**AN MPI TOOL TO MEASURE APPLICATION SENSITIVITY TO
VARIATION IN COMMUNICATION PARAMETERS**

by

EDGAR A. LEÓN BORJA

B.S., Computer Science, Universidad Nacional Autónoma de México, 2001

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

**Master of Science
Computer Science**

The University of New Mexico
Albuquerque, New Mexico

May, 2003

©2003, Edgar A. León Borja

Dedication

To Karla for her love.

Acknowledgments

I would like to thank Barney Maccabe for being a mentor to me, for giving me the opportunity to work with him, for motivating and directing the research, and for assistance with analysis and writing (this is not an all-inclusive listing); Jim Otto for his help in understanding the internals of GM; Roy Heimbach and Jonathan Atencio for their participation on the set up of my work environment on the Roadrunner Linux cluster; Ron Brightwell for his participation on debugging some of the Fortran applications; David Bader for providing some of the applications; Patrick Bridges for his editorial comments on several portions of this document; the Albuquerque High Performance Computing Center for their support of this research through hardware from and access to Roadrunner; and last but not least, Sandia National Labs for financial support.

**AN MPI TOOL TO MEASURE APPLICATION SENSITIVITY TO
VARIATION IN COMMUNICATION PARAMETERS**

by

EDGAR A. LEÓN BORJA

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

**Master of Science
Computer Science**

The University of New Mexico
Albuquerque, New Mexico

May, 2003

AN MPI TOOL TO MEASURE APPLICATION SENSITIVITY TO VARIATION IN COMMUNICATION PARAMETERS

by

EDGAR A. LEÓN BORJA

B.S., Computer Science, Universidad Nacional Autónoma de México, 2001

M.S., Computer Science, University of New Mexico, 2003

Abstract

This work describes an apparatus which can be used to vary communication performance parameters for MPI applications, and provides a tool to analyze the impact of communication performance on parallel applications. Our apparatus is based on Myrinet (along with GM). We use an extension of the LogP model to allow greater flexibility in determining the parameter(s) to which parallel applications may be sensitive. We show that individual communication parameters can be independently controlled within a small percentage error.

We also present preliminary results from applying this tool to a suite of parallel applications that represent a variety of (a) degree of exploitable concurrency, (b) computation to communication ratio and (c) frequency and granularity of inter-processor communications.

Contents

List of Figures	x
List of Tables	xii
1 Introduction	1
2 Background	3
2.1 Glenn’s Messages (GM)	3
2.2 The Message Passing Interface on GM (MPICH-GM)	8
2.3 The $Lo_s o_r g_s g_r GP$ communication parameters	9
3 Implementation	13
3.1 Latency (L)	13
3.2 Overhead ($o_s o_r$)	16
3.3 Gap per message ($g_s g_r$)	18
3.4 Gap per byte for long messages (G)	20

Contents

4	Validation and Calibration	23
4.1	Measurement Methodology	23
4.2	Results	26
5	Parallel Applications	32
5.1	Benchmark suite	32
5.1.1	Algorithmic	33
5.1.2	ASCI Purple	35
5.1.3	NPB 2	43
5.2	Sensitivity to communication parameters	46
5.2.1	Latency	46
5.2.2	Overhead	47
5.2.3	Gap	48
5.2.4	Gap per byte	49
5.2.5	Summary	50
6	Related Work	54
7	Conclusions	57
A	Sensitivity graphs per application	58
	References	63

List of Figures

2.1	The GM MCP and its interaction with user space	7
3.1	Implementation of latency and receive overhead	14
3.2	Implementation of send overhead	17
3.3	Implementation of send and receive gap	19
3.4	Implementation of Gap	21
4.1	Expected micro-benchmark signature of the LogP parameters	25
5.1	Force-decomposition of the force matrix	39
5.2	Sensitivity to latency	48
5.3	Sensitivity to receive overhead	49
5.4	Sensitivity to send overhead	50
5.5	Sensitivity to receive gap	51
5.6	Sensitivity to send gap	52
5.7	Sensitivity to Gap	53

List of Figures

A.1	Aztec sensitivity	59
A.2	FFTW sensitivity	59
A.3	IS sensitivity	60
A.4	LU sensitivity	60
A.5	ParaDyn sensitivity	61
A.6	Select sensitivity	61
A.7	SMG2000 sensitivity	62
A.8	SP sensitivity	62

List of Tables

4.1	Varying L for 8 byte messages	27
4.2	Varying o_s for 8 byte messages	28
4.3	Varying o_r for 8 byte messages	28
4.4	Varying g_s for 8 byte messages	29
4.5	Varying g_r for 8 byte messages	30
4.6	Varying G' for 4088 byte messages	30
5.1	Running time of applications in a 16-node cluster	47
5.2	Summary of application sensitivity to variation in communication parameters	52

Chapter 1

Introduction

Parallel architectures are driven by the needs of applications. Application developers and users often deal with a variety of problems regarding the performance of applications on specific parallel architectures. The origin of these problems is diverse and may be related to different aspects of the application and the platform, such as scalability, latency and bandwidth issues, balance between communication and computation, etc. We have created an apparatus to identify some of these problems regarding communication performance. This tool can be used to improve overall application performance and also to make decisions on which parallel architectures applications may or may not be suitable.

This work describes a tool to vary communication parameters on high-performance computational clusters. This tool is useful in analyzing the requirements and sensitivity of applications to communication performance. Our apparatus is based on the LogP model [13], which has been useful in characterizing communication performance [30, 23, 26, 27, 29]. This model has been extended in different ways to provide specific characterizations in terms of message size, type of network, etc. We also extend this model to specify a more fine-grain characterization of the network, providing greater detail in the sensitivity of applications to network performance.

Chapter 1. Introduction

Over the last few years, Myrinet [9] has become the high-performance cluster interconnect of choice. Consequently, we have implemented our tool on top of Myrinet networks. Together with Myrinet, the GM (Glenn’s Messages) message-passing system [31] has been widely used and, in many cases, has served as the basis for customized communication systems that use Myrinet. In this work, we have instrumented an extension of the LogP communication parameters into GM so we can vary communication performance. Despite the fact that we implement our apparatus in GM, we do not expect applications to be written in GM to benefit from it. Applications can be written in MPI (Message Passing Interface), which defines a standard, portable message-passing system [21]. MPI has been ported to a variety of platforms, in particular to Myrinet through GM.

Our apparatus allows a better understanding of parallel applications and may identify possible communication bottlenecks that would otherwise be hard to detect. This is done by determining which communication parameter(s) the application may be more sensitive to. In addition, it provides application designers with a better understanding of the architectural requirements of their applications, and if a certain platform (or upgrade) may or may not improve the performance of their applications in a significant way.

The remainder of this document is organized as follows. Chapter 2 provides the background of this work. In particular, Sections 2.1 and 2.2 give a brief overview of GM and MPICH-GM respectively. A description of the communication parameters used in our apparatus is presented in Section 2.3. Chapter 3 describes the instrumentation of the LogP parameters in GM. Chapter 4 describes our measurement methodology and shows the results of our empirical validation and calibration of the communication parameters. Chapter 5 describes the application suite as well as sensitivity results of applications to communication parameters. A discussion of related work is presented in Chapter 6. Finally, Chapter 7 presents our conclusions and outlines our intentions for future work.

Chapter 2

Background

This chapter provides an introduction to the background material related to this work. In particular, we describe the GM message-passing system, the implementation specifics of MPI on top of GM, and the model of communication performance used throughout this work and based on the LogP model.

2.1 Glenn’s Messages (GM)

GM is a low-level message-passing system created by Myricom¹ as a communication layer for its Myrinet networks. GM is comprised of a kernel driver, a user library, and a program that runs on the Myrinet network interface, called MCP (Myrinet Control Program). A Myrinet interface (also called Network Interface Controller or NIC) is, in turn, comprised of its core, the LANai² chip, and local memory. The LANai [32], contains an “E bus”

¹<http://www.myri.com>.

²*Lanai* is a Hawaiian word for a “veranda” or “porch”. Myricom started referring to Myrinet-interface chips by this name because a lanai is a transition between the home (the host computer) and the outside world (the Myrinet fabric). Lanai has been spelled “LANai” to include the idea of a Local Area Network.

Chapter 2. Background

interface (logic to/from the host), a RISC (Reduced Instruction Set Computer) processor and a packet interface (logic to/from the network).

GM provides reliable, in order delivery of messages. It allows any non-privileged user to use the network interface directly without the intervention of the host Operating System (OS). This technique, called OS-bypass, actually offloads part of the OS functionality to the network interface, avoiding the need to interrupt the OS to handle message sends or receives. GM has several internal queues to handle the communication between the LANai and the user. These queues reside in either host virtual memory or LANai memory.

GM provides both one and two-sided communication primitives. In the latter, whenever the sender process transmits a message, the message exchange occurs only if the receiver process has previously posted a buffer that matches the incoming message. A posted buffer matches an incoming message if both have the same size and priority. Send and receive operations are examples of a two-sided communication paradigm. In one-sided communication, in contrast, a message transfer takes place without the need to explicitly execute a receive operation on the receiver side, such is the case of remote put and get operations. The MPI implementation uses both types of primitives.

In GM, a message to be sent over the network is divided into fragments called packets. Although Myrinet does not impose any limitation on the size of the fragment to send, GM uses packets of at most 4KB length. This threshold represents the point in which GM reaches its maximum bandwidth.

GM implements two types of flow control, user to NIC and NIC to NIC. The user to NIC flow control is enforced at the user library level and is message-oriented. A user is allowed access to the network by opening a “port”. A port is a GM communication endpoint, and serves as the interface between the user and the network. When a user opens a port, GM assigns to her a number of send and receive tokens. A send token is needed by the user to send a message over the network. Similarly, a receive token is needed in order

Chapter 2. Background

to post a receive buffer. When the send or receive operation completes, the token is passed back to the user. The number of tokens assigned to a user determines how many slots there are to use in certain LANai queues, and is based on a page size of the host machine. By default GM 1.2.3 assigns 29 send tokens and 126 receive tokens to a single user.

The NIC to NIC flow control is enforced at the MCP level and is packet-oriented. GM implements an “ACK/NACK-based go back N” flow control protocol. GM establishes a connection between any pair of nodes that wish to communicate. When a node sends a packet for the first time, a sequence number is assigned to it. Upon reception of a packet the receiver checks if the expected sequence number matches the sequence number on the incoming packet. If it matches, the receiver acknowledges the sequence number to the sender. Otherwise, the receiver negative acknowledges the expected sequence number, indicating that the sender should retransmit the expected packet and all following packets and implicitly acknowledging all earlier messages.

