. Handlers are not atomic with respect to the application; therefore, the user must be wary of race conditions. Handlers may not block or busy wait, and must run to completion. Handlers are intended to be small maintenance functions.

Message progress that occurs independently in the background frees the user from resource management. While this scenario is ideal, it is not always feasible as some network interfaces present no means for user-level interrupt handling. For these networks, MATE must poll for progress. A single-threaded implementation will require that the user call the library to progress further.

If the user does not need to call a particular function from the library, then he may invoke `mate_progress()`. This may be a no-op on systems with independent progress.

```
mate_errcode_t mate_progress ();
```

This function always returns `MATE_SUCCESS`. Any trouble encountered during a message's progress will invoke the failure handler, if provided.

5.1.5 Miscellaneous

The buffer—supplied as the first parameter to the communication functions—must remain "alive" until submission (`mate_put()` and `mate_send()`) or completion (`mate_`

get() and mate_recv()). Buffers in static or heap space are usually fine; buffers in stack space (local variables) can be troublesome if the function that initiated the message returns before the event has been triggered.

Another issue is thread safety, which varies for the communication functions by implementation (details in the implementation's documentation). The mate_startup() and mate_shutdown() functions may only be called from the main thread, whereas mate_mypid() and mate_numprocs() are inherently thread-safe.

MATE is a small library, and as such does not have any function that can be easily built from another. It does not marshal data, thereby providing no direct support for heterogeneous environments or noncontiguous data. It does not have collective communication, which means it is impossible to take advantage of a particular network's capabilities for such. It does not present more than one private context for tags, and therefore cannot be used by libraries. MATE is merely a tool that supports the Progressive Messages model of communication, and as such provides unrestricted message semantics.

5.2 MPI Implementation

Because MATE (and indeed Progressive Messages) is so general, it can be implemented on a variety of systems. As a demonstration, MATE was built for the Message-Passing Interface. MPI guarantees completion of communication and accepts unexpected messages, greatly easing development. It does not, however, allow for user-level interrupt handling. Furthermore, many implementations of MPI are not thread safe, nor are they able to progress independently of user direction. These limitations require MATE to poll for progress.

An earlier attempt at building MATE over MPI was multithreaded; all calls to MPI were in a unique thread, separate from the user's code. While this appeared to function correctly, the performance was unacceptably poor. MATE was polling *and* thread-switching. Now MATE is single threaded.

MPI has non-blocking two-sided communication functions, MPI_Isend() and MPI_Irecv(), which return a "request" variable. MPI_Test() queries this variable and returns a boolean value of whether the communication has completed by that point. MATE essentially performs all communication using these functions. MATE's two-sided communication functions call their corresponding functions in MPI. They then pack the request variable, a copy of the user's schedule, a receipt of the communication, and a timestamp of the communication (from MPI_Wtime()) into a list of "pending communication".

Calls to mate_progress() check this list by querying the request variables and checking the timestamps against the current time. If a message has completed, then the

corresponding handler is invoked (if available). If an incomplete message is more than a second old, then it is cancelled (with `MPI_Cancel()`) and the timeout handler is invoked. If there is any trouble during `mate_progress()`, such as an MPI function's returning an error code, then the failure handler is invoked.

The MATE wild cards (`MATE_ANYSRC` and `MATE_ANYTAG`) are replaced by the MPI wild cards (`MPI_ANY_SOURCE` and `MPI_ANY_TAG`) in `mate_recv()`; they are switched back in `mate_progress()`. Differentiation between submission and completion in `mate_send()` is a little tricky since MPI requires the user to pick one or the other, not both. The equivalent of submission is a "buffered" send in MPI, whereas the equivalent of completion is a "synchronous" send. By default, MATE chooses the synchronous one through `MPI_Issend()`. If, however, the user does not specify a completion and timeout handler, then MATE invokes MPI's default send in the hopes that this may be optimised for buffered use. The reason the timeout handle must be blank for this substitution is that a buffered send is "completed" in MPI when the buffer is reusable, not necessarily when the other side has received it. If the user specifies both a completion and submission handler, then the latter is called only when the former is ready.

5.2.1 One-sided Communication

The Progressive Messages model does not natively incorporate one-sided communication because of the lack of progress notification. However, MATE includes this feature as a proof-of-concept. The semantic goals are similar for two-sided communication in that a user-defined schedule specifies when to perform what action.

MATE's one-sided communication is by far the most difficult feature to implement with MPI. Version 2 of the standard presents one-sided functions, but many implementations of MPI actually lack them, or only provide partial support. For this reason, MATE's one-sided functions are implemented entirely via the two-sided functions.

All of the functionality starts with a set of pre-posted internal `mate_recv()`'s against unique dedicated tags. The completion handlers associated with these `mate_recv()`'s perform the remote node's functionality.

For example, a `mate_get()` essentially requests the remote node to send back the required data. Therefore, the completion handler for a `mate_recv()` against the internal `MATE_GET_TAG` merely sends back the requested data and then reposts the `mate_recv()`. The local node, after having requested the remote node for the data, posts its own `mate_recv()` in anticipation.

To prevent conflict in the event of multiple simultaneous one-sided messages, the local node generates a unique tag for each message that is passed to the remote node; this tag essentially functions as a private channel with no fear of clashes.

As for `mate_put()`, the easiest approach is to send the data directly and have the remote node's completion handler copy it to its intended destination. This technique, known as an "eager" protocol [87], is used for small messages. Large messages—greater than one kilobyte in size—suffer greatly in performance from the intermediary copying. They instead require a "rendezvous" protocol.

For large messages, the local node first alerts the remote node of an impending message and then calls `mate_recv()`. The remote node responds with a newly generated unique tag along with all of the information from the local node, and then calls `mate_recv()` against the desired location using the unique tag. The local node, upon receiving the response, sends the actual data using the unique tag, thus completing a transfer with no intermediary copying. The reason the remote node responds with all of the local node's information is so that the local node may remain nearly stateless; if the response never comes through, then the local node holds minimal unused resources (though it will call the timeout handle).

One final item worth noting: MPI has "communicators" that act as private contexts for communication, just as virtual memory provides a private context for the address space. MPI programs are allowed more than one communicator. MATE uses its own to prevent clashes in case the user's application calls MPI directly.

5.2.2 A Note About Multiplexing

There have been previous attempts to bring event-driven communication to MPI via `select()` constructs for TCP [89]. That work used a separate thread to handle message progress independently of the user's main thread. The results from That work showed that the point-to-point performance was indeed slightly worse for bandwidth, latency and utilisation because of the thread-switching overhead. However, the user's overall performance was better because of reduced message waiting time.

That particular article did not address high-performance networks covered in Section 2.2 because InfiniBand and similar systems do not expose a simple file descriptor, and thus require more detailed (and non-portable) work to achieve effects equivalent to `select()`. As a result, the cited work relied on copying and kernel intervention to realise its goals, which this thesis specifically sought to avoid by means of user-managed buffers.

5.3 OpenFabrics Implementation

This implementation uses a fully connected topology; that is, each node is connected to each other node during startup using the "reliable connected" (RC) communication service [87]. Ideally a connection would only be established when one is necessary, but

this is not possible when one-sided messages are permitted as the remote node will not know that a message is incomming. While it is possible to use a "reliable datagram" (RD) communication service in InfiniBand, this construct is not available on other networks. As one of the design considerations for MATE was that the architecture should easily port to iWARP, among others, we are left with RC.

Because there is a connection between each pair of nodes, each node allocates a queue pair (QP) for each other node. There is also a shared receive queue (SRQ) among all QPs to facilitate message receipt. Furthermore, there are two completion queues (CQs), one for sent data and one for received.

InfiniBand has end-to-end flow control for channel semantics in hardware, which is necessary for reliability. If the receiver is not ready yet—no descriptor posted—the sender will slow down and retry. This mechanism makes a "miss" fairly expensive, so it would be better to pre-post a sufficient number of descriptors [85]. During startup, the receiver posts a number of descriptors for each node in the network. The sender maintains a count of "credits" for each destination; the count decreases when a send descriptor is posted, and increases when the descriptor is acknowledged. Should the count fall below a threshold, the sender alerts the receiver via a piggy-backed signal in a message's metadata; at this point the receiver posts more descriptors and the sender increments the credit count.

5.3.1 Receiving Data

There are three tables on each node, two for pending messages (outgoing and incoming) and one for unexpected messages (incoming). Entries in the pending message tables contain the local receipt, schedule, and timestamp of the message, along with a pointer to the transmitted message structure. This transmitted message structure, which constitutes the entries of the unexpected message table, contains the remote receipt, type of message, and a pointer to the InfiniBand memory registration for itself. The transmitted message structure also contains the data that is communicated during two-sided operations, and as such contains a field of raw data for eager protocol messages.

These tables are traversed by the host node CPU. Some high-performance networks contain a programmable NIC that can perform the work, thus offloading the responsibility from the CPU. Because taking advantage of this feature, however, limits the portability of the software architecture, MATE uses the host CPU. There is one optimisation here of note: the number of entries in the table that must be traversed greatly affects the latency [120]. Therefore, the table is not a flat list, but rather a hash table where the message tag is the key. The entries are also stored as a queue in order of message posting; the reason for this is explained in Section 5.3.4.

When a user posts a receive request via `mate_recv()`, the implementation first checks the table of unexpected messages to determine if the desired message has already

arrived. If present, the entry is removed and the completion handle (if any) is invoked. MATE also checks to see if the sending node has requested more receive descriptors—the credit count has fallen below the threshold—and complies as needed.

If the user's requested message is not in the unexpected message table, then MATE fills out a pending message entry and stores it in the incoming table.

5.3.2 Sending Data

When the user requests to send data via `mate_send()`, MATE checks the size of the message before acting. Small messages are transmitted via an eager protocol; MATE fills out a transmitted message structure such that the raw data field is a copy of the user's data. Large messages are communicated via a rendezvous protocol [118] in which the local node alerts the remote node of the local address. The remote node then uses an RDMA "read" to obtain the data and, upon completion, responds with a signal to alert the originating node that the communication has finished.

Regardless of protocol, the buffer used for communication must be pinned. As pinning is so expensive, it is desirable to avoid this step when possible. For a transmitted message structure, this implies a pool of pre-pinned buffers. Buffers other than the transmitted message structure are not as static. Luckily though, the principle of locality implies caching [119]; when the user requires that a buffer be pinned, MATE pins the whole set of pages containing the buffer and then stores the registration key in a table. Future pinning requirements consult this table for the key. Entries are maintained in a least-frequently-used manner to facilitate unpinning when too many buffers are registered.

One-sided operations—`mate_put()` and `mate_get()`—use an algorithm similar to the one found in [11]. If the local node has the key required to perform one-sided communication, then the operation simply proceeds via RDMA. If, however, the local node does not have the key, then it requests it from the remote node via two-sided communication. The remote node looks for the key in a table of pinned buffers (the buffer is pinned automatically if the search fails) and responds to the originating node with the key.

5.3.3 Progress Thread

The implementation is multithreaded to meet MATE's goal of independent progress. A progress thread is spawned at startup and blocks until the one of the completion queues (CQs) signals completion of a descriptor. (While it is possible for the thread to poll for completion, such a design is not particularly desirable for performance reasons [72].)

If the completed communication was a send operation, then the corresponding entry in the pending message table is removed and the completion handle invoked.

If the completed communication was a two-sided receive operation, then the pending message table is checked to see if the user has already requested this item. If so, then the completion handle is called; otherwise, an entry is stored in the unexpected message table. In either case, the descriptor is reposted to the receive queue.

For a request for a pinned buffer key—required in one-sided communication, as explained in the previous section—the node sends back the key.

If anything should go wrong during the execution, such as inability to allocate resources, then a failure handler is invoked where possible.

5.3.4 Timeout Thread

Timeouts are treated via a special thread that is spawned at startup. This thread sleeps for a fixed amount of time, one second by default. Upon waking, it scans the pending message tables for any entries that are older than the fixed amount of time. For each such entry that lists a timeout handle, the entry is removed and the handle invoked.

As mentioned earlier, all of the entries in the pending message tables are added in FIFO order. The reason is that scanning may stop immediately if an entry (for that particular hash key) is *not* older than the fixed amount of time. All remaining entries will be younger and thus are known not to have timed out. This scheme was introduced for better scalability with regard to the number of outstanding messages.

5.4 Summary

This chapter presented MATE (Message Alerts Through Events), a library that supports Progressive Messages communication on high-performance networks. We began with the specification to give an idea of how such a platform would actually look. MATE is intended to be a simple and small library, and only provides the features it must have.

With the specification, this chapter then covered implementation details for MPI and OpenFabrics. The MPI version was pretty straightforward, though it unfortunately needed polling because of MPI's progress dependencies. With notification solved, the implementation simply used MPI's non-blocking routines.

The OpenFabrics implementation was much more involved. We must pre-post receive descriptors and ensure that the sending process knows when the descriptors will be

exhausted. The recipient must be mindful of expected and unexpected messages and must segregate them accordingly. Further, the sender must cache the memory pinning to avoid expensive calls to the kernel. Altogether, the numerous “moving parts” of the OpenFabrics implementation made it a far more complicated product.

The next chapter will explore scalability simulations for Progressive Messages and performance results for MATE.

Chapter 6

Performance Results

The implementations of MATE described in the previous chapter demonstrate the portability of the Progressive Messages model. This chapter investigates the model's performance. Given that performance is a primary motivator for this thesis, we will consider very carefully what the impact is of event-driven communication. In particular, we want to know if software applications can scale because of the wait-free policy that anchors Progressive Messages.

Section 6.1 investigates the scalability through a simulation. It assesses wall-clock costs for messaging frameworks that require waiting. Our contention is that asynchronous communication leads to better scalability, among other benefits.

