OPTIMIZING COMMUNICATION FOR MASSIVELY PARALLEL PROCESSING

BY

SAMEER KUMAR

B. Tech., Indian Institute Of Technology Madras, 1999
M.S., University of Illinois at Urbana-Champaign, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

The current trends in high performance computing show that large machines with tens of thousands of processors will soon be readily available. The IBM Bluegene-L machine with 128k processors (which is currently being deployed) is an important step in this direction. In this scenario, it is going to be a significant burden for the programmer to manually scale his applications. This task of scaling involves addressing issues like load-imbalance and communication overhead. In this thesis, we explore several communication optimizations to help parallel applications to easily scale on a large number of processors. We also present automatic runtime techniques to relieve the programmer from the burden of optimizing communication in his applications.

This thesis explores processor virtualization to improve communication performance in applications. With processor virtualization, the computation is mapped to virtual processors (VPs). After one VP has finished computation and is waiting for responses to its messages, another VP can compute, thus overlapping communication with computation. This overlap is only effective if the processor overhead of the communication operation is a small fraction of the total communication time. Fortunately, with network interfaces having co-processors, this happens to be true and processor virtualization has a natural advantage on such interconnects.

The communication optimizations we present in this thesis, are motivated by applications such as NAMD (a classical molecular dynamics application) and CPAIMD (a quantum chemistry application). Applications like NAMD and CPAIMD consume a fair share of the time available on supercomputers. So, improving their performance would be of great value. We have successfully scaled NAMD to 1TF of peak performance on 3000 processors of PSC Lemieux, using the techniques presented in this thesis.

We study both point-to-point communication and collective communication (specifically all-to-all communication). On a large number of processors all-to-all communication can take several milli-seconds to finish. With synchronous collectives defined in MPI, the processor idles while the collective messages are

in flight. Therefore, we demonstrate an asynchronous collective communication framework, to let the CPU compute while the all-to-all messages are in flight. We also show that the best strategy for all-to-all communication depends on the message size, number of processors and other dynamic parameters. This suggests that these parameters can be observed at runtime and used to choose the optimal strategy for all-to-all communication. In this thesis, we demonstrate adaptive strategy switching for all-to-all communication.

The communication optimization framework presented in this thesis, has been designed to optimize communication in the context of processor virtualization and dynamic migrating objects. We present the streaming strategy to optimize fine grained object-to-object communication.

In this thesis, we motivate the need for hardware collectives, as processor based collectives can be delayed by intermediate that processors busy with computation. We explore a next generation interconnect that supports collectives in the switching hardware. We show the performance gains of hardware collectives through synthetic benchmarks.

To my parents,

Acknowledgements

I would like to thank my thesis advisor, Prof Kale, for his guidance, direction, motivation and continued support, without which this thesis would not have been possible.

I would like to thank the several PPL members, Chee-wai, Sayantan, Terry, Orion, Gengbin, Eric, Yan, Chao, Nilesh, Yogesh, Greg, Tarun, Filippo, David ... and the list goes on. It was a lot of fun working and hanging out with you guys. Chee-wai and Sayantan, I shall remember the several trips to Mandarin Wok we took to spice up intellectual aggression, and that made us quite immobile after. Eric and Yan, had a great time working with you guys on the CPAIMD project. Hope you guys will continue the great work.

Many thanks to the various members of the TCB group, specially Robert for helping me with my thesis and several papers in the past. Kirby and Mike, it was great working with you guys on BioCoRe. Jim and John, thanks for patiently answering all my questions on a variety of *fields*.

I would also like to thank my sister Pranati, my brother in-law Shyam and my two nieces Keerti and Kallika, who were both born during the PhD, for all the love and moral support that I needed over the years.

Finally, I would like to thank my parents for all their love and inspiration that was needed to accomplish this task.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Inter-processor communication is a well known hindrance to scaling parallel programs on large machines. With machines getting larger (e.g. Pittsburgh's Lemieux [42] has 3,000 processors and the IBM BlueGene-L would have 128K processors), the management of communication overheads is critical for the applications to achieve good performance on a large number of processors. Fortunately, many modern parallel computers have smart network interfaces with co-processors. This co-processor can potentially minimize main processor participation in communication operations, which would allow the main processor to compute while the messages are in transit. Examples of interconnects with co-processors are Quadrics QsNet [51, 52] and Infiniband [1].

In the presence of a co-processor, communication overhead can be effectively optimized through overlap of communication latency with computation. *Processor-virtualization* is an elegant mechanism to achieve such overlap [32]. Here the computation is divided into several *virtual processors* (objects or user-level-threads) with more than one virtual processor on each processor. Processor-virtualization leads to message-driven execution: since there are multiple objects (VPs) on a processor, there must be a scheduler that decides which one of them executes next. It schedules a VP, when there are messages available for that VP (thus the scheduling is driven by messages). After one virtual processor (VP) has finished computation and sent its messages, another VP can compute while the first one is waiting for responses to its messages, thus overlapping communication latency with computation. Simulation studies have shown that *message-driven execution* can exploit communication co-processors much more effectively than traditional MPI-style message-passing [21].

In this research, we explore the thesis that runtime optimization strategies can significantly improve performance of parallel applications. These include strategies for taking advantage of specific features and op-

portunities presented by individual communication architecture, schemes for efficient collective operations, and capability for automatically learning the communication patterns in an application and accordingly applying the appropriate optimizations to them.

With processor virtualization, these runtime schemes could optimize both object-level and processor-level communication. In this thesis, we explore both these levels of communication optimization. At the processor level, we examined and developed schemes for optimizing point-to-point communication and collective communication.

We optimize *point-to-point* communication by first evaluating the performance of the various communication interconnects available. The Charm runtime is then specialized and fine-tuned to use the strengths of each interconnect, and avoid its bottlenecks. The networks analyzed in this thesis include Quadrics QsNet, Mellanox Infiniband and Myricom Myrinet (Chapter 2). We analyze these networks through multiple communication benchmarks that measure the latency, bandwidth and CPU involvement of interprocessor communication. We also study the mechanisms of message passing supported on the different interconnects. For example, some networks support one-sided communication, while others only support MPI-style message passing.

Collective communication on large machines is a serious performance bottleneck. This thesis presents strategies to optimize all-to-all collective communication to enable applications to scale to large machines. Different techniques are needed to optimize all-to-all communication for small and large messages. For small messages (Section 3.1), completion time is dominated by the NIC and the CPU software overheads of sending the messages. This overhead can be reduced through techniques based on message combining. For large messages (Chapter 4), the cost is dominated by network contention, which can be minimized by smart sequencing of messages based on the understanding of the underlying network topology. Some of these techniques are also extended to *many-to-many* communication (Chapter 3.3), where many (but not all) processors exchange messages with many other processors.

On thousands of processors, all-to-all operations can take milliseconds to finish even with short messages. However, we show that the *CPU overhead* of these operations is typically a small fraction of the completion time. We therefore present an *asynchronous collective* interface in Charm++ and Adaptive MPI [23], where the idle time during a collective operation can be overlapped with computation (Chapter 6). We also demonstrate the performance gains of this asynchronous collective interface using a *sorting*

benchmark (Chapter 7).

The best strategy for all-to-all and many-to-many communication (Chapters 3 and 4) depends on the number of processors, message size, the underlying topology and whether CPU overhead or completion time of the collective operation is critical to the application. Since many scientific applications are iterative and the *principle of persistence* applies, the optimal collective communication strategy can be learned. Hence we believe that the runtime system should be able to adaptively learn the communication patterns in applications and apply the best strategy for that pattern. We present such dynamic strategy switching schemes in Chapter 6.

Processor virtualization in Charm++ enables object-level optimizations. In this thesis, we present the *streaming optimization* schemes that optimize the scenario where objects exchange several short messages (Chapter 5.6). These schemes are necessary to improve the performance of fine-grained applications such as parallel discrete event simulation (PDES). We also demonstrate better scaling to a larger number of processors with the streaming optimizations (Chapter 7).

The communication optimization schemes presented in this thesis have been coded as *strategies* in the **Communication Optimization Framework** (Chapter 6), which is now a core part of the Charm++ runtime. This framework has been designed to be a general communication optimization library, where a variety of processor-level and object-level strategies can be developed and deployed in a plug and play manner. The framework also enables easy addition and extension of communication optimization strategies. The strategies optimize object communication in the context of dynamically migrating objects. The framework also has built-in *adaptive learning* and *strategy-switching* capabilities. The communication patterns in applications can be learned by recording the sizes of messages and the destination objects for the messages from every object. This recorded information could then be used to choose the best strategy from a number of eligible strategies.

The performance gains of various communication optimizations strategies are shown through synthetic benchmarks and applications. Some of the applications studied in this thesis are NAMD [31] and CPAIMD [70]. NAMD is a classical molecular dynamics application used by thousands of computational biologists, while CPAIMD is a quantum chemistry application (Chapter 7). Such molecular dynamics applications consume a considerable share of the compute-time available on several supercomputer centers. Optimizing them would be of great value to the scientific community. These applications are communication

intensive on large processor configurations. Both NAMD and CPAIMD use the communication framework to optimize their communication operations. For example, both NAMD and CPAIMD compute 3D-FFTs which have all-to-all communication. CPAIMD also has all-to-all multicasts and several simultaneous reductions. In this thesis, we present performance improvements in NAMD and CPAIMD through various strategies in the communication framework.

The collective communication optimization strategies we present in this thesis require processing in the CPU. Eventhough asynchronous collectives minimize the CPU involvement, they do not completely eliminate it. If one of the intermediate processors is busy with computation the collective operation will be delayed. This motivates the need for hardware collectives, where the CPU only initiates the collective operation and is notified on completion. Thus, the collective operation is completely performed on the interconnect.

In Chapter 8, we explore a next generation network that efficiently implements collectives like multicasts and reductions in the switches of the interconnect. Most current interconnects use input-queued switches because they require relatively low buffer space and internal speedup in the switch. However, our proposed network uses output-queued switches, as input-queued switches have bad performance with the multicast operation [43, 56]. We show the performance gains of an output-queued network for hardware multicasts and reductions with synthetic benchmarks. We simulate this network using the *BigNetSim* network simulator. The network is also studied in a fat-tree configuration, which is currently the most popular network topology.

The communication optimizations presented in this thesis are demonstrated to have good performance on modern parallel machines like PSC Lemieux [42], NCSA Tungsten [68] and Turing [69]. However, these optimizations may have to be tuned to be effective on a very large machine like the IBM Bluegene/L with 128K processors. Other future directions include the development of strategy switches for other communication operations like streaming, broadcasts and reductions. It is also quite hard to accurately predict the performance of strategies analytically because network contention is hard to model and there are several sources of non-determinism in the system. Some of these are operating system daemons and background traffic from other jobs in the batch system. A detailed network simulation would have to be tied into the adaptive strategy-switch framework to be able to accurately predict the finish times of the various communication optimizations (Chapter 9).

1.1 Contributions

This thesis makes the following contributions :

- *Establishing the importance of CPU overhead for collective communication:* Our experiments show that collective communication operations can take several milliseconds to finish, while only a small fraction of that time involves CPU computation. Hence the remainder can be overlapped with computation. To make this overlap effective, the collective communication operation should consume the fewest CPU cycles. Hence collective operations should also be evaluated based on their CPU overheads, in addition to their completion times.

- *Strategies for all-to-all communication:* this thesis presents novel strategies for all-to-all communication. These strategies do not require the number of processors to be perfect squares or powers of two. The strategies are also analyzed with cost equations.

- *High performance machine layer on Quadrics QsNet*. We have developed a high performance machine layer on Quadrics Elan for the Charm runtime system, illustrating how the runtime system can take advantage of the specific strengths of the particular communication subsystem.

- Scaling NAMD: On 3000 processors of PSC Lemieux it achieved 1TF of peak performance and shared the Gordon Bell award at Supercomputing 2002. This involved development of techniques to optimize communication performance and overcome operating system overheads.

- *Adaptive strategy switching :* the communication optimization framework in the Charm RTS can dynamically switch the application to use the optimal strategy for its communication operations.

- *Exploration of hardware collectives:* We show the advantages of hardware collectives through synthetic benchmark simulations that emulate the NAMD multicast communication pattern. To support efficient hardware multicasts, we use an output-queued switch architecture.

1.2 Roadmap

The next chapter presents the design and performance study of cluster interconnects like QsNet, Myrinet and Infiniband. This analysis is required to optimize the Charm++ runtime on those interconnects and to

model their performance with cost equations. Chapters 3 and 4 present all-to-all communication optimization strategies, and analyze their performance with cost equations. In Chapter 5, we explore processor virtualization. We also present the *streaming* optimization for object-to-object communication. Chapter 6 presents the communication optimization framework with processor-level and object-level optimizations. Performance enhancements of applications with the communication framework are presented in Chapter 7. Chapter 8 presents our research on collective communication acceleration in network hardware. Summary and future directions are presented in Chapter 9.

Chapter 2

Quantitative Study of Modern Communication Architectures

Many modern communication architectures have a co-processor in the network interface. Examples of such interconnects are Quadrics QsNet [52], Mellanox Infiniband [1] and Myricom Myrinet [48]. These co-processors can perform remote DMA operations, that minimizes CPU participation in message passing. Network interfaces behave like standard I/O devices and communicate with the main CPU through the I/O bus. These devices also perform memory management to map application virtual memory to physical memory, as send and receive request have application virtual memory in them.

In this chapter, we present a study of such cluster interconnects, especially in the context of processor virtualization. The study presents latency and bandwidth and CPU overhead with several micro benchmarks. We also analyze the reasons for the performance bottlenecks in QsNet, Myrinet and Infiniband. This analysis has been used in the design of the Charm runtime system, which tries to avoid the bottlenecks presented in this Chapter.

The performance study is also used to design models that predict the performance of the above mentioned interconnects. In Chapters 3 and 4, we extensively use the model presented in this chapter to predict the performance of all-to-all communication optimization schemes.

Before we present the performance study, we briefly describe the architecture of a generic cluster network interface. (The description will help the reader understand the different terms and metrics in the performance study.) The key components of a network interface are shown in Figure 2.1. Such network interfaces have a network interface controller, which is a processor capable of running programmable threads to manage message passing. They also have a DMA engine to access data from main memory and send it

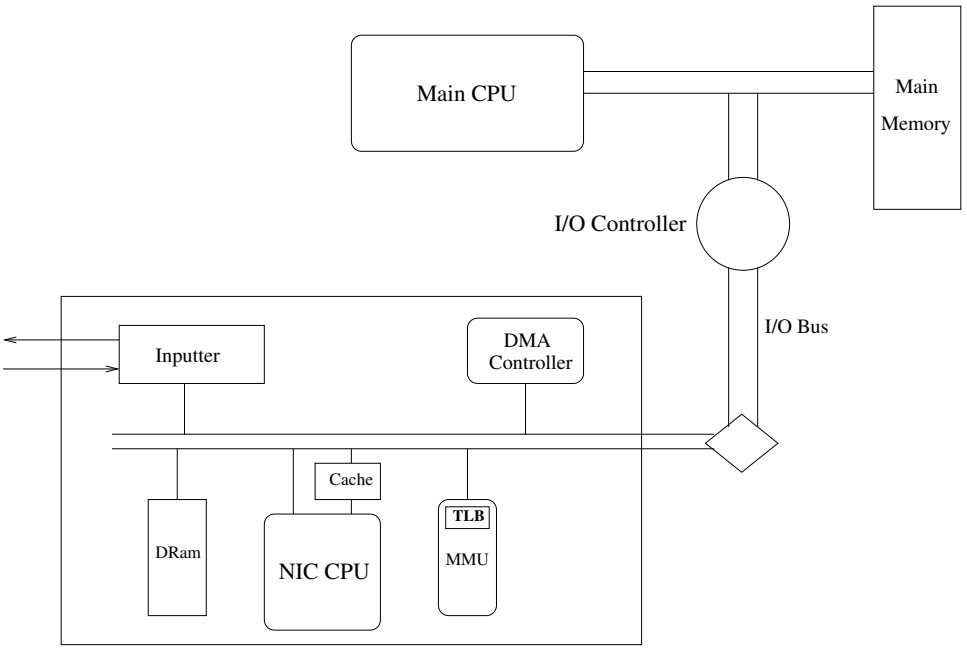


Figure 2.1: Architecture of a generic NIC with a co-processor

on the network, and access memory on remote nodes.

The memory management unit in the network adapter translates application virtual memory to physical memory. The memory management unit may also have a translation-lookaside-buffer (TLB) in hardware or software to optimize this translation, along with memory protection support for multi-tasking environments.

To send a message, the CPU issues a send transaction to the NIC through an I/O write operation. This send request will have a pointer to the application virtual memory and the rank of the destination processor. The send operation could be one of the following two types, (i) tagged MPI-style message passing, (ii) one-sided remote DMA operation.

Tagged messages do not have a destination memory address for the message, but just have a tag which is dynamically matched on the destination node. In some interconnects matching of tags is performed by the NIC, while on others it is done by the CPU. While NIC enabled tag matching could have a lower CPU overhead, it places an additional burden on the relatively slower NIC. As tag matching can be a complex operation, it directly affects the performance of the application. So in this chapter, we compare and both these types of network interfaces.

Many interconnects also support one-sided communication, which is more light-weight than tagged message passing. In a one-sided communication operation, the destination memory address for the message is

passed by the application during the send. This results in the send operation consuming minimum resources at the source and destination network interfaces. Though harder to program, one-sided communication is the efficient way of message passing on most modern networks.

After a *send* has been issued by the CPU, the MMU in the network interface translates the application virtual memory to physical pages. This has to be done for both tagged sends and one-sided communication. The NIC then issues DMA-writes to transfer these pages to the network. Contiguous pages in virtual memory may not translate to contiguous pages in physical memory, and therefore each message send may involve accessing different locations scattered in memory. Networks with built-in hardware support for scatter-gather can send all pages of a message in one transaction, while other networks however would have to issue several transactions to send each message.

Some network interfaces also require that the source and destination buffers be pinned and non-swappable, as they cannot handle page faults. Page faults which will stall the communication operation till the page is loaded from secondary storage. As this would require sophisticated logic in the NIC ASIC, it is avoided by many network interfaces. Only Quadrics QsNet supports page faults and does not require users to pin application memory before sending it.

To send the packets on the network the NIC may also have to compute a route to the destination. This happens in source routed networks where the NIC must find a complete route to the destination and encode it in the packet header. The routes are normally loaded into the network interface during initialization. The NIC on destination routed networks just encodes the destination address on the packet, leaving the underlying switching hardware to find the route dynamically.

At the receiver, the NIC first matches the tag of the incoming message with all the receives posted by the application, and then issues a DMA operation to move the message data to main memory. Even here the destination address would have to be translated from virtual to physical memory. If no posted receive matches the incoming tag, the NIC will have to allocate memory for the message and DMA it to the temporary buffer. This usually results in an expensive operation. For one-sided transactions, tag-matching is not necessary and data can directly be moved to the destination address.

On networks where tag matching is not performed by the NIC, the packets are usually just dumped into main-memory. The tag-matching would be performed later by the CPU, resulting in a copy to application buffers by the NIC. After the message has been successfully sent, the application is informed through an

interrupt or a memory flag.

The communication co-processor hence minimizes main processor involvement in message passing. The CPU is freed from the burden of packetizing the messages, implementing flow-control, discovering routes and recovering from errors. This lets the CPU work on the application and poll the network once in a while. We believe that processor virtualization is an excellent mechanism to exploit co-processor technology. We demonstrate the advantages of the communication co-processor for processor virtualization in Section 5.5.

The remainder of this Chapter presents performance studies of the popular interconnects, QsNet, Myrinet and Infiniband with several micro benchmarks. We present the latency and bandwidth of the above mentioned architectures. We begin with an evaluation of Quadrics QsNet.

2.1 Performance Study of QsNet

QsNet [51, 52] is a high-bandwidth low-latency clustering technology from Quadrics [57] that has been widely deployed. Several of the top 500 machine, like PSC's Lemieux [42] and ASCI-Q, are built upon Quadrics QsNet. The main advantage of QsNet is its programmable network interface called *Elan*. The Elan network interface has a communication co-processor and a remote DMA engine, similar to the generic network interface (Figure 2.1) presented in the previous section.

The Elan ASIC processor is fully functional and can freely access application memory. However, it is a 32 bit RISC processor and can only access 3GB of application virtual memory. Quadrics provides memory allocators which the applications should link with. Virtual to physical mapping is done completely in hardware, leading to a low overhead for message passing. The Elan co-processor can also throw interrupts to make the OS load pages from secondary storage into main memory, so it does not require the application to pin memory for message passing. This makes message passing truly a zero copy transaction.

We now present a performance evaluation of message latency and bandwidth on Quadrics QsNet. These experiments were performed on Lemieux using the Converse runtime system [36], which has been built upon the Elan TPort message passing library. Converse [28] is a light-weight message passing runtime, which used by the Charm++ system for message passing.

2.1.1 Message Latency

The published message latency for a zero byte message on Elan is $4.5\mu s$ [51, 52]. But we found that this was not the case when we ran Converse experiments on Lemieux. Converse is a light-weight runtime which provides low-level messaging primitives to the Charm runtime system. Converse executes a message scheduler that calls handlers on message arrival. These handlers execute user level code which may take several milliseconds to finish. Control is returned to the scheduler only after the handler has finished executing.

To improve performance, the Converse runtime posts several receives to match tags of incoming messages while a handler is executing. This is necessary as the the Elan NIC's performance is below par when arriving messages do not have receive buffers posted for them. The Elan NIC performs TAG matching for all messages. Arriving message that do not have receives posted for them are called unexpected messages. For such messages, the NIC allocates memory in a local buffer before receiving them. Later when the user posts a receive, there will be some additional copying overhead. This is shown by Table 2.1.

Message Size(b)	Converse	Converse Unexpected Messages
1024	17.3	22.8
4096	29.5	46.9
16384	72.1	144.9

Table 2.1: Converse Latency (μs)

However, we found that posting too many receives also affected the performance of the network interface. The Elan NIC stores posted receives in a linked list and traversing such a list can be slow in a 100 Mhz ASIC. To evaluate the effect of posted receives on message latency, we ran the ping-pong benchmark (both in MPI and Converse) with a varying number of posted receives. For MPI, these additional receives were posted with a tag that was different from the one used for ping-pong, which forced the NIC to traverse a long list before matching each message. The results are shown in Table 2.2. The best one-way message latency for Converse is $6.02\mu s$, a little more than $4.69\mu s$ for MPI. This additional overhead is due to timer and scheduling overheads in the Converse runtime system.