Upon reception of an ACK (Acknowledgment) or NACK (Negative Acknowledgment), the sender frees all packet descriptors with lower sequence numbers from the sent queue (packet descriptors yet unacknowledged). Upon reception of an NACK, all the packet descriptors with sequence numbers greater than or equal (go back N) are enqueued to the send queue again (packets to be sent). Since the ACKs and NACKs are not delivered reliably, GM also uses timeouts to re-send messages that have not been acknowledged.

The MCP is a state-based program implemented using four state machines: SDMA, SEND, RDMA and RECV. Figure 2.1 illustrates the relationship between these machines. Depending on the state of the “system” the MCP will trigger events for the different state machines interfaces to manage and control the flow of messages between the network and the user, and vice-versa. The state of the system is allocated in the registers ISR (Interface Status Register) and STATE. Register IMR is also used to mask the part of the system’s state in ISR.

Chapter 2. Background

Upon reception of a packet, the RECV state machine checks the packet header. If the header is not valid, the packet is dropped. If the header is valid, the RECV machine DMA's the packet to a buffer or staging area in LANai memory (at this point the packet is called a “chunk”). The LANai memory has two buffers for send chunks and other two for receive chunks. The RECV machine notifies the RDMA machine when chunks in the receive buffers have been written. The RECV machine is also in charge of intercepting ACKs and NACKs and to do the appropriate tasks specified by the flow control mechanism.

The RDMA state machine is in charge of dealing with message chunks in the LANai memory. The RDMA machine checks sequence numbers and performs the appropriate tasks according to the flow control mechanism. It also looks for user receive buffers in the receive queue that matches the message size and priority of the incoming message. If there is a matching buffer for the incoming message, the message is DMAed to the matching buffer and an event describing the receive is enqueued in a receive event queue in host memory. If there is no matching buffer for the incoming message and the message size is greater than a constant (128 bytes), the message is dropped; otherwise, the message is attached to the event describing the receive and is enqueued in the receive event queue.

The user is in charge of polling the receive event queue to check for receive events (and other events). The user performs this action by using one of the library receive functions: `gm_receive()`, `gm_blocking_receive()` and `gm_blocking_receive_no_spin()`.

The user sends a message by invoking one of the GM library functions: `gm_send_with_callback()`, `gm_send_to_peer_with_callback()` and `gm_directed_send_with_callback()`. These functions pass the send descriptor (or token) to the send queue in LANai memory. Note that prior to the send, the user should allocate the message in DMAable memory at the host (this can be done by using the library functions `gm_malloc()` or `gm_register_memory()`), to ensure that the DMA engine would be able to transfer the message directly from user space to LANai memory.

Figure 2.1: The GM MCP and its interaction with user space in a message transmission and reception. Dash rectangles refer to buffer space, square rectangles denominate either a state machine or a queue. Rounded rectangles, identify the areas where the $Lo_s o_r g_s g_r GP$ parameters are implemented.

The SDMA state machine removes an entry from the send queue and begins the transfer of the message to the LANai memory. The transfer is done in chunks that fit in the send

buffers, as the buffers become available. The SDMA machine will also assign sequence numbers to chunks and notify the SEND machine about the chunks that are ready to be sent. The SEND machine then injects the chunks as packets to the network, pre-pending the correct route to the destination node, and recording the send to the sent list.

2.2 The Message Passing Interface on GM (MPICH-GM)

MPICH-GM is a port of MPICH on top of GM, created by Myricom. MPICH is a portable implementation of MPI, the message-passing interface standard. MPICH-GM is a two-level protocol, it uses an *eager* protocol for the transmission of small messages, and a *rendezvous* protocol for long messages. The two-level protocol reflects the trade-offs necessary to achieve low latency and high bandwidth.

The eager protocol allows the transmission of small messages, even when a receive buffer has not been posted on the receiver. The receiver temporarily stores the incoming message until the message is consumed. This technique allows low-latency but low-bandwidth due to the extra copy at the receive side. This protocol is non-blocking since it allows the sender to complete even when there is no matching receive.

To avoid significant overhead in memory copies for long messages, the rendezvous protocol implements a 3-way handshake. The sender transmits a request-to-send (RTS) to the receiver, the receiver replies back to the sender with a clear-to-send (CTS), and finally the sender transmits the messages. The reply from the receiver contains the virtual memory address in which the message should be delivered, thus the sender performs a remote put operation to move the message to its destination. In GM, the remote put operation is implemented via the function `gm_directed_send_with_callback()`. Thus, this protocol achieves higher bandwidth but incurs greater latency due to the handshake.

The transition point between protocols occurs at 16KB by default, although this point

can be modified at run time. This number represents the transition point between short and long messages in MPICH-P4, although the optimal crossover in GM is 4KB. The 16KB threshold has been kept because some MPI applications incorrectly rely on this number and may fall into deadlock if it is changed.

An interesting feature of MPICH-GM is the ability to change the behavior of blocking receive MPI calls at run time (as a parameter to `mpirun`). Three modes are provided: *polling*, *blocking* and *hybrid*. By default GM uses the polling method. Under this method, the MPI blocking receive call translates into the GM function `gm_receive()`, in which the network device is polled until an event is found. This method achieves low latency, but high CPU utilization.

In the blocking method, the MPI function sleeps in the kernel. At the appearance of an event, the network interface delivers an interrupt to the host to awaken the sleeping function. This behavior is achieved through the GM function `gm_blocking_receive_no_spin()`. This method allows low CPU utilization, but the interrupt and context switch costs increase the latency.

In the hybrid method, the MPI function polls for 1 millisecond and then goes to sleep. This method uses the GM function `gm_blocking_receive()`.

2.3 The $Lo_s o_r g_s g_r GP$ communication parameters

Over time, many models of parallel computation have been proposed and no consensus of a unifying model has been reached. It has been difficult to create a model that allows algorithms to be independent of a specific technology and at the same time to take full advantage of the underlying architecture. The fact that the technology for parallel computation is constantly changing between generations makes this task even more complicated.

Many of these models are overly simplistic (PRAM, Parallel Random Access Machine)

Chapter 2. Background

or overly specific (Network models). More recent models such as BSP (Bulk Synchronous Parallel) and LogP (Latency overhead gap Processors) provide a middle ground being general enough to allow algorithms based on these models to be portable and specific enough to reflect critical trends in underlying distributed parallel computers such as computational clusters. As such, they serve as a basis for developing high-performance and portable algorithms.

In this work, we have chosen to use the LogP model to characterize communication performance rather than as a model to develop efficient portable parallel algorithms. LogP provides a better characterization of the network than its predecessor, the BSP model. The LogP model was created a few years after the BSP model, and overcomes some limitations presented by the BSP [13].

The idea of using the LogP model to characterize communication performance is not new and has been successfully used in several studies [30, 23, 26, 27, 29]. LogP has also been extended to provide a finer characterization of the network resulting in new communication parameters such as Gap for long messages, send overhead, receive overhead, etc.

The LogP model is a model for distributed-memory multiprocessors and abstracts the parallel architecture into four parameters:

Latency (L) : an upper bound on the time to transmit a small message (a word or a small number of words) from its source to its destination.

overhead (o) : the time that the host processor is engaged in sending or receiving a message and cannot do any other work.

gap (g) : the minimum time interval between consecutive message transmissions or consecutive message receptions at a node. The reciprocal of g corresponds to the available per-node bandwidth

Chapter 2. Background

Processors (P) : the number of nodes in the system.

This model is asynchronous, processors do not share a common clock, and the latency may vary (although bounded by the parameter L). The model considers that the network has a finite capacity, i.e., at most $\lceil L/g \rceil$ messages may be on the network at any given time.

The LogP model does not differentiate between short and long messages, thus it does not take into consideration the usage of special devices to support the send of long messages. To address this issue, an extension of this model was created, the *LogGP* model [1]. The new parameter, G , defines the time per byte for a long message. As in the LogP model, the reciprocal of G characterizes the available per processor communication bandwidth for long messages.

Under the LogGP model, the time to transmit a small message from one process to another in different nodes takes $o + L + o$ cycles, o cycles of overhead in the send processor, plus the network latency from the sender NIC (Network Interface Controller) to the receiver NIC, plus o cycles of reception overhead. If RTT is the round-trip-time of a message transferred between two nodes, then:

$$RTT = 2(o + L + o) \quad \text{solving for } L, \quad (2.1)$$

$$L = \frac{RTT}{2} - 2o \quad (2.2)$$

The time to transmit a long message of k bytes takes $o + (k - 1)G + L + o$. First, the sender processor initiates the transfer incurring in o , subsequent bytes are sent every G cycles. The last byte enters the network at time $o + (k - 1)G$ and arrives at the host $L + o$ cycles later.

The communication parameters we have decided to use as a characterization of communication performance are based on the LogGP model. We further distinguish overhead and gap as composed of two parts each: the send and receive parts. Considering that the

Chapter 2. Background

send and receive operations are in general not symmetric, we consider two parameters to model the overhead: the send overhead, o_s , and the receive overhead, o_r . These two are considered separately and independently of each other. Under this network characterization, equations 2.1 and 2.2 become:

$$RTT = 2(o_s + L + o_r), \quad (2.3)$$

$$L = \frac{RTT}{2} - o_s - o_r \quad (2.4)$$

An interesting issue arises while considering the gap. We should be able to control the gap independently of the communication primitive being used. By definition, the gap is the minimum time interval between consecutive sends or consecutive receives; if we control just the send side of the gap, in a many-to-one communication semantics, the time between consecutive message receptions at the receiver may be shorter than the gap; if we control just the receive side, in a one-to-many communication semantics, the time between message transmissions at the sender may be shorter than the gap. Thus, we consider the gap as being composed of two parameters: the send gap, g_s , and the receive gap, g_r . We consider these two separately and independently of each other.

In summary, we characterize the communication performance of a parallel machine in terms of seven parameters: L , o_s , o_r , g_s , g_r , G and P . These parameters allow us a high-degree of flexibility to determine the particular parameter(s) to which an application may be sensitive to.

Chapter 3

Implementation

We instrument the communication parameters based on GM version 1.2.3. Timing measurements are done at two levels: (1) at the MCP level, using the Real-Time-Clock (RTC) in the Myrinet interface, and (2) at the host user level, using the function `gettimeofday`. The Myrinet interfaces used in this work have an RTC reference period of $\frac{1}{2}\mu s$. The resolution of `gettimeofday` is less than $1\mu s$.

3.1 Latency (L)

The latency, or time to transmit a small message from NIC to NIC, is increased by delaying the notification of delivery of messages to the user. Although a message has been received, the user is not notified of the delivery until the increased latency has been met. To avoid increasing any other communication parameters, we implement the added latency using a delay queue as described below.