Section 6.2 presents some network metrics that may be of interest. In particular, it tests MATE's latency and bandwidth over an InfiniBand-compatible network. Improving these "unit tests" is not a primary goal of this thesis, though it is always desirable to have fast communication. This section explains some issues encountered with the experimental testbed we used.

6.1 Simulating Scalability

Chapter 3 presented a common example problem of work stealing in a scientific loop. The three messaging frameworks described in this thesis handle the communication in very different ways. The message-passing model must explicitly check for progress after completing an atomic unit of work. The message-driven model can handle incoming requests and responses concurrently with a work unit, but must wait for sent data to be acknowledged by the network for reliability. The Progressive Messages model does not wait for anything provided it can still perform some unit of work.

It is fruitful to see what effect the delay scenarios have on scalability in really large systems. Because we do not have access to a machine with 16K processors, we must

simulate the impact. The workload in this simulation is a randomly generated list of 128K positive integers that is right-tail normally distributed. (The non-uniform distribution reflects extreme cases where one work unit may take far longer than average.) The workload is initially split among the processes in the system.

Consider a single step in our simulation[1], where a step represent all processes' completion of a single work unit:

```
// simulate a step for a given model
simstep:{[balance] // "balance" parameter is a higher-
                   // order function for which messaging
                   // framework is being tested
 // take a step of progress and increment the clocks
 // by the size of waiting process, if one exists;
 // remove the finished process from the process
 // queue
 clocks+:0f^first each procs; `procs set 1_'procs;
 // generate list of procs with workloads below
 // the threshold; these are "sources"
 sources:where threshold>sum each procs;
 // for each source, randomly select "targets"
 // for rebalancing
 targets:(count sources)?numprocs;
 // balance any (source,target) pairs with the
 // higher-order function
 if[not all null targets; (balance .) each
    sources,'targets];
 }
```

There are two globals variables in the above function: `clocks` is an array that represents the elapsed wall-clock times for each process, and `procs` represents each process's assigned work units in time required to complete the task. The `procs` variable is an arrays of arrays in which the first index represents the process id and the second index corresponds to a specific task in the queue.

The above function progresses every process through one work unit and records the impact to the wall clock. It then determines which processes have work sizes that are below the threshold ("sources") and randomly chooses other processes to request work from ("targets"). Note that targets are not exclusive; it is possible for multiple sources to pick the same target, which is expected in a real-world scenario.

[1]All code in this chapter is written with q/kdb+, a proprietary programming language used in the financial industry for analyzing time-series data. It is very similar in ideology to array programming languages APL and J.

6.1.1 Message-Passing Scalability

The `balance` parameter from above is a higher-order function for the particular messaging framework that will be simulated in balancing the load between a source and target. We will look at the message-passing version first since it is the most complicated to simulate:

```
// balance with message passing
balancemp:{ // x and y are implicit parameters:
            // x is a single source
            // y is the paired target
 // don't trade with self: return false if process
 // has selected itself
 if[x=y;:0b];
 // temporary placeholders for the target
 // and source process queues
 py:procs y;
 px:procs x;

 // target can only handle request after
 // a work unit is complete
 while[(clocks[x]>clocks y)and 0<count py;
    clocks[y]+:first py; py:1_py];
 // source must wait for target to receive request;
 // this prevents look-ahead corruption too
 newclock:max clocks x,y;
 // source can only handle reponse after
 // a work unit is complete
 while[(newclock>clocks x)and 0<count px; clocks[x]+:first
     px; px:1_px];

 // account for network latency on both target and source
 clocks[x,y]+:latency;
 // prevent thrashing by checking that target will
 // have enough to do
 if[(3*threshold)<sum py; :0b];
 // target keeps second half
 procs[y]:last p:(0,(1+count py)div 2)cut py;
 // source gets target's first half
 procs[x]:px,first p;
 // return true
 1b
 }
```

The first issue to be mindful of in the message-passing model is that the target can only handle requests after a work unit has completed; furthermore, the source can only handle responses after a work unit has completed (if there is any work left to do). Thus, the clocks of both processes must advance to represent a completed work unit. The message-passing model additionally suffers from the network latency of waiting for acknowledgements.

The code then will perform a sanity check so that the source will remain reasonably above the threshold after splitting its workload. (The very first attempt at coding this simulation did not include a thrashing check, so the model did not scale at all beyond a handful of processes.) If the sanity check is satisfied, then the balancer partitions the workload and returns true (`1b`).

At this point it may be fruitful to ask what a reasonable threshold and latency might be for the simulation. Obviously, more latency simply leads to worse results for all models. We also want the latency to be significantly less than the average time to complete a single work unit, otherwise load balancing would be pointless. The randomly generated workload was the same for all simulation runs and had a median of six. (The normally-distributed random numbers were multiplied by ten so that they could be rounded down to integer values for readability while testing the software.) Because the median was six, the latency was simply set to 0.5.

As for the threshold, the simulation runs tried a few different values. The lower the threshold, the better the scalability is for every communication model simply because the processes are not trying to request work far in advance of when they actually need it. That is, a high threshold contributed to more thrashing.

6.1.2 Event-Driven Scalability

We now turn our attention to event-driven simulations. Processes with event-driven communication do not need to complete a workload before handling a request or response.

In message-driven communication, a process must still wait for acknowledgements when sending a message.

```
// balance with message driven
balancemd:{ // x and y are implicit parameters:
            // x is a single source
            // y is the paired target
 // don't trade with self: return false if process
 // has selected itself
 if[x=y;:0b];
 // temporary placeholders for the target
 // and source process queues
```

```
 py:procs y;
 px:procs x;

 // account for network latency on both target and source
 clocks[x,y]+:latency;
 // prevent thrashing by checking that target will
 // have enough to do
 if[(3*threshold)<sum py; :0b];
 // target keeps second half
 procs[y]:last p:(0,(1+count py)div 2)cut py;
 // source gets target's first half
 procs[x]:px,first p;
 // return true
 1b
 }
```

The final simulation is for Progressive Messages. Unlike with the message-driven model, processes with Progressive Messages do not wait for network acknowledgements when sending a message. That means the process is *always* computing.

```
// balance with progressive messages
balancepm:{ // x and y are implicit parameters:
            // x is a single source
            // y is the paired target
 // don't trade with self: return false if process
 // has selected itself
 if[x=y;:0b];
 // temporary placeholders for the target
 // and source process queues
 py:procs y;
 px:procs x;

 // prevent thrashing by checking that target will
 // have enough to do
 if[(3*threshold)<sum py; :0b];
 // target keeps second half
 procs[y]:last p:(0,(1+count py)div 2)cut py;
 // source gets target's first half
 procs[x]:px,first p;
 // return  true
 1b
 }
```

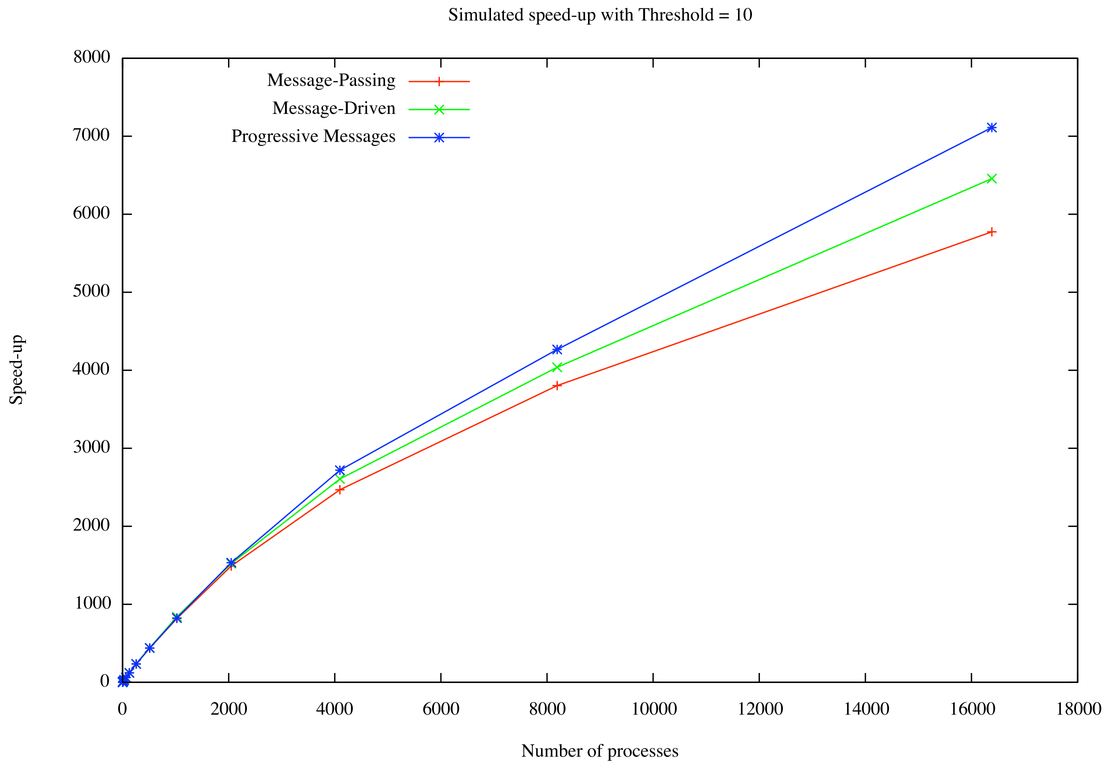

Figure 6.1: Speed-ups for when load-balancing threshold is 10

6.1.3 Results

The simulation was run across the three messaging models for the varying threshold levels. Each run used the same set of randomly generated data that fit a right-tail normal distribution (all numbers were positive). As described above, a wall clock was incremented according to the implications of a model—message-passing waits for a work unit to complete before handling requests and responses, whereas event-driven models can overlap their computation and communication.

Figures 6.1–6.3 present the scalability graphs. While linear scalability is always the goal, it is in practice often unrealizable for large sets of processes. That is, the efficiency drops as the universe of compute nodes increases. The reason is that there is extra overhead from state synchronization in the system, plus there is the possibility of thrashing. Indeed, larger threshold values lead to less scalability because the processes are requesting work too soon, leading to unnecessary communication and state changes.

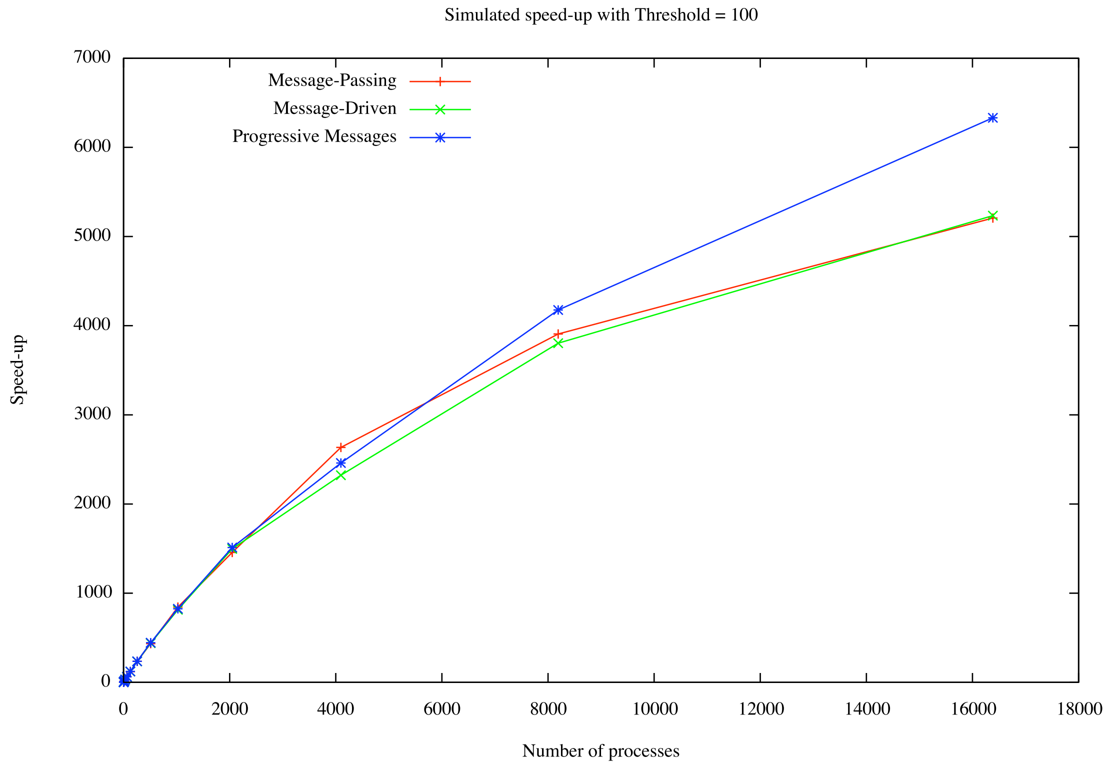

Figure 6.2: Speed-ups for when load-balancing threshold is 100

6.2 Network Metrics

Beyond scalability, the high-performance computing industry uses a number of standard metrics to describe the communications system itself. By far the most common is bandwidth, the amount of information that can be transfered in a given time. Another metric is latency, the amount of time required for a message to reach its destination. A third and less reported number is utilization, the percentage of time the software requires the CPU. This section gives an overview of these metrics.

6.2.1 Bandwidth

Bandwidth, the rate of data movement, is usually measured as the number of bits per second. It is not constant; it grows as the message size increases, up to a point. The maximum bit rate is the figure usually quoted in the literature, though not all applications realize this much because bandwidth is lower for small messages [66]. A typical bandwidth plot vs packet size is demonstrated in Figure 6.4.