We found that as the number of receives posted increased from 1 to 33 (Table 2.2), the 16 byte message latency increased from $6\mu s$ to $18.5\mu s$ for Converse and from $4.69\mu s$ to $17.8\mu s$ for MPI. We believe that this increase is because the Elan network interface has a shared data and instruction cache. So looping over a large receive list on message arrival flushed the NIC's I-cache (Table 2.3), thus degrading performance.

Message Size(b)	#Receives Posted	Converse	MPI
16	1	6.02	4.69
16	5	7.27	5.66
16	9	8.34	6.93
16	17	10.9	10.9
16	33	18.5	17.8
64	1	7.27	5.96
64	5	8.28	7.23
64	9	9.48	8.20
64	17	12.0	12.3
64	33	19.7	19
256	1	9.89	8.89
256	5	11.1	10.0
256	9	12.2	11.2
256	17	15.3	15.4
256	33	22.9	22.7

Table 2.2: Latency (μs) vs No. of receives posted

Hence, there is a tradeoff between posting more receives and running the risk of unexpected message handling in the NIC. The optimal number of receives posted is specific to each application, and the Converse runtime system is tunable in this respect.

#Receives Posted	#Cache Misses
1	86017
5	92475
9	103037
13	174060
17	1008003
33	6539278

Table 2.3: Receives Posted vs Cache Misses

So far we have shown the message latency using one *processor per node*. But, the nodes on Lemieux have four *processors per node* (PPN) and applications usually use all of them. To compute the latency when all 4 processors are being used, we ran ping-pong with each processor exchanging messages with the corresponding processor in the remote node. Table 2.4 shows the message latency for PPN=1 and PPN=4 with 9 receives posted by each processor. The latency reported in Table 2.4 is slightly more than that reported in Table 2.2. This is because of the additional timer overhead of computing the CPU overhead and idle time in the ping-pong benchmark. Observe that the message latency is much higher when PPN=4, $17\mu s$ for a 16

Message Size	Latency(μs)		CPU Overhead(μs)	
	PPN = 1	PPN = 4	PPN = 1	PPN = 4
16	9.49	17.04	5.59	5.3
64	10.5	19.36	5.29	5.36
256	13.4	24.5	6.47	6.05
1024	18.4	42.81	6.04	6.26
4096	29.7	83.2	6.69	6.52

Table 2.4: Converse Latency vs CPU Overhead

Message Size	Latency(μs)		CPU Overhead(μs)	
	PPN = 1	PPN = 4	PPN = 1	PPN = 4
16	12.4	27.17	11.5	9.9
64	12.9	31.81	11.99	10.35
256	15.13	41.37	13.13	12.46
1024	26.24	77.47	12.6	12.08
4096	51.23	154.1	13.47	12.94

Table 2.5: Converse with two way traffic

byte message compared with $9.5\mu s$ for PPN=1. The latency is further increased possibly because there are 36 receives (9 by each processor) posted on the NIC.

For message driven execution, CPU overhead is a more critical parameter, as the remaining time can be overlapped with other computation. Table 2.4 also presents the CPU overhead of the ping-pong benchmark for both PPN=1 and PPN=4. This CPU overhead (e.g. $5.6\mu s$ for 16 bytes and PPN=1) includes send, receive and the converse RTS overheads. The CPU overhead is similar for both PPN=1 and PPN=4 and does not change much with the message size, perfect for message driven execution. CPU overhead is obtained by subtracting the idle time from the round trip time and then dividing the remainder by two. Observe from Table 2.4, that for PPN=1 the CPU overhead for a 256 byte message is more than the overhead for a 1024 byte message. This is because messages smaller than 288 bytes are first copied by the elan library into the network interface and then sent from there, thus incurring a higher CPU overhead.

As parallel applications tend to have bi-directional traffic, we computed the CPU overheads and latencies with bi-directional traffic and the results are presented in Table 2.5. Observe that with bi-directional traffic, 4 processors per node and many receives posted, the latency for short messages is 27 μs, which is a significant message latency. We recommend that applications that send several short messages should use message combining to improve performance, as presented in Chapter 3.1.

	Main-Main	Elan-Elan	Elan-Main
One Way Traffic	290	319	305
Two Way Traffic	128	319	305

Table 2.6: Elan node bandwidth (MB/s)

2.1.2 Node Bandwidth

The QsNet network can support a full duplex bandwidth of 319 MB/s. This bandwidth is achievable only if messages are sent from Elan memory, as PCI I/O restricts the main memory bandwidth to about 290 MB/s. Further when processors are simultaneously sending and receiving, this bi-directional traffic restricts the throughput to about 128 MB/s each-way. Heavy contention for DMA and PCI by messages in both directions is responsible for this loss of network throughput.

Table 2.6 shows the achievable network bandwidth for different placements of the sources and destinations of messages. For example, Main-Main represents both source and destination of the message are in Main memory, while Elan-Main indicates that the source data is in Elan memory and destination address is in Main memory. Observe that sending messages from Elan memory is faster. For applications to achieve this higher throughput, first the message has to be DMAed into Elan memory at a bandwidth of about 305MB/s, and then sent from there. But, this memory copy overhead can nullify the advantage of sending the message from Elan memory. However, multicast operations can be optimized by copying the message once into Elan memory, and then sending it to multiple main memory destinations (Chapters 3 and 4).

Multi-ping performance: To measure the maximum bandwidth available for each message size we ran the multi-ping benchmark. In this benchmark, node-0 sends 128 messages of a fixed size to node-1, and node-1 only sends a single response back. Simultaneously, node-1 sends 128 messages to node-0 and after receiving all of them node-0 sends one response back. The multi-ping benchmark shows the degree of pipelining in the network interface. It reflects the gap (g) in the LogP model [12, 47, 3]. A highly pipelined NIC would achieve full throughput for very short messages, as it could pipeline the startup of the next message while current message is in flight. Table 2.7 shows the performance of the multi-ping benchmark, where the throughput gets close to the for 4KB messages. As mentioned before, PCI contention restricts bandwidth to about 128MB/s, though achievable bandwidth shown at 256KB is 124MB/s.

Bandwidth to distant nodes: We also observed that the network bandwidth drops when sending to far away nodes, with the ping-pong benchmark. A similar drop in throughput on the ASCI-Q machine (inter-

Message Size	Time per Message (μs)	Bandwidth (MB/s)
16	9.8	1.6
64	11.6	5.5
256	13.6	18.8
1024	21.4	47.9
4096	44.4	92.3
16K	139	118
64K	532	123
256K	2109	124

Table 2.7: Converse Network Bandwidth with two way traffic

#Nodes	Elan-Main
4	300
16	292
64	267
256	233

Table 2.8: Elan node bandwidth (MB/s) with two-way traffic

connected by QsNet) has been reported in [18]. Table 2.8 shows the network bandwidth as a function of the size of the fat tree. The bandwidth is the lowest when messages go to the highest level of switches. For example on a fat-tree of size 64, node 0 sends a message to node 32, or node 12 sends a message to node 61, etc.

We believe this drop in network throughput is due to the small packet size (320 bytes) used by the QsNet network protocol, which only allows one packet to be in flight from a given source. On receiving the header of the packet, the receiver optimistically sends an acknowledgment and the next packet is only sent when the acknowledgment arrives at the sender. However, if the acknowledgment is delayed there will a stall in the network channel leading to a drop in throughput [36, 18]. When the sender and receiver are far away in the network this delay is quite likely, which seems to explain the throughput drop in Table 2.8.

2.2 Performance Study of Myrinet

Myrinet [48, 17] is a simpler network, as its ASIC controller cannot access application memory directly [17]. Address translation is done by the NIC through a software TLB, and so each TLB miss results in an operating system interrupt. The interrupt handler translates the page, updates the Myrinet TLB and ensures memory

protection in a multi-tasking environment.

When a send is issued, the message is sent page by page. Each is page is copied into a local buffer and sent from there. The copying and sending is pipelined at the page level to achieve a good bandwidth. Myrinet also requires that the send and receive buffers of a message be pinned in un-swappable memory. So, applications should either send all messages from pinned memory, or pin a small partition and send by copy data into the pinned partition. The Charm runtime (developed in collaboration with Gengbin Zheng) is built using the latter scheme. This adds to the CPU overhead of message passing as messages have to be copied in and out of pinned memory. We now present a performance analysis of Myrinet.

2.2.1 Latency

In Myrinet, message TAG matching is done in the main CPU. Hence, posting more receives does have an impact not application performance, though the CPU overhead is marginally increased. the message latency for a short message with varying number of posted receives is shown in Table 2.9.

Message Size(b)	#Receives Posted	MPI Latency (μs)
16	1	6.5
16	5	6.5
16	9	6.5
16	17	6.5
16	33	7.0
16	65	7.0

Table 2.9: Message latency vs Number of posted receives for MPI

The message latency with increasing message size is shown by Table 2.10. These results were obtained from the ping-pong benchmark on two nodes of the Tungsten cluster [68]. Observe that the CPU overhead here increases with message size. This is because, the runtime has to copy the message into a pinned buffer before sending the message. On the receiving side, each message has to be copied into an application buffer from a pinned buffer. Hence there are two memory copies for each message send operation.

The message latency and CPU overhead for two-way traffic is shown in Table 2.11. Observe that the CPU overhead nearly doubles, as the processor spends time copying for both send and receive operations.

Message Size	Latency(μs)	CPU Overhead(μs)
16	8.7	0.7
64	8.5	0.7
256	11.3	0.7
1024	16.5	0.9
4096	33.3	1.2
16K	93.4	3.0
64K	308	12.0
256K	1149	112

Table 2.10: Converse Latency vs CPU Overhead for Myrinet for ping-pong

Message Size	Latency(μs)	CPU Overhead(μs)
16	9.9	1.2
64	10.2	1.2
256	14.0	1.4
1024	18.8	1.6
4096	40.4	3.0
16K	103	6.1
64K	316	24.8
256K	1146	283

Table 2.11: Converse Latency vs CPU Overhead for Myrinet with two-way traffic

2.2.2 Bandwidth

Myrinet 2000 has a network bandwidth of 2 Gbps (250 MB/s) each-way and the Charm runtime is able to achieve this peak bandwidth. We measured network bandwidth using the multi-ping benchmark. As mentioned before, this benchmark measures the degree of pipelining in the network. The bandwidth at each message size reflects the maximum bandwidth achievable at that message size.

Table 2.12 shows the network bandwidth achieved for the converse runtime. Observe that the time per message initially stays close to the 6μs, and then grows with with message size at network throughput. The Myrinet NIC sends messages in page by page [17]. For messages smaller than a page, the NIC first translates the application virtual address and then schedules the DMA to copy the message from application memory to NIC memory. The message is then sent from NIC memory, page-by-page. When several messages are sent, this copying from main memory to NIC memory is pipelined with packet transmission on the network. So, for short messages the dominant stage in the pipeline is NIC startup and the copy from main-memory to NIC memory. This makes the message latency depend on the I/O bandwidth (1064MB/s vs network

Message Size	Time per Message (μs)	Bandwidth (MB/s)
16	5.7	2.8
64	5.7	11.2
256	6.0	42.7
1024	6.7	153
4096	17.2	238
16K	66.7	246
64K	263	249
256K	1053	249

Table 2.12: Converse one-way multi-ping throughput

Message Size	Time per Message (μs)	Bandwidth (MB/s)
16	9.4	1.7
64	9.2	7.0
256	9.7	26
1024	11.4	90
4096	25.4	161
16K	71.8	228
64K	269	244
256K	1063	247

Table 2.13: Converse multi-ping throughput with two way traffic

bandwidth of 250MB/s), which is probably why the message latency grow slowly. For large messages, the network becomes the dominant stage in the pipeline and determines the message latency.

Table 2.13 shows the performance of the multi-ping benchmark with two way traffic, where both processors send a stream of messages to each other simultaneously. Observe that the two-way times are within 10 μs of the one-way times, suggesting a constant additional overhead in the NIC. From Table 2.13, we can also infer the absence of PCI contention as maximum achieved throughput is similar for both one-way and two-way communication. In fact, nodes on tungsten support PCIX-133 which has a peak bandwidth of about 1064MB/s. Hence PCI is not a bottleneck here, as it was on Lemieux.

We have also observed that unlike Elan, Myrinet bandwidth does not change with the size of the network. However, Myrinet uses a CLOS network which is less dense than a fat-tree network, resulting in more contention for all-to-all traffic [66].

2.3 Performance Study of Infiniband

Infiniband [1] is a standard for high performance computing networks. The standard presents a wide variety of guidelines and protocols for vendors to produce network interfaces and switches, allowing different vendors to produce network interfaces and switches. The standard also has support for efficient hardware multicasts in the switching network. The popular vendor for Infiniband is Mellanox Technologies [45]. In this section, we use Infiniband to represent Mellanox Infiniband.

The Infiniband network interface architecture is also similar to the generic network interface we presented in Figure 2.1. It has a network interface controller (NIC) and DMA engines to move data between the channels and system memory. The Infiniband NIC can freely access application virtual memory, but cannot handle page faults. So the application memory should be pinned and un-swappable. Both message passing and one-sided communication through remote-DMA are supported in Infiniband. The Charm runtime (developed by Greg Koenig) is implemented on top of the Virtual Machine Interface (VMI) [49]. All results presented in this section are based on this runtime.

2.3.1 Message Latency

In Infiniband, message tag matching is performed by the CPU. Hence, posting several receives does not affect the performance of the network interface. This is demonstrated by Table 2.14.

Message Size(b)	#Receives Posted	MPI Latency (μs)
16	1	5.0
16	5	5.0
16	9	5.0
16	17	5.0
16	33	5.0
16	65	5.0

Table 2.14: Message latency vs Number of posted receives for MPI on Infiniband

As Infiniband requires that the application virtual memory is pinned and un-swappable, the Charm VMI runtime maintains a pool of pinned buffers which can be reused by the application. This is unlike the Charm runtime on Myrinet, which managed pinning through copying. The Infiniband network supports a high full duplex bandwidth of 10Gbps, which is close to the memory bandwidth on many systems, we avoid copying to get better throughput.

Message Size	Latency(μs)	CPU Overhead(μs)
16	10.3	0.6
64	11.2	0.5
256	11.6	0.6
1024	14.6	0.6
4096	20.8	0.6
16K	42.7	0.8
64K	131	0.9

Table 2.15: Converse Latency vs CPU Overhead for ping-pong with one-way traffic

Message Size	Latency(μs)	CPU Overhead(μs)
16	10.3	0.5
64	11.2	0.5
256	11.6	0.5
1024	14.9	0.6
4096	21.4	0.7
16K	43.3	0.9
64K	135	1.0

Table 2.16: Converse Latency vs CPU Overhead for ping-pong with two-way traffic

The latency and CPU overhead for ping-pong with one-way traffic is shown in Table 2.15, and it clearly shows a low CPU overhead. The message latency here is a little higher than the MPI latency ($10\mu s$ vs $5\mu s$ for MPI) due to overheads in the Converse and VMI runtimes. The message latency and CPU overhead with two-way traffic is presented in Table 2.16.

2.3.2 Bandwidth

The published bandwidth for Infiniband 4X is 10Gbps or 1.25GB/s. To evaluate the bandwidth on Infiniband, we ran the multi-ping benchmark on the architecture Opteron cluster, and the results are shown by Table 2.17. The maximum achievable bandwidth is about 650MB/s, possibly due to the contention in the PCI chip-set on the Opteron nodes.

The multi-ping performance with two-way traffic is shown in Table 2.18. The throughput saturates at about 317MB/s for 64KB messages, and then drops for larger messages. The lower throughput at 64KB is probably because of PCI contention from traffic in both directions. The drop in throughput after 64KB messages could be because the memory allocator ran out of pinned memory and the pinning/un-pinning overheads are also reflected here. Development of sophisticated memory allocators for Infiniband is left as

Message Size	Latency(μs)	Bandwidth (MB/s)
16	4.0	4.0
64	4.0	16.0
256	4.1	62.4
1024	5.3	193
4096	9.8	418
16K	27.5	596
64K	101	649
256K	401	654

Table 2.17: Converse Multi-ping performance with one-way traffic

Message Size	Latency(μs)	Bandwidth (MB/s)
16	4.8	3.3
64	5.1	12.5
256	6.9	37.1
1024	9.7	106
4096	19.0	216
16K	52.4	313
64K	207	317
256K	961	273

Table 2.18: Converse Multi-ping performance with two-way traffic

future work for this thesis.

2.4 Communication Model

From, the last three sections we can infer that the communication performance depends on several parameters like, CPU startup overhead, network interface overheads, channel throughput and I/O throughput. Some of these overheads also depend on the hardware and software protocols used. There are also several sources of non-determinism which could make these overheads non-linear. For example, in QsNet, when several receives are posted the message latency increases due to cache misses.

However, under normal circumstances when the response of the network interface is linear, it is still possible to model the performance of the interconnect. In this section, we first present a simple model for a network interface with a linear response and then propose extensions to it. We also plot the predicted times with the actual times and show that the predictions are quite accurate.

For a network with a linear response, the time to send a point-to-point message, denoted by T_{ptp}, is

given by.

$$T_{ptp} = \alpha + m\beta + C + L \tag{2.1}$$

- α is the total processor and network startup overhead for sending each message.
- β is the per byte network transfer time. The byte here is being sent out from main memory.
- C is the network contention delay experienced by the message.
- L is the network latency. As L is small on tightly coupled parallel machines, we will ignore it in our cost equations
- m is the size of the message.

Improving Model Accuracy: The above model is quite simple to predict message latency on modern network interfaces. We have observed that the interconnects use several protocols to communicate messages. Some of these are hardware optimizations are software protocols which depend on the message size. For example, in QsNet messages smaller than 288 bytes are sent inline with the rendezvous. For larger messages, first a rendezvous is sent and after the rendezvous has been acknowledged the message is DMAed to the destination, resulting in a higher α overhead. We believe it is possible to accurately model the performance of a NIC with several pairs of α, β values depending on the message size.

Measuring model parameters: the ping-pong benchmark is a natural choice be to compute the α and β values. However, often in applications, several messages are sent in a burst. A classic example of this bursty traffic is all-to-all communication. Moreover, the model presented in this section will mainly be used to predict the performance of all-to-all communication strategies in Chapters 3 and 4. With bursty all-to-all traffic, software startup overheads in the network interface could be pipelined with packets on the wire, thus reducing these overheads.

Therefore, we use the multi-ping benchmark to estimate α and β values of the network. In the multi-ping benchmark, processors send bursts of messages to each other and the pipelined latency and throughput is computed.

Modeling QsNet. We can model QsNet with just two curves! The two curves along with the performance of multi-ping benchmark are shown in Figure 2.2. For short messages smaller than 288 bytes (Prediction

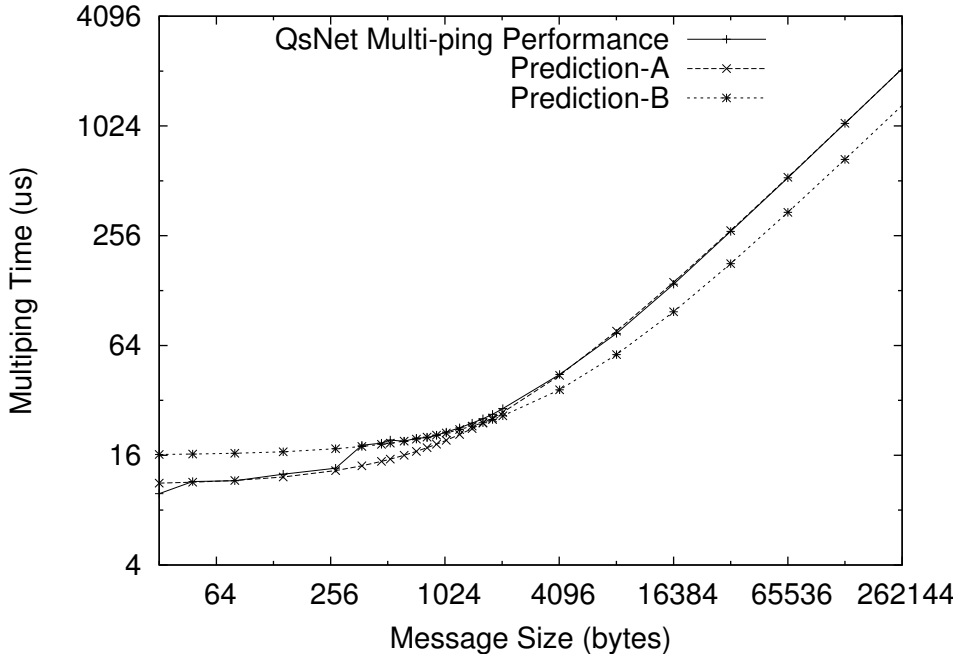


Figure 2.2: Multi-ping predicted and actual performance on QsNet

A), the message is sent along with the rendezvous, resulting in a smaller α overhead of $11\mu s$. As short messages are copied into NIC memory and sent from there, the throughput is determined by PCI bandwidth (125MB/s or $\beta = 8ns/byte$).

For messages larger than 288 bytes (Prediction-B), first a rendezvous is sent and when it is acknowledged the data is DMAed out, resulting in a higher α overhead of $16\mu s$. Fortunately these messages also have a higher throughput of about 200 MB/s ($\beta = 5ns/byte$). This could be because of the absence of PCI contention when the sender message is waiting for the acknowledgment to the rendezvous.

However, when messages larger than 2KB, the behavior of the system is back to Prediction-A. This is probably because the rendezvous of a queued message is acknowledged while it still in the queue. Figure 2.2 clearly demonstrates this.