GM maintains a queue of events in host memory (receive event queue) which is filled up by the MCP when an event occurs. The GM user, through the GM library, polls this

Chapter 3. Implementation

queue for new events such as the completion of a receive. The queue is accessed through a port.

To add arbitrary delays to the inherent latency of the system without modifying any other LogP parameters, we implement a delay queue, see Figure 3.1. This queue has the same size as the receive event queue ($GM_NUM_RCV_QUEUE_SLOTS \approx GM_NUM_SEND_TOKENS + GM_NUM_RCV_TOKENS$) and it is accessed through the same port (the delay queue could have been implemented directly in the receive event queue to avoid extra space). When an event is inserted in the event queue, the event is delayed. The time in which the event should be delivered in observance of latency delay (time of arrival + delay) is inserted into the delay queue, and every time the user polls for events, it actually polls the delay queue for events ready to be delivered. Only events ready to be delivered according to the delay queue, are delivered to the user.

Figure 3.1: Implementation of L and o_r . A new event is inserted in the event queue at time τ , o_r has been increased by δ and L has been increased by Δ . If $\Delta = 0$, the event will be delivered at time $\tau + \delta$.

In a message exchange, the latency delay parameter is set in both parties to the desired

Chapter 3. Implementation

added delay value x . Thus, the new latency, L' , in observance of the added delay is given by:

$$\begin{aligned} L' &= \frac{RTT'}{2} - o_s - o_r && \text{by (2.4)} \\ &= \frac{RTT + 2x}{2} - o_s - o_r \\ &= \frac{RTT}{2} - o_s - o_r + x \\ &= L + x && \text{by (2.4)} \end{aligned} \tag{3.1}$$

Thus, we increase the latency by x time reference (μs).

To implement the added latency, we modified the implementations of two library functions. The function `gm_open( )` is in charge of opening and initializing a port to a user. This includes the initialization of pointers to LANai and host-memory queues. In this function we allocate and initialize the delay queue, and also set the arbitrary delay latency value from the user through a file.

The added latency was instrumented in the GM library function `gm_receive( )`, which is in charge of polling the receive event queue. This function returns an event if there is an event to deliver in the event queue or no event if the event queue is empty. We modified the function as follows: when a new event is inserted in the event queue, the function checks if the event is a receive event; if so, the new time in observance with the latency delay is calculated and inserted in the delay queue. Once the new time is calculated, the delay queue is polled for events ready to be delivered. When there are no events pending in the event queue, the delay queue is polled to check for events ready to be delivered.

An important issue arises due to the `gm_unknown( )` library function. The user may pass almost all the events to this function to be handled. For example, the user may be

Chapter 3. Implementation

polling only for `RECV_EVENT` (normal receive event) and the rest of the events passed to this function. If a `FAST_RECV_EVENT` (receive event for small messages) arises, `gm_unknown()` converts this event to a `RECV_EVENT`. To perform the conversion, `gm_unknown()` replaces the type of the event and rewind the current event queue pointer to the previous slot.

To avoid adding delay latency twice for an event such as a `FAST_RECV_EVENT` that was passed to `gm_unknown()`, the delay queue implementation checks for `gm_unknown()` calls by detecting if the current receive queue pointer gets rewound, and if so, just delivers the event to the application without computing the new delay time.

3.2 Overhead (o_s o_r)

The host overhead, or time the host processor is engaged in sending or receiving a message, has been divided in two parts: send overhead (o_s) and receive overhead (o_r). The former refers to the time the host processor is engaged in sending a message. The latter refers to the time the host processor is engaged in receiving a message. In our implementation they can be varied independently of each other, allowing greater flexibility in determining their effect in parallel applications.

The receive overhead has been implemented by putting the host processor to work in a loop (if necessary) after the message has been DMAed into user memory but just before the delivery of the event to the application. Specifically, upon a message reception, the corresponding event is recorded in the receive event queue, in which the user polls for a new event. As Figure 3.1 shows, we delay the completion of the polling function (`gm_receive()`) the amount of time specified by the receive overhead delay. The delay pseudo code follows:

Chapter 3. Implementation

```
while (time_ready_to_deliver > current_time)
    get_current_time(&current_time)
```

The send overhead has been implemented by adding a similar delay loop after the user has initiated the send (through `gm_send_with_callback()`, `gm_send_to_peer_with_callback()` or `gm_directed_send_with_callback()`) and before the transfer of the message to the LANai memory has been initiated, see Figure 3.2. The implementation was added to the library functions `gm_*send_*with_callback()`.

Figure 3.2: Implementation of o_s . Before the message gets written to LANai SRAM, the user is delayed by o_s .

Modifying the send and receive overhead in this fashion allows latency to remain unaffected. Let L be the latency with communication parameters unaffected, and L' the resultant latency after adding a delay of $x \mu s$ to send overhead at both parties in a message exchange, then:

$$L' = \frac{RTT'}{2} - o'_s - o_r \quad \text{by (2.4)} \quad (3.2)$$

$$= \frac{RTT + 2x}{2} - (o_s + x) - o_r \quad \text{thus,} \quad (3.3)$$

$$= \frac{RTT}{2} - o_s - o_r \quad (3.4)$$

$$= L \quad \text{by (2.4)} \quad (3.5)$$

Chapter 3. Implementation

We can use the same argument to see that latency remains unaffected when adding a delay to receive overhead.

The arbitrary increase on the overhead is not independent of the gap. When increasing the send overhead, the gap increases as well because the time between consecutive message sends will be increased by the extra time to execute every send. The same argument is true for receive overhead.

3.3 Gap per message (g_s g_r)

The gap, or minimum time interval between consecutive sends or receives, can be more specifically described as a parameter composed of two parts: send gap (g_s) and receive gap (g_r).

The send gap has been implemented by delaying the send of a message if not enough time has passed after the last send. The receive gap has been implemented by delaying the transfer of the event corresponding to the message reception to the receive event queue if not enough time has passed after the last reception.

The pseudo code for the implementation of the gap is presented below.

```
// time reference is 1/2 us.
while (RTC < ready)
    spin();
ready = RTC + 2*gap_delay
deliver receive event or
pass send descriptor to SEND
```

The send gap implementation was added to the SDMA machine, see Figure 3.3(a). The logic of this machine can be seen as a two-level protocol. At one level, a message is

Chapter 3. Implementation

queued normally using the SEND state machine, at the other a shortcut is taken and the message is sent to the network directly from the SDMA machine.

If the message to send has size greater than `GM_MAX_SHORTCUT_MESSAGE_LEN` (256 bytes) and certain idle state, σ , has been reached, the send is performed by the macro `SHORTCUT_IF_POSSIBLE`. If the current state of the SDMA machine is not σ or the message size is greater or equal than `GM_MAX_SHORTCUT_MESSAGE_LEN`, the send is queued normally (it goes through the SEND machine). We have added the send gap delay in both protocols.

Figure 3.3: Implementation of g_s and g_r . In (a), consecutive insertions of receive events to the event queue is delayed by g_r . In (b), consecutive message sends are delayed by g_s .

The receive gap implementation was added to the RDMA state machine, see Figure 3.3(b). As in the send gap, the logic of the RDMA machine can be seen as a two-level protocol depending on the message size. For messages of size less than `GM_MAX_FAST_RECV_BYTES` (128 bytes), there is no need to have a receive buffer ready, the message itself fits within an event (receive token). The event is DMAed to the receive event queue (in user space) through which the user can access the message.

For messages larger than `GM_MAX_FAST_RECV_BYTES`, the event does not include the message. Once the message has been copied to the user specified buffer, the event will be passed to the event queue. In both receive protocols we added the delay before the event is passed to the receive event queue.

3.4 Gap per byte for long messages (G)

The Gap, or time per byte for a long message, is implemented by adding a delay, G , after every byte sent. To achieve this, we first calculate the number of bytes in every packet to be sent. Then, we add a delay, according to the length of the packet, after the packet has been sent. If the length of the packet is n bytes, then the delay is:

$$n \cdot G = G' \tag{3.6}$$

G' can be calculated using the LogP signature for bursts of bulk messages. If a message exceeds 256 bytes, meaning that a packet exceed 256 bytes, then it is considered a bulk message and the SEND state machine is delayed after the packet was sent as shown in Figure 3.4. It is interesting to note that GM partitions the message data into equal sized chunks, each fitting in one packet without exceeding the minimum number of packets that the message comprises. For example, if 4097 bytes are to be sent, one packet will have ~ 2048 bytes and the other ~ 2049 bytes, instead of 4096 and 1 respectively.

In order to calculate G' , we use the LogP signature for bursts of bulk messages. At the steady state, we calculate G' as the average time to send a message (just as in the LogP model for g). Thus,

Chapter 3. Implementation

Figure 3.4: Implementation of G . After every packet send and before the next send, delay the SEND state machine by G' . Note that G' depends on the size of the packet (G_1 for packet and G_2 for next_packet).

$$BW = \frac{1}{G} \quad (3.7)$$

$$BW = \frac{n}{G'} \quad \text{by (3.6)} \quad (3.8)$$

We increase the size of the message, n , up to a point where the bandwidth does not increase anymore. This point is exactly the packet size, GM_MTU (4096 bytes). Since the MCP does not support division operations, the implementation of the Gap uses bit shifting.

The delay of a packet of size n is:

$$G' = (n \gg x) \mu s = (n/2^x) \mu s \quad \text{thus,}$$

$$BW = \frac{n}{G'} = \frac{n \text{ bytes}}{\frac{n}{2^x} \mu s} = 2^x \text{ MB/s}$$

Therefore, a delay with key equal to x changes the bandwidth to 2^x MB/s.

The implementation of this parameter was added to the SEND state machine in the MCP. The corresponding pseudo code is presented below.

Chapter 3. Implementation

```
// sml = send-message limit
// smp = send-message pointer
// these vars are not the actual registers
packet_length = sml - smp - header_size

// RTC = Real-Time Clock
// time reference is equal to 1/2 us
if (packet_length > 256) {
    while (RTC < ready_to_send)
        spin();
    ready_to_send = RTC + 2*(packet_length >> x);
}

// send the packet by writing to register SMLT
// (Send-Message Limit, with the Tail)
SMLT = sml;
```

Chapter 4

Validation and Calibration

This chapter describes the methodology used to empirically validate and calibrate our tool and presents the validation results. We empirically validate our tool for each communication parameter by measuring the error between the desired parameter value and the measured or observed value.

4.1 Measurement Methodology

To extract the communication parameters of a given parallel machine, we use a micro-benchmark developed by Culler and others [14]. It was implemented on GM and it is based on the Myricom's `logp_test` program. A brief description of this micro-benchmark follows.