To assess how much lower the bandwidth is for smaller messages, we can investigate how quickly the bandwidth rises. A faster-rising bandwidth means faster transfers

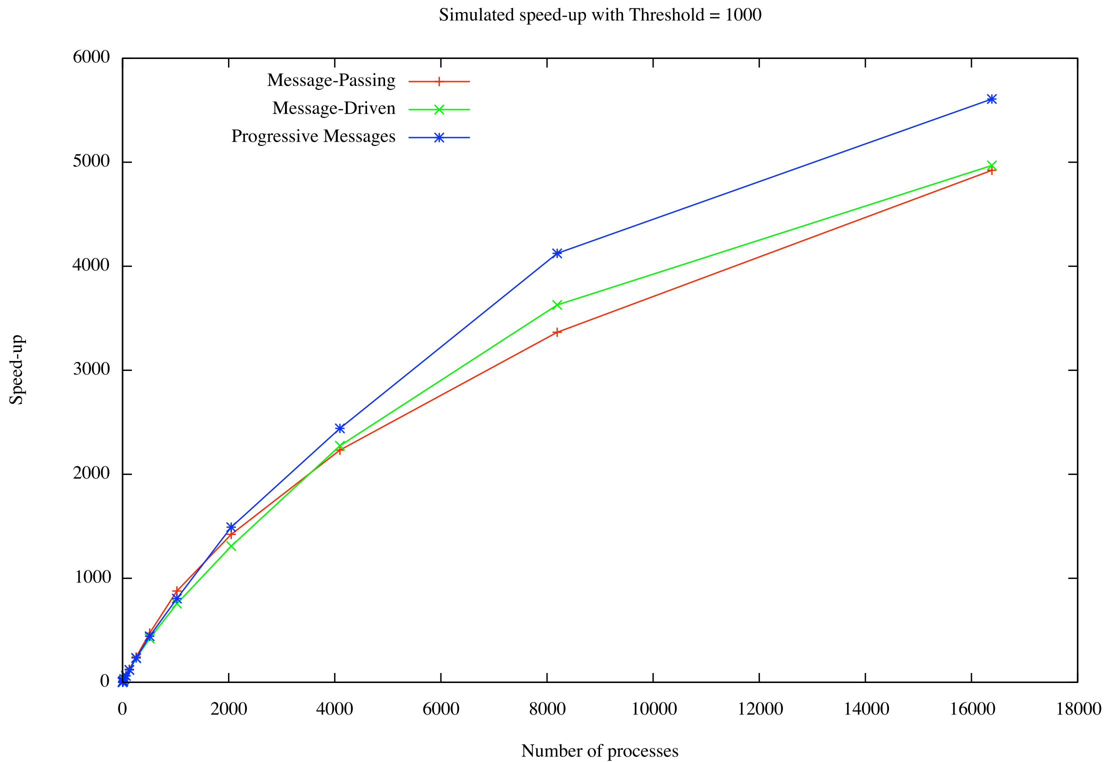

Figure 6.3: Speed-ups for when load-balancing threshold is 1000

for the same message size. A simple metric that captures this acceleration is the "half-power point", designated by $n_{1/2}$, which is the message size at which half of the maximum bandwidth becomes available. A lower number means a faster rising bandwidth. See Figure 6.5 for an example.

6.2.2 Latency

Latency is a measure of delay, expressed as a unit of time. While bandwidth can be increased by adding multiple lines of communication, latency is fundamentally limited by the underlying system. Its main components are (see Figure 6.6):

1. the software running on the host CPU,
2. the bus that must be traversed to reach the NIC,
3. the NIC itself,
4. the cable connecting the NIC to the switch,
5. the switch itself.

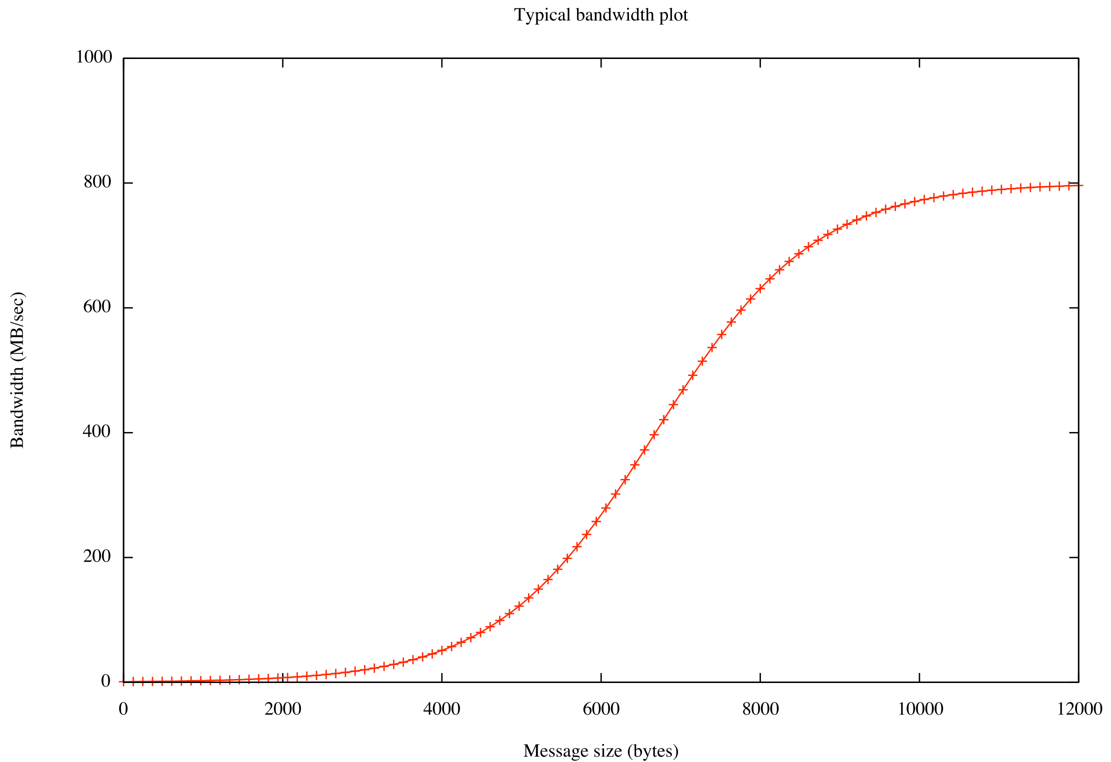

Figure 6.4: Typical bandwidth plot resembles a sigmoid curve

The switch traditionally is not an issue. And beyond shortening the length, there is not much improvement available for the cable. However, it is possible to remove the switch and one of the cables if the network's topology is such that nodes are connected directly to each other. This is an additional argument in favour of a torus topology (see the fault tolerance argument in Section 2.2.3).

The software can be speeded up by having a faster CPU and by bypassing the kernel—that is why high-performance networks operate at the user level. Furthermore, RDMA removes the need for copying; see Section 2.2.2.

As for the bus, one suggestion is to remove it altogether and plug the NIC directly into the CPU (see Section 2.2.4). This design has the added benefit of using the host node's CPU instead of requiring an ASIC on the NIC, thereby making the network faster and cheaper. This design however, increases the CPU utilization.

6.2.3 CPU Utilization

CPU utilization traditionally has not been quoted much, though it is growing in importance as offloading becomes more common (see Section 2.2.1). This metric represents the percentage of time the process requires the CPU. A lower number

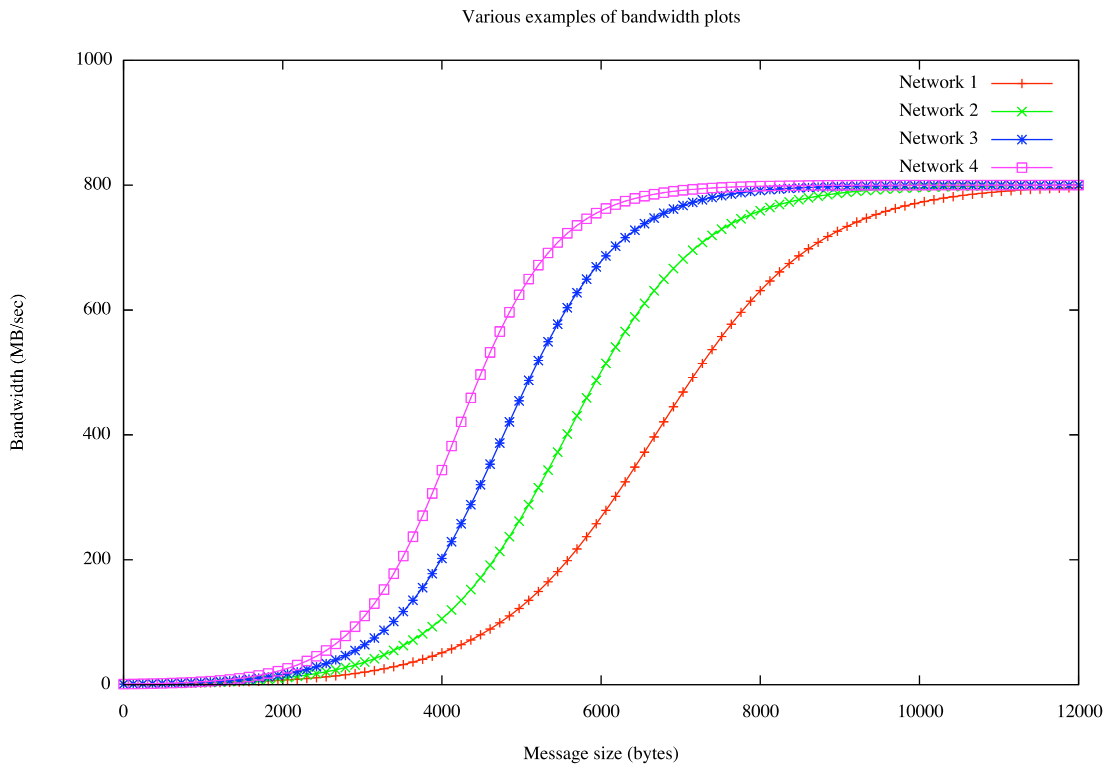

Figure 6.5: The bandwidth of Network 4 rises faster than the bandwidth of Network 1, even though the maximum bandwidth is the same

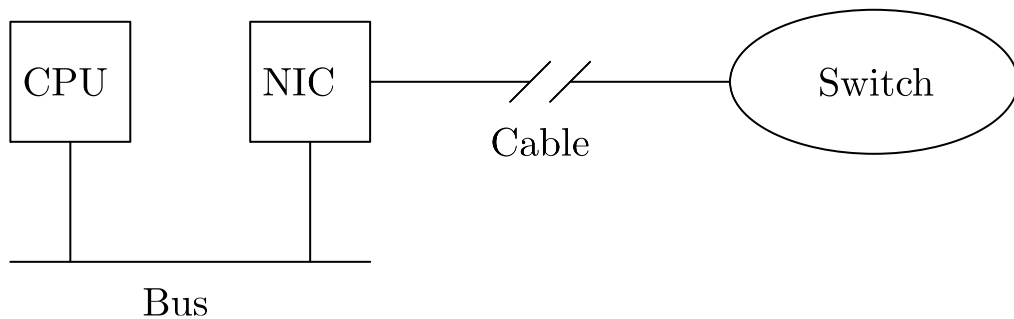

Figure 6.6: A message travels through the bus, NIC, cable, and switch

indicates that the CPU is free to do more work, which increases the amount of communication that may be overlapped with computing work. Overlapping has the effect of diminishing (hiding) the consequences of latency in the application. Thus, the "effective latency", particularly for large messages, will be much lower.

CPU utilization is essentially the usage figure obtained from the POSIX `top` command. This can be computed by comparing iterative calls to `getrusage()` and `gettimeofday()`. Note that utilization is only valid for that process on that core; it does not consider the presence of other cores, particularly in regards to multi-threaded applications.

6.2.4 Ping Tests

To test the effectiveness of MATE, we use a series of ping tests based on the standard Intel MPI Benchmarks [69] and NetPIPE [112]. These tests are point-to-point between a pair of processes; they measure bandwidth, latency, and utilization. Most of the tests have the option of waiting for either submission or completion.

The most common benchmark test is a "ping pong", in which node 0 sends a message to node 1, and then node 1 responds with a message back to node 0; the figure quoted is one-half the time required for this trip (see Figure 6.7). This test gives a sense of unidirectional communication.

Another popular test, "ping ping", sees both nodes simultaneously sending a message to each other. The objective here is to determine how well the network responds to an incoming message.

The "ping put" test measures one-sided push, whereas the "ping get" measures one-sided pull. The latter does not have a "ready" event as buffer resuse exists only for sending data.

6.2.5 Other Tests

While the ping tests are common, they do not reflect the effects of communication on the wider system. Other benchmarks, such as NAS [9] and HPC Challenge (HPCC) [88], investigate how "balanced" the whole computing environment is, particularly for parallel applications. Much of these tests involve "kernels" which are common computing tasks for technical codes, like computational fluid dynamics. These benchmarks have been ported to multiple platforms, so it feasible that MATE could target these in the future.

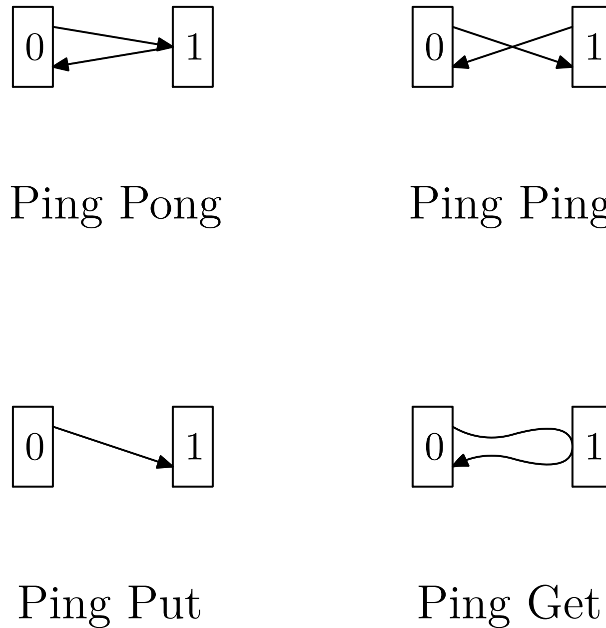

Figure 6.7: Conceptual message paths during the ping tests

6.2.6 Results

We performed ping tests on the MATE / MPI and MATE / IB[2] implementations described in Chapter 5. The testbed was an InfiniPath cluster (see Section 2.2.4) featuring dual-core 2.2 GHz Opterons. Note that InfiniPath has no offloading; its CPU utilization is 100%, as confirmed by the tests. This characteristic significantly hampers MATE's performance because of the requisite polling.