Predicting Myrinet Performance: The multi-ping performance of Myrinet can also be predicted using two curves (*Prediction-A* and *Prediction B* shown in Figure 2.3). Prediction-A corresponds to (α, β) values of $(9.2\mu s, 2.6ns/byte)$, while (α, β) for Prediction-B are $(5.7\mu s, 4ns/byte)$. The actual predicted performance is the maximum of the two. The slope of the Prediction-A is close to PCI-X bandwidth, while the slope of the Prediction-B is close to the network bandwidth. With short messages in both directions the

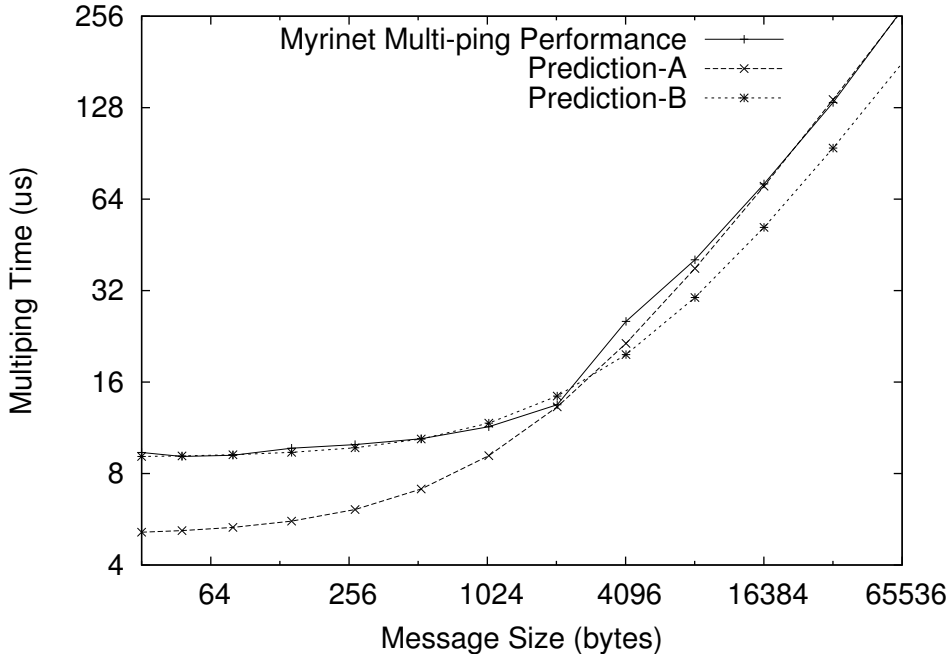


Figure 2.3: Multi-ping predicted and actual performance on Myrinet

network interface appears to have a higher α overhead (Prediction-A). But with large messages, the startup overheads can probably be overlapped with the incoming message, resulting in a lower alpha overhead for Prediction-B.

Figures 2.2 and 2.3 show that two pairs of values for α and β can model the performance of QsNet and Myrinet. However, the cost equations in the rest of the thesis will only use one α and β. The dependence of α and β on message size is avoided for simplicity. Further, when dealing with machine specific and network-interface costs, we also use the following parameters:

- β_{em} is the per byte network transfer time from Elan memory.
- γ is the per byte memory copying overhead for message combining.
- δ is the per byte cost of copying data from main memory to Elan memory.
- P is used to represent the number of processors (or nodes) in the system.

Chapter 3

All-to-All Communication

All-to-all collective communication is an example of a complex collective communication operation involving all the processors. It is a well known performance impediment. In this chapter, we present optimization schemes for all-to-all communication.

All-to-all communication can be classified as all-to-all *personalized* communication (AAPC)[10, 39, 61, 7, 55, 63, 15, 62, 65, 30] or all-to-all *multicast* (AAM)[74, 73, 25, 9, 37]. In AAPC each processor sends a *distinct* message to each other processor, while in AAM each processor sends the *same* message to every other processor in the system. All-to-all multicast is a special case of all-to-all personalized exchange. But as it is a simpler problem, faster implementations of it are possible. MPI defines the primitives *MPI_Alltoall* for all-to-all personalized communication and *MPI_Allgather* for all-to-all multicast.

All-to-all communication is used in many algorithms. AAPC is used by applications like Fast Fourier Transform and Radix Sort, while AAM is needed by algorithms such as Matrix Multiplication, LU Factorization and other linear algebra operations [74]. Molecular dynamics applications like NAMD[54, 34] and computational quantum chemistry applications like CPAIMD[70] have the *many-to-many communication* pattern which is a close cousin of the all-to-all communication pattern. NAMD and CPAIMD have the many-to-many personalized communication and the many-to-many multicast patterns. In many-to-many communication many (not all) nodes send messages to many (not all) other nodes. In this Chapter, we first present strategies for all-to-all communication and then extended them to optimize many-to-many communication.

All-to-all collective communication is commonly used with both small and large messages. For example, each processor in NAMD [54] sends relatively short messages (about 2-4KB) during its many-to-many multicast operation. In CPAIMD, processors send large messages (160 KB) during the many-to-many mul-

ticast operation.

However, different techniques are needed to optimize all-to-all communication for small and large messages. For small messages, the cost of the collective operation is dominated by the software overhead of sending the messages. This can be reduced by message combining. For large messages, the cost is dominated by network contention. Network contention can be minimized by smart sequencing of messages based on the underlying network topology. This chapter mainly deals with all-to-all optimization schemes for short messages, while optimizations for large messages are presented in Chapter 4.

We present the performance results of our strategies on QsNet, which uses a fat-tree network topology. Fat-tree networks, as described in detail in [41, 52, 22], are easy to extend and have a high bisection bandwidth. Hence, they are the preferred communication network for many modern parallel architectures. Network contention on fat-trees has been extensively studied in [22] for the CM5 data network. We use this analysis in the design of our all-to-all communication optimization strategies. The optimization strategies we describe are general, as they can be applied to any fat-tree network and do not restrict the number of processors to powers of two.

Further (in this Chapter for short messages and in Chapter 4 for large messages), we describe additional optimizations specific to QsNet. Sending messages directly from NIC memory substantially increases the network bandwidth, as it avoids DMA and PCI contention. We use this feature in our AAM strategies. As the performance results in Chapter 4 show, the direct strategies scale to 256 nodes (1024 processors) of Lemieux with an *effective bandwidth* of 511 MB/s per node (or 255.5 MB/s each way).

In this thesis, we emphasize *CPU overhead* as an important metric for evaluating collective communication strategies. Most related work has studied collective communication operations from the point of view of completion time, while we believe that computation overhead is an equally important factor. In this thesis, we evaluate the CPU overhead and completion time for all our strategies. We also design strategies with better CPU overheads where ever possible. This is one of our major contributions.

We now present combining strategies to optimize AAPC and AAM for short messages. Chapter 4 will present direct strategies to optimize all-to-all communication with large messages.

3.1 Combining Strategies for Short Messages

The cost of implementing all-to-all communication (AAPC and AAM), by each processor directly sending messages to all (P-1) destinations is given by Equation 3.1 using the model presented in Section 2.4.

$$T_{all-to-all} = (P-1)\alpha + (P-1)m\beta + C \tag{3.1}$$

As presented in Section 2.4, the parameters α and β depend on the message size. However, for simplicity, the cost equations in this Chapter do not show this dependence. With relatively short messages, the cost presented in Equation 3.1 is dominated by the software overhead (α) term.

We use message combining to reduce the total number of messages, making each node send fewer messages of larger size, which are routed along a virtual topology in multiple phases. In each phase, the messages received in the previous phases are combined into one large message, before being sent out to the next set of destinations in the virtual topology. After the final phase, each node has received every other node's data. With these strategies, the number of messages sent out by each node is typically much smaller than P, thus reducing the total software overhead. We present three combining strategies: 2-D Mesh, 3-D Mesh and Hypercube.

3.1.1 2-D Mesh Strategy

In this scheme, the messages are routed along a 2-D mesh. In the first phase of the algorithm, each node exchanges messages with all the nodes in its row. In the second phase, messages are exchanged with column neighbors.

In the case of AAPC, in the first phase each node sends all messages destined to a column to its row neighbor in that column. In the second phase, the nodes sort these messages and send them to their column neighbors.

In all-to-all multicast, in the first phase each node multicasts its message to all the nodes in its row. In the second phase, the nodes combine all the messages they received in the previous round and multicast the combined message to the nodes in their respective columns.

In both cases, each message travels two hops before reaching its destination. In the first phase, each node sends $\sqrt{P}-1$ messages of size $\sqrt{P}m$ bytes for AAPC and m bytes for AAM. In the second phase,

each node sends $\sqrt{P}-1$ messages but of size $\sqrt{P}m$ bytes in both cases.

The completion time for AAPC with the 2-D Mesh strategy $T_{2d-mesh-aapc}$, is shown by Equation 3.2. Here, $C_{2d-mesh-aapc}$ represents the network contention experienced by the messages.

$$T_{2d-mesh-aapc} = 2 \times (\sqrt{P}-1) \times (\alpha + \sqrt{P}m\beta) + C_{2d-mesh-aapc} \tag{3.2}$$

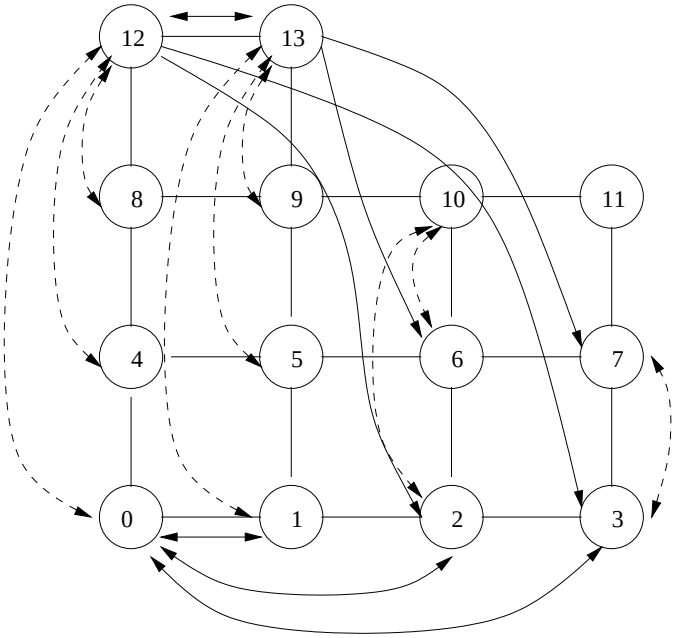
In all-to-all multicast with the 2-D mesh strategy, both phases are multicasts along rows and columns respectively. The completion time for AAM with the 2-D mesh strategy, $T_{2d-mesh-aam}$ is shown in Equation 3.3.

$$T_{2d-mesh-aam} = 2(\sqrt{P}-1)\alpha + (P-1)m\beta + C_{2d-mesh-aam} \tag{3.3}$$

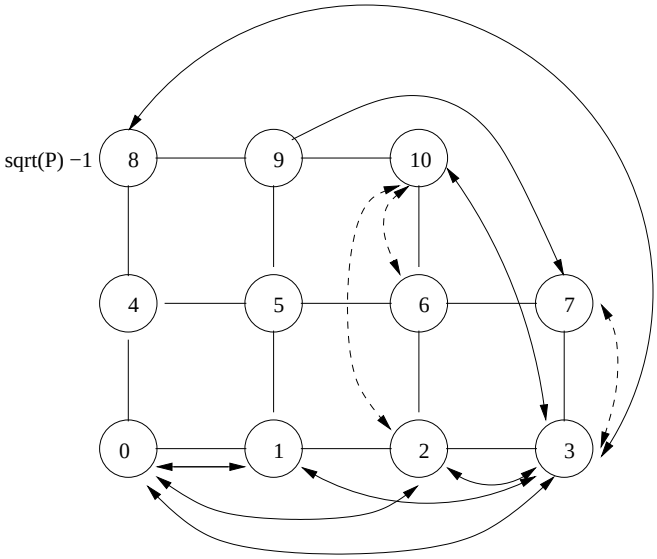
When the number of nodes is not a perfect square, the 2-D mesh is constructed using the next higher perfect square. This gives rise to *holes* in the 2-D mesh. Figure 3.1(a), illustrates our scheme for handling holes in a 2-D mesh with two holes. The dotted arrows in Figure 3.1(a) show the second stage. The role assigned to each hole (which are always in the top row), is mapped uniformly to the remaining nodes in its column. So if node (i,j) needs to send a message to column k and node (i,k) is a hole, it sends that message to node $(j \bmod (nrows-1), k)$ instead. Here $nrows$ is the number of rows in the 2-D mesh. Thus in the first round node 12 sends messages to nodes 2 and 3. No messages are sent to a rows with no nodes in them. Dummy messages are used in case nodes have no data to send.

Observe that $\lceil\sqrt{P}\rceil - 1 \leq NROWS \leq \lceil\sqrt{P}\rceil$, whereas number of columns is always $\lceil\sqrt{P}\rceil$. (If $NROWS \leq \lceil\sqrt{P}\rceil - 1$ then the next smaller square 2-D mesh would have been used). Thus the number of processors that would have sent messages to the hole is at most $\lceil\sqrt{P}\rceil - 1$, and the processors in the hole's column (that share its role) is at least $(NROWS - 1) = \lceil\sqrt{P}\rceil - 2$. Hence the presence of holes will increase the number of messages received by processors in columns containing holes by one (nodes 2,3,6,7 in figure 3.1(a)) or two (node 3 in figure 3.1(b)). Figure 3.1(b) shows the worst case scenario when processor 3 receives two extra messages. The worst case happens when the number of rows is $\lceil\sqrt{P}\rceil - 1$ and there is only one hole.

In the second phase (when processors exchange messages along their columns), these processors will exchange one or two messages less and the total $2(\lceil\sqrt{P}\rceil - 1)$ will remain unchanged. So the α factor of



(a) 2-D Mesh Topology



(b) Mesh: Worst case with holes

Figure 3.1: 2-D Mesh Virtual Topology

Equation 3.2 remains the same while the β factor will only increase by $2(\sqrt{P}).m.\beta$ for AAPC and $2.m.\beta$ for AAM, which is insignificant additional overhead for large P.

$$T_{2d-mesh-aapc} = 2(\lceil\sqrt{P}\rceil - 1)\alpha + 2Pm\beta + C_{2d-mesh-aapc} \tag{3.4}$$

$$T_{2d-mesh-aam} = 2(\lceil\sqrt{P}\rceil - 1)\alpha + (P+1)m\beta + C_{2d-mesh-aam} \tag{3.5}$$

Optimal Mesh: It is possible that other meshes can be constructed if P is not a perfect square. Consider a 2-D mesh of x columns and y rows. Hence, $xy = P$, while the number of messages exchanged for an all-to-all operation with this 2-D mesh is $x + y - 2$. Simple arithmetic shows that

$$x + y \geq 2 * \lfloor\sqrt{P}\rfloor$$

Our bound on the number of messages is actually just two more than this 2-D mesh lower bound.

3.1.2 3-D Mesh

We also implemented a virtual 3-D Mesh topology. In this topology messages are sent in three phases along the X, Y and Z dimensions respectively.

In the first phase of AAPC, each processor sends messages to its $\sqrt[3]{P} - 1$ neighbors along the X dimension. The data sent contains the messages for the processors in the plane indexed by the X coordinate of the destination. In the second phase, messages are sent along the Y dimension. The messages contain data for all the processors that have the same X and Y axes but different Z axis as the destination processor. In the third and final phase data is communicated to all the Z axis neighbors. The cost of AAPC with the 3-D Mesh topology is given by Equation 3.6. Here, in all phases each processor sends $\sqrt[3]{P} - 1$ messages of size $\sqrt[3]{P^2}m$.

$$T_{3d-mesh-aapc} = 3(\sqrt[3]{P} - 1) \times (\alpha + \sqrt[3]{P^2}m\beta) + C_{3d-mesh-aapc} \tag{3.6}$$

AAM has three similar phases. In phase 1 of AAM, each processor multicasts its message along the x-axis of the 3-D Mesh. In phase 2, processors combine the messages received in the previous round and

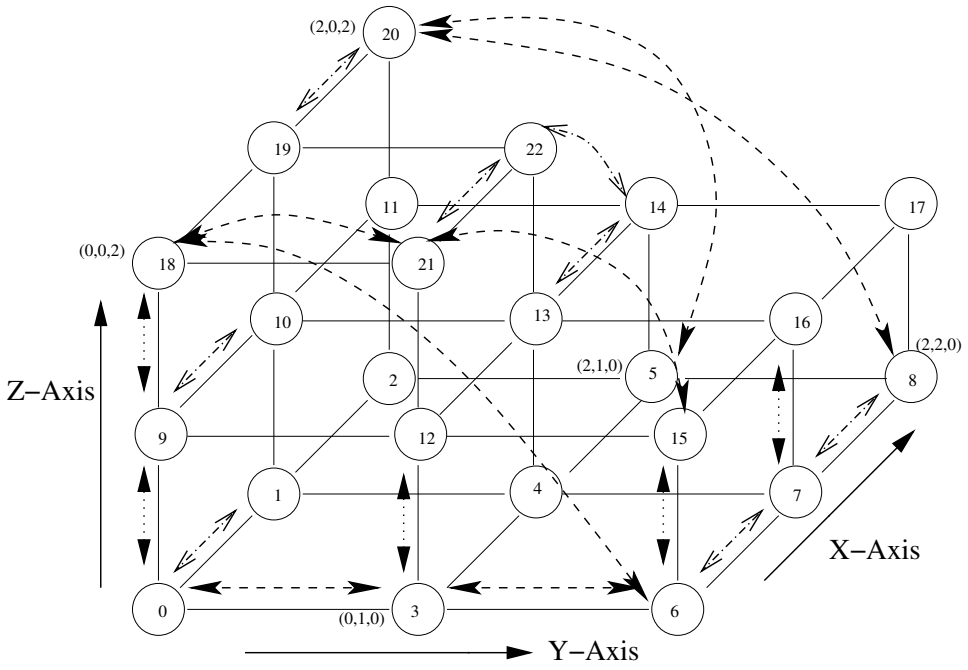


Figure 3.2: 3-D Mesh Topology

multicast the combined message along the y dimension. In phase 3, processors combine the messages received in the second round and multicast it to all neighbors along the z-dimension. In this strategy, each processor sends $\sqrt[3]{P} - 1$ messages of sizes m, $\sqrt[3]{P}m$ and $(\sqrt[3]{P})^2$ in phases 1,2 and 3 respectively. The cost of AAM with the grid virtual topology is given by equation 3.7.

$$T_{3d-mesh-aam} = (\sqrt[3]{P} - 1) \times (3\alpha + m\beta(1 + \sqrt[3]{P} + \sqrt[3]{P^2})) + C_{3d-mesh-aam} \tag{3.7}$$

When the number of processors is not a perfect cube, the next larger perfect cube is chosen. If X, Y, Z are the sizes of the 3-D Mesh in the x,y,z dimensions respectively, then $X = Y = \lceil \sqrt[3]{P} \rceil$, and $Z \leq \lceil \sqrt[3]{P} \rceil$. Figure 3.2 shows a 3-D Mesh with 23 processors.

Holes are mapped in a similar fashion as they were in the 2-D Mesh strategy. Holes are mapped to the corresponding processors in the inner planes (processors with same x and y axis but different z coordinates). Messages to holes will be sent in phases 1 and 2. Holes are mapped as follows: a message from processor (x_1, y_1, z_1) to the hole (x_2, y_2, z_2) is sent to the destination $(x_2, y_2, x_2 \bmod (Z - 1))$ in phase 1 and to $(x_2, y_2, y_2 \bmod (Z - 1))$ in phase 2. As shown in figure 3.2, in phase 1 processor 20 (2,0,2) sends its messages to the processors 5 (2,1,0) and 8 (2,2,0).

Simple arithmetic shows that $Z \geq Y - 3$, and so if $X = Y \geq 3$, a representative for a hole will receive atmost 4 more messages (2 in each of the phases). With AAPC these messages will be of sizes $\lceil \sqrt[3]{P} \rceil^2 m$ bytes in both phases. With AAM however, these messages will be of size m bytes in phase 1 and $\lceil \sqrt[3]{P} \rceil m$ bytes in phase 2. These terms are insignificant and get dominated by the other terms in the cost equation.

The cost for 3-D Mesh strategy with holes for AAPC is given by Equation 3.8 and for AAM is given by 3.9. The parameter λ_h is 1 if there are holes in the 3-D Mesh and 0 otherwise.

$$T_{3d-mesh-aapc} \approx 3\lceil \sqrt[3]{P} \rceil \alpha + (3P + (4\lambda_h - 3)\lceil \sqrt[3]{P} \rceil^2) m\beta + C_{3d-mesh-aapc} \tag{3.8}$$

$$T_{3d-mesh-aam} \approx 3\lceil \sqrt[3]{P} \rceil \alpha + (P + 2\lambda_h \lceil \sqrt[3]{P} \rceil) m\beta + C_{3d-mesh-aam} \tag{3.9}$$

3.1.3 Hypercube

The hypercube (Dimensional Exchange) scheme consists of $log_2(P)$ stages. In each stage, the neighboring nodes in the same dimension exchange messages. In the next stage, these messages are combined and exchanged between the neighbors in the next dimension. This continues until all the dimensions are exhausted.

Thus in the first phase of AAPC, each processor combines the messages for $P/2$ processors and sends it to its neighbor in that dimension. In the second phase, the messages destined for $P/4$ processors are combined. But now, each processor has the data it received in phase 1 in addition to its own data. Thus it combines $2 \times (P/4) \times m$ bytes and sends it to its neighbor. The overall cost is given by Equation 3.10. Observe the equation also include a memory copying overhead from message combining (the γ term). As the hypercube strategy sends only one message in each phase, the message combining overheads also show up in the cost equations. The strategies 2-D Mesh and 3-D Mesh send several messages in each phase, and so they can go in a pipeline, where the next message is copied while the first one is on the wire.

$$T_{hcube-aapc} = log_2 P \times (\alpha + \frac{P}{2} m(\beta + \gamma)) + C_{hcube-aapc} \tag{3.10}$$

In the first phase of AAM however, each node sends its multicast message (size m bytes) to its neighbor. In the second phase, each node combines the message it received in the previous round with its message and sends $2m$ bytes to its neighbor. In the third phase, the messages from the previous rounds are combined

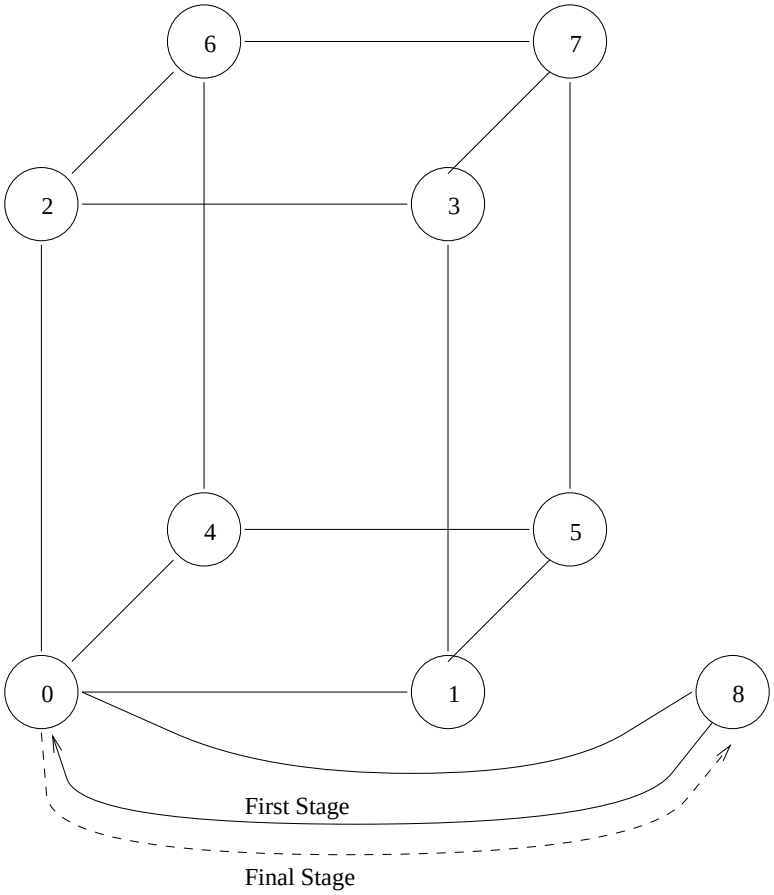


Figure 3.3: Hypercube Virtual Topology

with the local message leading to a message size of $(m + m + 2m = 4m)$. In round i, the message size is $2^{i-1}m$. The overall cost is given by the Equation 3.11.

$$T_{hcube-aam} = log_2 P\alpha + (P-1)m(\beta+\gamma) + C_{hcube-aam} \tag{3.11}$$

With imperfect hypercubes, when the number of nodes is not a power of 2, the next lower hypercube is formed. In the first step, the nodes that are outside this smaller hypercube send all their messages to their corresponding neighbor in the hypercube. For example, in Figure 3.3, node 8 sends it messages to node 0 in the first stage. Dimensional exchange of messages then happens in the smaller hypercube, where all the messages for node 8 are sent to node 0. In the final stage, node 0 combines all the messages for node 8 and sends them to node 8. If there are holes, many nodes will have twice the data to send.