The micro-benchmark issues a sequence of M request messages of a fixed length, and measures the average time per issue, the *message cost*. The receiving node sends a reply (of the same length as the request) to the sender for every message received. The sender's pseudo code is presented below.

Chapter 4. Validation and Calibration

```
start timer
  repeat M times
    issue request
stop timer
... handle remaining replies
```

For small M (initial phase) no replies are handled (and hopefully the network has not reached its capacity), therefore the message cost is only the send overhead, o_s . For larger M , the sender begins to receive replies (transition phase) and the message cost increases due to the receive overhead, o_r . This cost keeps increasing to the point where the network has reached its capacity (steady phase). At this point, the sender cannot inject a new message before draining a reply from the network. Thus, after sending a message, the sender waits a period of time, *idle*, then receives a reply from the network, and then sends the next message. Therefore, the gap, g , is comprised of:

$$g = o_s + \textit{idle} + o_r \quad (4.1)$$

which is just the time interval between consecutive sends.

Three stages or phases can be identified from the collection of graphs shown in Figure 4.1. In the initial phase the message cost is just o_s . The transition phase begins either at the reception of the first reply or when the network capacity has been reached, whichever comes first. The reception of the first reply should occur after the round-trip-time, RTT , of the first request, i.e., after the transmission of $\frac{RTT}{o_s}$ number of messages. The steady phase is reached when the network is full and the message cost is just the gap.

In GM the transition phase is reached earlier than expected (when M reaches the number of send tokens, `GM_NUM_SEND_TOKENS`), due to the limitation on the number of sends imposed by the token mechanism.

It is easy to measure g and o_s using the signature graph shown in Figure 4.1: g is measured as the average message cost at the steady phase; o_s is measured as the average

Figure 4.1: Expected micro-benchmark signature of the LogP parameters.

message cost in the initial phase. To calculate o_r from equation 4.1, it is necessary to determine the value of *idle*. Since this value is not known, a delay Δ between message sends is added. As Figure 4.1 shows, for $\Delta > idle$, equation 4.1 becomes:

$$g' = o_s + \Delta + o_r \quad (4.2)$$

Having o_r , is easy to calculate the latency, L , from:

$$\frac{RTT}{2} = o_s + L + o_r \quad (4.3)$$

RTT is easily measured by taking the average of a number (100) of ping-pong trials with the same message size used in the micro-benchmark.

The resulting micro-benchmark follows:

Chapter 4. Validation and Calibration

```
start timer
  repeat M times
    issue request
    compute for Delta time
stop timer
... handle remaining replies
```

The parameters measured depend on the message size used by this micro-benchmark; for example, to get the latency, we use a small message size as defined in the LogP model. However, we can also measure the “latency” for bulk messages although not defined in the original model. Thus, the parameters L , o_s , o_r and g are functions of the message size n : $L(n)$, $o_s(n)$, $o_r(n)$ and $g(n)$.

To measure G (gap per byte), we first measure $g(k)$, where k is the number of bytes at which the maximum bandwidth is reached (4KB in GM); then, G is calculated using equation 3.6 where $G' = g(k)$ (see section 3.4). Therefore, with this micro-benchmark and a ping-pong test to get the round-trip-time, we are able to obtain the values of the LogP parameters.

4.2 Results

The results presented in this section were gathered using two computational nodes. Each node has a 400 MHz Intel Pentium II processor with 512 MB of main memory and a Myrinet LANai 7.2 network interface card with 2 MB of memory. Nodes were connected through a Myrinet network. GM version 1.2.3 and a Linux 2.2.14 kernel were used.

To empirically validate and calibrate the communication parameters, we vary each parameter over a fixed-range of values while leaving the remaining parameters unmodified. Using the micro-benchmark described in the previous section, we measure the value of all

Chapter 4. Validation and Calibration

the parameters and verify that the average error difference (*Err*) between the desired value of the varied parameter and the measured value is small (less than 9%). We also verify that the remaining parameters remain fairly constant and with low standard deviation (*Std*).

Tables 4.1, 4.2 and 4.3 show the results of varying L , o_s and o_r respectively for 8 byte messages. The $+<parameter>$ label in these tables represents the desired added value of the varied parameter to the system, for example if the inherent system's $o_s = 1.34\mu s$, then an added value of $20\mu s$ results in a desired value (*Goal*) of $21.34\mu s$. The *Measured values* column represents the measured values of the communication parameters using the micro-benchmark. The added overhead in Tables 4.2 and 4.3 is not independent of the gap (g). When increasing send overhead, the gap increases as well because the time between consecutive message sends will be increased by the extra time the host processor takes to execute every send. The same is true for receive overhead.

Table 4.1: Varying L for 8 byte messages. All units are given in μs .

				Measured values				
	%Err	+L	Goal	L	o _s	o _r	g	RTT
		0		18.72	1.38	4.31	23.91	48.84
	2.40	10	28.72	29.41	1.39	4.28	24.11	70.19
	2.89	20	38.72	39.84	1.39	4.30	24.03	91.08
	5.87	30	48.72	51.58	1.39	4.29	23.94	114.54
	6.39	40	58.72	62.47	1.39	4.30	24.19	136.33
	3.75	50	68.72	71.30	1.36	4.36	24.06	154.07
	5.44	60	78.72	83.00	1.34	4.35	24.13	177.40
	5.51	70	88.72	93.61	1.39	4.30	23.90	198.64
	4.81	80	98.72	103.47	1.39	5.01	23.94	219.77
	5.49	90	108.72	114.69	1.33	4.37	23.86	240.81
	5.20	100	118.72	124.89	1.40	4.28	24.00	261.16
	5.40	110	128.72	135.67	1.39	4.31	23.94	282.76
	4.97	120	138.72	145.61	1.40	4.30	23.97	302.64
Avg	4.84				1.38	4.36	23.99	
Std	1.21				0.02	0.19	0.09	

Chapter 4. Validation and Calibration

Table 4.2: Varying o_s for 8 byte messages. All units are given in μs .

				Measured values			
%Err	+ o_s	Goal	o_s	o_r	g	RTT	L
	0		1.34	4.33	23.92	48.82	18.73
2.12	10	11.34	11.58	4.12	25.39	68.42	18.49
0.14	20	21.34	21.31	3.82	31.22	88.72	19.21
0.54	30	31.34	31.51	4.18	41.39	108.65	18.62
3.39	40	41.34	42.74	3.07	51.29	128.48	18.41
0.02	50	51.34	51.33	4.19	61.23	151.54	20.24
0.42	60	61.34	61.60	4.09	71.21	169.05	18.83
0.15	70	71.34	71.23	4.17	81.19	191.97	20.57
0.00	80	81.34	81.34	4.34	91.26	212.04	20.32
0.33	90	91.34	91.04	4.67	101.28	232.01	20.28
0.01	100	101.34	101.35	4.18	111.24	252.01	20.46
0.09	110	111.34	111.44	4.35	121.27	271.38	19.90
0.13	120	121.34	121.18	4.63	131.14	290.76	19.55
Avg	0.61			4.16			19.50
Std	1.05			0.39			0.82

Table 4.3: Varying o_r for 8 byte messages. All units are given in μs .

				Measured values			
%Err	+ o_r	Goal	o_r	o_s	g	RTT	L
	0		4.29	1.38	24.20	48.82	18.73
1.47	10	14.29	14.08	1.40	25.19	68.39	18.71
1.73	20	24.29	23.87	1.38	27.62	89.00	19.24
2.77	30	34.29	33.34	1.38	36.64	108.77	19.66
2.55	40	44.29	43.16	1.41	45.84	129.11	19.97
2.30	50	54.29	53.04	1.41	55.25	149.09	20.09
2.69	60	64.29	62.56	1.41	64.49	168.52	20.27
2.44	70	74.29	72.48	1.35	73.67	191.41	21.87
3.76	80	84.29	81.12	1.40	82.83	212.28	23.61
3.05	90	94.29	91.41	1.36	92.29	232.47	23.45
3.25	100	104.29	100.90	1.38	101.66	252.20	23.81
0.70	110	114.29	113.49	1.37	110.83	271.51	20.87
2.16	120	124.29	121.60	1.41	120.13	292.47	23.21
Avg	2.41			1.38			21.03
Std	0.83			0.02			1.91

Chapter 4. Validation and Calibration

Tables 4.4 and 4.5 show the results of varying g_s and g_r respectively for 8 byte messages. The *Goal* column represents the desired value of the varied parameter; naturally a desired value less than the inherent system's value cannot be achieved using our tool. The values in column RTT_1 differ from RTT (round-trip-time) in the number of trials used in the ping-pong test: 1 in the former, and 100 in the latter. Having RTT_1 instead of RTT allow us to keep the latency (L_1) constant when varying the gap. Otherwise, the gap would be injected to RTT between consecutive message transmissions and receptions. Since just one trial is considered in RTT_1 , the value of the latency, L_1 , is greater than L due to *warm-up issues*.

Table 4.4: Varying g_s for 8 byte messages. All units are given in μs .

			Measured values				
	%Err	Goal	g	o _s	o _r	RTT ₁	L ₁
		0	23.91	1.36	4.33	125.07	56.84
		10	24.18	1.36	4.30	125.06	56.85
		20	27.70	1.35	4.33	125.06	56.84
	10.83	30	33.25	1.41	4.25	126.06	57.36
	1.95	40	40.78	1.38	4.28	125.06	56.86
	0.72	50	50.36	1.40	4.26	124.07	56.36
	0.52	60	60.31	1.52	4.15	124.06	56.35
	0.17	70	70.12	1.50	4.17	124.06	56.34
	0.23	80	80.18	1.39	4.27	125.05	56.85
	0.20	90	90.18	1.36	4.33	126.06	57.33
	0.29	100	100.29	1.41	4.27	125.06	56.85
	0.21	110	110.23	1.40	4.31	125.06	56.81
	0.15	120	120.18	1.37	4.37	125.06	56.78
Avg	1.52			1.40	4.27	124.98	56.80
Std	3.31			0.05	0.06	0.63	0.31

Table 4.6 shows the results of varying G' for messages of size 4088 bytes. Column κ represents the input key values, in which a system with key x has a bandwidth (BW) of 2^x MB/s. The RTT is not independent of G (and thus G'), because the messages involved are bulk messages. Consequently, L is not independent of G .

Chapter 4. Validation and Calibration

Table 4.5: Varying g_r for 8 byte messages. All units are given in μs .