Figures 6.8–6.11 show the results of ping pong and ping ping for both native MPI and MATE / MPI. The polling damages MATE's latency, though the maximum bandwidth is not affected. However, the native MPI does rise faster; its $n_{1/2}$ is 4KB versus 8KB for MATE / MPI. One peculiarity is that InfiniPath's 1.5μsec latency only exists for regular MPI sends, which return as soon as the buffer is reusable. Tests with "synchronous" sends (which wait until the data has been received) took around 12μsec.

InfiniPath's MPI has no one-sided message capability, so the ping put and the ping get have only been tested with MATE / MPI. The put test (Figures 6.12 and 6.13) features better latency than the ping ping because it invokes MPI's send via the eager protocol and takes advantage of the peculiar performance skew noted in the last paragraph. The get test (Figures 6.14 and 6.15) closer resembles the ping pong since it sends the request and receives the result.

The final set of tests is on MATE / IB (Figures 6.16 and 6.17). InfiniPath has *verbs*

[2]The implementation was created when OpenFabrics was known as OpenIB, hence the name "MATE / IB".

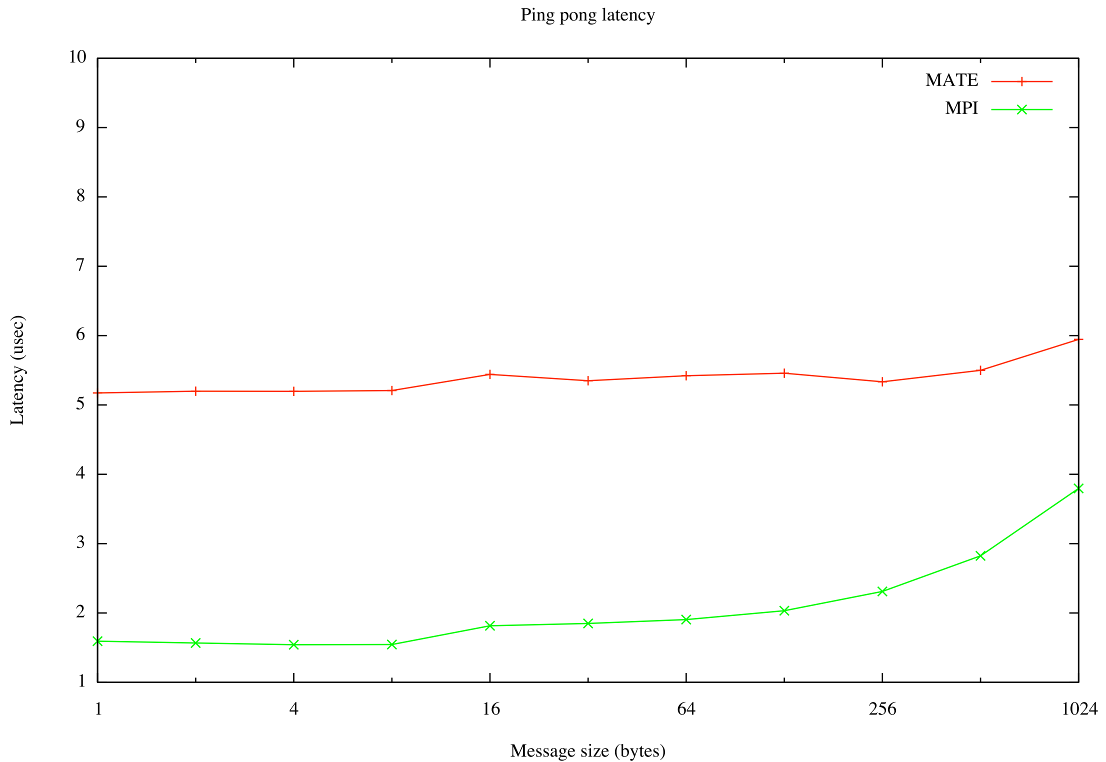

Figure 6.8: Ping pong latency between two nodes

API of its own; the only available stack is an early beta emulation released before OpenFabrics was fully codified.[3] The stack implementation had a bug that prevented proper reuse of pinned buffers. This means that MATE / IB's rendezvous and one-sided communication could not handle more than a few iterations during the ping test (the test requires at least a thousand iterations to amortize the startup costs). Thus, only the eager protocol communication results are available, and even then their actual potential cannot be fully realized on the testbed cluster.

6.3 Summary

This chapter presented some performance results for the Progressive Messages model. We began by comparing the scalability of each of the three models presented in this thesis over a work-stealing simulation. The message-passing model must wait for a work unit to complete before handling any communication, which leads to significant delays in responding to work requests; both event-driven models do not have this problem. However, the message-driven model must wait for network acknowledgements when sending a message, which causes some delay before the process can

[3]InfiniPath's vendor was bought-out shortly after releasing the early beta.

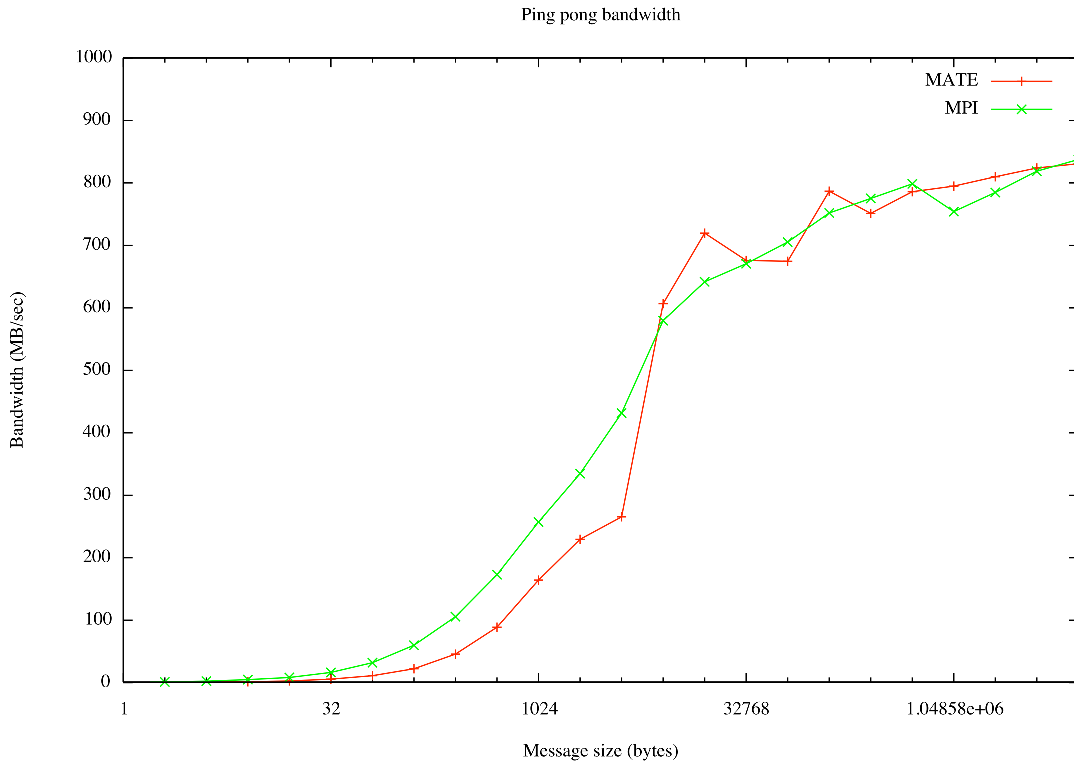

Figure 6.9: Ping pong bandwidth between two nodes

return to computing a work unit. Only the Progressive Messages model is free from both types of delays; it can fully overlap communication and computation, which leads to better scalability.

The other major result in this chapter is the "unit tests" of common network metrics using MATE. While not the primary focus of this thesis, lower latency and faster-rising bandwidth (lower $n_{1/2}$) are always desirable properties to have. The results do indicate worse latency, though it is difficult to tell whether how much of that impact comes from the peculiarities of the testbed network. In either case, applications that require scaling to over many processes will often sacrifice latency for the goal of more efficiency.