AAPC cost of hypercube with holes is shown in Equation 3.12. Here $\lambda_h = 1$ if there are holes and 0 otherwise. The AAM cost is shown by Equation 3.13.

$$T_{hcube-aapc} \approx log_2 P \times [(1+\lambda_h)\alpha + (\frac{1+\lambda_h}{2}) \times Pm(\beta+\gamma)] + C_{hcube-aapc} \tag{3.12}$$

$$T_{hcube-aam} = (log_2 P + \lambda_h)\alpha + (1+\lambda_h)(P-1)m(\beta+\gamma) + C_{hcube-aam} \tag{3.13}$$

3.1.4 All-to-All Personalized Communication performance

Figures 3.4 and 3.5 present the performance of AAPC using both the communication framework (Chapter 6) and MPI on Lemieux, using 4 processors per node. The strategies 2-D Mesh and 3-D Mesh do better than direct sends for messages smaller than 1000 bytes on both 512 and 1024 processor runs. For very small messages, the indirect strategies are also better than MPI all-to-all. Also notice the jump for direct sends at message size of 2KB. This is because our runtime system switches from statically to dynamically allocated buffers at this point. MPI has a similar and much larger jump, which further increases with the number of processors.

Although the indirect strategies are clearly better than direct sends for messages smaller than 1KB, they are worse than MPI for the range 300b to 2KB. However, two factors make our strategies superior to MPI.

Scalability: Figure 3.6 shows the *scalability* of our schemes compared with MPI for the all-to-all operation

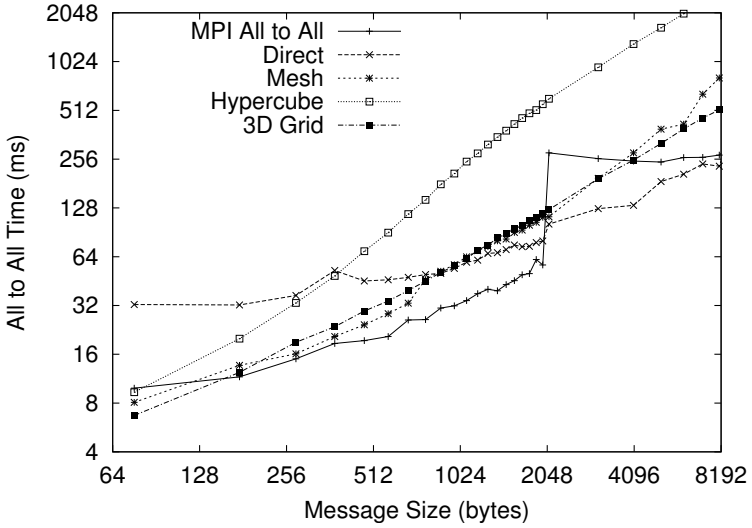


Figure 3.4: AAPC Completion Time (ms) (512 Pes)

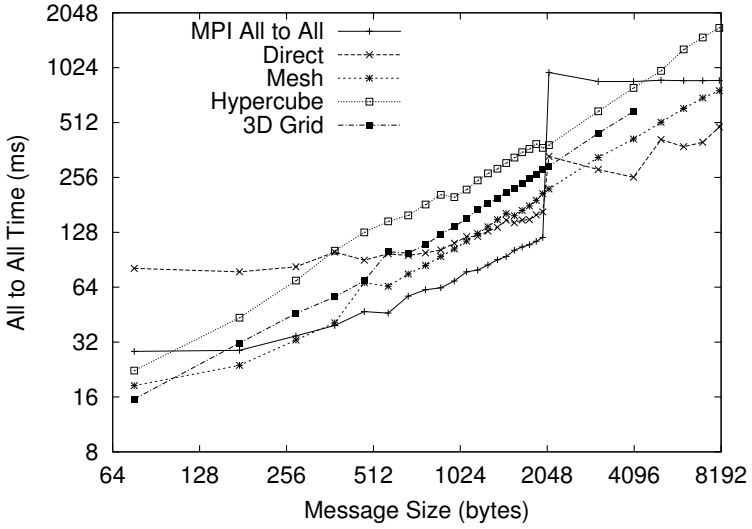


Figure 3.5: Completion Time (ms) on 1024 processors

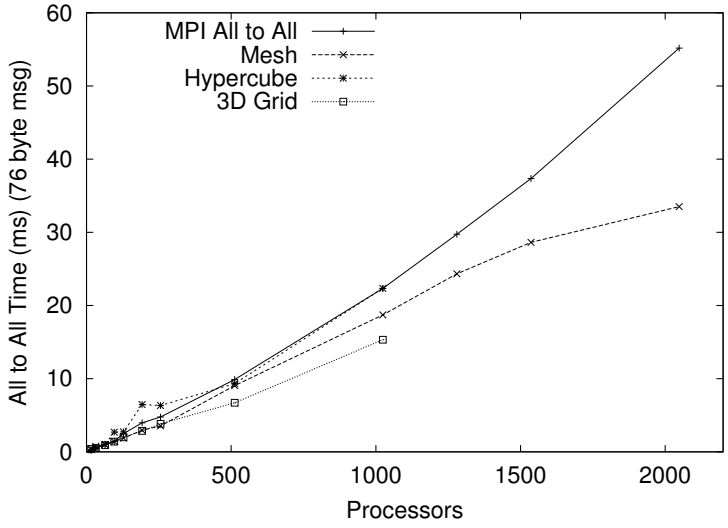


Figure 3.6: AAPC time for 76 byte message

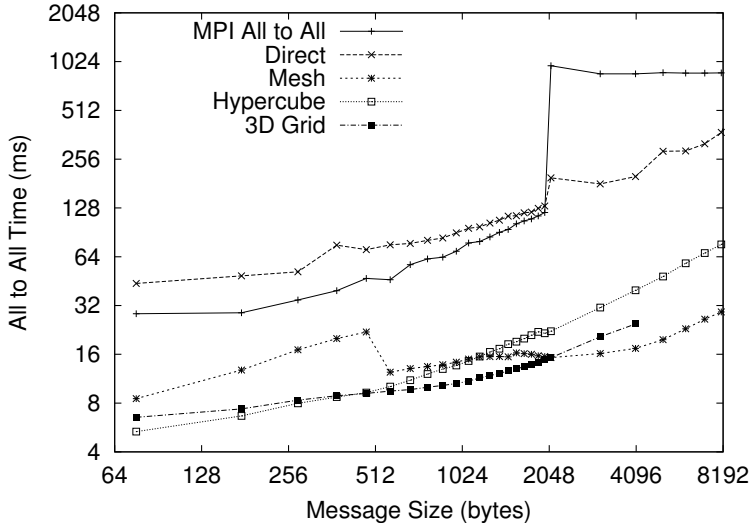


Figure 3.7: CPU Time (ms) on 1024 processors

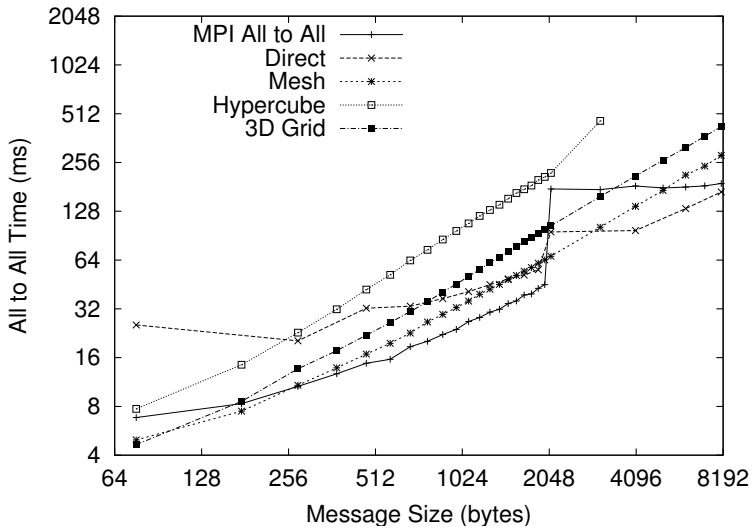


Figure 3.8: AAPC Completion Time (ms) (513 Pes)

with a message size of 76 bytes. The Hypercube strategy does best for a small number of processors (this is not clearly seen in the linear-linear graph). But as the number of processors increases, 2-D mesh and 3-D Mesh improve, because they use only two and three times the total amount of network bandwidth respectively, while for hypercube the duplication factor is $\log p/2$. MPI compares well for a small number of processors but for 64 or more processors our strategies start doing substantially better (e.g. 55 ms vs 32 ms on 2048 processors).

CPU Overhead: Probably the most significant advantage of our strategy arises from its use of a message-driven substrate on machines with a communication co-processor. In contrast to MPI, our library is asynchronous (with a non-blocking interface), and allows other computations to proceed while AAPC is in progress. Figure 3.7 displays the amount of *CPU time* spent in the AAPC operations on 1024 processors. This shows the software overhead of the operation incurred by the CPU. Note that this overhead is substantially less than the overall time for our library. For example at 8KB, although the 2-D mesh algorithm takes about 800 ms to complete the AAPC operation, it takes less than 32 ms of CPU time away from other useful computation. This is possible because of the communication co-processor in the Quadrics Elan NIC, and hence the low CPU overhead of communication operations (Chapter 2).

In our implementation, we have two calls for the AAPC interface. The first one schedules the messages and the second one polls for completion of the operation. On machines with support for "immediate mes-

sages" — those that will be processed even if normal computation is going on — and on message-driven programming models (such as Charm++), this naturally allows for other computations to be carried out concurrently. In other contexts, user programs or libraries need to periodically call a polling function to allow the library to process its messages.

Another interesting perspective is provided by the performance data on on 513 processors with 3 processors per node, shown in Figure 3.8. Note that all the strategies perform much better here (compare with Figure 3.4). We believe this is due to OS and Elan interactions when all 4 processors on a node are used (Chapter 2).

3.1.5 All-to-All Multicast Quadrics Optimizations

In AAM with the 2-D Mesh and the 3-D Mesh topologies, processors send the same message to their neighbors in the topology. This partial multicast can be optimized for QsNet by copying the messages into the network interface and sending them from there. The new cost equations are shown below. Observe the δ terms, which represent the cost of copying messages from main memory to NIC memory.

$$T_{2d-mesh-aam} \approx 2\sqrt{P}\alpha + Pm\beta_{em} + \sqrt{P}m\delta + C_{2d-mesh-aam} \tag{3.14}$$

$$T_{3d-mesh-aam} \approx 3\sqrt[3]{P}\alpha + Pm\beta_{em} + (\sqrt[3]{P})^2 m\delta + C_{3d-mesh-aam} \tag{3.15}$$

With the hypercube topology however, each processor sends a distinct message in each phase. Hence we cannot take advantage of the lower transmission overhead β_{em}, which depends on the same message being sent several times. But, we can use a *hybrid* approach with hypercube exchange for $log_2 P - \zeta$ stages and then direct exchange on ζ-dimension sub cubes. For $\zeta = 2$, the number of messages would only increase by 1, and for $\zeta = 3$ it would increase by 4. However the per byte term would be reduced substantially, as most of the data is sent in the last few stages. The new cost equation is given by Equation 3.16. For simplicity we have not included the holes term in this equation. Equation 3.16 has three parts to it, (i) hypercube cost for $log_2 P - \zeta$ stages, (ii) direct cost within the ζ-subcube with messages of size $(P/2^\zeta)m$ bytes, (iii) cost of copying this message into the network interface. The optimal value of ζ depends on the number of nodes P and the size of the message m. The term P' represents $P/2^\zeta$ in Equation 3.16.

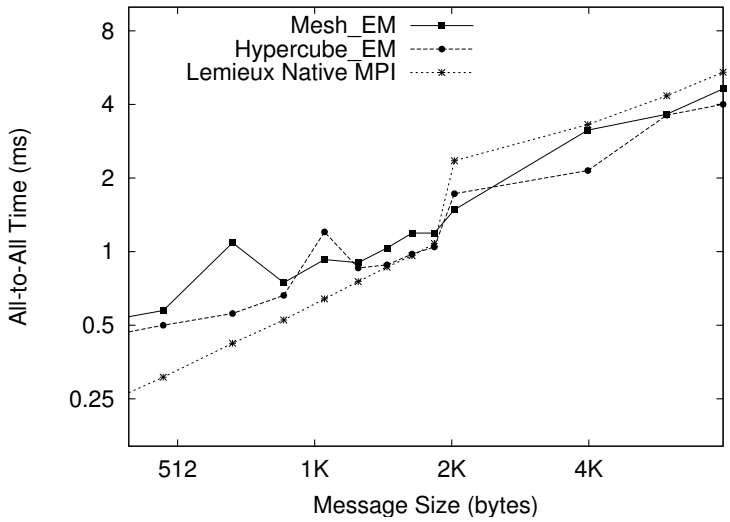


Figure 3.9: AAM Performance for short messages on 64 nodes

$$T_{hcube} = (log_2 P - \zeta)\alpha + (P' - 1)m(\beta + \gamma) + (2^{\zeta} - 1)(\alpha + P' m \beta_{em}) + P' m \delta + (P-1)mC_{hcube-aam} + L_{hcube-aam} \quad (3.16)$$

3.1.6 All-to-all Multicast Performance

Figures 3.9 and 3.10 show the short message performance (completion time) of the strategies (combining strategies and Lemieux MPI), on 64 and 256 nodes respectively. The 2-D mesh strategy presented in these graphs (2-D Mesh_EM) sends all the messages from Elan memory (EM). Hypercube_EM, shown in the plots, directly sends messages from Elan memory in the last three stages, i.e. parameter $\zeta = 3$. Observe that MPI does better than our strategies for very short messages, because of scheduling and timer overheads in the Charm runtime system. But for messages larger than 2KB on 64 nodes and 400 bytes on 256 nodes, Hypercube_EM starts doing better.

Figure 3.11 shows the advantage of copying the message into Elan memory, on 256 nodes. Here, 2-D Mesh_MM shows the performance of the 2-D mesh strategy sending all its messages from main memory. For Hypercube_MM, there are no direct stages, i.e. $\zeta = 0$. Sending messages from Elan memory (EM) substantially improves the performance of the 2-D mesh strategy on 256 nodes. Hypercube also benefits from direct stages that send messages from Elan memory.

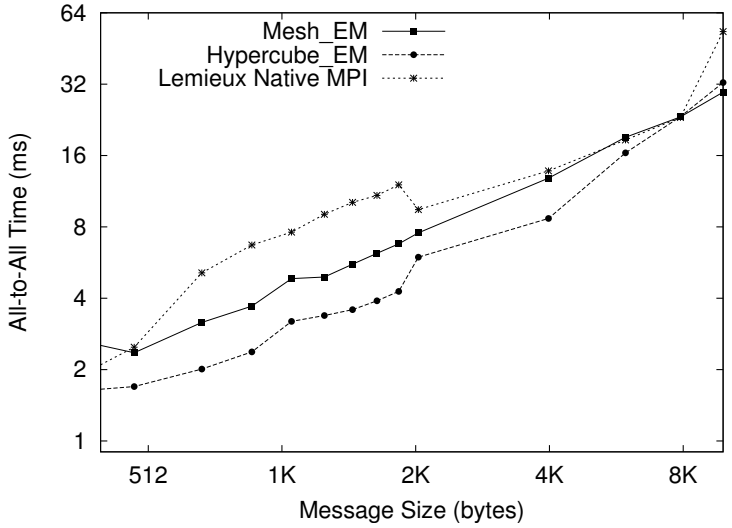


Figure 3.10: AAM Performance for short messages on 256 nodes

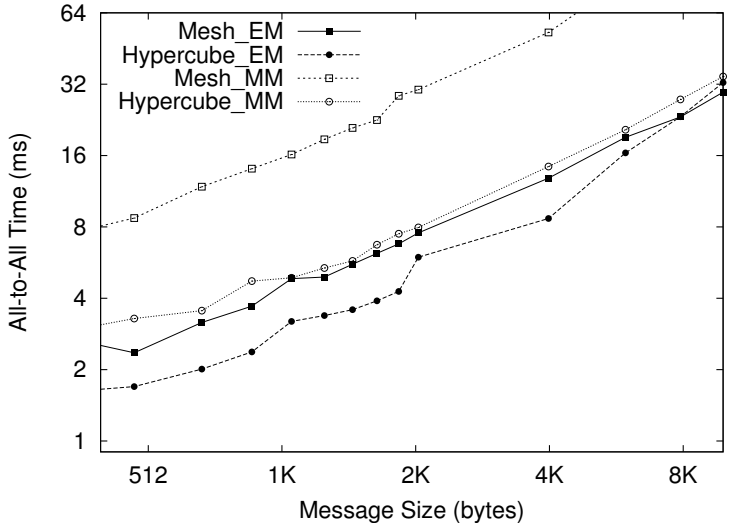


Figure 3.11: Effect of sending data from Elan memory (256 nodes)

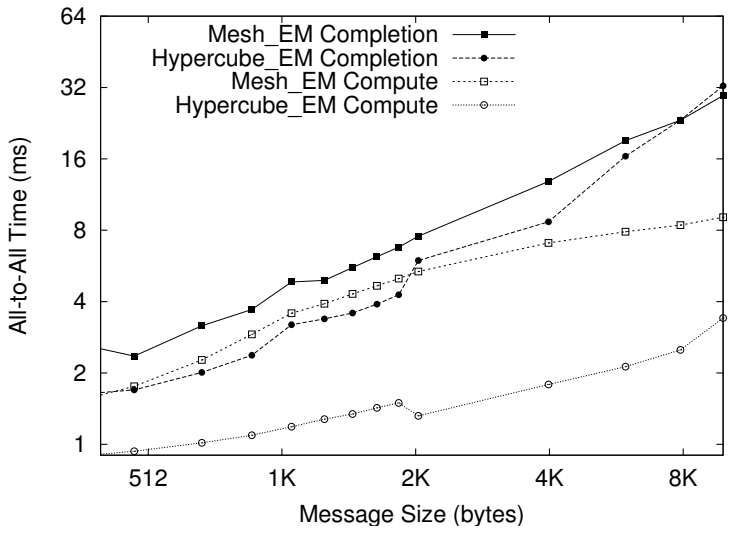


Figure 3.12: Computation overhead vs completion time (256 nodes)

Figure 3.12 shows the computation overhead of Hypercube and 2-D Mesh strategies. Notice that the computation overhead is much less than the completion time. This justifies the need for an asynchronous split-phase interface, as provided by our framework.

3.2 Comparing predicted and actual performance

In this section, we present the effectiveness of our cost equations. We compare the predicted performance of AAPC strategies like 2D-Mesh and Direct with actual performance on PSC Lemieux. Earlier in this chapter, we used 4 processors per node for many of our results. To minimize non-determinism from operating system daemons and other sources, we just use one processor on each node for these runs. Figures 3.13 and 3.14 show the predicted times and the actual times for 2D-Mesh strategy and Direct strategy. We used the communication model presented in Section 2.4 to model QsNet performance on PSC Lemieux. As presented in Section 2.4, the α and β values depend on the message size. For combining strategies we also have to add a $2.5\mu s/message$ combining overhead as all messages have to be malloced, collected and combined, before they are sent on the network. This overhead is omitted from the cost equations for simplicity. The plots show that our cost equations closely model the actual performance of QsNet.

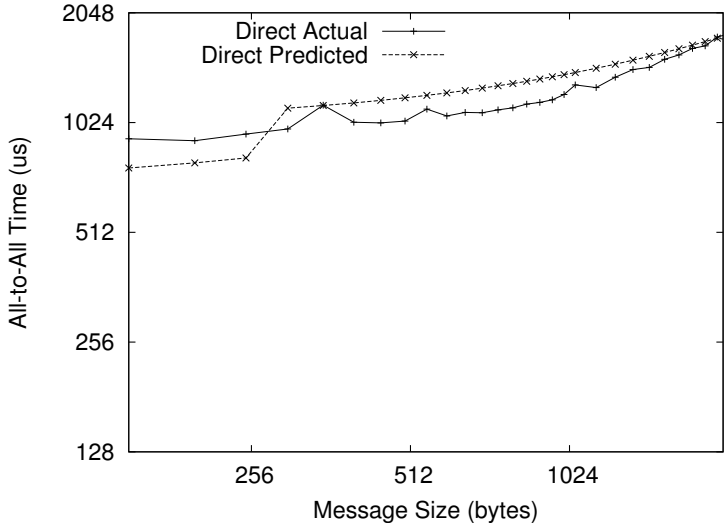


Figure 3.13: Direct strategy performance

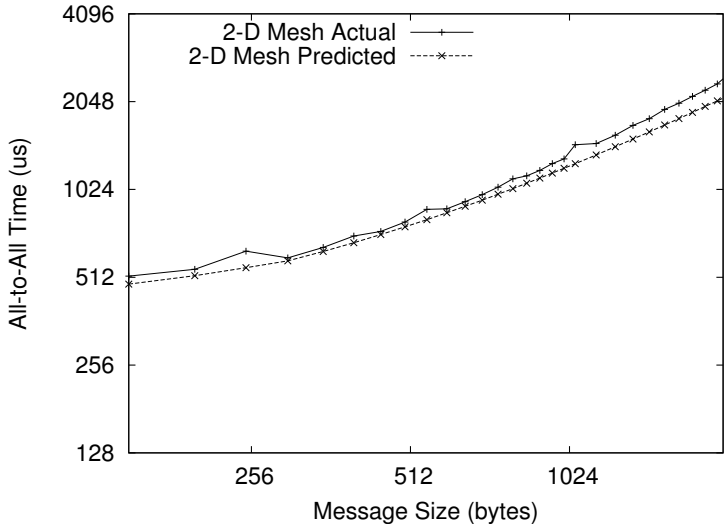


Figure 3.14: 2D Mesh topology performance

3.3 Many-to-Many Collective Communication

In many applications, each processor in the collective operation may not send messages to *all* other processors. We call this variant of all-to-all communication as *many-to-many communication*. Applications like NAMD, Barnes Hut Particle simulator[71], Euler Solver and Conjugate grid solver etc. have the many-to-many communication pattern. Here the degree (δ) of the communication graph, which is the number of processors each processor communicates with, becomes another important factor to consider while optimizing these applications. We present the analysis of two classes of the many-to-many communication.