			Measured values			
%Err	Goal	g	o_s	o_r	RTT ₁	L ₁
	0	24.13	1.36	4.32	125.06	56.84
	10	23.92	1.39	4.29	126.06	57.34
	20	23.79	1.38	4.94	125.06	56.19
1.27	30	29.62	1.41	4.28	125.06	56.83
2.43	40	39.03	1.37	4.31	125.06	56.84
3.08	50	48.46	1.41	4.26	125.06	56.85
3.30	60	58.02	1.39	4.29	125.06	56.84
3.31	70	67.68	1.41	4.24	126.06	57.37
3.33	80	77.34	1.39	4.29	125.06	56.84
3.33	90	87.00	1.39	4.28	125.06	56.85
3.34	100	96.66	1.40	4.27	125.06	56.84
3.33	110	106.34	1.41	4.26	126.06	57.35
3.26	120	116.09	1.37	4.31	127.06	57.84
Avg	2.99		1.39	4.33	125.44	56.98
Std	0.66		0.01	0.18	0.65	0.40

The measurement units for all the tables follow: L , o_s , o_r , g_s , g_r , G' and RTT in μs ; and BW in MB/s. The %Err label represents the percentage error difference between

Table 4.6: Varying G' for 4088 byte messages. G' represents the accumulated gap per-byte for 4088 byte messages. 2^κ represents the desired bandwidth (BW) in MB/s. The remaining units are given in μs .

					Measured values				
%Err	κ	BW	Goal	G'	o_s	o_r	BW	RTT	L
	7	128	31.93	76.29	1.55	3.27	53.57	290.06	140.21
	6	64	63.87	77.22	1.53	4.00	52.93	290.23	139.57
9.22	5	32	127.75	115.97	1.57	4.21	35.24	291.65	140.03
9.38	4	16	255.50	231.54	1.53	4.09	17.65	304.25	146.49
9.47	3	8	511.00	462.62	1.59	3.80	8.83	514.17	251.68
8.01	2	4	1022.00	940.16	1.58	3.92	4.34	1033.59	511.28
6.94	1	2	2044.00	1902.16	1.60	3.88	2.14	2088.05	1038.53
Avg	8.60				1.56	3.88			
Std	1.10				0.02	0.30			

Chapter 4. Validation and Calibration

the desired value of a communication parameter, v_i , and the measured value, v_o , and is calculated as follows:

$$\%Err = \frac{|v_i - v_o|}{v_i} \cdot 100 \quad (4.4)$$

In summary, we have empirically calibrated and validated our apparatus to show that we can control the communication parameters within a percentage error of no more than 9%. We have also shown that the parameters can be varied independently of each other.

Chapter 5

Parallel Applications

This chapter provides a brief description of the benchmark suite used to test the apparatus as well as initial results of application sensitivity to variation in communication parameters. The benchmark suite consists of parallel applications written in MPI that approximate the performance a user can expect from a portable parallel program on a distributed memory parallel computer.

5.1 Benchmark suite

The benchmark suite is grouped in the following categories: *Algorithmic*, *ASCI Purple* and *NPB 2*. The first category represents common problems with well-known algorithms, namely *FFTW* and *Select*. The second category is composed by a subset of the ASC (Advanced Simulation and Computing) Purple benchmark codes [28]. Each of the benchmark programs represents a particular subset and/or characteristic of the expected ASC workload, which consists of solving complex scientific problems using a variety of state-of-the-art computational techniques. The applications used in this work are: *Aztec*, *ParaDyn* and *SMG2000*. Finally, the third category is a subset of the NAS (Numerical Aerodynamic

Simulation) Parallel Benchmarks 2 [6, 37], a widely-recognized suite of benchmarks originally designed to compare the performance of highly parallel computers with that of traditional supercomputers. The applications used in this work are: *IS*, *LU* and *SP*.

5.1.1 Algorithmic

The two applications presented in this section are FFTW and Select. These applications represent common problems with well-known algorithms. FFTW computes the Discrete Fourier Transform in one or more dimensions. Select finds the *i*th smallest element of a sequence.

FFTW. FFTW (Fastest Fourier Transform in the West) is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size.

A Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform and its inverse. The DFT is important in the area of frequency analysis because it takes a discrete signal in the time domain and transforms that signal into its discrete frequency domain representation. There are several variations of what the DFT really computes (normalized, unnormalized, forward, backward, etc.); in particular FFTW defines [20] the one-dimensional forward transform of a complex array X of size n , the array Y , where

$$Y_i = \sum_{j=0}^{n-1} X_j e^{-2\pi i j \sqrt{-1}/n} \quad (5.1)$$

Evaluating these sums directly would take $O(n^2)$ operations. An FFT algorithm computes the same result in only $O(n \log n)$ operations. For multi-dimensional transforms,

Chapter 5. Parallel Applications

the definition follows. Let X be a d -dimensional complex array whose elements are $X[j_1, j_2, \dots, j_d]$, where $0 \leq j_s < n_s$ for all $s \in \{1, 2, \dots, d\}$. Let also $\omega_s = e^{2\pi\sqrt{-1}/n_s}$, for all $s \in \{1, 2, \dots, d\}$. The forward transform computes a complex array Y , whose structure is the same as that of X , defined by

$$Y[i_1, i_2, \dots, i_d] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \dots \sum_{j_d=0}^{n_d-1} X[j_1, j_2, \dots, j_d] \omega_1^{-i_1 j_1} \omega_2^{-i_2 j_2} \dots \omega_d^{-i_d j_d} \quad (5.2)$$

The computation of the transform is based on the Cooley-Tukey algorithm [12]. In its most basic form, this method first computes the Fourier transforms of the even indexed numbers and of the odd indexed numbers, and then combines those two results to produce the Fourier transform of the whole sequence.

To parallelize the computation of the transform, the data is distributed according to the rows (first dimension) of the data: each processor gets a subset of the rows of the data. The FFTW implementation “adapts” itself to the underlying architecture in which is running (minimizing running time rather than number of operations), yielding to one of the fastest implementations known [19].

FFTW version 2.1.3 is used in this work with the following input parameters: benchmark a forward in-place normal-order transform of size $128 \times 128 \times 128$. In-place refers to using the same array for input and output, and normal-order specifies that the output data has the same ordering as the input data.

Select. The Selection problem is a generalization of finding the minimum, maximum or median of a set of elements. The problem consists of finding the element of rank i from a set, i.e., the i th smallest element of the sequence. Finding the elements of rank 1, n and $\lceil n/2 \rceil$ from a sequence of size n , correspond to finding the minimum, maximum and median respectively. The selection problem is known to admit a linear-time sequential

algorithm based on the divide-and-conquer strategy.

Parallel deterministic and randomized (those making use of random-number generators) algorithms have been devised to solve this problem. The algorithm used in this work is a well-known randomized algorithm due to Rajasekaran and Reif [35, 36], and implemented by Bader and Jája [4, 2].

Given a set X of n elements distributed evenly among the processors (each processor holds n/p elements) and a rank k , the algorithm works iteratively by choosing two *splitters* k_1 and k_2 , which partition the input data in three sets G_0 , G_1 and G_2 , such that:

$$G_0 = \{x \in X \mid x < k_1\} \quad (5.3)$$

$$G_1 = \{x \in X \mid k_1 \leq x \leq k_2\} \quad (5.4)$$

$$G_2 = \{x \in X \mid x > k_2\} \quad (5.5)$$

The choice of the splitters should guarantee with high probability the membership of k in the middle set. It is desirable to have the middle set significantly smaller than the other two. The process is repeated iteratively on the set holding k until it is “small enough”, in which case the problem is solved sequentially in one processor.

Bader and Jája’s *Select* version 1.0 is used in this work with the following input: find the median of a set of 4, 194, 304 random, uniformly distributed, integers.

5.1.2 ASCI Purple

The ASCI Purple benchmark suite is composed by 12 benchmarks, each representing a particular subset of the expected ASC workload. The ASC workload consists of solving complex scientific problems using a variety of state-of-the art computational techniques. These benchmarks are written in Fortran, C and C++. Message passing is done via MPI and threading via OpenMP. The ASCI Purple subset used in this work is comprised by Aztec, ParaDyn and SMG2000.

Chapter 5. Parallel Applications

Aztec. Aztec is a parallel iterative solver for linear systems of equations arising from the approximation of partial differential equations. Aztec was developed at Sandia National Labs by Ray Tuminaro and others [40].

Aztec is an iterative library for solving the linear system of equations $Ax = b$ where A is a user-supplied sparse matrix commonly produced by finite difference and finite element approximations to systems of partial differential equations, b is a user supplied vector, and x is the vector to be computed. The heart of Aztec lies in a set of data transformation tools to perform matrix-vector multiplication in parallel [41], which play an important role in the solution of large sparse linear systems by iterative methods.

The sparse matrix-vector product, $Au = v$, is the major kernel operation of Aztec. To perform this operation in parallel, a distribution of the matrix and vectors into the available processors is needed. A subset of elements of v is assigned to each processor, this subset is called the processor's *update set*. Each processor is in charge of computing its update set. Every processor also stores the nonzero elements of the matrix rows corresponding to its update set.

The update set can be further divided into two subsets: *internal* and *border*. Processor P_i can compute its internal components without any communication, i.e., the components of u needed to calculate the internal set are also owned by P_i . The border set computations require elements from the u vector that are not owned by P_i and thus require communication (requests are made to the owner processors of those needed components). All of those components required by a processor to compute its border set comprise another set called the *external set*.

Aztec also provides a number of Krylov iterative methods for the solution of the linear system of equations. These methods are conjugate gradient (CG) [39], generalized minimum residual (GMRES) and stabilized bi-conjugate gradient (BiCGSTAB).

Aztec version 2.1 is used in this work with the following input parameters: solve a

Chapter 5. Parallel Applications

linear system of equations using a preconditioned conjugate gradient squared algorithm on a discretization grid of size $100 \times 100 \times 100$.

ParaDyn. ParaDyn is a parallel version of the serial DYNAMO code used to simulate metals and alloys using the Embedded Atom Method (EAM) [15, 18]. DYNAMO was authored by two of the EAM creators, Stephen Foiles and Murray Daw at Sandia National Labs and Clemson University respectively. ParaDyn [34] was written by Steve Plimpton at Sandia National Labs.

The EAM method has been used in the field of Molecular Dynamics to simulate liquids, solids and molecules. Under this scheme, every atom is considered as a point mass ruled by Newton's equations of motion. The goal is to compute the forces on every atom at every timestep for a number of steps. The force equations are derived from the energy of the atoms, which in turn can be written as a function of their positions.

The force is typically a non-linear function of the distance between pairs of atoms and may be either long-range or short-range. In long-range forces where each atom interacts with all others, the computation of the force per atom results in a quadratic number of operations. For short-range forces, just a small region surrounding every atom is needed to compute its force. Paradyn uses short-range forces. The specific equations to compute the energy and force [34] follow.