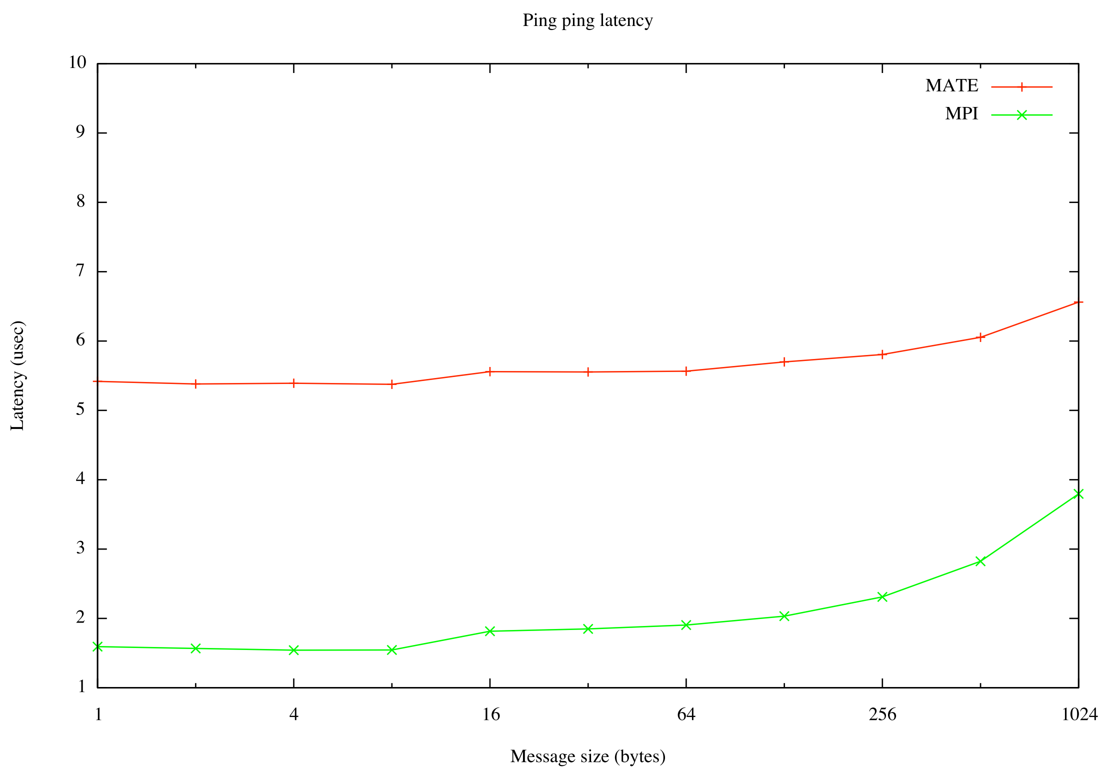

Figure 6.10: Ping ping latency between two nodes

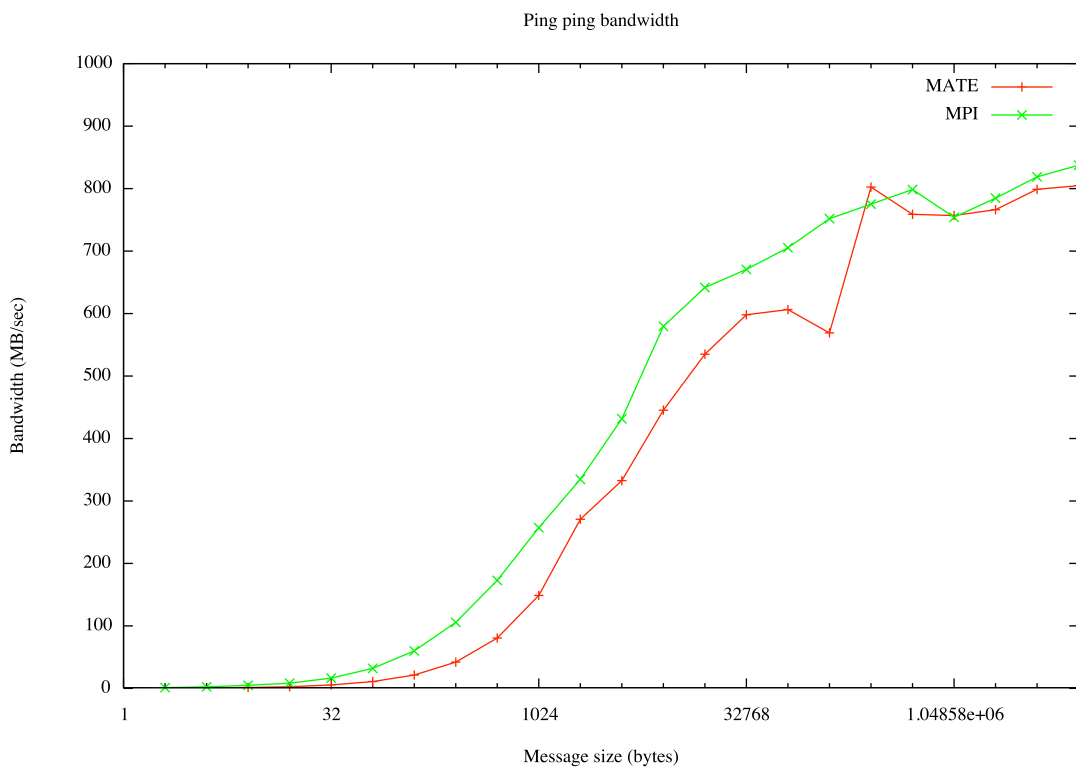

Figure 6.11: Ping ping bandwidth between two nodes

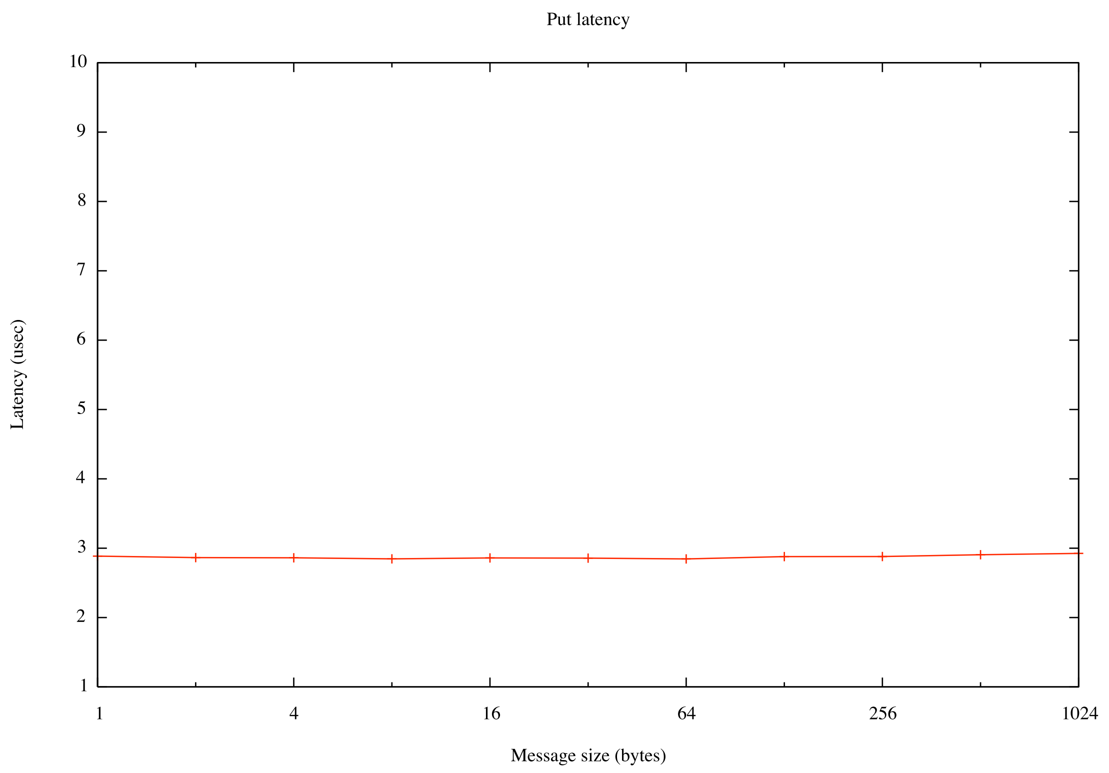

Figure 6.12: Put latency between two nodes (OpenFabrics)

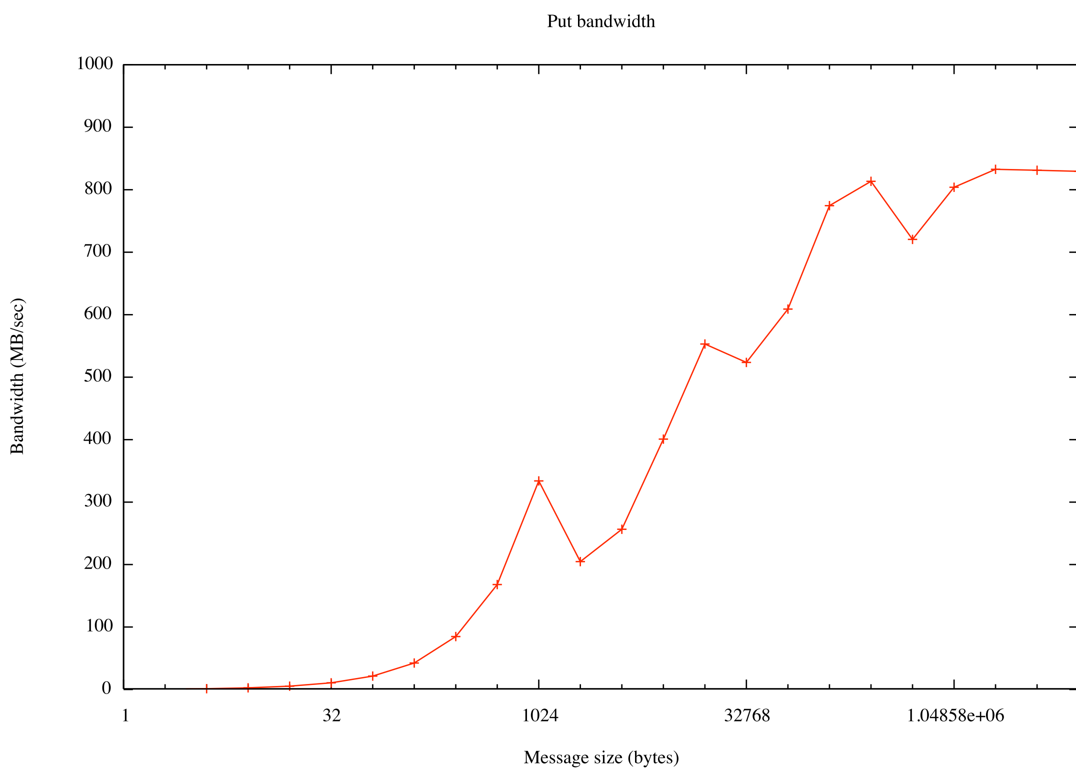

Figure 6.13: Put bandwidth between two nodes (OpenFabrics)

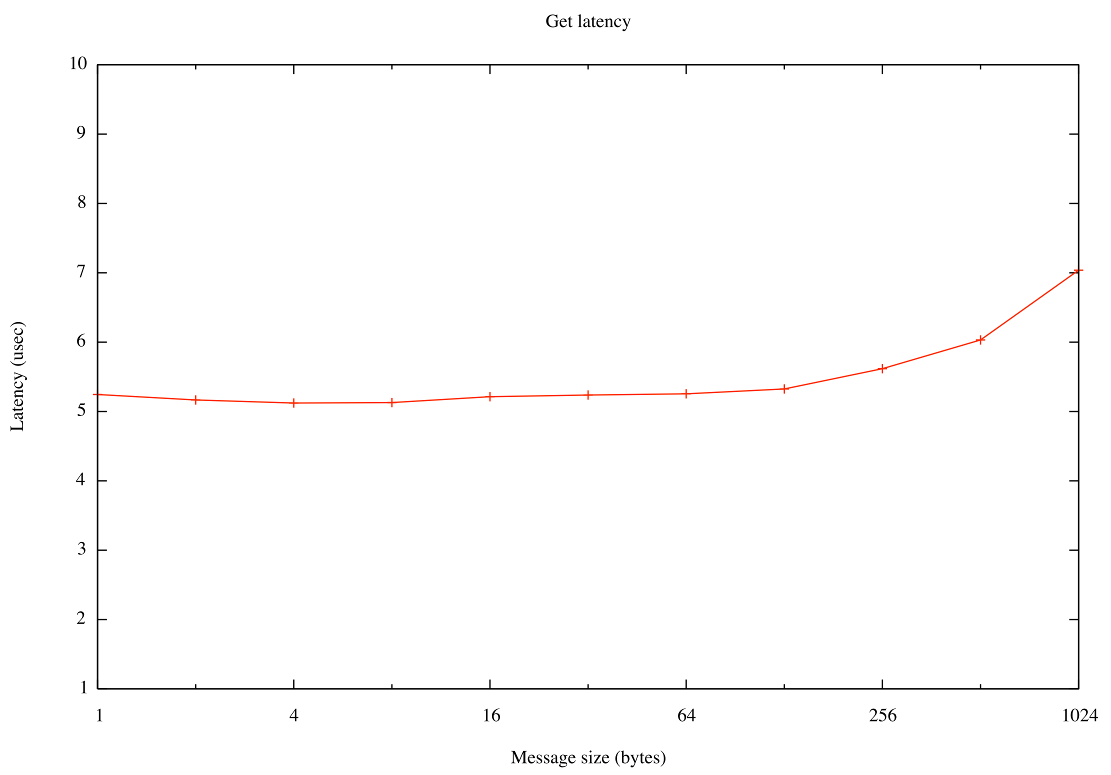

Figure 6.14: Get latency between two nodes (OpenFabrics)

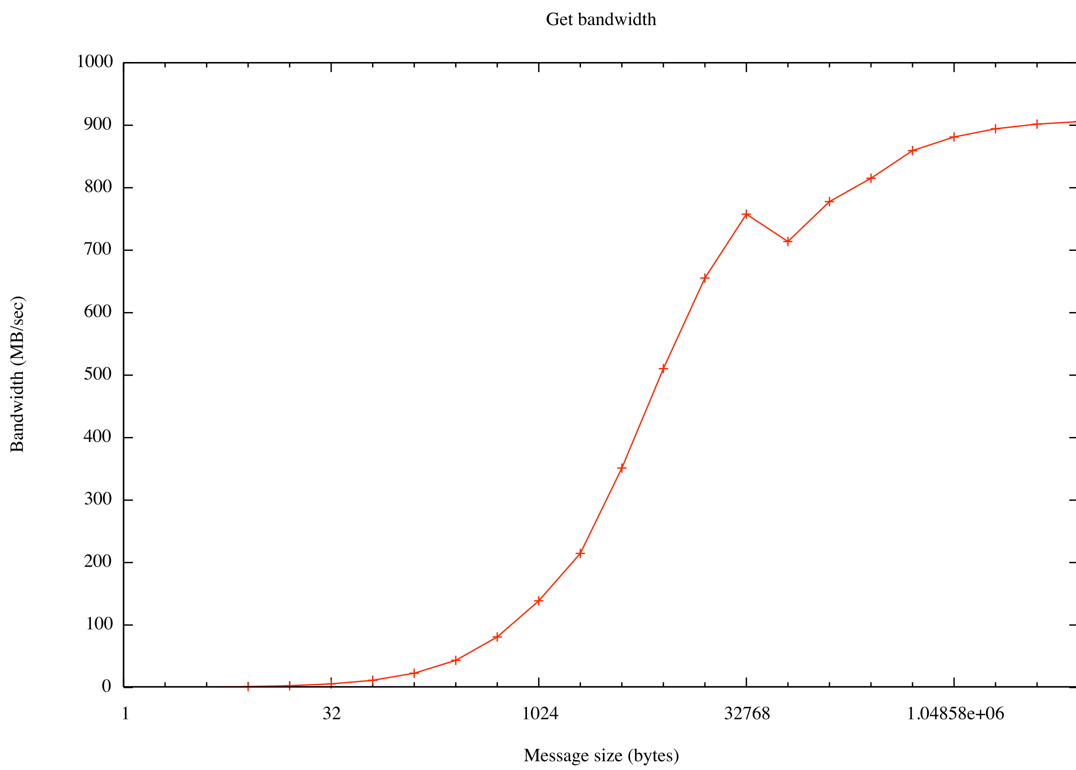

Figure 6.15: Get bandwidth between two nodes (OpenFabrics)