3.3.1 Uniform Many-to-Many Communication

In many applications, processors only communicate messages with a subset of processors. For processor p_i, we use S_i to represent the subset of processors it communicates with. We also use δ_i, where $\delta_i = |S_i|$, to represent the size of this subset for p_i. In many applications, δ_i is the same for each processor or there is only a small variance in it. Such a many-to-many communication pattern is termed as uniform many-to-many communication.

In uniform many-to-many communication, all processors send and receive around the same number of messages. All-to-all communication is a special case of this class. We first present our analysis of uniform many-to-many *personalized* communication (UMMPC). Here each processor exchanges a similar number of *distinct* messages with other processors.

An example of UMMPC is the neighbor-send application shown in figure 3.15. Here processor i sends messages to processors $(i + k) \bmod P$ for $k = 1, 2, .., \delta$.

For UMMPC, the cost equations 3.1, 3.2, 3.6,and 3.10 of Section 3.1 are modified:

$$T_{2d-mesh} \approx 2\sqrt{P}\alpha + 2\delta m\beta \quad (3.17)$$

$$T_{3d-mesh} \approx 3\sqrt[3]{P}\alpha + 3\delta m\beta \quad (3.18)$$

$$T_{hypercube} \approx log_2 P \times (\alpha + \delta m\beta) \quad (3.19)$$

$$T_{direct} = \delta \times (\alpha + m\beta) \quad (3.20)$$

In the 2d-mesh strategy, each processor sends δ messages and each of these messages is transmitted

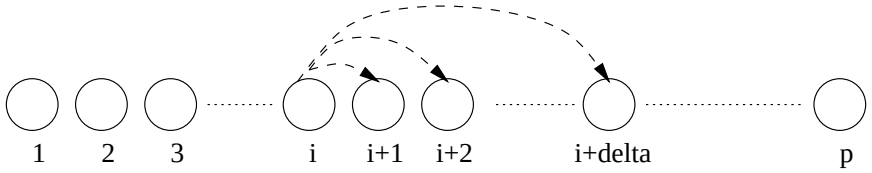


Figure 3.15: Neighbor Send Application

twice on the network. So the amount of per-byte cost spent on all the messages in the system is $2P\delta m\beta$. Since the MMPC is uniform, this cost can be evenly divided among all the processors. The resulting cost equation for the 2d-mesh topology is given by Equation 3.17. By a similar argument we get equations 3.18 and 3.19. Also observe that δ appears only in the β part of the equations 3.17, 3.18 and 3.19. This is because, in each virtual topology, the number of messages exchanged between the nodes is fixed. If there is no data to send, dummy messages must be sent instead.

Figures 3.16 and 3.17 show the performance of the strategies with the degree of the graph δ being varied from 64 to 2,048. In this benchmark, each processor i sends δ messages to processors $(i + k) \bmod P$ for $k = 1, 2, .., \delta$. For small δ the direct strategy is the best. Observe that the direct strategy is more prominent in the 76 byte plot, as these messages get sent with the Elan rendezvous resulting in a lower α overhead.

Comparison between the 2-D Mesh and 3-D Mesh strategies is interesting. The former sends each byte twice, while the latter sends each byte three times. But the α cost encountered is less when the 3-D Mesh strategy is used. For small (76 byte) messages, the α (per-message) cost dominates and the 3-D Mesh strategy performs better. For larger (476 byte) messages, the 3-D Mesh strategy is better until the degree is 512. For larger degrees, the increased amount of communication volume leads to dominance of the β (per-byte) component, and so the 2d-mesh strategy performs better.

Similar optimization schemes and models can also be obtained for uniform many-to-many multicast. As uniform communication implies there are no hot-spots in the system, we use the same analogy presented for UMMPC to extend the equations 3.3, 3.7, 3.11 incorporating the new parameter δ. The new equations are presented below:

$$T_{2-DMesh} \approx 2\sqrt{P}\alpha + \delta m\beta \tag{3.21}$$

$$T_{3-DMesh} \approx 3\sqrt[3]{P}\alpha + \delta m\beta \tag{3.22}$$

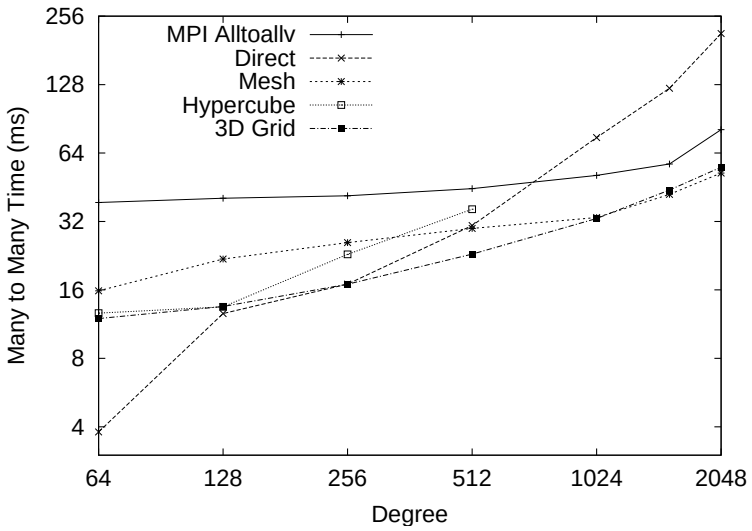


Figure 3.16: MMPC completion time with varying degree on 2048 processors for 76 byte messages

$$T_{Hypercube} \approx log_2 P \times \alpha + \delta m \beta \tag{3.23}$$

$$T_{Direct} = \delta \times (\alpha + m\beta) \tag{3.24}$$

3.3.2 Non-Uniform Many-to-Many Communication

In non-uniform MMPC there is a large variance in the number of messages each processor sends or receives; for example, some processors may be the preferred destinations of the messages. There may also be a variance in the *sizes* of messages processors exchange. With processor-virtualization, non-uniform many-to-many communication can be optimized through communication load-balancing. Heavy communication objects can be smartly placed among processors to make the communication more uniform. But, a general model for non-uniform many-to-many communication is non trivial and is not the emphasis of this thesis. Instead we, present case studies of several complex many-to-many communication patterns and optimizations for them.

Many of these optimizations have been motivated by the CPAIMD (Car Parinello Ab-Initio Molecular Dynamics [70]) application. This application has several many-to-many communication patterns. The application consists of states (3D grids of points) which have the wave functions of electrons in the system.

The computation moves between Fourier space and real space through 3D-FFTs. With several states this would result in several simultaneous 3D-FFT operations. Each 3D-FFT operation requires a trans-

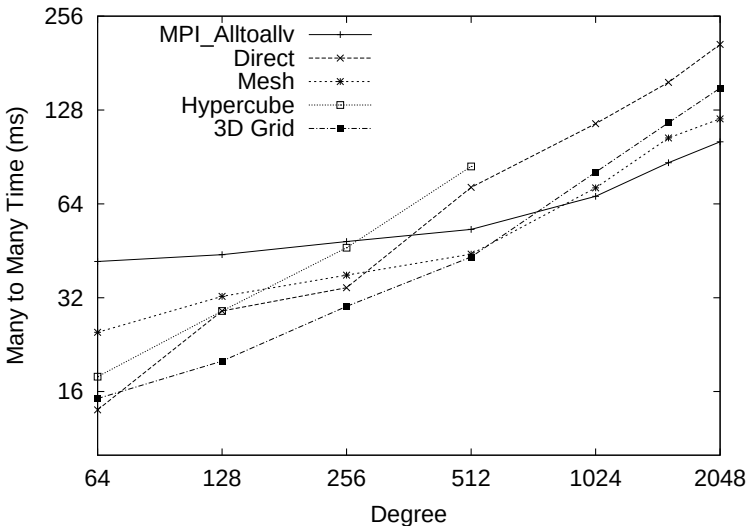


Figure 3.17: MMPC completion time on 2048 processors with 476 byte messages

pose, which is an all-to-all operation within each state. When the application is run on more processors than the number of states, the simultaneous transposes become a many-to-many operation. As the states in Fourier space are sparse the many-to-many operation is non uniform. This complex many-to-many operation involves short messages exchanged between application objects. On receiving each message some computation also needs to be performed, so not only does the communication operation have to be optimized, it also has to be effectively pipelined with computation. We use the *streaming strategy* (Chapter 5.6) to optimize this operation and the performance improvements are presented in Chapter 7.

The CPAIMD application also has an all-to-all multicast operation on a small number of processors, while on larger number of processors this becomes an irregular many-to-many multicast operation. This multicast however involves big messages. For the all-to-all multicast case, we use the direct optimizations like k-prefix and k-shift which we will presented in Chapter 4. On larger number of processors, the many-to-many multicast operation is optimized through the ring strategy. The performance improvements of these optimizations is presented in chapter 7.

NAMD motivates optimizing another complex many-to-many multicast operation. In NAMD, the cells of the atom grid (called patches) multicast atom coordinates to the compute objects, which compute the interaction between the cells. When the number of patches is much smaller than the number of processors this operation is a non-uniform many-to-many multicast operation. Due to a tight critical path in NAMD,

this multicast cannot be optimized through software techniques like trees or rings, because intermediate processors on the tree may get busy executing computes and delay the multicast messages. This operation can however be optimized through hardware multicast support in the switches of the interconnect. This is described in detail in Chapter 8.

3.4 Related Work

The indirect strategies based on virtual topologies have been presented before. The 2-D Mesh and 3-D Mesh strategies has been presented in [10], while hypercube strategy has been presented in [55, 38]. A hybrid algorithm that combines direct and indirect strategies is presented in [62]: it combines the direct Scott's [58] optimal 2-D Mesh communication strategy with the recursive partitioning strategy which is similar to our hypercube. We have also developed a hybrid strategy, which is a hybrid of the hypercube strategy and the prefix send strategy (Chapter 4). The schemes to handle holes [30] and the analysis of the strategies based on the CPU overhead is our contribution.

Chapter 4

Collective Communication: Direct Strategies for Large Messages

When messages are large, combining strategies offer little benefit for the all-to-all operation. However, the communication cost can be optimized by using topology dependent optimizations that minimize network contention. In this section, we develop such strategies for fat-tree networks. We first analyze network contention on fat-tree networks and then present contention free communication schedules. The direct strategies that we present next take advantage of such communication schedules.

4.1 Fat Tree Networks

QsNet uses a *fat-tree* (more specifically, 4-ary n-tree) interconnection topology. We now present the definition of a generic fat-tree [53].

Definition: *A fat-tree is a collection of vertexes connected by edges and is defined recursively as follows*

- *A single vertex is a fat-tree. This vertex is also the root of the fat-tree.*
- *If* $v_1, v_2, ..., v_i$ *are vertexes and* $T_1, T_2, ..., T_j$ *are fat-trees, with* $r_1, r_2, ..., r_j$ *as roots, a new fat-tree can be constructed by connecting with edges of the vertexes* $v_1, v_2, ..., v_i$ *and* $r_1, r_2, ..., r_j$ *in any manner. The roots of the new fat-tree are* $v_1, v_2, ..., v_i$*.*

The graph k-ary n-tree has been defined in [53]. It is a type of fat-tree which can be defined as follows:

Definition: *A k-ary n-tree is a fat-tree that is composed of two types of vertexes:* $P = k^n$ *processing nodes and* nk^{n-1} *switches. The switches are organized hierarchically with n levels that have* k^{n-1} *switches at*

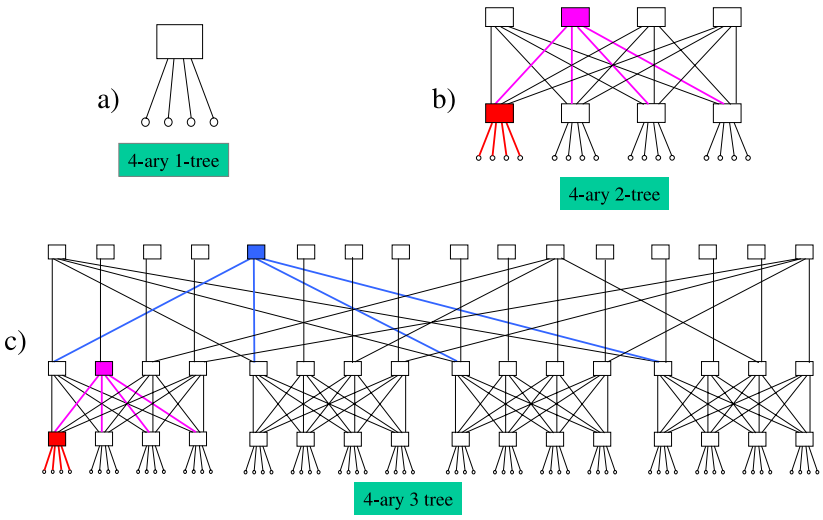


Figure 4.1: Fat Tree topology of Lemieux

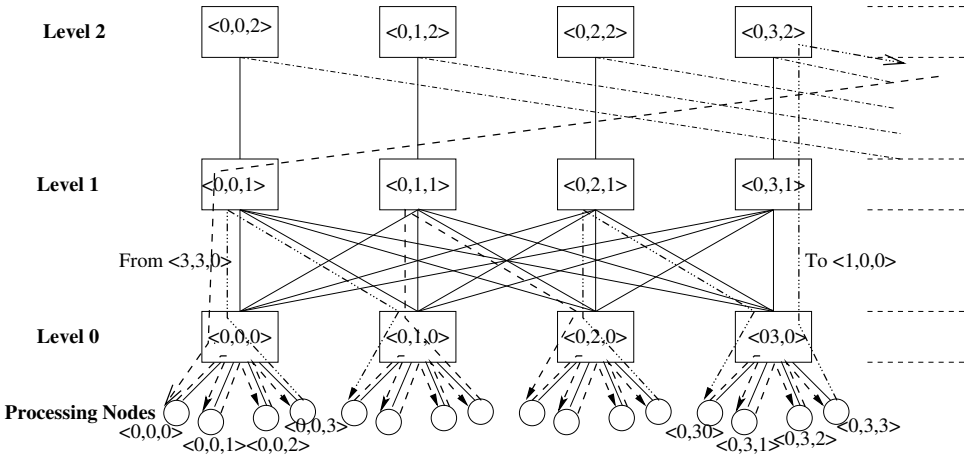
each level. Each node can be represented by the n-tuple $\{0, 1, ..., k-1\}^n$*, while each switch is defined as an ordered pair* $\langle w, l \rangle$ *where* $w \epsilon \{0, 1, ..., k-1\}^{n-1}$ *and* $l \epsilon \{0, 1, ..., n-1\}$. Here the parameter l represents the level of each switch and w identifies a switch at that level. The root switches are at level $l = n - 1$, while the switches connected to the processing nodes are at level 0.

- *Two switches,* $\langle w_0, w_1, ..., w_{n-2}, l \rangle$ *and* $\langle w'_0, w'_1, ..., w'_{n-2}, l' \rangle$ *are connected by an edge iff* $l' = l + 1$ *and* $w_i = w_{i'}$ *for all* $i \neq n - 2 - l$

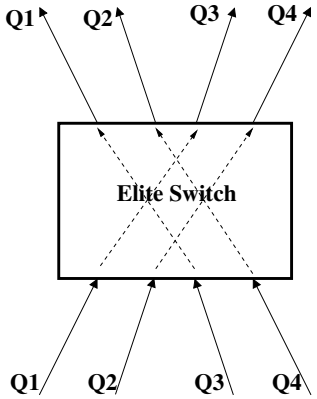
- *There is an edge between the switch* $\langle w_0, w_1, ..., w_{n-2}, 0 \rangle$ *and the processing node* $\{p_0, p_1, ..., p_{n-1}\}$ *iff*

$$w_i = p_i \; for \; all \; i \; \epsilon \; \{0, 1, ..., n-2\}$$

The bisection bandwidth of fat-trees is $O(k^n)$ or $O(P)$. Figure 4.2(a) shows the first quarter of a 64 node fat-tree, with nodes and switches labeled using the above definition. The switches $\langle w_0, w_1, 2 \rangle$ are the root nodes while the switches $\langle w_0, w_1, 0 \rangle$ are connected to the processing nodes $\langle w_0, w_1, i \rangle$.



(a) First Quarter of a 4-ary 3-tree



(b) Contention free top level switches

Figure 4.2: Fat Tree topology

Routing on a fat-tree has two phases: (i) Ascending phase: here the message is routed to one of the common ancestors of the source and the destination, (ii) Descending phase: here the message is routed through a fixed path from the common ancestor to the destination node. Network contention happens mainly in the downward descending phase[22]. Many communication schedules on fat-trees are *congestion free*, i.e. they have no contention during the downward descending phase. The following lemmas present congestion free permutations, where each node sends a message to a distinct destination node. Proofs of these Lemmas have been presented in detail in [22]. We only briefly restate the Lemmas and the outlines of the proofs here.

Lemma 1 *Cyclic shift by 1, where each processor P_i sends a message to the processor $P_{(i+1) \bmod P}$, is*

congestion free.

The proof is straightforward. Only 1/4th of the traffic at the lowest level will go up to the next level and the rest will remain at the lowest level. The traffic that goes up will never compete for the same output link at any level of switches. Figure 4.2(a) also shows the congestion free schedule of the *cyclic-shift-by-1* operation on a 64 node fat-tree.

Lemma 2 *All quarter permutations that preserve the order of messages within a quarter are congestion free.*

In a quarter permutation, all messages from a source quarter go to the same destination quarter. The destination quarter for each quarter is also distinct. For example, Q1, Q2, Q3, Q4 sending messages to Q3, Q4, Q1, Q2 is a quarter permutation. In a quarter permutation, all messages go to the top of the fat tree. At the topmost-level switches, each incoming packet is destined to a different quarter and hence a different output port. So, there will be no contention at the topmost-level switches. Figure 4.2(b) demonstrates this.

Message order is preserved if processor $P_{i,l}$ in quarter Q_i only sends a messages to the corresponding processor $P_{j,l}$ in quarter Q_j. In this scenario the message from $P_{i,l}$ at the topmost switch will use the path used by the message from $P_{j,l}$ (or a translated path) to the topmost switch, hence there will be no network congestion. In fact [22] describes shuffle and exchange quarter permutations that are all congestion free.

Definition: *A permutation is said to map a tree to itself when hierarchical groupings are preserved: siblings remain siblings, first cousins remain first cousins, k^{th} cousins remain k^{th} cousins.*

Lemma 3 *If a permutation maps a fat-tree into itself it is congestion free.*

This is the generalization of Lemma 2 where at each level of switches the traffic is a quarter permutation preserving the order of messages. So it is congestion free. The following Lemmas present communication schedules that map the fat-tree into itself. Hence they are also congestion free.

Lemma 4 *Hypercube dimension exchange is congestion free.*

Lemma 5 *Prefix-Send, where each processor P_i in stage j sends a message to the processor $P_{i\oplus j}$, is congestion free iff the total number of processors P is a power of two.*

Lemma 6 *Cyclic shift by $k = a * 4^j$ (i.e. processor P_i sends a message to either of the processors $P_{i \pm k}$), is congestion free iff a=1,2,3 and $k \leq P$.*

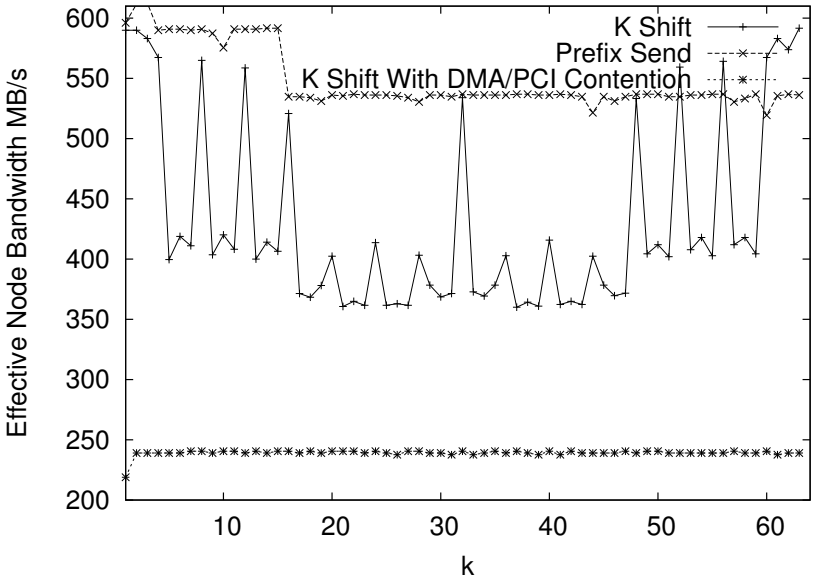


Figure 4.3: Effective Bandwidth for cyclic-shift and prefix-send on 64 nodes

Since the performance of these permutations is important for their use as steps of our multicast strategies, we analyzed them empirically. Figures 4.3 and 4.4 show the performance of the cyclic-shift and the prefix-send permutations on 64 and 256 nodes of Lemieux. In both permutations, in the k^{th} step each node P_i sends a message to its neighbor at distance k. For cyclic shift this neighbor is $P_{(i+k) \bmod P}$, while it is $P_{i \oplus k}$ for prefix-send. In the figures, the x-axis shows the distance k and the y-axis displays the effective bandwidth. The bandwidth for prefix-send is more stable than cyclic shift. Observe that for $k > 16$ the bandwidth of prefix-send drops from 610 MB/s to 534 MB/s and for $k > 64$ it drops to 470 MB/s.