The energy E of atom i is given by:

$$E_i = \sum_j \phi(r_{ij}) + F \left(\sum_j \rho(r_{ij}) \right) \quad (5.6)$$

where ϕ is a pair-potential interaction between atoms i and j and F is an embedding function representing the energy to embed atom i in the electron density ρ provided by its surrounding j atoms.

Chapter 5. Parallel Applications

The force $\vec{F}$ on atom i is the derivative of the previous equation and is given by:

$$\vec{F}_i = - \sum_j (\phi'_{ij} + F'_i \rho'_{ij} + F'_j \rho'_{ji}) \vec{r}_{ij} \quad (5.7)$$

where F'_i is the derivative of atom i 's embedding function F , and ρ'_{ij} is the derivative at atom i of the electron density due to atom j .

There exists several methods to compute the forces in parallel, in particular two methods are provided by Paradyn: *atom-decomposition* and *force-decomposition* [34, 33].

On a parallel computer with P processors and N atoms to simulate, the following parameters must be computed every timestep: total force on every atom (computed as the sum of the forces on the atom due to all other atoms) and current position, among others. Let two vectors of length N , x and f , correspond to the position and total force of the atoms respectively.

In the atom-decomposition method, we assign N/P atoms to every processor as well as two subvectors of x and f each of length N/P , corresponding to the position and force parameters of its assigned atoms. Thus, processor z , $0 \leq z < P$, is associated with subvectors x_z and f_z and is in charge of computing the required parameters for each one of its N/P atoms.

Consider an $N \times N$ force matrix F , where entry (ij) represents the force on atom i due to atom j . To compute the total force on atom k , it is necessary to compute row k of F , and then sum all the entries. Thus, every processor is assigned a subset of the rows f_z , corresponding to its N/P atoms.

At the beginning of every timestep every processor has an updated copy of the position vector x . Processor z computes its N^2/P entries of F adding them into its force vector, f_z . Using the forces just computed, it updates its atom positions, x_z , and broadcast them

Chapter 5. Parallel Applications

to all the other processors to get ready for the next timestep.

This algorithm is expensive due to the all-to-all communication step, where every processor sends N/P data to $P - 1$ processors, i.e., $O(N)$ data, which dominates the running time for P bigger than some constant. The force-decomposition algorithm due to Plimpton, was created to overcome this problem.

The force-decomposition algorithm logically decomposes the force matrix into P sub-matrices or blocks (instead of rows as in atom-decomposition) of size $N/\sqrt{P} \times N/\sqrt{P}$. Processor $P_z, 0 \leq z < P$, is assigned the sub-matrix F_z corresponding to the z th block using row-order. Figure 5.1 shows an example of the decomposition of the force matrix with 32 atoms and 16 processors.

Figure 5.1: Force-decomposition of the force matrix with 16 processors and 32 atoms.

As in the atom-decomposition method, every processor is assigned N/P atoms, we

Chapter 5. Parallel Applications

say that processor P_z “owns” the z th subvector of the atoms. Again, every processor is in charge of computing and updating the positions and forces on the atoms it owns.

The force matrix, F , differs slightly from the atom-decomposition method in that entry (ij) corresponds to the force on atom i from vector x due to atom j from vector y . x corresponds to the position vector using F ’s row order, while y corresponds to the position vector using column order. In the example above, processor 6 is assigned the matrix formed by the subvectors $\{4,5,6,7\}$ (columns) and subvectors $\{2,6,10,14\}$ (rows). Thus, every submatrix F_z has two subvectors associated with it (its row and column indices) x_α and y_β . Subvector f_α corresponds to the total force of subvector α .

At the beginning of every timestep, processor z has updated positions of its x_α and y_β subvectors. Using this information, P_z computes its submatrix elements and adds them row-wise into vector f_α (no need to explicitly store matrix elements). At this point, every processor has partially computed the total force on its N/P elements (the rest resides in its row-peers). To compute the total force of its N/P elements, every processor in a row perform a regular personalized communication [3] with its row-peers, i.e., every processor sends (and receives) N/P elements to (from) $\sqrt{P} - 1$ processors, thus every processor exchanges $N/P \times \sqrt{P} = N/\sqrt{P}$ data.

Using the information on f_z , processor z updates its x_z and y_z subvectors. To prepare for the next timestep, P_z needs updated positions of its x_α and y_β from the owners. To achieve this, P_z broadcasts its x_z to its row α and its column β . At this point we are ready to begin the next timestep.

Unlike the atom-decomposition method, where all-to-all communication is needed, the force method requires only personalized communication and scales as $N/\sqrt{P}$ (vs. N for the atom-decomposition method).

The 1998 version of ParaDyn release 1 is used in this work with the following input parameters: run 3000 steps of a *Cu* (Copper) grain boundary with 1720 atoms using the

Chapter 5. Parallel Applications

force-decomposition method.

SMG2000. SMG (Semicoarsening Multigrid) [10] is a parallel solver for the linear systems arising from finite difference, finite volume, or finite element discretization of the diffusion equation $-\nabla \cdot (D\nabla U) + \sigma U = B$, on logically rectangular grids.

SMG uses a multigrid method, based on the alternation between smoothers (relaxation methods) and coarse grid corrections, to solve the linear system [38]. Before describing how SMG works, the next few paragraphs briefly describe smoothers and coarse grid corrections. For a detailed description of these methods, refer to MGNet [16].

Let $Au = b$ be the given linear system to solve and n the number of unknowns. The matrix A is symmetric, positive definite, and connections have the standard "nearest-neighbor" 9-point stencil form. If we use Gaussian elimination to solve it, about $n^3/3$ operations are needed, which may need a tremendous amount of computing power for large n . Several methods have been devised to give an approximation of the solution quickly. These methods are based on the fact that A can be split as $A = S - T$. Then, our linear system of equations can be written as $Su = Tu + b$. Therefore, given an initial value of u , say u_k , the iteration

$$Su_{k+1} = Tu_k + b, \quad \text{or} \quad (5.8)$$

$$u_{k+1} = u_k + S^{-1}(b - Au_k) \quad (5.9)$$

gives an approximation to u . Depending on the split criteria of A , we have several methods that converge to the solution at different rates:

1. S = diagonal part of A (Jacobi's method)
2. S = triangular part of A (Gauss-Seidel method)
3. S = combination of 1 and 2 (successive overrelaxation or SOR)

Chapter 5. Parallel Applications

S is called a *preconditioner* and should be a simple and invertible matrix. Gauss-Seidel generally converges faster than Jacobi and requires less storage. Faster convergence to the solution can be achieved by introducing an overrelaxation factor w . When $w = 1$ we have Gauss-Seidel. The optimal choice of w never exceeds 2. These methods constitute some of the iterative methods for linear systems of equations and are commonly used as smoothers in multigrid methods.

An alternative approach consists of constructing S^{-1} from a problem similar to A but with fewer gridlines. Assume A_c is constructed just like A , but from a smaller grid. Thus, the dimension of the matrix A_c is less than the original dimension. Although the inverse of A_c is easier to compute than the inverse of A , we cannot replace A_c for S in equation 5.8 due to dimension incompatibilities. To fix this problem, prolongation (interpolation) and restriction operators are used. The prolongation operator, P , is used to transfer an error correction from the coarse to the fine grid, while the restriction operator, R , is used in transferring residuals from the fine to the coarse grid. Let $S_c = PA_c^{-1}R$, this equation can now replace S^{-1} in 5.9. This method is called coarse grid correction.

We now proceed to the SMG algorithm in 2D for linear systems. We assign red/black colors to the unknowns in the grid, Ω , as follows: unknowns in even columns (planes in 3D) are colored red and will be used for the coarse grid. Odd columns are colored black. Quantities in the coarse grid will use a $2h$ superscript. Quantities in the fine grid will use an h superscript. This is called *semi-coarsening* since the coarse grid is coarser in only one of the dimensions. A simplified version of the algorithm is outlined below.

V(v1,v2)-cycle

if level = coarsest then

0. Solve coarse grid equation directly

else

1. Pre-smoothing: perform v_1 smoothing steps on $A^h u^h = b^h$.
2. Compute residual: $r^h = b^h - A^h u^h$.
3. Restrict: $b^{2h} = Rr^h$.

Chapter 5. Parallel Applications

4. Recursively solve $A^{2h}u^{2h} = b^{2h}$.
5. Interpolate: $v^h = Pu^{2h}$.
6. Correct: $u^h = u^h + v^h$.
7. Post-smoothing: perform v_2 smoothing steps on $A^hu^h = b^h$.

In a distributed memory machine with P processors, the grid Ω is partitioned into P non-overlapping subgrids Ω_q . Each process is responsible for calculations that result in changes of the approximate solution within its subgrid. To perform these calculations, a processor may need to receive values of the current approximate solution from other processes.

The particular parameters used in this work for this application are: the solver is SMG in 3D with 1 pre and post smoothing steps and diffusion coefficients (0.1, 1.0, 10.0). Each processor has an $20 \times 20 \times 20$ subgrid. The global problem is: $2 \cdot 20 \times 4 \cdot 20 \times 2 \cdot 20$ and is solved on $2 \times 4 \times 2$ processors.

5.1.3 NPB 2

The NPB 2 benchmarks represent a complementary set of programs to the NPB 1. NPB 1 benchmarks are specified algorithmically and are implemented mostly by computer vendors. NPB 2 is based on Fortran 77 and MPI and unlike NPB 1 is intended to be portable with little or no tuning. NPB 2 [37] consists of four parallel kernels and three simulated application benchmarks. For the purposes of this work, we selected three (one kernel and two applications) benchmarks from NPB version 2.3. We briefly describe these benchmarks here; a complete description can be found in the literature [5, 6, 37]. The remainder of this section is divided into the description of the synthetic applications followed by the kernel benchmark.

The synthetic applications discussed here are LU and SP. These applications are simulated applications from the field of Computational Fluid Dynamics (CFD), which has

Chapter 5. Parallel Applications

driven significant work on parallel architectures. Although they lack the complexities associated with real CFD programs, such as pre- and post-processing, they retain their essential computation and data movement.

CFD involves the numerical solution of a system of non-linear partial differential equations (PDEs) in two or three spatial dimensions. These equations are known as the Navier-Stokes equations and represent the laws of conservation of mass, momentum and energy applied to a fluid medium in motion. When given certain conditions, they represent a physical problem.

To solve the system of equations using a computer, discretization is necessary. This is done using a spatial discretization procedure based on finite difference approximations. The discretization process uses structured meshes, characterized by a logical ordering of its points and a rectilinear coordinate system. The resulting system is a linear system of equations, whose solution requires a tremendous amount of processing power and storage if solved by matrix-inversion. Therefore, the solution is obtained through iterative methods.