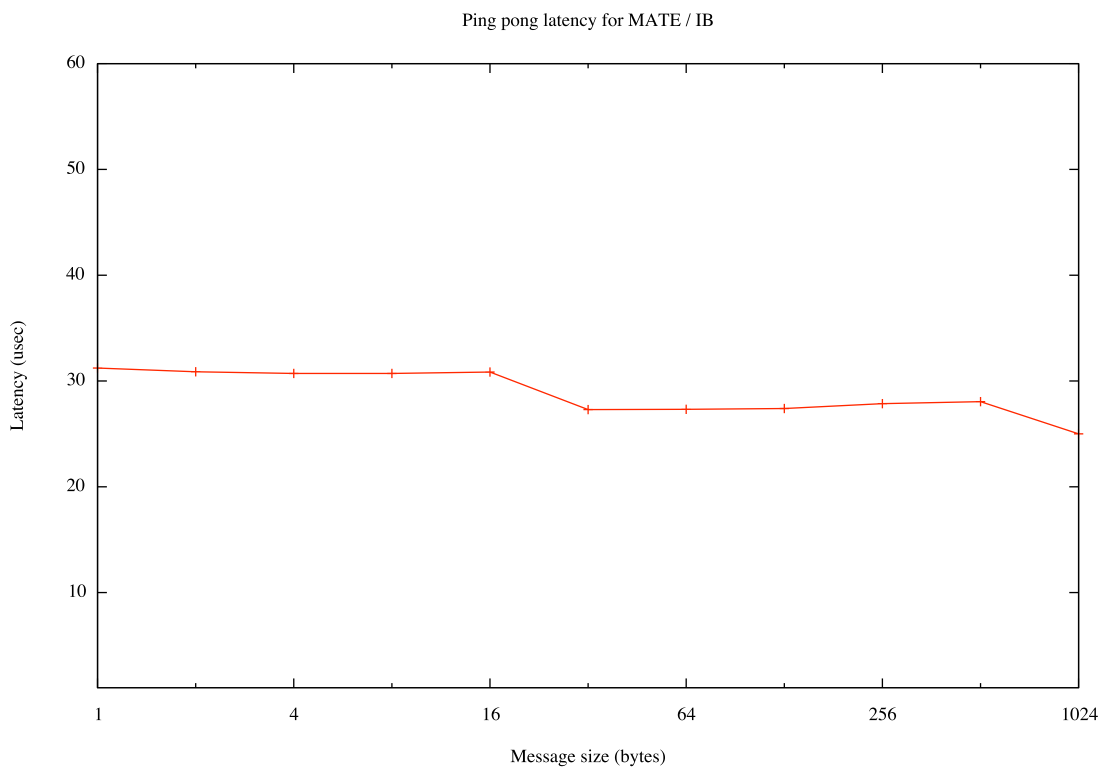

Figure 6.16: Ping pong latency between two nodes using OpenFabrics IB stack

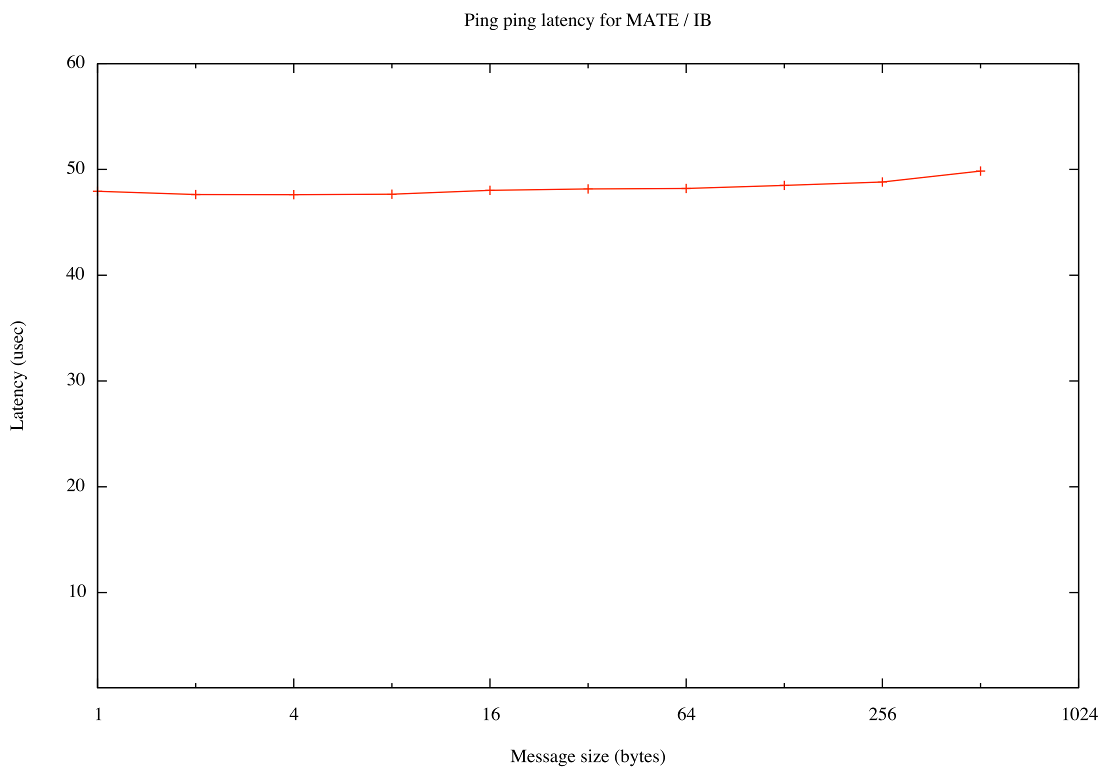

Figure 6.17: Ping ping latency between two nodes using OpenFabrics IB stack

Chapter 7

Conclusion

This thesis introduced the Progressive Messages model of communication. It is an event-driven framework that allows user applications to observe message termination. The semantics are an extension to the message-driven model that alerts the application about changes in the message progress. Both of these event-driven models can be contrasted to the message-passing model, which has no event notification.

This research was motivated by the trouble in parallelising a time-consuming loop. The message-passing approach requires that the dynamic load-balancing logic execute within the main loop. That design takes time away from the computation, which affects scalability. The message-driven approach improves the design by handling new messages only on arrival. But it must at least wait for acknowledgements when sending data. The solution is to employ concurrent and as-needed processing of messages. The message-driven model has this for successful receives. This thesis went a step further to use alerts on all types of message progress.

We simulated the scalability of all three models in a dynamic load-blanacing application. Progressive Messages was found to be more efficient than either message-passing or message-driven because the software can respond to work requests at anytime. That means the communication and computation phases are overlapped, which leads to greater efficiency.

Because the literature does not define the precise semantics of message-passing or message-driven models, we identified them as Cypher-Leu (formal MPI) and a subset of CSP, respectively. We then covered how applications are programmed with these models and hopefully demonstrated the expressiveness of Progressive Messages.

To develop software with the Progressive Messages model, we implemented the MATE library with both MPI and OpenFabrics. This API specifies an action through a schedule that maps events and callbacks. The "unit test" performance metrics for MATE indicate some added latency, though it is difficult to tell how much of this overhead comes from the software and how much comes from the hardware or device

drivers.

7.1 Future Work

The research in this thesis targets high-performance computing, hence the emphasis on networks like InfiniBand. However, given the generic semantics of Progressive Messages, event notification should be applicable to web services. Indeed, AJAX shares some similar properties in its client-server communication. On the server-server front, it is quite likely that an HTTP-based protocol could replace REST and SOAP with message tracking. Any overhead of a MATE-like API would be unnoticeable because of the existing high latency of the Internet.[1]

One immediate application given the ability to track messages is fault-tolerance. It is conceivable that an application could build error-handling routines that respond to unexpected message events. This is analogous to exception handling in many programming languages. The user would simply annotate his software with callbacks to handle edge cases in communication.

Another obvious direction for further research is in hardware. The profuse requirement for design patterns in InfiniBand demonstrates the lack of support for common communication needs in VIA-inspired networks. It is possible that a new network could drop RDMA and instead focus on event notification at the user level. This is similar in spirit to the design philosophy of InfiniPath, which was created explicitly for MPI.

A fourth potential area of investigation is direct support for message tracking in a domain-specific programming language. Just as the object-oriented paradigm led to languages with abstract data types and the functional paradigm gave way to languages with higher-order functions, perhaps the Progressive Messages model could inspire a language with event-based communication.[2] This can work well if the intended domain is technical computing or even web development.

7.2 Final Thoughts

Just as in the package-delivery industry, progress monitoring through event notification allows users to develop applications that know their messages' states. Specifically geared towards parallel and distributed computing applications, the Progressive Messages model provides notification of message state changes. The application can reserve a callback for each state-change possibility, which leads to greater scalability

[1] This is why web developers can get away with scripting languages.

[2] This is similar to how Go is derived from CSP, or how Erlang follows the Actor model.

since the communication is overlapped with the computation.

Appendix A

Dynamic Typing in Remote Memory Access

This thesis was inspired by an earlier attempt to bring dynamic typing to remote memory access. The immediate results of this initial study were made available by expanding the MPI-1 bindings [97] for the Ruby programming language [91] to include the RMA[1] features of MPI-2 [5]. The goal here was the same as for Progressive Messages—to make it easier to take advantage of modern network features.

Dynamic typing is common in very high-level programming languages. Users can quickly and easily create new programs as well as modify existing ones, albeit by sacrificing some performance and potential safety. Dynamic typing is already common in distributed systems programming, such as with Erlang [4] and other message-driven platforms.

This appendix presents the only known MPI-2 RMA bindings for a scripting language. The work was not pursued beyond implementation as RMA is not believed to be a sufficient programming model for either performance or ease of use. As mentioned above, Progressive Messages came from the experience in creating these bindings.

A.1 The Ruby Bindings of MPI-2

Communication and synchronization in MPI-2's RMA are separate. Therefore, the user calls communication routines in MPI Ruby during "epochs" defined by synchronizations. All communication is guaranteed to have finished by the end of the epoch, but not any time earlier. During an epoch, the user may issue any number of communications.

[1] MPI-2 refers to `put` and `get` routines as Remote Memory Access.

In MPI-2, communication is to contiguous memory and mimics accesses to a remote array. Therefore, the target memory in the Ruby bindings is an array. When communicating data, the user specifies the index of the array where the results will be written to or read from. Because arrays in Ruby are dynamic and heterogeneous, communication is not restricted by size or type. Also, any process may usually perform communication at any time to any index, although there are some restrictions as explained in section A.1.2. The complete bindings appear in Figures A.1 and A.2.

A.1.1 Window Creation

A "window" must be created before any communication may take place. This window is the logical representation for the remote array. Window creation is collective; all processes must call the function. It takes as parameters an object (representing the initial array) and an MPI communicator. The array may be empty as represented by `[]`. The effects of all remote memory operations are realized through this array, either fetching or storing data at a specific index. The user must begin an epoch before issuing any communications through the window as the window is not active until synchronization.

```
win = MPI::Win.create(ary, comm)
                        # ary is of type Array
                        # comm is of type MPI::Comm
                        # win is of type MPI::Win
```

A.1.2 One-sided Communication

MPI Ruby's remote memory access occurs through one-sided communication between two processes. The semantics for the Ruby bindings differ from traditional MPI in that all access is based on a logical array—initialized through `MPI::Win.create()`—rather than a buffer. All accesses, whether to read or write data, occur through specific indices of the array. Thus to the user, all RMA actions appear like operations on a remote array.

The semantics for the MPI functions mimic those of regular Ruby arrays. Because Ruby arrays are dynamic and heterogeneous, there is no restriction on what the size or type of the data must be, or where data may be stored. If the index provided by the user is greater than the index of the final element in the array, then the array will expand to accommodate the data. The user may also employ negative indices to count backwards from the end of the array. Furthermore, the user may provide data of differing types to different elements of the array.

To place data in a remote array, the user calls `put()` as a member function of a window object. The parameters of this method are the data to be placed, the destination's

`MPI::Win.create ( ary, comm )`
Creates and returns a window of type MPI::Win based on an initial object of type Array and a communicator of type MPI::Comm.

`MPI::Win#put( obj, dest, index )`
Begins one-sided communication to the index at the destination. Returns true on success.

`MPI::Win#get( src, index )`
Begins one-sided communication from the index at the source. Returns an object of type MPI::Getrequest.

`MPI::Win#accumulate( obj, dest, index, op )`
Accumulates data at the index of destination using the given MPI operation. Returns true on success.

`MPI::Getrequest#object()`
Returns the object of a "get request". Valid only after synchronization.

`MPI::Win#object()`
Returns the array represented by the window. Valid only after synchronization.

`MPI::Win#set_attr( keyval, obj )`
Establishes an attribute for key value (of type MPI::Keyval) and returns true on success.

`MPI::Win#get_attr( keyval )`
Returns attribute based on key value if exists; returns nil otherwise.

`MPI::Win#delete_attr( keyval )`
Deletes the (key value, attribute) pair and returns true on success.

Figure A.1: MPI Ruby RMA bindings for communication

`MPI::Win#fence()`
Barrier synchronization for the window. All remote operations are guaranteed to have completed afterwards. Returns true on success.

`MPI::Win#group()`
Returns communicator's group (of type MPI::Group) used to create the window.

`MPI::Win#start( to_group )`
Starts access epoch (active target synchronization) and returns true on success.

`MPI::Win#complete()`
Completes access epoch (active target synchronization) and returns true on success.

`MPI::Win#post( from_group )`
Begins exposure epoch and returns true on success.

`MPI::Win#wait()`
Ends exposure epoch and returns true on success.

`MPI::Win#test()`
Checks exposure epoch and returns true if completed, false otherwise.

`MPI::Win#lock( lock_type, rank )`
Starts access epoch (passive target synchronization) on process's window. Returns true on success.

`MPI::Win#unlock( rank )`
Completes access epoch (passive target synchronization) on process's window. Returns true on success.

Figure A.2: MPI Ruby RMA bindings for synchronization

process id, and the index of the destination's logical array. Any valid Ruby index is permissible, data of any type or size is allowed, and any valid MPI destination rank is legal.

```
MPI::Win#put(obj, dest, index)  # Fixnum dest and index
```

To extract data from a remote array, the user calls `get()`. The arguments are simply the remote process's id and the index of the array. Again, any index valid in Ruby is valid for MPI Ruby, and the `get()` method follows the Ruby semantics for arrays. This function, however, returns a "request" object rather than the actual value. The reason is that the results of `get()` might not have finished until after the synchronization. Once the user has signalled the end of the "epoch" with a synchronization, the data may be retrieved from the "request" object.

```
req = MPI::Win#get(src, index)  # Fixnum src and index
obj = req.object()              # MPI::Getrequest req
```

Because `put()` merely places the data, there is another function used to perform a remote computation. This function, called `accumulate()`, is similar to `put()` except that an MPI operation is also provided in the argument list. As with the traditional MPI semantics, any of the standard operators may be used; however, only the standard-defined operators are allowed. No user-defined operators may be used with `accumulate()`. However, some user-defined types with overloaded operators may be used.

```
MPI::Win#accumulate(obj, dest, index, op)
                                # Fixnum dest and index
                                # MPI::Op op
```

It is important to note that, as with the traditional MPI semantics, using the same index more than once during an epoch leads to a race condition. The only exception is `accumulate()`, in which the operations are guaranteed to be atomic. The only guarantee in the order of completion for communication is that all `get()` operations will finish before any `put()` or `accumulate()` calls. (There are some cases when the user would like to read an element whose index is determined relative to the target array's boundaries. Writing to the array may expand the boundaries, which could result in unintended values from `get()`'s. So `get()` will finish before `put()`.) However, there is no order within a set of `get()` or `put()` operations.