The drop in throughput is due to the a naive packet protocol that is used by QsNet. The NIC sends a packet and stalls on an acknowledgment. Full utilization can only be achieved if this acknowledgment is received before the entire packet has been sent out. On large networks it is likely that the acknowledgment will not arrive on time, leading to a loss of throughput. This issue is dealt in more detail in Chapter 2.

For cyclic shift observe that the peaks in throughput occur only at the values of k given by Lemma 6. For other values of k, network contention impairs throughput. On 64 nodes, the effective bandwidth at the peaks in the plots varies between 560 and 580 MB/s. However, on 256 nodes the peak bandwidth drops and

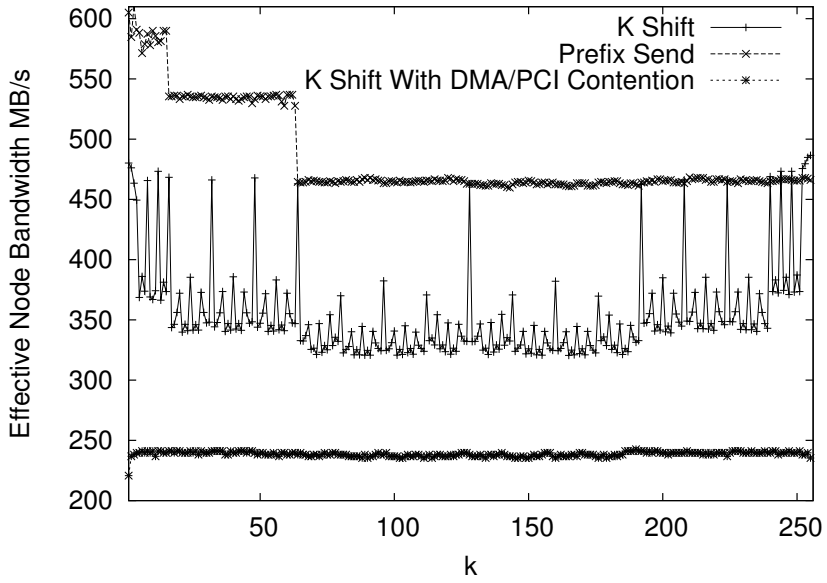


Figure 4.4: Effective Bandwidth for cyclic-shift and prefix-send on 256 nodes

varies between 460 and 485 MB/s. This is because in the *cyclic-shift-by-k* operation nodes at the boundaries send messages to distant nodes, restricting the network throughput at at the peaks. Again wire and switch delays are responsible for this loss of network throughput.

Figures 4.3 and 4.4 also show the performance of the *cyclic-shift-by-k* permutation with messages sent from main memory. DMA and PCI contention bring the effective bandwidth down to a steady 240MB/s. Observe that network contention makes no difference to the effective per-node bandwidth as the node bandwidth here is much lower than the network capacity. This shows the usefulness of sending messages from NIC memory, as node bandwidth is more than doubled.

4.2 All-to-All Personalized Communication

Unlike the combining strategies described in Section 3.1, which were applicable for both AAPC and AAM, the direct strategies we describe now are specific to AAPC or AAM. Only the prefix-send strategy happens to be applicable to both AAPC and AAM, for the Quadrics QsNet network. The important difference here is that network contention is more severe in the case of AAM, where each processor sends the same message to everyone enabling it to be copied and sent from NIC memory. The throughput from Elan memory is much higher than the network throughput from main memory (figures 4.3 and 4.4), but also more sensitive

to network contention. Therefore, our AAM strategies are more complex, as presented in the next section.

With AAPC however, there is no gain in sending messages from Elan memory as each message is different. So the cost of copying the message into Elan memory will nullify the gain of sending the message with a higher bandwidth. QsNet has minimal network contention, when data is sent from main memory. This because PCI bandwidth is not enough to fully load the network. So, both prefix-send and cyclic-shift-by-k perform well in this scenario. We use these permutations in the design of our AAPC strategies.

4.2.1 Prefix-Send Strategy

In this strategy each node exchanges its message with its prefix neighbor $p \oplus i$, in the i^{th} step. This strategy requires that the total number of nodes is a power of two. Equation 4.1 shows the cost of the prefix-send strategy.

$$T_{Prefix} = (P-1)(\alpha + m\beta) + C_{Prefix} \tag{4.1}$$

From Lemma 5, the prefix-send strategy is congestion free. For QsNet, C_{Prefix}, which is the network contention in prefix-send, is actually insignificant because messages are being sent from main memory.

4.2.2 Cyclic Send

In the cyclic send approach, each processor sends a messages in the cyclic-shift order. In stage i each processor p sends a message to the processor $(p + k) \bmod P$. Since all messages are sent from main memory, network contention has no significant effect with QsNet. For other networks this strategy could lead to heavy network contention. The cost of AAPC with the *Cyclic Send* strategy is given by equation 4.2.

$$T_{Cyclic} = (P-1)(\alpha + m\beta) + C_{Cyclic} \tag{4.2}$$

As both these strategies have been presented in literature before [22, 55], we just present them here as a reference. We now present the more interesting problem of all-to-all multicast, where our schemes significantly improve performance on QsNet.

4.3 All-to-All Multicast

With AAM messages can be copied into the network interface, resulting in a much higher node bandwidth. However as described in Section 4.1, there are two other bottlenecks: (i) Network contention, (ii) Lowered throughput to distant nodes due to wire/switch delays.

Our direct AAM strategies handle both these issues. Network contention can be avoided by using prefix-send and cyclic-shift permutations with values of k satisfying Lemma 6. All-to-all multicast can implemented by P-1 such permutations.

Loss of network throughput to far-away nodes is addressed by the k-prefix strategy (Section 4.3.3). This strategy minimizes data exchange with far away nodes, enabling it to scale to 256 nodes.

Another issue relevant to messages being sent from Elan memory is that of contiguous nodes. Lemma 5 does not hold in the presence of missing nodes in the system. The k-shift strategy tries to optimize this scenario.

4.3.1 Ring Strategy

In this strategy, messages are sent along a ring formed by all the nodes in the system. In every stage of the ring strategy, node p receives a message and forwards that message to its neighbor $((p+1) \bmod P)$ in the ring. This strategy is the same as *cyclic-shift-by-1* operation, repeated P-1 times. So by Lemma 1 it is congestion free. The cost of the ring strategy is given by

$$T_{ring} = (P-1)(\alpha + m\beta) + C_{ring} \tag{4.3}$$

Even though the ring strategy is congestion free, it cannot take advantage of lower network transmission time β_{em}. This is because in each iteration every node sends a different message. Hence the ring strategy is obviously worse than the other strategies. However, we introduce it as a background for the k-prefix strategy (Section 4.3.3).

4.3.2 Prefix-Send Strategy

The prefix-send strategy for AAM is the same as prefix-send with AAPC, except that messages are sent from Elan memory. The cost of prefix-send with AAM is shown by Equation 4.4, where the cost of copying

the message into Elan memory is also included.

$$T_{Prefix} = (P-1)(\alpha + m\beta_{em}) + m\delta + C_{Prefix} \tag{4.4}$$

Prefix-Send strategy has two main disadvantages, (i) it forces the number of nodes P to be a power of 2, (ii) it sends data to distant nodes. As mentioned earlier, wire length delays would limit the throughput of the prefix-send strategy. The k-prefix strategy has been designed to address both these problems.

4.3.3 k-Prefix Strategy

The k-prefix strategy is a hybrid of the *ring* and the *prefix-send* strategies. Here, k is a power of two and P is a multiple of k. We divide the fat-tree into partitions of size k. Prefix-send is used to send multicast messages within the partition, while ring strategy is used to exchange messages between neighbor partitions. Each node in the partition is involved in a different ring across all the partitions.

In the first $k-1$ phases, each node exchanges its message with its $k-1$ prefix neighbors within the partition. So, in phase i (where $0 \leq i \leq k-1$) node p exchanges a message with the node $p \oplus (i+1)$. In the k^{th} phase, node p sends a message to the node $(p+k) \bmod P$ forming the ring across partitions.

In the next iteration, node p multicasts the message it received from node $p-k$ in the previous iteration, to the same k neighbors. This is repeated P/k times, until all the messages have been exchanged.

By Lemma 5 the first $k-1$ phases are congestion free. Since k is a power of two, by Lemma 6 the last phase is also congestion free. Hence, k-prefix is congestion free. The cost of the k-Prefix strategy is given by equation 4.5.

$$T_{k-Prefix} = (P-1)(\alpha + m\beta_{em}) + (P/k)m\delta + C_{k-Prefix} \tag{4.5}$$

In $k-1$ out of k phases of this strategy, messages are sent to nearby nodes (at most k away). This makes the k-prefix strategy have a high throughput on a large number of nodes.

4.3.4 k-Shift Strategy

Both prefix-send and k-prefix are very sensitive to missing nodes in the system. On Lemieux, it is often hard to find contiguous nodes. Moreover, since the Elan hardware skips these missing nodes while assigning

processor ids to the programs running on the nodes, it confuses the optimization strategies affecting performance. We now describe the k-shift strategy which performs better in the presence of missing nodes in the system.

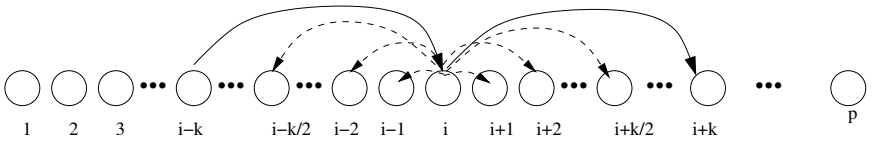


Figure 4.5: K-Shift Strategy

The k-shift strategy takes advantage of Lemma 6. In k-shift strategy each node p sends messages to k nodes $\{(p - \lceil (k-1)/2 \rceil, ..., p-2, p-1, p+1, p+2, ..., p + \lfloor (k-1)/2 \rfloor), p+k\}$. Each message that node p gets from node $p-k$, can be copied into its NIC before it is sent to the k neighbors. This is repeated for P/k iterations to complete the collective operation.

For $k = \{1, 2, 3, 4, 8\}$ we get a contention free schedule if P is a multiple of k. Other values of k will have contention in some or all stages of the strategy. Figure 4.5 shows the k-shift communication schedule. The cost of k-shift strategy is shown by the following equation.

$$T_{k-Shift} = (P-1)(\alpha + m\beta_{em}) + (P/k)m\delta + C_{k-Shift} \tag{4.6}$$

The equation also includes the cost of copying the message into the Elan NIC. Due to this additional overhead, larger values of k would have a better performance. So we use $k = 8$ in all our performance runs.

In the k-shift strategy, most messages are sent to successive nodes. So having a few non-contiguous nodes, results in network contention only in some (not all) phases of the strategy.

4.4 Performance

In this section, we only present the performance results of the AAM direct strategies and they are shown in Figures 4.6 and 4.7. The k-prefix strategy shows the best performance overall. For messages larger than 40KB, k-prefix performs two times better than Lemieux MPI. The ring strategy, which only sends messages from main memory, has performance very similar to that of MPI. (Performance results presented in Chapter 3 indicated that combining strategies perform better than direct strategies for messages smaller

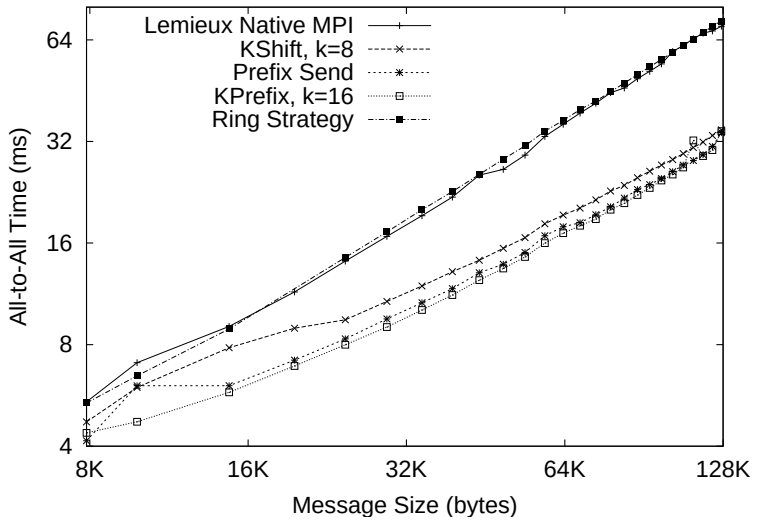


Figure 4.6: AAM Performance(ms) for large messages on 64 nodes

than 8KB. Therefore, the performance of direct strategies has only been shown for message sizes greater than 8KB.)

To keep the nodes synchronized during the collective multicast operation, we have inserted global barriers in our direct strategies after every message is sent. However, these barriers make the computation overhead the same as the completion time. But, k-shift and k-prefix can be altered to perform barriers after k messages have been sent and received. The altered strategies return control to the Charm++ scheduler after sending k messages. The scheduler can schedule useful computation until k messages have been received from the node's neighbors. Control is returned to the strategy, which first performs a barrier and then executes its next step. The performance of the altered k-prefix strategy is shown by *kprefix-lb* in figures 4.8 and 4.9. Here *lb* represents *less barriers*. This modification causes a drop in performance, because nodes are not completely synchronized any more. But the computation overhead is improved a lot. On 64 nodes, the performance of kprefix-lb is comparable to that of k-prefix. On 128 nodes this performance is worse than k-prefix, but still better than the MPI.

Table 4.1 shows the bandwidth of the collective multicast operation for 256 KB messages. In the absence of missing nodes, k-prefix scales best among all the strategies, with an effective bandwidth of 511MB/s on 256 nodes. As mentioned earlier, lower throughput to distant nodes results in k-shift and prefix-send not performing as well as k-prefix. But, in the presence of missing nodes k-shift performs best. This is because

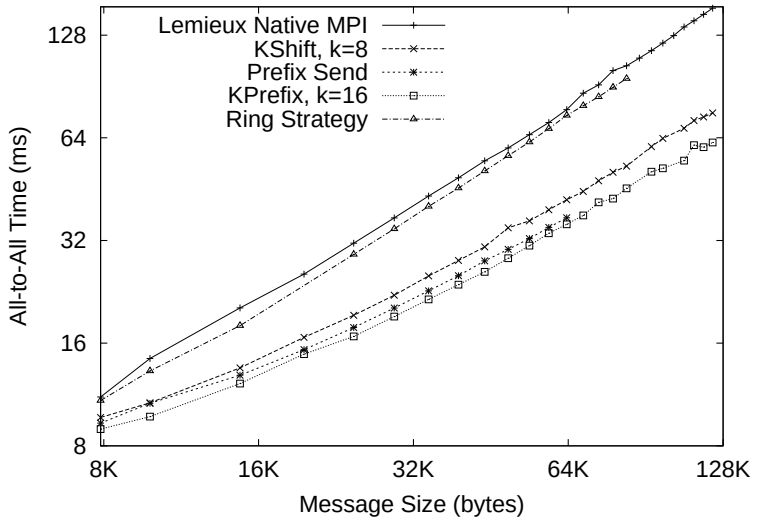


Figure 4.7: AAM Performance(ms) for large messages on 128 nodes

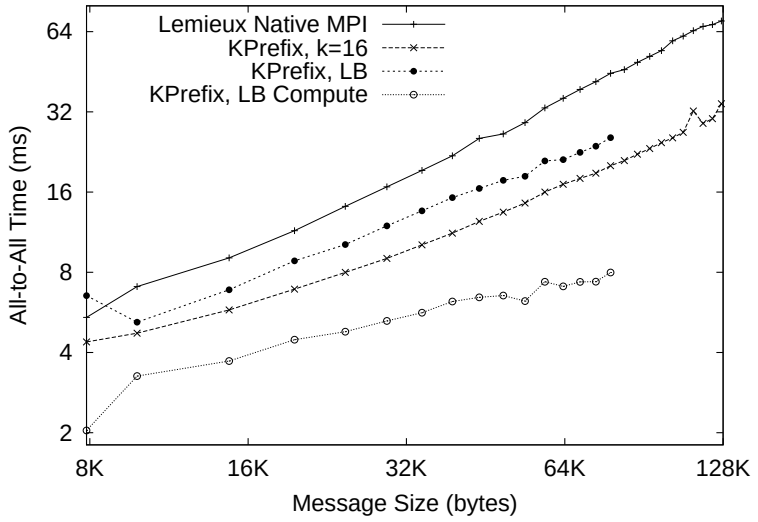


Figure 4.8: CPU Overhead Vs Completion Time (ms) on 64 nodes

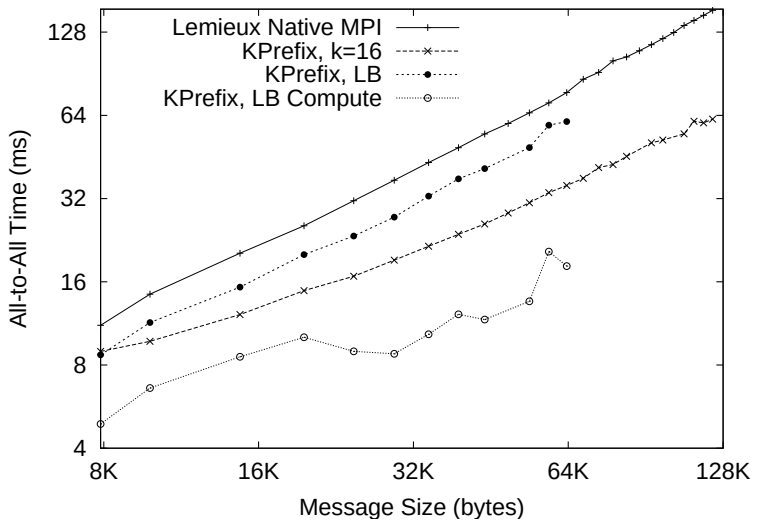


Figure 4.9: CPU Overhead Vs Completion Time (ms) on 128 nodes

Nodes	Native MPI	k-Shift	k-Prefix	Prefix-Send
64	225	507	531	520
128	198	432	519	428
144	187	433	521	-
192	-	416	516	-
256	190	405	511	429
128 (1 missing node)	143	392	338	316
128 (2 missing nodes)	112	399	373	-
240 (1 missing node)	138	394	346	-

Table 4.1: All-to-all multicast effective bandwidth (MB/s) per node for 256 KB messages

missing nodes cause only some (not all) phases of the k-shift strategy to have congestion, resulting in better performance. On Lemieux, a large number of contiguous nodes are often hard to find. The k-shift strategy can be used in such a scenario.

4.5 Related Work

Much of the related work for all-to-all communication with large messages has been specific to architectures. All-to-all communication on 2d Meshes, Tori and Hypercube architectures have been studied in [74, 73, 39, 61, 7, 55, 26, 15, 16, 62, 58]. All-to-all multicast on cluster of workstations is presented in [25]. The LS strategy presented in [25] is best suited for small clusters of work stations. The paper also presents the ring

algorithm which is also analyzed by us.

Contention free permutations on fat-tree networks are also presented in [53, 52]. Prefix-send is presented in [22, 55] as a solution for all-to-all personalized communication. The analysis presented in [22] requires that the entire fat tree is available for the application, forcing P to be a power of 2. In a large machine like Lemieux, several nodes are often down and powers of two nodes may not be available. Hence such restrictions may be hard to meet.

In contrast, our all-to-all communication strategies do not restrict the number of nodes to be powers. We present two new strategies, k-prefix and k-shift, to optimize collective multicast on fat-tree networks. The strategy k-prefix scales well to a large number of nodes as it minimizes data exchange with far-away nodes, while the performance of k-shift is not affected much by missing nodes in the network.

Moreover, our AAM strategies take advantage of the higher bandwidth available in QsNet by sending messages from Elan memory. Finally, we also analyze collective communication from the point of view of *completion time* and *computation overhead*. We found that the kprefix-lb strategy has a low CPU overhead and is suitable for applications that can overlap the all-to-all multicast with computation, and the completion time of the AAM does not affect the critical path.

Chapter 5

Charm++ and Processor Virtualization

Charm++ [29] is a parallel programming platform that supports processor virtualization. In Charm++, the application is divided into a large number of chunks (user-level threads [23] or C++ objects) which act as the *virtual processors*. The Charm runtime system maps these VPs to physical processors, relieving the user from the burden of task assignment to physical processors. The main advantage of processor virtualization is that while one VP is waiting for replies to its messages, other VPs can execute. Adaptive overlap of communication and computation is now implicit, enabling applications to scale to a large number of processors. In fact, processor virtualization is ideally suited for interconnects with co-processors, as demonstrated in Section 5.5.

Powered by processor virtualization, the Charm++ runtime system supports dynamic load-balancing [27] by migrating VPs, automatic checkpointing [75] and fault-tolerance [8], out-of-core execution and the ability to change the number of processors used by the application [33].

The first few chapters of this thesis have presented processor-level communication optimizations, but the rest of the thesis explores communication optimization in the presence of processor virtualization. Processor virtualization introduces a new dimension to communication optimization, allowing object-level communication to be optimized in addition to processor-level communication.

Objects in Charm++ can migrate between processors, and so optimization schemes have to be aware of this migration. Both the streaming optimization presented in this Chapter and the all-to-all communication optimizations presented in the next Chapter support dynamic migrating objects.

We begin with a brief overview of the Charm++ language and runtime. (Details of writing Charm++ programs can be found in the Charm++ manual [14]).

5.1 Charm++ Basics

In a Charm++ program, the user partitions work into Chares which are the virtual processors in the program. The user writes an interface (.ci) file, where he defines the Chares and entry functions for them. These entry methods can be invoked remotely on the Chares through *proxies*, which are the Charm++ equivalents of stubs in CORBA. The .ci file is used to create .decl.h and .def.h files. These header files have generated code for the chare proxies, similar to the way CORBA generates stub files from the IDL. Unlike CORBA, Charm++ is asynchronous and the proxies are normally used to send messages to the chares.

The Charm++ language has several chare constructs, of which *chare-arrays* are the most widely used. The next section describes chare-arrays.

5.2 Chare Arrays

Chare Array [40] objects enable processor virtualization in Charm++. Array elements are regular C++ objects scattered across all the processors. They can dynamically migrate between processors at *any time* during the program, and not just at synchronized steps. Array objects are addressed by the Array ID and the object index in that array. The Charm++ language has constructs for one dimensional arrays, two dimensional arrays and can also support any user specified element index. The objects are accessed through an array proxy and the index of the element in the array.

The Charm++ array manager maintains a hash-table that maps the array index to the processor on which that element resides. As array objects can migrate between processors, this hashtable points to the last known processor of the object. Load-balancing is achieved by migrating heavily-loaded array objects to lightly-loaded processors.