SP. SP (Scalar-Pentadiagonal) uses the diagonal form of the approximate factorization algorithm, leading to the solution of a sequence of multiple, independent scalar pentadiagonal systems, each oriented along the three mutually orthogonal directions of the computational space. The solutions to these are based on a multi-partition scheme [11], in which a fixed multiple of partitions are assigned to each processor. The number of messages per relaxation step (iteration) is moderate and the granularity of the communications is high [17]. The particular parameters used in this work for the SP benchmark follow: the size of the discretization grid is set to $36 \times 36 \times 36$, and the number of time steps to 400.

LU. LU (Lower-Upper Symmetric Gauss-Seidel) uses the Symmetric Successive Over-Relaxation algorithm (SSOR) [7] resulting in the solution of regular-sparse, 5×5 block

Chapter 5. Parallel Applications

lower and upper triangular systems. The solution to these are based on a diagonal pipelining method called *wavefront* [7]. The number of messages per iteration is high and the granularity of the communications is kept low [17]. The particular parameters used in this work for the LU benchmark follow: the size of the discretization grid is set to $33 \times 33 \times 33$, and the number of time steps to 300.

IS. IS (Integer Sort) sorts N keys in parallel using a parallel version of *bucket sort* [37]. A simplified version of this algorithm follows. Let P be the number of processors and N the number of keys. The keys are generated using a pseudo-random number generator and must be uniformly distributed in memory. The keys are evenly distributed over the processors, such that every processor holds N/P keys in its local memory. Each processor maintains P “small” buckets (of size N/P^2) and one “big” bucket (of size N/P). The algorithm works as follows:

1. Each processor separates the keys in its memory into its small buckets.
2. Perform an all-to-all personalized communication [3] of the i th small bucket to the big bucket in the i th processor.
3. Each processor sorts its big bucket.

Step 1 takes N/P computational steps; in step 2 every processor sends N/P^2 keys to $P-1$ processors, involving $(P-1)(N/P^2) = O(N/P)$ communication steps; step 3 could be done by using merge sort and thus involving $N/P \log(N/P)$ computational steps. Thus this algorithm runs in $O((N/P) \log(N/P))$. Note that this algorithm assumes the keys are uniformly distributed, otherwise all the keys could fall into one big bucket. The particular parameters used in this work for the IS benchmark follows: Sort 2^{23} keys.

5.2 Sensitivity to communication parameters

This section describes the effects of varying the communication parameters on our application suite. This effect is quantified by measuring the running time slowdown of applications due to changing communication parameters. The variation in communication parameters is done by increasing each parameter independently of the others.

We use a 16-node Myrinet cluster to quantify the sensitivity of applications to communication parameters. Each node has a 400 MHz Intel Pentium II processor with 512 MB of main memory and a Myrinet LANai 7.2 network interface card with 2 MB of memory. Nodes were connected through a Myrinet network. Our results were gathered using GM version 1.2.3 and a Linux 2.2.14 kernel.

The running time of the applications from our benchmark suite in the 16-node cluster is given in Table 5.1. None of the communication parameters have been increased from the system's inherent values.

The sensitivity results are presented in two ways: on a parameter basis, and on an application basis. In the former, a single parameter is varied for all the applications (see Figures 5.7, 5.2, 5.5, 5.3, 5.6 and 5.4). In the latter most of the parameters are varied for a single application (see Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7 and A.8). Since the latter presents the same results as the former but in a format more suitable for analyzing the sensitivity of a particular application, the latter results have been added as an appendix; see Appendix A.

5.2.1 Latency

Reducing latency, or time to transmit a small message from NIC to NIC, has been studied extensively in the past as a way to improve network performance. Programmers and developers have put a lot of effort in hiding this parameter in their parallel applications.

Chapter 5. Parallel Applications

Table 5.1: Running time of applications in a 16-node cluster. The particular parameters used to run the applications are also included.

Application	Description	Input Parameters	Running Time (secs)
FFTW	Fast Fourier Transform	Transform of size $128 \times 128 \times 128$	00.325249
Select	Find the Median	2^{22} random, uniformly distributed integers	00.000162
Aztec	CG Iterative Solver	Discretization grid of size $100 \times 100 \times 100$	31.431667
ParaDyn	EAM Metals Simulator	3000 steps of Cu with 1720 atoms	11.354635
SMG2000	Semicoarsening Multigrid	Discretization grid of size $40 \times 80 \times 40$	04.551155
IS	Integer Bucket Sort	2^{23} keys	01.686667
LU	Lower-Upper Symmetric Gauss-Seidel Solver	Discretization grid of size $36 \times 36 \times 36$	18.771667
SP	Scalar-Pentadiagonal Solver	Discretization grid of size $33 \times 33 \times 33$	21.193333

As a consequence, most of the applications discussed here are fairly insensitive to added latency, see Figure 5.2. Select, SMG2000 and ParaDyn show sensitivity to added latency. Among these, only Select has a clear dependence to this parameter due to the fine grain type of communication and high number of communication steps performed by this application.

5.2.2 Overhead

Host overhead, or time the host processor is engaged in sending or receiving a message, is injected every time a message is sent or received; the more messages a processor transfers in a period of time, the bigger the impact of this parameter. Figures 5.4 and 5.3 show application sensitivity to send and receive overhead respectively. Select and SMG2000 show

Figure 5.2: Sensitivity to latency. This figure plots the run time slowdown of applications vs. the added latency.

the strongest sensitivity to these communication parameters due to their high communication frequency —we say that these applications have a “high-dependency” to overhead. ParaDyn and LU are also sensitive but in a diminished way —we say that these applications have a “low-dependency” to overhead. There is no significant difference between the send and receive overhead sensitivity due to the symmetry of the applications in terms of messages sent and received.

5.2.3 Gap

The gap or time interval between consecutive sends or receives dominates those applications in which the time interval between message transfers is less than the gap. Those applications sending messages faster than the gap are penalized by the send gap as Figure 5.6 shows. Similarly, those applications receiving messages faster than the gap are penalized

Figure 5.3: Sensitivity to receive overhead. This figure plots the run time slowdown of applications vs. the added receive overhead.

by the receive gap, see Figure 5.5. As in the case of the overhead, Select and SMG2000 show the highest sensitivity to send and receive gap, while ParaDyn and LU show a lower dependency. However, overhead has a more significant impact on the applications than the gap.

5.2.4 Gap per byte

The Gap, or gap per byte for long messages, identifies those applications in which a significant amount of message transfers involve bulk messages. As Figure 5.7 shows, FFTW and IS show the strongest sensitivity, due to the size of the messages being sent. The rest of the applications show some sensitivity to this parameter. FFTW and IS are good examples of the current trend to cluster small messages into one bigger message, if possible, to avoid communication overhead and to take advantage of the continuing increase in

Figure 5.4: Sensitivity to send overhead. This figure plots the run time slowdown of applications vs. the added send overhead.

network bandwidth¹.

Unlike the other graphs in which the x axes is given in μs , Figure 5.7 plots application slowdown vs. variation on bandwidth given by 2^i MB/s where i corresponds to the x axes.

5.2.5 Summary

The parallel codes used in this work represent a wide variety of types of applications with regard to: (a) degree of exploitable concurrency, (b) computation to communication ratio and, (c) frequency and size of inter-processor communications. We have measured the impact of varying each communication parameter independently on our application suite. Table 5.2 shows a summary of the application sensitivity to communication parameters in a per-application basis; a more detailed characterization can be found in Appendix A.

¹The data rate for Myrinet-2000 links is 250 MB/s (2 Gb/s).

Figure 5.5: Sensitivity to receive gap. This figure plots the run time slowdown of applications vs. the added receive gap.

We found that the applications used in this work have no significant difference between sensitivity to send parameters (overhead and gap) and receive parameters, due to the symmetry of their communication patterns. Also, overhead has a more significant impact on applications than the other communication parameters. Applications are fairly insensitive to latency. Bandwidth or Gap per byte has been identified as a significant parameter to application performance due to the current trend of programmers to cluster small messages together into one message, reducing a significant amount of communication overhead and taking advantage of the increasing communication bandwidth. The ratio of communication and computation is also significant. Applications with very small ratio are not dependent to communication performance.

Chapter 5. Parallel Applications

Figure 5.6: Sensitivity to send gap. This figure plots the run time slowdown of applications vs. the added send gap.

Table 5.2: Summary of application sensitivity to variation in communication parameters. $\times$, $\checkmark$ and $\checkmark$ refer to no-dependency, low-dependency and high-dependency respectively.

		L	o_s	o_r	g_s	g_r	G
<i>Algorithmic</i>	FFTW	$\times$	$\times$	$\times$	$\times$	$\times$	$\checkmark$
	Select	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$
<i>ASCI Purple</i>	Aztec	$\times$	$\times$	$\times$	$\times$	$\times$	$\checkmark$
	ParaDyn	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$
	SMG2000	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$
<i>NPB 2</i>	IS	$\times$	$\times$	$\times$	$\times$	$\times$	$\checkmark$
	LU	$\times$	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$
	SP	$\times$	$\times$	$\times$	$\times$	$\times$	$\checkmark$

Figure 5.7: Sensitivity to Gap. This figure plots running time slowdown of applications vs. the decreased bandwidth, given by 2^i , where i goes from 6 to 1.

Chapter 6

Related Work

This work provides an apparatus to instrument LogP communication parameters in high-performance computational clusters. To provide greater insight in determining the parameters to which an application is sensitive, we consider the send and receive overhead as two independent parameters. This further separation is also applied to the gap. We use a micro-benchmark based on the work by Culler and others [14] to measure the communication parameters. Also, we provide preliminary results on application sensitivity to communication performance of a benchmark suite of parallel applications.

The related work can be classified in three groups: (1) work focused on the sensitivity of applications to specific communication parameters, (2) work focused on the analysis and measurement of communication parameters, and (3) work focused on the analysis of application performance through analytical models.

The first group has goals similar in spirit to ours, but in different contexts [22, 30, 8, 26]. Holt and others studied the performance of network interface controllers in cache coherent DSM (Distributed Shared Memory) machines [22]. They found that controller *occupancy* is critical to the performance of these machines. Controller occupancy refers to the time in which the controller is busy with one action and cannot perform another, and

Chapter 6. Related Work

partially corresponds to the latency and gap in the LogP model.