A.1.3 Attribute Caching

There may be a case in which the user would like to associate data with a window and recall it in the future. MPI allows for this "attached" data, known as an attribute. Attributes are uniquely identified by a key value. To cache an attribute, the user

creates a new key value with `MPI::Keyval.create()` and then calls the manipulation functions as appropriate.

```
# MPI::Keyval keyval
MPI::Win#set_attr( keyval, obj )
MPI::Win#get_attr( keyval )
MPI::Win#delete_attr( keyval )
```

A.1.4 Synchronization

Synchronization in MPI identifies epochs and guarantees that window creation and one-sided communication have completed. There are three different types of synchronization in MPI. The first is a collective barrier in which no process is allowed to continue until all communication has completed. This represented by the function `fence()`.

```
MPI::Win#fence()
```

A more scalable method of synchronization is one in which a process establishes a list of hosts and clients against a window with which to communicate. This is a more general form of synchronization and is useful for cases when communication is fairly static and limited. A host declares an "exposure" epoch against a group of processes that may perform remote operations, whereas a client declares an "access" epoch against a group of processes that may be the target of remote operations. One-sided communication is guaranteed to have completed by the end of the epoch.

```
# designate exposure epoch from an MPI::Group
MPI::Win#post(from_group)
MPI::Win#wait()

# designate access epoch to an MPI::Group
MPI::Win#start(to_group)
MPI::Win#complete()
```

It is important to note that `post()` does not block; however, `start()` *may* block. Therefore, when two processes must open both access and exposure epochs against each other, `post()` must be called before `start()`.

A third method, known as "passive target synchronization", relaxes the need for the target process to identify an exposure epoch. The client process identifies the access epoch with the functions `lock()` and `unlock()`. These names are misleading in that they do not provide a means for handling critical sections as in shared-memory programming. The `lock()` function, for example, is not required to block.

However, the user may request a lock type of `MPI::Win::EXCLUSIVE` (as opposed to `MPI::Win::SHARED`) to indicate that the RMA operations are to be atomic among the client processes. As with all other one-sided communication, though, there is no guarantee as to which process performs communication first, nor is there an order for the completion of operations with a process's access epoch. One may place each operation within its own access epoch to force order within a process, but MPI has no mechanism to order operations among different processes.

```
MPI::Win#lock(lock_type, rank)
MPI::Win#unlock(rank)
```

After a synchronization, the array represented by the window will have been changed to reflect the `put()` and `accumulate()` operations. The new array may be retrieved by calling the `object()` method.

```
obj = MPI::Win#object()
```

A.2 Design and Implementation

Arrays in Ruby are dynamic and heterogeneous, but the traditional MPI specification only allows for statically sized buffers with fixed displacement units. The traditional semantics also do not include a facility for resizing the memory window. Therefore, MPI Ruby cannot simply be a set of wrappers over the C functions. Auxiliary information must be sent in advance to the target so that the appropriately sized buffers may be allocated to receive the incoming data.

Ruby provides a C API that may be used to embed the interpreter in other applications. For our library, the Ruby interpreter has been extended using this API. Thus, the implementation is written in C and uses the existing MPI functions for communication.

In MPI Ruby, each process maintains a table of the auxiliary information for all incoming messages. As will be explained momentarily, this table must be statically sized. Because the number of processes within a communicator is fixed, the table is indexed by process id. Therefore, this table lists, per each process,

- the index of the array whose element is the subject of the communication,
- the size of the data to be placed (or a negative value to request a `get()`), and
- the `accumulate()` operation to be performed (`REPLACE` in the case of `put()`).

Here is an example of the table that each process maintains:

process 0	index 1, size 512, op REPLACE	index 17, size 1024, op SUM
process 1		
process 2	index 0, size −1, -	
⋮		
process n-1	index 3, size 20, op REPLACE	

Of course, the target must be alerted that it has incoming data so that it may perform the necessary network communications to receive the appropriate messages. To both accomplish this alert and fill the table of outstanding messages, MPI Ruby uses the C `MPI_Put()` function with the table as a memory window. Because a memory window in C must be statically sized, the table must be statically sized, as mentioned above.

During synchronization, a process checks its table for non-zero size entries. For positive entries (indicating `put()` or `accumulate()`), the process allocates a buffer of the appropriate size and then attempts to receive the data; for negative entries (indicating `get()`), the process sends to the requester both the size of the array element and the actual data. Once all data has been transferred, a process packs the buffers into the array at the indicated index using the appropriate operator.

Here is an outline of `accumulate()`, which is simply a generalized `put()`:

1. Pack (index, size, op) tuple into temporary buffer array.
2. Call `MPI_Put()` to remotely place the buffer into the target's table of outstanding messages.
3. Call `MPI_Isend()` to schedule the object's data disbursement.

The `get()` command is as follows:

1. Pack (index, −1, -) tuple into temporary buffer array.
2. Call `MPI_Put()` to remotely place the buffer into the target's table of outstanding messages.
3. Call `MPI_Irecv()` to obtain the data's size.
4. Return a `Getrequest` object to encapsulate the data to be retrieved later.

The receipt of the actual data for a "get request" is performed during synchronization. The `fence()` function is as follows:

1. Call `MPI_Win_fence()` to ensure outstanding message table is complete.

2. For each `put()` request, allocate enough space and call `MPI_Irecv()`.
3. For each `get()` request, transmit size of data element with `MPI_Isend()`.
4. Wait until all new outstanding messages, both outgoing and incoming, are done.
5. Send actual data of `get()` requests with `MPI_Isend()`.
6. Obtain data to fulfill `get()` with `MPI_Irecv()` and pack into array of `Getrequest` object.
7. Using data resulting from `accumulate()` / `put()` requests, apply appropriate operation and store result into array.

With multiple `MPI_Send()`'s and `MPI_Recv()`'s, it is possible that there could be a conflict among the tags used in MPI Ruby. There is a simple solution to address this issue. Because multiple accesses to an index within an epoch is illegal, according to our specification, the index may be used as the tag. To further ensure that there is no conflict between the tags used in MPI Ruby and the tags used by the user, a duplicate communicator is employed rather than the original. Thus, as long as there is no violation of the requirements, there will be no conflict in tag use.

When MPI Ruby was created, neither of the major open-source implementations of MPI-2 (Open MPI and MPICH2) had completed passive synchronization. Therefore, the `lock()` and `unlock()` functions were never implemented.

A.3 Examples

Because Ruby is a very expressive language, its MPI bindings allow for many of the high-level features present in arrays. For example, if one were to write to an out-of-bound index, the array would expand to accept the element. Also, one may read from negative indices to wrap around the end of the array. Furthermore, practically any data is allowed at any element in the array. The MPI bindings were created with these novelties in mind. Here is a demonstration:

```
world = MPI::Comm::WORLD
pid = world.rank

win = MPI::Win.create([1, "Text", {"hash"=>5}], world)
win.fence

root = 0
case pid
when 0
  win.put([1, 2, 3], root, 4)
```

```
when 1
  req = win.get(root, -1)
when 2
  win.put(2, root, 1)
when 3
  win.accumulate(1, root, 0, MPI::Op::SUM)
end

win.fence

# win.object on the root is now:
#       [2, 2, {"hash"=>5}, nil, [1, 2, 3]]
# req.object on process 1 is now:
#       {"hash"=>5}
```

A nice feature in RMA is that there is no risk of deadlock should two processes communicate with each other simultaneously. In addition, the scalable synchronization of MPI-2 may be used when communication does not involve all processes. Here is an example to illustrate both of these features:

```
# "data" has been computed and now must be
# traded between processes 0 and 1

if ( pid <= 1 )
  target = 1 - pid
  group = win.group.incl([target])
  win.post(group)
  win.start(group)
  win.put(data, target, 0)
  win.complete
  win.wait
end
```

A.4 Comparing RMA Speeds of Ruby and C

The remote memory access functions of Ruby may be easier to use than those of C, but the Ruby bindings' implementation will be slower because it introduces overhead on top of the C library. This section examines the performance differences between the Ruby and C versions.

Ultimately, we will examine the amount of time for two processes to swap data of varying sizes to determine the effects of Ruby relative to the time for communication. The easiest means for testing is to time a loop calling `put()` and then to calculate the

average. However, because communication is separate from synchronization in RMA, the message-passing must be completed with a call to `fence()` in each iteration. Thus, the test merely appears as

```
start = MPI.wtime
for i in 1..max do
  win.put(data, target, 0)
  win.fence
end
finish = MPI.wtime
time = finish - start
average = time/max
```

The calls to `fence()` incur their own overhead. So before this test was run (on a shared memory machine), the call to `put` was commented out to determine the time for synchronization. This value (0.596 ms for C and 1.369 ms for Ruby) was then subtracted from all measured timings of the communication. The results of both C's and Ruby's communication appear in Table A.1. Note that there are no times in Ruby for messages of size greater than or equal to 128 KB. The reason is that the program crashed, possibly because of either Ruby's or MPI Ruby's method of handling buffers for the dynamic types.

Message size (KB)	C time (ms)	Ruby time (ms)
1	0.574	2.900
2	0.612	3.654
4	0.773	3.740
8	0.906	4.217
16	1.525	7.143
32	2.913	10.779
64	5.476	14.254
128	9.113	N/A
256	17.283	N/A

Table A.1: Communication times for `MPI_Put()` in Ruby and C

Figure A.3 shows a plot of this data. It is interesting to note that the growth of the C version is linear, whereas the Ruby version's growth is much less. As the message size increases, the overhead effects of Ruby diminish compared to the time required to communicate the message. Had the Ruby version not crashed, it is possible that the performance difference between Ruby and C would be negligible for megabyte-size messages.

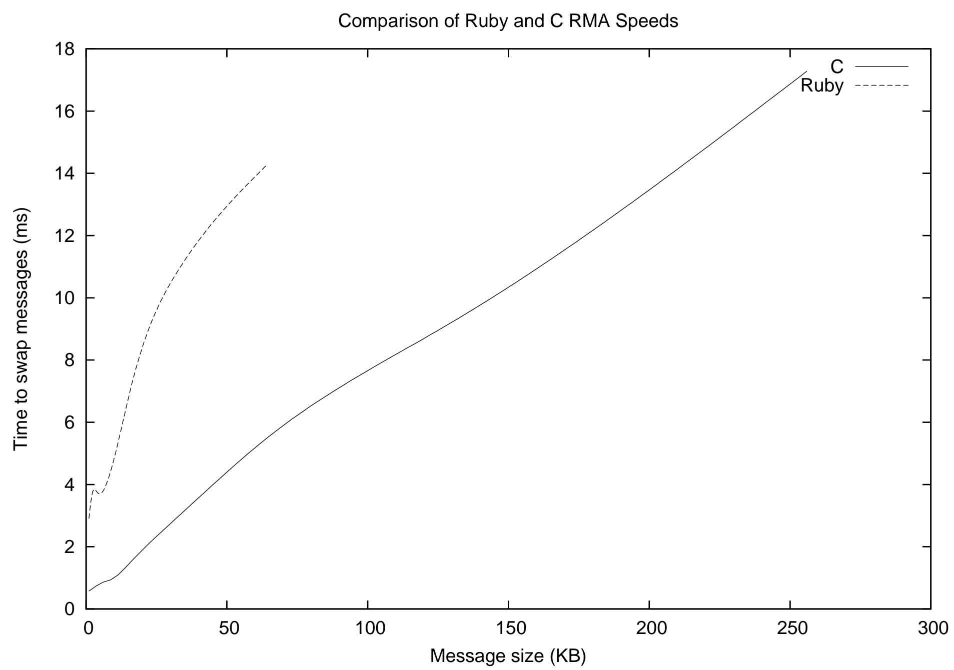

Figure A.3: Communication times for `MPI_Put()` in Ruby and C

Bibliography

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the USENIX Annual Technical Conference*, pages 289–302, 2002.

[2] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] Don Anderson. *HyperTransport Architecture.* Addison-Wesley, 2003.

[4] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang.* Prentice-Hall, second edition, 1996.

[5] Christopher C. Aycock. MPI Ruby with Remote Memory Access. In *Proceedings of the Thirteenth IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, pages 280–281, 2004.

[6] Christopher C. Aycock. Why Pretend? *HPCwire*, 15(40), 2006.

[7] Christopher C. Aycock. Innovation and Commoditization in High Performance Computing. *HPCwire*, 16(3), 2007.

[8] J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Science*, 335:131–146, May 2005.

[9] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS Parallel Benchmark Results. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, 1992.

[10] Pavan Balaji, Wu chun Feng, and Dhabaleswar K. Panda. Bridging the Ethernet-Ethernot Performance Gap. *IEEE Micro*, 26(3):24–40, 2006.

[11] Christian Bell and Dan Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03).* IEEE Computer Society, 2003.

[12] Christian Bell, Wei-Yu Chen, Dan Bonachea, and Katherine Yelick. Evaluating Support for Global Address Space Languages on the Cray X1. In *Proceedings of the 18th International Conference on Supercomputing (ICS 2004)*. ACM Press, 2004.

[13] David Bell and Jane Grimson. *Distributed Database Systems*. Addison-Wesley, 1992.

[14] Siegfried Benkner and Thomas Brandes. Efficient Parallel Programming on Scalable Shared Memory Systems with High Performance Fortran. *Concurrency and Computation: Practice and Experience*, 14(8-9):789–803, 2002.

[15] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, Bill Pugh, P. Sadayappan, Jaime Spacco, and Chau wen Tseng. Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC '03)*. Springer-Verlag, 2003.

[16] Boris Bialek. DB2 Enterprise Server Edition Utilizing InfiniBand Technology. In *OpenIB Developers Workshop*, 2005.

[17] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[18] Dan Bonachea. GASNet Specification, v1.1. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[19] Dan Bonachea and Jason Duell. Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations. *International Journal on High Performance Computing and Networking*, 1(1/2/3):91–99, 2004.