The array manager supports basic collective operations on arrays, which have been proved to work efficiently in the presence of migrations [40]. The Charm runtime also supports collective operations on subsets of arrays through array sections (developed in collaboration with Gengbin Zheng). For example, we can create an array section proxy and multicast a message to a subset of the array.

The TCharm (threaded-charm) runtime can bind array objects to user level threads, which can also migrate between processors like other array objects. Processor virtualization in *Adaptive MPI* is achieved through these user level threads. The effectiveness of AMPI is presented in [23].

5.3 Delegation

Message management libraries can be easily be developed in Charm++ through *delegation* (developed in collaboration with Orion Lawlor). Normally, messages in Charm++ are sent by the runtime, when the user makes an entry function call on a proxy. The runtime marshals the parameters of the function call into a message and sends it to the processor where the destination object lives.

Delegation allows marshaled messages to be passed to a delegation group object. Groups (also known as branch office chares) are Charm++ objects which have one member on every processor. They are ideally suited for developing system libraries. Delegation forwards application messages to the library group object instead of the Charm runtime. To be delegated, the library has to inherit from CkDelegateManager interface and implement the methods defined in it. The class declaration of CkDelegateManager is shown below :

```
class CkDelegateMgr : public IrrGroup {
  public:
    ......................
    virtual void ArraySend(... ,int ep, void *m, CkArrayIndexMax &idx, CkArrayID a);
    virtual void ArrayBroadcast(... , int ep, void *m,CkArrayID a);
    virtual void ArraySectionSend(... , int ep, void *m, CkArrayID a, CkSectionID &s, );
    ........................
};
```

The delegation base class has virtual methods to send point-to-point array messages, broadcasts to the entire array and section sends to send data to a subset of the array. A delegation group could implement any of these methods to receive application messages.

The Charm++ proxy class has a pointer to the CkDelegateManager base class, which can dynamically point to the delegation library child class pointer. To enable delegation, the user has to set up the array proxy to point to the delegated library through the CkDelegateProxy call. Once the call is made All message invocations on the proxy now go to the library and not the Charm++ runtime.

The delegated library can perform optimizations like message combining with other messages etc., before sending data out on the network. The communication framework interfaces with the user application through the delegation interface. As delegation works through inheritance, it has the minimal overhead of

one virtual pointer indirection.

5.4 Optimizing Communication with Chare Arrays

A Charm++ (or Adaptive MPI) program typically has tens of thousands of array objects or threads living on thousands of processors. These objects communicate with each other using point-to-point messages, or participate in collective communication operations. These objects can also freely migrate between processors at will.

The next Chapter presents a communication optimization framework, that optimizes the above mentioned scenario of processor virtualization and dynamic migrating objects. These optimizations are in addition to basic implementations of many collectives provided by the Charm++ array interface. The purpose of this framework is to enable easy development of communication strategies in the Charm++ runtime, and the ability to dynamically switch between the different schemes. The framework has been designed to optimize the following :

Point-to-point communication This can be a source of serious overhead when many array objects send several short messages.

Collective communication collective communication specifically all-to-all affects the scaling of applications to a large number of processors.

Migration When an object migrates, the source processor has to be notified about this migration so that it does not expect messages from that object any more. On the destination processor messages from newly arrived objects also have to be handled. Another way of handling migration would be to forward messages of the migrated object to the processor from where it migrated.

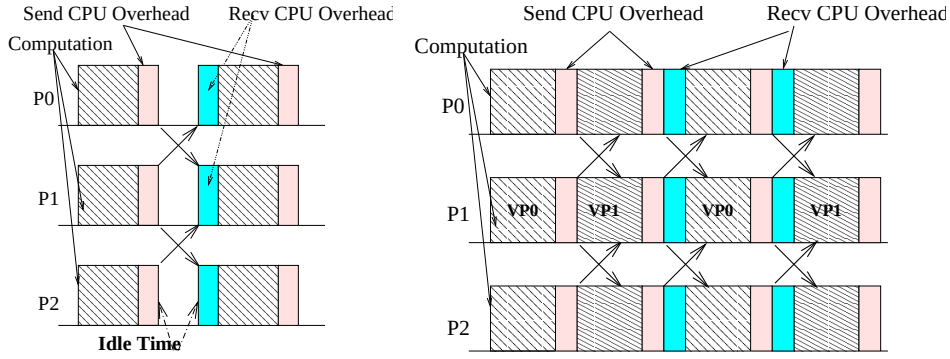
Array Sections The communication framework should support collective operations on subsets of array elements.

5.5 Processor Virtualization with Communication Co-processor

With processor virtualization there are multiple VPs on each processor. Hence, the runtime system can effectively overlap the communication latency of one virtual processor with computation from another VP. The presence of the communication co-processor in modern network interfaces makes this overlap even

more significant, as the CPU overhead of a message send operation a small fraction of the total latency.

For example in Elan, a 4KB message takes $51\ \mu s$ to reach its destination (Table 2.4). However, the CPU overhead is only about $13\ \mu s$, which includes both the send and receive CPU overheads. The remaining time can be used for other computation. The advantage of this overlapping is demonstrated by a 6-point stencil benchmark. Here, each processor (or virtual processor in our case) computes and then communicates (sends and receives) with 6 neighbors, two in each dimension.



(a) Without Virtualization (b) With Virtualization

Figure 5.1: Timeline for the neighbor-exchange pattern

In traditional MPI style of programming, this program could be written as computation followed by communication with no overlapping. The communication operation involves the exchange of 6 messages. Figure 5.1(a) demonstrates this style of programming, though only two messages are shown. Here, the shaded rectangles show computation and the solid rectangles show the CPU overhead of communication. The send CPU overheads are shown by the light Grey sold rectangles while the receive overheads are shown by the dark Grey solid rectangles. Observe that there is idle time between the computations. This scenario does not effectively utilize the low CPU overhead of network interface with co-processors.

In Charm++ with processor virtualization, the above program can have multiple virtual processors on each physical processor. After a virtual processor has sent its messages and is waiting for reply messages, another virtual processor can compute (Figure 5.1(b)). Here each processor has two virtual processors. On processor P1, VP0 computes and sends its messages. While VP0 is waiting for messages from its neighbors (on processors P0 and P1 and possibly others), VP1 can start computing. (As our VPs are user level threads there is only a small context switch overhead). Thus, communication latency is effectively overlapped with

computation. Moreover, by the time VP1 finishes, VP0 has received all its messages (only 2 out of 6 are shown here) enabling it to compute again with no delay or a short delay.

Processors	NVP=1	NVP=8
8	192	105
64	16.4	12.7
512	6.5	5.7

Table 5.1: Time (ms): 3D stencil Computation of size 240^3 on Lemieux

Table 5.1 shows the performance of 3D 6-pt stencil on Lemieux (these results are based on the AMPI [23] framework developed by Chao Huang). Here NVP is the number of virtual processors per real processor. So for 8 processors and an NVP of 8 we have 64 total virtual processors in the program. We ran the program with an NVP of 1 and an NVP of 8, corresponding to the two columns in table 5.1. Observe that NVP=8 performs better, as it makes good use of the communication co-processor. Moreover, notice that the performance gain for NVP=8 drops with the increase in the number of processors. This is because the application becomes more fine grained on a larger number of processors, which leads to shorter messages and less overlap of communication with computation.

5.6 Object-to-Object Communication

In the Charm++ runtime, when a message is received a handler is executed for it. We use *grain-size* to represent the amount of computation that needs to be executed when a message is received. Several parallel applications are fine-grained, i.e. the amount of computation for each message is quite small. Message handlers in such applications may only have few memory accesses, which typically take hundreds of nano-seconds on modern processors. These handlers may also send several short messages, which could further increase the communication overhead of the application. For example, in a distributed memory cache manager, requests for stale memory blocks, block writes and block invalidates involve quick processing and short messages. Parallel network simulation involves processing events which usually move packets through various components of the network. For example, the switch handler processing a packet would lookup a routing table and deposit the packet on the destination port. Another example of such a fine grained parallel application is *union-find* which tries to find the root of the current tree, and hence requires fast remote memory accesses through a series of destinations.

Fine grained communication is expensive when the messages are sent to a remote processors. Even on the fastest of networks, message processing times are of the order of a few microseconds, e.g., the pipelined latency with the Converse runtime in Quadrics Elan3 is about $9.8\mu s$ and $4.8\mu s$ for Infiniband (Chapter 2). When handlers only take hundreds of nanoseconds to finish, the application would spend most of its time in sending and receiving messages. This high communication overhead may affect the scalability of applications to a large number of processors. In this chapter, we present strategies that improve scaling microsecond and sub-microsecond granularity applications in Charm++.

5.7 Streaming Optimization

Our optimization strategies for fine-grained communication take advantage of the fact that much of the overhead in sending short messages is independent of the size of the message. In Charm++, they are mainly from memory allocation, scheduling and network overhead. Much of this overhead can be amortized if several messages are sent together as one message. In this section, we present the effectiveness of message combining to reduce the overhead of sending several short messages.

With a large number objects (VPs) on each processor and each of them sending several short messages, it is possible that several of these messages are destined to the same processor. This fine-grained communication can be treated as a message stream which is optimized through message combining. The streaming messages can be inserted into buckets, based on their destination processors. At the timeout, or when a bucket fills up, or when the source processor goes idle, the messages in a bucket are combined and sent out to the destination processor as one message. We term this scheme as the *streaming* optimization scheme. Streaming optimization ensures that the per-message overheads of message passing are amortized across the several messages that were combined into one. The messaging overhead of Charm++ array messages with and without the streaming optimizations are presented in Table 5.2 with a bucket size of about 500 messages. Observe that streaming results are significantly in lower. The streaming performance presented here, shows the per-message overhead in the Charm runtime that did not get amortized by message combining. Short array message packing, presented in the next section eliminates some of this overhead.

Processor	Interconnect	Charm++ Default	Streaming Performance
3 Ghz Xeon	Myrinet	9.6 us	1.7 us
1.6Ghz IA64	NUMA Connect	5.5 us	2.5 us
1.5Ghz IA64	Myrinet	14.8 us	3.8 us
1 Ghz Alpha	ELAN3	16.2 us	3.2 us
2 Ghz Mach G5	Myrinet	14 us	2.8 us

Table 5.2: Streaming Performance on two processors with bucket size of 500

5.7.1 Short Array Message Packing

Charm++ is a dynamic and message driven runtime system with migratable objects. It has load-balancers, tracing, message priorities and numerous other features built-in. The Charm++ runtime is organized into several layers with each layer adding functionality to the runtime. Object messages pass through these different layers, each adding some overhead to message processing in addition to the low level network processing overheads.

The Charm++ runtime also sends messages in an envelope, which results in a 100 byte header and a footer for prioritized messages. The header stores the destination object id, priority, queuing strategy, destination handler and several other message parameters. For short messages (smaller than a 100 bytes), this can be a significant source of overhead.

In some situations, the information in the envelope is the same across several messages between two objects. This redundant information needs to be sent only once. Therefore, we enhance the performance of the streaming optimization by stripping envelopes for short array messages. The common envelope information is sent once with the combined message.

This scheme also hides other overheads in the Charm runtime: (i) scheduling overhead is minimized by calling the entry methods inline without priorities, and (ii) the processor for the most recent array element is cached, avoiding hash-table overheads when several messages are exchanged between the same pair of objects.

We call the above optimizations as *short message packing*. The performance of streaming with short array message packing is presented in Table 5.3 (Here we have kept the bucket size at 500 to show the maximum performance gains). Short message packing leads to a substantial reduction in message overheads on some platforms. Short message packing substantially reduces the messaging overheads in the Charm++ runtime. The next section presents performance of streaming by varying the bucket size and the number of

Processor	Interconnect	Charm++ Default	Streaming	Short Message Packing
3 Ghz Xeon	Myrinet	9.6 us	1.7 us	1.2 us
1.6Ghz IA64	NUMA Connect	5.5 us	2.5 us	0.95 us
1.5Ghz IA64	Myrinet	14.8 us	3.8 us	2.3 us
1 Ghz Alpha	ELAN3	16.2 us	3.2 us	2.1 us
2 Ghz Mach G5	Myrinet	14 us	2.8 us	2.5 us

Table 5.3: Short message packing performance on various architectures with a bucket size of 500

processors.

5.8 Ring Benchmark

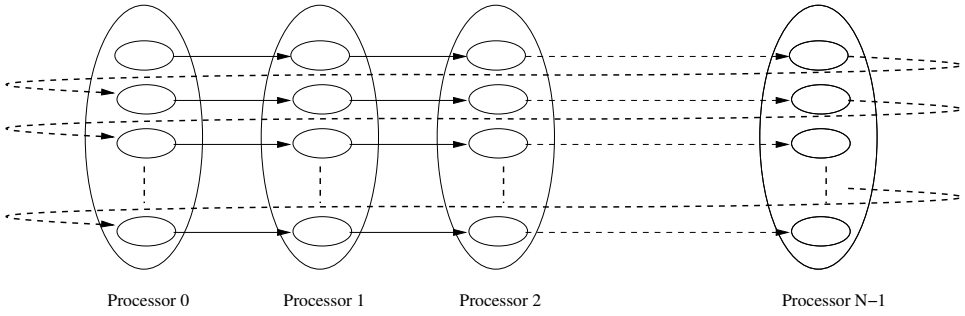


Figure 5.2: The Ring benchmark

In the ring benchmark, array elements simultaneously send messages to their neighbors along a ring. As the array elements are inserted on processors in a round robin order, the processors also send messages along a ring. Each processor sends a different message for each element residing on it. Figure 5.2 shows a schematic description of the ring benchmark.

The effectiveness of the streaming optimizations is shown through the ring benchmark. The performance of streaming with the ring benchmark on NCSA Tungsten [68] is shown in Tables 5.4, 5.5, 5.6 and 5.7. Observe that streaming performance is not so good for bucket sizes of 5 and below. Mesh-streaming presented in the next section addresses this problem.

Processors	Charm++ default	Streaming	Short Message Packing
2	9.4 us	7.7 us	6.3 us
4	10.0us	10.9us	9.4 us
16	17.8us	11.4us	10.1us
64	27.7	- us	12.1us

Table 5.4: Ring benchmark performance with a bucket size of 1 on NCSA Tungsten Xeon cluster

Processors	Charm++ default	Streaming	Short Message Packing
2	9.4 us	3.5 us	2.4 us
4	16.8us	4.0 us	3.5 us
16	17.1	4.2 us	3.4 us
64	20.1	4.8 us	4.2 us

Table 5.5: Ring benchmark performance with a bucket size of 5 on NCSA Tungsten Xeon cluster

Processors	Charm++ default	Streaming	Short Message Packing
2	9.3 us	1.9 us	1.4 us
4	17 us	1.9 us	1.3 us
16	17 us	2.0 us	1.4 us
64	17 us	2.6 us	1.5 us

Table 5.6: Ring benchmark performance with a bucket size of 50 on NCSA Tungsten Xeon cluster

Processors	Charm++ default	Streaming	Short Message Packing
2	9.4us	1.7 us	1.2 us
4	16.9us	1.8 us	1.2 us
16	17.1	1.9 us	1.3 us
64	17.4	1.9 us	1.3 us

Table 5.7: Ring benchmark performance with a bucket size of 500 on NCSA Tungsten Xeon cluster

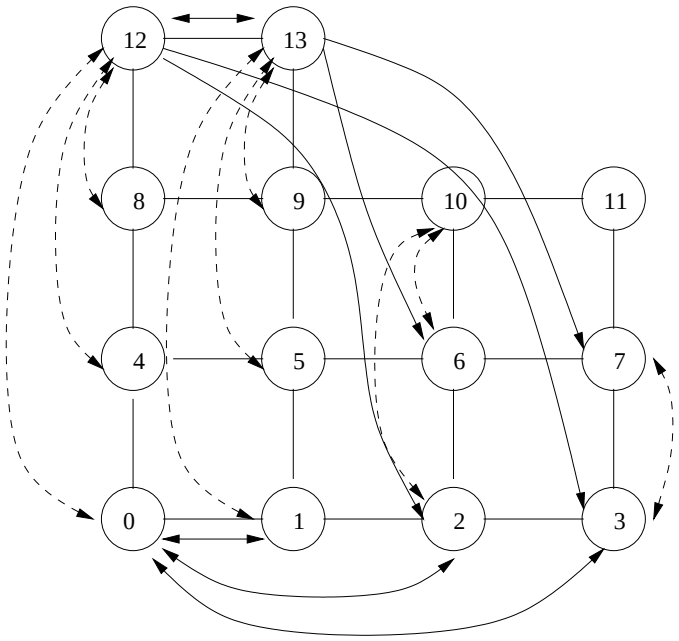


Figure 5.3: 2D Mesh virtual topology

5.9 Mesh Streaming

Streaming strategy requires that several messages be sent between pairs of processors so that per-message overheads are amortized. Often objects send out several messages, but these messages are spread out to many processors. Can we optimize this scenario, where only a few messages are exchanged between pairs of processors?

We can use message combining along virtual topologies (Chapter 3) to optimize streaming communication. On a virtual topology, several processor's messages are sent to an intermediate processor from where they are routed to their destinations. Hence more messages can be combined and a fixed sized bucket would get filled earlier. For example, with the 2-D Mesh virtual topology messages destined to $\sqrt{P}$ processors are combined and sent to each row neighbor in the first phase. In the second phase, these messages will be routed to their correct destinations. Since messages are short, the cost of sending them twice is not significant. But the amortized cost of sending messages is lower because more messages can be combined together. (*Mesh Streaming* was developed in collaboration with Greg Koenig from the Parallel Programming Laboratory.) Mesh-streaming uses the scheme presented in Section 3.1 for irregular meshes (Figure 5.3).

The performance of mesh streaming with the ring benchmark is presented in Table 5.8. In the ring

Processor	Interconnect	Charm++ default	Streaming	Mesh
3 Ghz Xeon	Myrinet	9.6 us	1.7 us	1.9 us
1.6Ghz IA64	NUMA Connect	5.5 us	2.5 us	2.7 us
1.5Ghz IA64	Myrinet	14.8 us	3.8 us	4.1 us
1 Ghz Alpha	ELAN3	16.2 us	3.2 us	3.0 us
2 Ghz Mach G5	Myrinet	14 us	2.8 us	2.9 us

Table 5.8: Mesh-Streaming performance comparison with a short message and a bucket size of 500 on 2 processors

Processors	Messages per processor pair	Charm++	Streaming	Mesh Streaming
4	1	10.1 us	11.3 us	14.1 us
4	2	9.4 us	6.5 us	12.2 us
4	4	9.4 us	4.1 us	7.1 us
4	10	11.5 us	2.8 us	4.6 us
4	50	14.1 us	2.1 us	3.0 us
16	1	13.6 us	13.4 us	11.8 us
16	2	13.6 us	7.6 us	8.3 us
16	4	14.3 us	4.6 us	5.5 us
16	10	23.2 us	3.1 us	5.1 us
16	50	26.3 us	2.6 us	4.3 us
64	1	13.7 us	18.1 us	8.8 us
64	2	13.8 us	8.7 us	6.2 us
64	4	13.9 us	5.3 us	4.6 us
64	10	15.6 us	3.3 us	4.2 us
64	50	14.6 us	2.4 us	3.6 us

Table 5.9: Performance of all-to-all benchmark with a short message on PSC Lemieux

benchmark, each processor sends messages to only one other processor. But still mesh-streaming performance is similar to other streaming optimizations. In the next section, we show that mesh performs better the other streaming schemes in the scenario where each object sends messages to objects residing on several processors.

5.10 All-to-all benchmark

To demonstrate the performance gains on mesh-streaming, we ran a benchmark where each array object sends k point-to-point messages to every other array object. Even here, array elements are arranged in a round robin order. So for streaming, k becomes the natural bucket size as processor pairs would exchange k or more messages, depending on number of objects on each processor. Table 5.9 shows the performance

of streaming and mesh-streaming optimizations respectively for the all-to-all benchmark. Here both the schemes pack short messages.

Mesh-streaming is indeed more effective when a small number of messages are exchanged between processors. As shown in table 5.9, on 64 processors mesh-streaming has the best performance with bucket sizes less than 10.

Chapter 6

Communication Optimization Framework

We have presented several communication optimization schemes in the previous chapters. Processor level strategies are presented in Chapters 3 and 4, and object-based optimizations are presented in Chapter 5.6. These schemes optimize both collective communication and point-to-point communication.

Applications developed in Charm++ or Adaptive MPI should be able to take advantage of the above optimizations easily. The *Communication Optimization Framework* presents an interface for applications to elegantly use the optimization schemes presented in this thesis. The framework is general and can support a variety of optimizations. Some of the communication **operations** currently supported are *all-to-all* communication, *broadcast*, *section multicast* and *streaming*. Optimizations for these operations are implemented as *strategies* in the framework. A **strategy** is as an optimization algorithm for a communication operation. As Charm++ and AMPI applications are developed with dynamic migrating objects, both processor-level and object-level optimizations are supported in the framework, with migration support in the object-level strategies.

It can be a tedious task for the programmer to choose the strategies for his application's communication operations. To save the programmer from this burden, the framework can dynamically choose the best strategy for a communication operation. As many scientific applications exhibit the *principle of persistence* [32], the characteristics of a communication operation can be observed and later used to choose a strategy from a list of applicable strategies for that operation. The details of this novel feature are presented in Section 6.4.

A block diagram of the Charm runtime system with the communication framework is shown by Figure 6.1. The communication framework functionally operates at a level between Charm++ and Converse (which is the light-weight message passing library [28]). While it is fully aware of Chare-arrays and other Charm++ constructs, it uses Converse messages to communicate. This design choice was made to enable

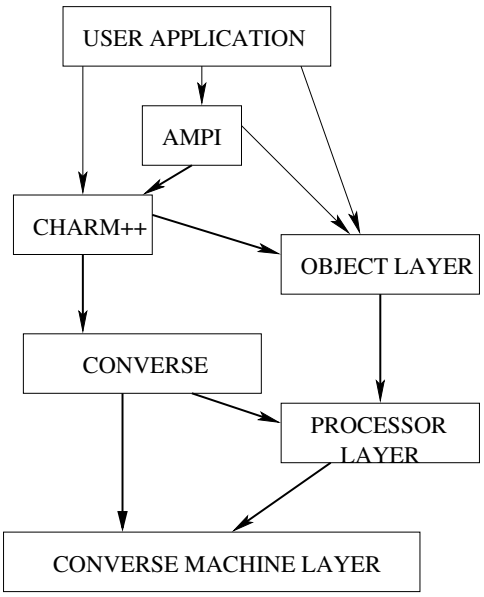


Figure 6.1: The Communication Optimization Framework

the communication optimizations to use the light-weight and relatively low-latency Converse runtime. At the end-points, after the all intermediate messages have been processed, the Charm++ object entry methods are invoked by the communication library.