Martin and others studied the impact of LogGP communication parameters on parallel applications [30]. The applications they used were written in Split-C on top of Generic Active Messages. They found the host overhead to be critical to application performance. Our work, in contrast, uses GM as the communication layer and focuses on applications written in MPI. Although the applications used in our work differ from those used by Martin, the results are similar. Applications were fairly insensitive to latency and sensitive to overhead.

Bilas and others studied the impact of communication parameters in SVM (Shared Virtual Memory) systems [8]. They found that the overhead of generating and delivering interrupts is critical in a SVM system. Kielmann and others artificially increased latency and bandwidth simulating communication performance of WANs (Wide Area Networks), to analyze collective communication operations on this type of networks [26].

The second group provides different micro-benchmarks to measure communication parameters based on the LogP model [23, 26, 27]. Iannello and others consider the send and receive overhead separately. The latency, overhead and gap for bulk data transfers are dependent on the message size. They use two micro-benchmarks based on the work by Culler [14], one measures the o_s and g , and the other, $o_s + o_r$. The goal of this study is to analyze the performance of the FM (Fast Messages) library in different platforms [23].

Kielmann and others introduce another extension to the LogP model, the *parameterized LogP*, suitable for WANs. In this model, the overhead and gap are also considered as functions of the message size. The overhead is split in send and receive overhead. The latency is defined as an end-to-end latency from process to process, which differs from the original LogP model. This work presents improved algorithms for collective communication on WANs [26].

A micro-benchmark to measure the parameterized LogP was also developed [27]. The

Chapter 6. Related Work

aim of this micro-benchmark is the ability to measure the communication parameters without saturating the communication links (except when measuring the gap for messages of size 0), i.e., use fewer messages than the micro-benchmark developed by Culler [14]. This micro-benchmark was implemented in MPI and thus called *MPI LogP benchmark*.

The third group analyzes application performance using analytical performance models [24, 25]. Unlike our work where no knowledge about an application is needed, these models are application centric and require an in-depth understanding of the key performance characteristics of the application. The models are parametrized in terms of computational and communication performances of an individual system, allowing the prediction of application performance on current and future generation machines.

These analytical models can be useful in several ways. Since a detailed understanding is required to create a model, suggestions on improving application performance may arise and can be validated immediately with the appropriate changes to the model. An accurate model for an application can also be used to validate system performance. When installing a new system there are often a number of refinements that need to be done in hardware and software, the expected runtime of an application on this particular system should prove helpful to determine whether or not tuning is needed.

Unlike our work, the analytical models do not need to run the applications to predict their performance. Furthermore, there is no need to have the machine (may not exist yet) in which the application is to be run. Although our apparatus is application independent and is accurate according to the machine in which the analysis is done, the results may be hard to interpolate to another system (due to the multidimensionality of performance) and is limited to communication performance. Analytical models take into consideration a wider spectrum of performance parameters, but the accuracy of the model depends on an in-depth knowledge of the application and the system in which is to be analyzed.

Chapter 7

Conclusions

We have created an apparatus to vary communication performance, based on the LogP model, on high-performance computational clusters. This tool is useful to designers of parallel architectures and message-passing developers to answer the question: What do applications really need? It can also be useful to parallel application developers and users to identify the source of performance degradation of an application on parallel architectures.

We have empirically calibrated and validated our apparatus using a micro-benchmark showing that we can control the communication parameters within a percentage error of no more than 9%. We have also shown that the parameters can be varied independently of each other. We have used our apparatus on a variety of applications to determine their sensitivity to communication performance, thus validating our tool on applications as well.

Appendix A

Sensitivity graphs per application

This chapter has been included as an appendix considering that the information here is the same as in Section 5.2. However, it provides a different view of the data, a graph focuses on an application rather than a communication parameter. The emphasis here is to determine which communication parameters have an impact on a specific application.

The Gap per byte, has not been included in these graphs due to the difference in which the Gap is measured (MB/s rather than μs). See Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7 and A.8.

Appendix A. Sensitivity graphs per application

Figure A.1: Aztec sensitivity.

Figure A.2: FFTW sensitivity.

Appendix A. Sensitivity graphs per application

Figure A.3: IS sensitivity.

Figure A.4: LU sensitivity.

Appendix A. Sensitivity graphs per application

Figure A.5: ParaDyn sensitivity.

Figure A.6: Select sensitivity.

Appendix A. Sensitivity graphs per application

Figure A.7: SMG2000 sensitivity.

Figure A.8: SP sensitivity.

References

- [1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*, pages 95–105, June 1995.
- [2] David A. Bader. An improved randomized selection algorithm with an experimental study. Technical Report AHPCC-TR-99-014, Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM, September 1999.
- [3] David A. Bader, David R. Helman, and Joseph JáJá. Practical parallel algorithms for personalized communication and integer sorting. Technical Report CS-TR-3548 and UMIACS-TR-95-101, University of Maryland, College Park, MD, November 1995.
- [4] David A. Bader and Joseph JáJá. Practical parallel algorithms for dynamic data redistribution, median finding, and selection. Technical Report CS-TR-3494 and UMIACS-TR-95-74, University of Maryland, College Park, MD, July 1995.
- [5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrisnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, March 1994.
- [6] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995.
- [7] E. Barszcz, R. Fatoohi, V. Venkatakrisnan, and S. Weeratunga. Solution of regular, sparse triangular linear systems on vector and distributed-memory multiprocessors. Technical Report RNR-93-007, NASA Ames Research Center, Moffett Field, CA, April 1993.

References

- [8] Angelos Bilas and Jaswinder Pal Singh. The effects of communication parameters on end performance of shared virtual memory clusters. In *Proceedings of the ACM/IEEE SuperComputing (SC '97) Conference*, San Jose, CA, November 1997.
- [9] 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.
- [10] Peter N. Brown, Robert D. Falgout, and Jim E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.
- [11] J. Bruno and P. R. Cappello. Implementing the beam and warming method on the hypercube. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 1073–1087, Pasadena, CA, January 1988.
- [12] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, April 1965.
- [13] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '93)*, pages 1–12, San Diego, CA, May 1993.
- [14] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. Assessing fast network interfaces. *IEEE Micro*, 16(1):35–43, 1996.
- [15] M. S. Daw and M. I. Baskes. Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals. *Phys. Rev. B*, 29:6443–6453, 1984.
- [16] Craig C. Douglas. *MGNet Home Page*, January 2003. <http://www.mgnet.org>.
- [17] Rod Fatoohi and Sisira Weeratunga. Performance evaluation of three distributed computing environments for scientific applications. In *Proceedings Supercomputing '94*, pages 400–409, Washington, DC, November 1994.
- [18] S. M. Foiles, M. I. Baskes, and M. S. Daw. Embedded-atom-method functions for the fcc metals Cu, Ag, Au, Ni, Pd, Pt and their alloys. *Phys. Rev. B*, 33:7983–7991, 1986.
- [19] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE Intl. Conference on Acoustics, Speech, and Signal Processing (ICASSP '98)*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

References

- [20] Matteo Frigo and Steven G. Johnson. *FFTW User's Manual*, November 1999. <http://www.fftw.org/doc/>.
- [21] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
- [22] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Stanford University, January 1995.
- [23] Giulio Iannello, Mario Lauria, and Stefano Mercolino. Cross-platform analysis of Fast Messages for Myrinet. In Dhabaleswar K. Panda and Craig B. Stunkel, editors, *Proceedings of the 2nd International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC '98)*, volume 1362 of *Lecture Notes in Computer Science*, pages 217–231, Las Vegas, NV, February 1998. Springer.
- [24] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–48, Denver, CO, 2001. ACM Press.
- [25] Darren J. Kerbyson, Harvey J. Wasserman, and Adolfo Hoisie. Exploring advanced architectures using performance prediction. In *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02)*, pages 27–37, Kohala Coast, Big Island, Hawaii, January 2002.
- [26] Thilo Kielmann, Henri E. Bal, and Sergei Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '00) Conference*, Cancun, Mexico, May 2000.
- [27] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast measurement of LogP parameters for message passing platforms. In *4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, Cancun, Mexico, May 2000.
- [28] Los Alamos National Laboratories, Sandia National Laboratories, and Lawrence Livermore National Laboratories. *The ASCI Purple Benchmarks*, November 2001. <http://www.llnl.gov/asci/purple/benchmarks/>.

References

- [29] Edgar A. León, Arthur B. Maccabe, and Ron Brightwell. Instrumenting LogP parameters in GM: Implementation and validation. In *Workshop on High-Speed Local Networks (HSLN)*, Tampa, FL, November 2002.
- [30] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA '97)*, pages 85–97, Denver, CO, June 1997.
- [31] Myricom, Inc. *The GM API*, October 1999. http://www.myri.com/scs/GM/doc/gm_toc.html.
- [32] Myricom, Inc. *LANai 7*, June 1999. <http://www.myri.com/vlsi/LANai7.pdf>.
- [33] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. Technical Report SAND91-1144, Sandia National Laboratories, Albuquerque, NM, May 1993.
- [34] Steven J. Plimpton and Bruce A. Hendrickson. Parallel molecular dynamics with the Embedded Atom Method. In J. Broughton, P. D. Bristowe, and J. M. Newsam, editors, *Materials Research Society (MRS) Proceedings*, volume 291 of *Materials Theory and Modeling*, pages 37–42, Pittsburgh, PA, 1993.
- [35] S. Rajasekaran and J. H. Reif. Derivation of randomized sorting and selection algorithms. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithms Derivation and Program Transformation*, chapter 6, pages 187–205. Kluwer Academic Publishers, Boston, MA, 1993.
- [36] Sanguthevar Rajasekaran. Randomized selection on the Hypercube. *Journal of Parallel and Distributed Computing*, 37(2):187–193, 1996.
- [37] William Saphir, Rob Van der Wijngaart, Alex Woo, and Maurice Yarrow. New implementations and results for the NAS parallel benchmarks 2. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.
- [38] S. Schaffer. A semi-coarsening multigrid method for elliptic partial differential equations with highly discontinuous and anisotropic coefficients. *SIAM Journal on Scientific Computing*, 20(1):228–242, 1998.
- [39] John N. Shadid, Scott A. Hutchinson, and Harry K. Moffar. Parallel performance of a preconditioned CG solver for unstructured finite element applications. In *Proceedings of the Colorado Conference on Iterative Methods*, Breckenridge, CO, April 1994.

References

- [40] Ray S. Tuminaro, Mike Heroux, Scott A. Hutchinson, and John N. Shadid. Official Aztec user's guide – version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, Albuquerque, NM, November 1999.
- [41] Ray S. Tuminaro, John N. Shadid, and Scott A. Hutchinson. Parallel sparse matrix-vector multiply software for matrices with data locality. Technical Report SAND95-1540J, Sandia National Laboratories, Albuquerque, NM, July 1995.