[20] Robert A. Breyer and Sean Riley. *Introduction to Fast Ethernet and Ethernet Switching*. Ziff-Davis, 1995.

[21] Ron Brightwell, Doug Doerfler, and Keith D. Underwood. A Preliminary Analysis of the InfiniPath and XD1 Network Interfaces. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[22] Ron Brightwell, Bill Lawry, Arthur B. MacCabe, and Rolf Riesen. Portals 3.0: Protocol Building Blocks for Low Overhead Communication. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02)*, pages 164–173. IEEE Computer Society, 2002.

[23] Ron Brightwell and Keith D. Underwood. An Analysis of the Impact of MPI Overlap and Independent Progress. In *Proceedings of the 18th annual international conference on Supercomputing (ICS 2004)*, pages 298–305. ACM Press, 2004.

[24] Darius Buntinas and William Gropp. Designing a Common Communication Subsystem. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2005.

[25] Darius Buntinas and William Gropp. Understanding the Requirements Imposed by Programming-Model Middleware on a Common Communication Subsystem. Technical Report ANL/MCS-TM-284, Argonne National Labs, 2005.

[26] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. NFS over RDMA. In *NICELI '03: Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 196–208, 2003.

[27] Don Cameron and Greg Regnier. *Virtual Interface Architecture.* Intel Press, 2002.

[28] François Cantonnet, Yiyi Yao, Smita Annareddy, Ahmed S. Mohamed, and Tarek A. El-Ghazawi. Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003).* IEEE Computer Society, 2003.

[29] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[30] Lei Chai, Ranjit Noronha, and Dhabaleswar K. Panda. MPI over uDAPL: Can High Performance and Portability Exist Across Architectures? In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 19–26, 2006.

[31] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP.* Academic Press, 2001.

[32] Chi-Chao Chang, Grzegorz Czajkowsk, Thorsten von Eicken, and Carl Kesselman. Evaluating the Performance Limitations of MPMD Communication. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing.* ACM Press, 1997.

[33] B. Chapman, F. Bregier, A. Patil, and A. Prabhakar. Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems. *Concurrency and Computation: Practice and Experience*, 14(8-9):713–739, 2002.

[34] Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, and Katherine Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS 2003)*, pages 63–73. ACM Press, 2003.

[35] Yaofei Chen, Rose Dios, Ali Mili, Lan Wu, and Kefei Wang. An Empirical Study of Programming Language Trends. *IEEE Software*, 22(3):72–79, 2005.

[36] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault Tolerant High Performance Computing by a Coding Approach. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 2005.

[37] Ron Choy, Alan Edelman, John R. Gilbert, Viral Shah, and David Cheng. Star-P: High Productivity Parallel Computing. In *Proceedings of the Eighth Annual Workshop on High Performance Embedded Computing (HPEC 2004)*, 2004.

[38] Wu chun Feng, Justin (Gus) Hurwitz, Harvey Newman, Sylvain Ravot, R. Les Cottrell, Olivier Martin, Fabrizio Coccetti, Cheng Jin, Xiaoliang (David) Wei, and Steven Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters, and Grids: A Case Study. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[39] Salvador Coll, Duato Duato, Fabrizio Petrini, and Francisco J. Mora. Scalable Hardware-Based Multicast Trees. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2003.

[40] Salvador Coll, José Duato, Francisco J. Mora, Fabrizio Petrini, and Adolfy Hoisie. Collective Communication Patterns on the Quadrics Network. *Performance Analysis and Grid Computing*, pages 93–107, 2004.

[41] John R. Corbin. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls*. Springer-Verlag, 1991.

[42] Andy Currid. TCP Offload to the Rescue. *Queue*, 2(3):58–65, 2004.

[43] Robert Cypher and Eric Leu. The Semantics of Blocking and Nonblocking Send and Receive Primitives. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 729–735, 1994.

[44] Dennis Dalessandro, Pete Wyckoff, and Gary Montry. Initial Performance Evaluation of the NetEffect 10 Gigabit iWARP Adapter. In *2006 IEEE International Conference on Cluster Computing*, pages 1–7, 2006.

[45] DAT Collaborative. *uDAPL Specification*. `http://www.datcollaborative.org/udapl.html`.

[46] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. The direct access file system. In *FAST '03: 2nd USENIX Conference on File and Storage Technologies*, pages 175–188, 2003.

[47] Lloyd Dickman, Greg Lindahl, Dave Olson, Jeff Rubin, and Jeff Broughton. PathScale InfiniPath: A First Look. In *HOTI '05: Proceedings of the 13th Symposium on High Performance Interconnects*, pages 163–165, 2005.

[48] Jack Dongarra, Thomas Sterling, Horst Simon, and Erich Strohmaier. High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions. *Computing in Science and Engineering*, 7(2):51–59, 2005.

[49] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, 1998.

[50] Roy Thomas Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[51] Samuel A. Fineberg and Don Wilson. Performance Measurements of a User-Space DAFS Server with a Database Workload. In *NICELI '03: Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 185–195, 2003.

[52] FIX Protocol Organization. *The FIX Protocol Standard*. `http://www.fixprotocol.org/`.

[53] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[54] Ian Foster and Carl Kesselmen. The Globus Toolkit. In *The Grid: Blueprint for a New Computing Infrastructure*, pages 257–278. Morgan Kaufmann, 1999.

[55] Basilio B. Fraguela, Jia Guo, Ganesh Bikshandi, María J. Garzarán, Gheorghe Almási, José Moreira, and David Padua. The Hierarchically Tiled Arrays Programming Approach. In *Proceedings of Seventh Workshop on Languages, Compilers and Run-Time Support for Scalable Systems*, 2004.

[56] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[57] Jesse James Garrett. *Ajax: A New Approach to Web Applications*. `http://www.adaptivepath.com/`.

[58] David Geer. Chip Makers Turn to Multicore Processors. *IEEE Computer*, 38(5):11–13, 2005.

[59] Patrick Geoffray. A Critique of RDMA. *HPCwire*, 15(33), 2006.

[60] Pam F. Gorder. Multicore Processors for Science and Engineering. *Computing in Science and Engineering*, 9(2), 2007.

[61] William Gropp. Learning from the Success of MPI. In *Proceedings of the 8th International Conference on High Performance Computing (HiPC '01)*, pages 81–94. Springer-Verlag, 2001.

[62] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI–The Complete Reference: Volume 2, The MPI Extensions*. MIT Press, 1998.

[63] Attila Gürsoy and Laxmikant V. Kale. Performance and Modularity Benefits of Message-Driven Execution. *Journal of Parallel and Distributed Computing*, 64(4):461–480, 2004.

[64] David B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(1):10–22, 1992.

[65] Shawn Hansen and Sujal Das. Fabric-Agnostic RDMA—Fabric-Agnostic RDMA with OpenFabrics Enterprise Distribution. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[66] Ralph Hartley. Transmission of Information. Technical report, Bell Systems, 1928.

[67] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[68] Lars P. Huse and Ole W. Saastad. The Network Agnostic MPI - Scali MPI Connect. In *Proceedings of the 10th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 294–301. Springer-Verlag, 2003.

[69] Intel GmbH, Germany. Intel MPI Benchmarks. Users Guide and Methodology Description, 2004.

[70] Interconnect Transport API Working Group. *Interconnect Transport API Specification*. `http://www.opengroup.org/icsc/native/`.

[71] Jae-Wan Jang and Jin-Soo Kim. Design Issues and Performance Comparisons in Supporting the Sockets Interface over User-Level Communication Architecture. *The Journal of Supercomputing*, 39(2):205–226, 2007.

[72] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, Darius Buntinas, Rajeev Thakur, and William Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication over InfiniBand Clusters. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, 2004.

[73] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, William Gropp, and Rajeev Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. In *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004)*. IEEE, 2004.

[74] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A User-Centred Approach to Functions in Excel. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming (ICFP '03)*, pages 165–176. ACM Press, 2003.

[75] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: Parallel Programming with Message-Driven Objects. In *Parallel Programming Using C++*, pages 175–213. MIT Press, 1996.

[76] Humaira Kamal, Brad Penoff, and Alan Wagner. SCTP versus TCP for MPI. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005.

[77] Charles H. Koelbel. *The High Performance Fortran Handbook*. MIT Press, 1993.

[78] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@HOME – massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, 2001.

[79] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 2006.

[80] Reuven M. Lerner. At the Forge: Introducing SOAP. *Linux Journal*, March 2001.

[81] Eliezer Levy and Abraham Silberschatz. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys*, 22(4):321–374, 1990.

[82] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

[83] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabaleswar K. Panda, David Ashton, Darius Buntinas, William Gropp, and Brian Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, pages 16–25. IEEE Computer Society, 2004.

[84] Jiuxing Liu, Amith R. Mamidala, and Dhabaleswar K. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, pages 10–19. IEEE Computer Society, 2004.

[85] Jiuxing Liu and Dhabaleswar K. Panda. Implementing Efficient and Scalable Flow Control Schemes in MPI over InfiniBand. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, pages 183–190. IEEE Computer Society, 2004.

[86] Jiuxing Liu, Abhinav Vishnu, and Dhabaleswar K. Panda. Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation. In *Proceedings of the 18th Annual International Conference on Supercomputing*. ACM Press, 2004.

[87] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 295–304. ACM Press, 2003.

[88] Piotr R. Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[89] Supratik Majumder, Scott Rixner, and Vijay S. Pai. An Event-Driven Architecture for MPI Libraries. *The Los Alamos Computer Science Institute Symposium*, 2004.

[90] Amith R. Mamidala, Jiuxing Liu, and Dhabaleswar K. Panda. Efficient Barrier and Allreduce on IBA Clusters Using Hardware Multicast and Adaptive Algorithms. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing (Cluster 2004)*. IEEE, 2004.

[91] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly & Associates, 2002.

[92] Charles R. Maule. Ethernet Interconnects—iWARP Ethernet. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[93] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, 1976.

[94] Bertrand Meyer and Karine Arnout. Componentization: The Visitor Example. *IEEE Computer*, 39(7):23–30, 2006.

[95] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium (IPPS/SPDP '99)*, pages 533–546. Springer Verlag, 1999.

[96] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal on High Performance Computing Applications*, 20(2):203–231, 2006.

[97] Emil Ong. MPI Ruby: Scripting in a Parallel Environment. *Computing in Science and Engineering*, 4(4):78–82, 2002.

[98] Oracle Corporation. Oracle9i Real Application Clusters: Cache Fusion Delivers Scalability. An Oracle White Paper, 2002.

[99] Joachim Parrow. Expressiveness of Process Algebras. *Electronic Notes in Theoretical Computer Science*, 209:173—186, April 2008.

[100] David A. Patterson. Latency Lags Bandwidth. *Communications of the ACM*, 47(10):71–75, October 2004.

[101] Kevin Pedretti and Ron Brightwell. A NIC-Offload Implementation of Portals for Quadrics QsNet. In *Fifth LCI International Conference on Linux Clusters*, 2004.

[102] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22:46–57, 2002.

[103] Khoi Anh Phan, Zahir Tari, and Peter Bertok. A Benchmark on SOAP's Transport Protocols Performance for Mobile Applications. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1139–1144, 2006.

[104] C. Phillips and Ronald H. Perrott. Problems with Data Parallelism. *Parallel Processing Letters*, 11(1):77–94, 2001.

[105] Gordon Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.

[106] Renato John Recio. Server I/O Networks Past, Present, and Future. In *NICELI '03: Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence*, pages 163–178, 2003.

[107] RMDA Consortium. *Architectural Specifications for RDMA over TCP/IP*. `http://www.rdmaconsortium.org/`.

[108] RNIC Programming Interface Working Group. *RNIC Programming Interface Specification*. `http://www.opengroup.org/icsc/rnicpi/`.

[109] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[110] Duncan Roweth and Ashley Pittman. Optimised Global Reduction on QsNet II. In *HOTI '05: Proceedings of the 13th Symposium on High Performance Interconnects*, pages 23–28. IEEE Computer Society, 2005.

[111] Tom Shanley. *InfiniBand Network Architecture*. Addison-Wesley, 2002.

[112] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. NetPIPE: A Network Protocol Independent Performace Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[113] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI–The Complete Reference: Volume 1, The MPI Core*. MIT Press, 1998.

[114] Lawrence Snyder. *Programming Guide to ZPL*. MIT Press, 1999.

[115] Sockets API Extensions Working Group. *Extended Sockets API Specification*. http://www.opengroup.org/icsc/sockets/.

[116] Shinji Sumimoto, Hiroshi Tezuka, Atsushi Hori, Hiroshi Harada, Toshiyuki Takahashi, and Yutaka Ishikawa. The Design and Evaluation of High Performance Communication Using a Gigabit Ethernet. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*, pages 260–267, 1999.

[117] Sun Microsystems. Internet Engineering Task Force, RFC 1094: NFS: Network File System Protocol Specification, March 1989.

[118] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Symposium on Principles and Practice of Parallel Programming (PPOPP'06)*, 2006.

[119] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pindown Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *Proceedings of 12th International Parallel Processing Symposium*, pages 308–314. IEEE Computer Society, 1998.

[120] Keith D. Underwood and Ron Brightwell. The Impact of MPI Queue Usage on Message Latency. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP 2004)*, pages 152–160. IEEE Computer Society, 2004.

[121] Steven P. VanderWiel, Daphna Nathanson, and David J. Lilja. Complexity and Performance in Parallel Programming Languages. In *Proceedings of the 1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS '97)*. IEEE Computer Society, 1997.

[122] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266. ACM Press, 1992.

[123] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.

[124] Weikuan Yu, Ranjit Noronha, Shuang Liang, and Dhabaleswar K. Panda. Benefits of High Speed Interconnects to Cluster File Systems: A Case Study with Lustre. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.