To elegantly support both object and processor optimizations, the framework has two layers, with the first layer performing object-level optimizations, while the second layer is for processor-level optimizations. The object layer of the communication library calls the processor layer for inter-processor communication. The object layer can combine several messages destined to objects that reside on the same processor, thus reducing the α cost of the communication operation. The processor layer can then perform other optimizations like sending messages along a virtual topology.

Charm++ and Converse programs can easily access the communication framework (also called Communication Library). MPI programs can use these optimizations through Adaptive MPI [23]. Several of the MPI calls in Adaptive MPI use strategies in the communication library.

The communication framework has been designed with several interacting modules in the C++ programming language. The class hierarchy of the framework is presented in Figure 6.2. There are two manager classes which coordinate the strategies across processors, the *ComlibManager* class for object-level coordination and the *ConvComlibManager* class to manage the processor-level functionality of the communication framework.

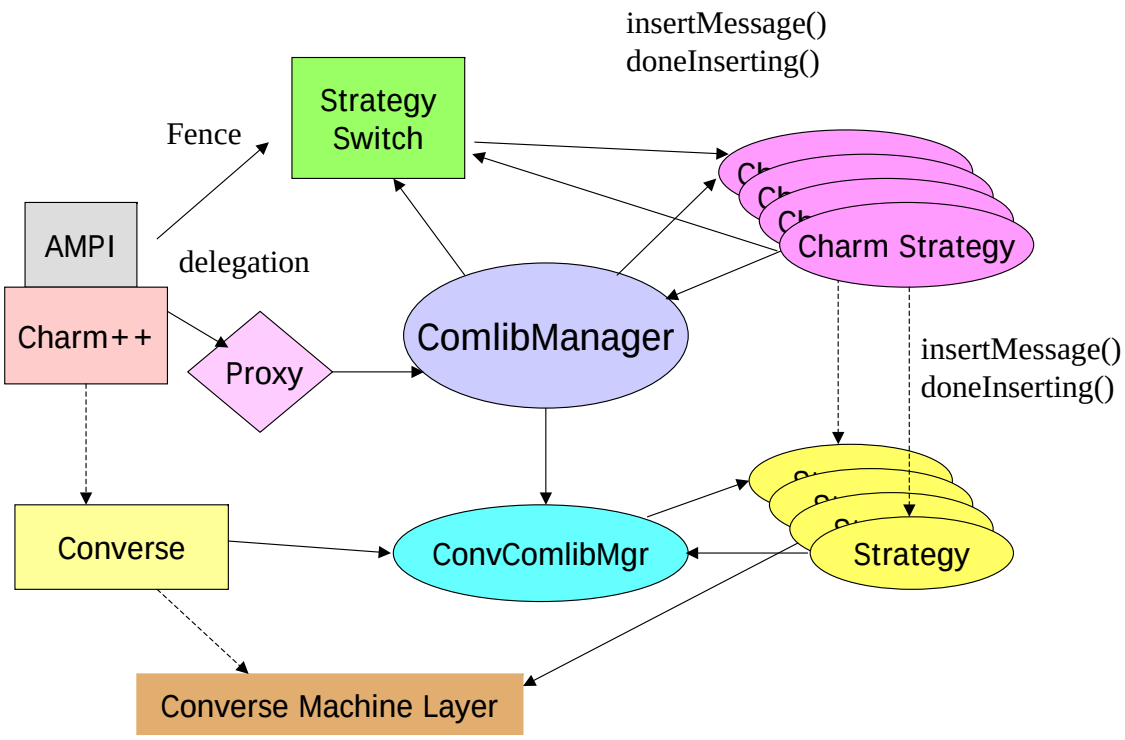


Figure 6.2: Class Hierarchy

The *delegation framework* in the Charm runtime (Chapter 5.3) redirects application messages to the communication library. These messages are then passed to the strategy which calls Converse routines to send the messages to their destination processors.

There are two levels in the strategy classes too: subclasses of *Strategy* optimize processor-level communication, while subclasses of *CharmStrategy* optimize object-level communication. The object-level strategies can call one or more processor-level strategies to optimize processor-to-processor communication.

Application communication patterns, referred to as **calls**, are delegated to **instances** of different object level strategies. A communication **call** should be distinguished from an **operation**, as it depends on where in the application code the operation occurs. For example, an array object can participate in multiple all-to-all operations, leading to several calls to the all-to-all operation. Each of these calls may use to a different

strategy instance. Strategy instance pointers corresponding to the different calls are stored in the instance-table on all the processors. In Figure 6.2, several instances of CharmStrategy are shown which invoke one or more instances of processor level strategies.

A call *c* is associated with both a strategy instance *si* and a proxy *p*. This association is set up by the programmer during startup. All entry method invocations on *p* will be passed to *si*. The proxy *p* also stores the index of *si* in the instance table, which lets the communication framework pass the message to the correct strategy instance (*si*).

The *StrategySwitch* class chooses the best strategy for a communication operation using dynamic application statistics. It can choose both Charm++ and Converse level strategies, or a combination of both. The StrategySwitch modules (discussed in detail in Section 6.4) should be designed along with the strategies they can switch. These strategies should also register the same StrategySwitch class with the communication framework at startup. During the next fence step, the StrategySwitch will choose the best strategy for that call.

We now begin a more detailed description of the communication framework by presenting the strategy module.

6.1 Communication Optimization Strategy

Optimization algorithms are implemented as Strategies in the communication library. Strategies can be implemented at the Object (Charm++) level or the processor (Converse) level. Code reuse is possible by having a few object managers perform object level optimizations and then call several other processor level optimization schemes. For example, to optimize all-to-all communication the processor level strategies could use the different virtual topologies presented in Chapter 3.

All processor (Converse) level strategies inherit from the *class Strategy* defined below and override its virtual methods.

```
//Converse or Processor level strategy
class Strategy : public PUP::able{
public:
    //Called for each message
```

```
    virtual void insertMessage(MessageHolder *msg);
    //Called after all chares and groups have finished depositing their
    //messages on that processor.
    virtual void doneInserting();
    virtual void beginProcessing(int nelements);
};
```

The class method *insertMessage* is called to deposit messages with the strategy. MessageHolder is a wrapper for converse messages. When a processor has sent all its messages, *doneInserting* is invoked on the strategy.

```
//Charm++ or Object level strategy
class CharmStrategy : public Strategy{
 protected:
    int isArray;
    int isGroup;
    int isStrategyBracketed;
    ............
    ............
public:
    //Called for each message
    virtual void insertMessage(CharmMessageHolder *msg);
    //Called after all chares and groups have finished depositing their
    //messages on that processor.
    virtual void doneInserting();
    virtual void beginProcessing(int nelements);
};
```

Charm++ level strategies also have to implement the insertMessage and doneInserting methods. Here insertMessage takes a CharmMessageHolder which is a Charm++ message wrapper. The call to beginProcessing initializes the strategies on each processor. This additional call is needed because the constructor

of the strategy is called by user code in main::main on processor 0. Along with initializing its data, beginProcessing can also register message handlers, as the communication library strategies use Converse to communicate between processors. The flags *isArray* and *isGroup* store the type of objects that call the strategy and the flag *isStrategyBracketed* flag specifies if the CharmStrategy is bracketed or not. Bracketed strategies require that the application deposits messages in brackets demarcated by the calls ComlibBeginIteration and ComlibEndIteration. Bracketed strategies are discussed in detail in Section 6.2.1.

6.2 Supported Operations and Strategies

The communication framework currently supports four different communication operations namely, (i) many-to-many communication, (ii) broadcast, (iii) section multicast, (iv) streaming. Table 6.1 shows the different strategies that optimize these communication operations. Some of these are converse strategies while others are object strategies. We now present in detail the strategies optimizing the above mentioned operations.

Operation	Object Strategy	Processor Strategy
Many-to-many personalized	EachToManyStrategy	Mesh, Grid, Hypercube, Direct
Many-to-many multicast	EachToManyMulticastStrategy	Mesh, Grid, Hypercube, Direct
Broadcast	BroadcastStrategy	Binomial tree, Binary tree
Section Multicast	DirectSection, RingSection, TreeSection	
Streaming	Streaming, MeshStreaming, PrioStreaming	

Table 6.1: Communication Operations supported in the Framework

6.2.1 EachToManyStrategy

The class *EachToManyStrategy* optimizes all-to-all personalized communication using virtual topologies described in Chapter 3.1. The topologies 2-D Mesh, 3-D Mesh and Hypercube have been implemented. EachToManyStrategy manages the object level communication by fist combining all object messages being sent to the same processor into one message and then calling the routers to optimize processor-to-processor communication. Different virtual topologies have been implemented as Converse *routers*. EachToManyStrategy can be initialized to chose one such topology. For example, with the mesh router, the strategy on each processor first sends messages to its row neighbors. After having received its row messages each processor sends the column messages. After having received the column messages an iteration of the strategy finishes.

All local messages are delivered as soon they are received. EachToManyMulticastStrategy is a variant of the EachToManyStrategy that can multicast messages to arrays using virtual topologies. It uses multicast routers for processor communication.

EachToManyStrategy requires that all local messages have been deposited before they can be packed into row and column messages. Hence it needs to be a *bracketed* strategy. Bracketed strategies require each of the participating objects to deposit their intended messages within brackets. Calls to *ComlibBeginIteration* and *ComlibEndIteration* create a bracket. The call ComlibBeginIteration sets up the delegation framework to forward user messages to the correct strategy instance. User messages then get passed to the insertMessage entry function of the strategy. When all local objects have called ComlibEndIteration, doneInserting is invoked on the strategy.

Bracketed strategies are typically needed when the communication optimization requires local source objects to reach a barrier. At this local barrier the communication framework invokes doneInserting on that strategy, which the calls the converse level strategy.

Non-bracketed strategies have no such restriction. They process messages as soon as they arrive. so, non-bracketed strategies should not expect a doneInserting to be invoked on them. They must all process messages in the insertMessage call itself.

6.2.2 Streaming Strategy

This strategy optimizes the scenario where objects sends several small messages to other objects. The StreamingStrategy collects messages destined to the same processor after a timeout or when certain number of messages have been deposited. These messages are combined and sent as one message to that destination, thus sending fewer messages of larger sizes. The timeout is a floating-point parameter to the StreamingStrategy. It needs to be specified in milliseconds, with a default of 1ms. Micro-second timeouts can also be specified by passing values less than 1. For example, 0.1 represents $100\mu s$.

The Streaming Strategy by default is a non-bracketed strategy. *Non-bracketed* strategies do not require the objects to call beginIteration and endIteration. Such strategies do not have to wait for all local messages, before processing those messages. As messages may wait for timeout potentially leading to loss of throughput, the streaming strategy also has a bracketed variant which flushes buckets on the endIteration call.

6.2.3 Section Multicast and Broadcast Strategies

The direct multicast strategies can multicast a message to the entire array or a section of array elements. The direct multicast strategies are non-bracketed, and the message is processed when the application deposits is. These strategies do not combine messages, but they may sequence the destinations of the multicast to minimize contention on a network. For example, the RingMulticastStrategy sends the messages along ring resulting in good throughput as the ring permutation is contention free on many communication topologies (Chapter 4).

For section multicast, the user must create a section proxy and delegate it to the communication library. Invocations on section proxies are passed on to the section multicast strategy.

6.3 Accessing the Communication Library

Users of Charm++ and Adaptive MPI can access the communication library to improve the performance of their applications. With Adaptive MPI (AMPI) the communication framework is accessed transparently. AMPI automatically initializes several strategies to optimize several MPI calls like MPI_Alltoall, MPI_Allgather etc. In Chapters 3 and 4, we show that the CPU overhead of an all-to-all operation is much smaller than its completion time. We have hence provided asynchronous collective communication extensions in AMPI through the MPI_Ialltoall call, which lets the application compute while the all-to-all is in progress. The EachToManyStrategy is called by AMPI for MPI_Alltoall and the MPI_Ialltoall calls.

In Charm++, however, the user must create the strategies in the program explicitly. Charm++ programs are normally based on communicating arrays of chares [40], that compute and then invoke entry methods on local or remote chares by sending them messages. These array elements send messages to each other through proxies. The messages are passed to the Charm++ runtime which calls lower level network APIs to communicate. To optimize communication in Charm++, the user can redirect a communication *call* to go through an instance of a strategy.

To access the communication framework, the user first creates and initializes a communication library strategy. He then needs to make a copy of the array proxy and associate it with that strategy. The user can create several instances of the same strategy, to optimize different communication calls in his application. Each communication operation is now associated with a proxy. The exact sequence of calls is shown below.

```
//In main::main()
//Create the array
aproxy = CProxy_Hello::ckNew();

Strategy *strategy = new EachToManyStrategy(USE_MESH, srcarray, destarray);
//Register the strategy
ComlibAssociateProxy(aproxy, strategy);



//Within the array object
//First proxy should be delegated
ComlibBeginIteration(aproxy);
aproxy[index].entry(message); //Sending a message
.....     //sending more messages
.....
ComlibEndIteration(aproxy);
```

The above example shows the EachToManyStrategy. Notice the ComlibBeginIteration and the ComlibEndIteration calls, which demarcate the bracket. After main::main, the Communication Framework broadcasts the strategies along with the data passed to them from the user. On each processor a *beginProcessing* is called to initialize the strategies, after which messages are passed to the strategy.

6.4 Adaptive Strategy Switching

Many scientific applications tend to be iterative, strongly exhibiting the principle of persistence. This suggests that the communication patterns of such applications can be learned dynamically at runtime. Self tuning schemes have been presented in the past [24, 2], for example Atlas [24] tunes the numerical algorithms to a specific hardware at compile time. Atlas is a linear algebra system, that chooses algorithms based on the hardware properties like cache size, processor speed etc. The choice is made through several tuning benchmark runs which are an input to the compiler, which selects the the best algorithm.

The communication framework, however, chooses strategies at runtime based on dynamic application statistics it collects. The communication characteristics of the application are searched in a database of patterns and the best strategy is obtained. The framework can then switch the application to use this strategy.

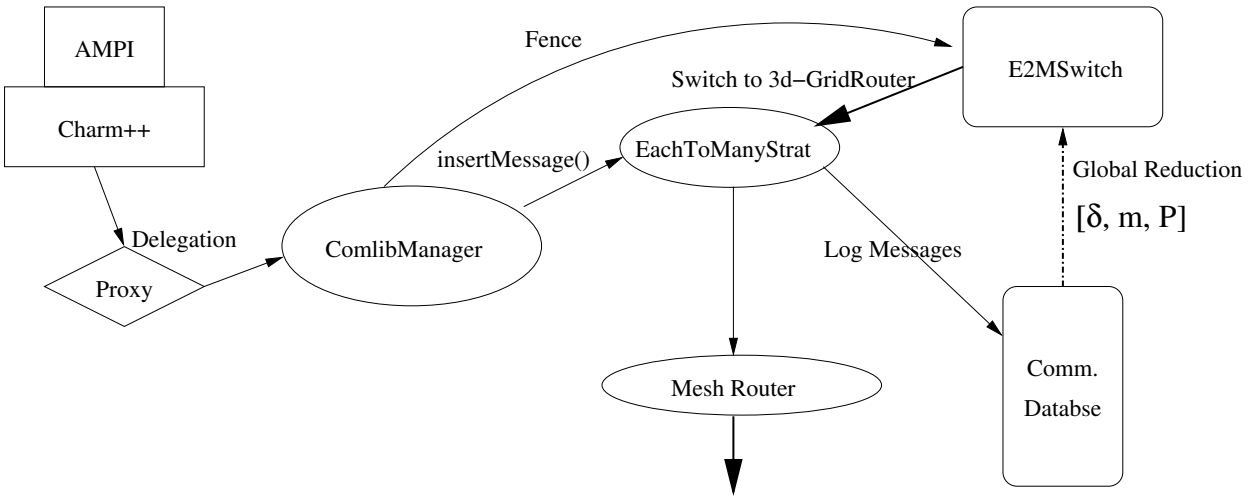


Figure 6.3: Strategy Switching in All-to-All communication

We have designed a *strategy-switch* in our communication framework, that can dynamically switch the application messages to the best strategy. Figure 6.3 illustrates strategy switching in all-to-all communication with the EachToManyStrategy. This strategy first optimizes object communication and then calls one of several routers (*MeshRouter* in Figure 6.3) to optimize processor level all-to-all communication.

Strategies or groups of strategies can have their own strategy-switch class. For example, *E2MSwitch* can switch EachToManyStrategy to use the router with the lowest overhead using the equations presented in Chapter 3.3. The application can start with a default router and at the fence step, which are global barriers like load-balancing, the communication framework could switch the strategy to use the heuristically optimal virtual topology.

For all-to-all communication, the best strategy depends on the size of the message, the number of processors and the degree of the communication graph (Chapter 3.3). (For large messages, the network topology is also important). This can be represented as the tuple $[\delta, m, P]$, the degree of communication on that processor, the average size of the messages exchanged, and the number of processors participating in the all-to-all operation. Strategies in the communication framework log application messages in the communication database. At the next fence step, the $[\delta, m, P]$ tuple is retrieved from the communication database on

each processor and globally averaged through a reduction to get $[\delta_{av}, m_{av}, P_{av}]$. The switch module uses this global average to choose the optimal strategy. In Figure 6.3, E2MSwitch switches the virtual topology to *3-D Mesh* from *2-D Mesh*. The performance of the chosen strategy for all-to-all personalized communication, on the Turing cluster [69] with increasing message size is shown in Figure 6.4. Observe that the switch point between 2D-Mesh and Direct is quite accurate.

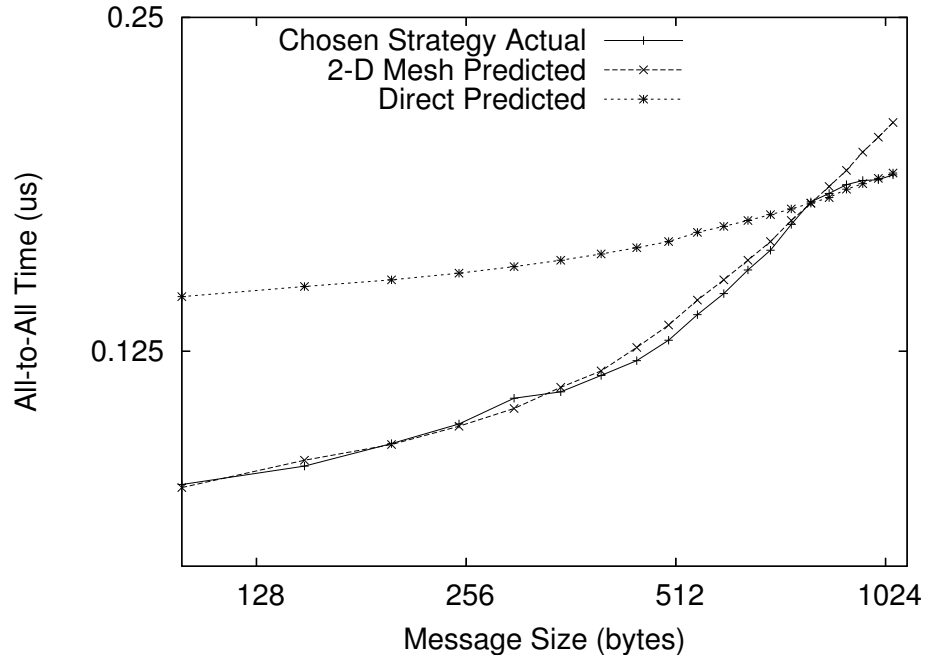


Figure 6.4: All-to-all strategy switch performance 16 nodes of the Turing cluster

6.5 Handling Migration

Array objects in Charm++ can migrate between processors, and so the communication framework has to explicitly handles object migration. There are two types of object migration in Charm++ :

- *Any-time migration*, where objects can migrate at any-time between processors. This involves packing the stack and heap allocated memory of the VP. The array manager in Charm++ supports this mechanism. Anytime migration is useful for fault tolerance; if the CPU cooling system throws an interrupt that the CPU is overheated and prone to faults, all work has to be immediately moved out of it.

- *Systolic migration.* The migration here occurs only at known and regular time intervals, e.g. centralized load balancing. After all migrations have been completed the application will run without any migrations till the next load-balancing step.

In the communication optimization framework, both types of migration are supported, but the framework is only optimized for systolic migration. Any-time migration is supported through message forwarding till the next systolic fence. At the fence, all strategies are reconfigured with the new object maps.

The decision not to optimize any-time migration was made keeping performance in mind. In contrast, an array broadcast mechanism [40] for any-time migration is presented. This scheme sends every broadcast via processor 0 to ensure that array elements receive each broadcast message only once. This serialization does not add any serious overhead for broadcast (since it adds only one more hop to the broadcast and processor 0 needed the broadcast message anyway), but for an all-to-all broadcast this serialization will make processor 0 a bottleneck restricting throughput. In the communication framework, the strategies have no additional overhead for the no-migration and systolic migration scenarios, as this are currently the common cases in the Charm++ runtime. Our broadcast and multicast schemes have been designed not to have overheads such as the serialization mentioned above. With migration, messages are temporarily forwarded back to the source processor and at the fence step, all object maps are reconstructed to compensate the migrations.

Each strategy in the communication framework has an application designated set of source and destination objects. We handle source and destination migration separately. Source migration is a problem for the bracketed strategies which wait for all objects on a processor to deposit their messages. So if an object migrates, its deposited messages will be received on the new processor. The strategy object on the source processor either has to be notified about this migration, or the messages of the migrated object have to be forwarded back to the source processor. We use the latter scheme.

The communication library uses a two step scheme to handle source migration:

- When object migration occurs, the communication library forwards messages of a migrated source object back to the processor where the object resided at the last fence. We call this processor the *designated processor P*. For each array element **e** , its designated processor will be a processor where **e** was some time in the past. Under normal circumstances, **P** will be the processor on which **e** resides. The designated-processor for an array element only changes at the fence step.

As each communication call is associated with a persistent proxy that the array object must use to invoke the call, the framework stores the designated processor of that array element in this proxy. Migrating objects should pack and carry this proxy with them, allowing the communication framework on the new processor to locate the designated processor of that object.

- As more elements migrate, the performance of the communication library will degrade. So a *fence* step is needed that reconstructs all the object maps. This is particularly useful for periodic global migrations, e.g. centralized load-balancing. Such a global barrier can be a communication library fence, where the array element maps are reconstructed for the strategies. In addition users can also call a Communication Framework fence explicitly.

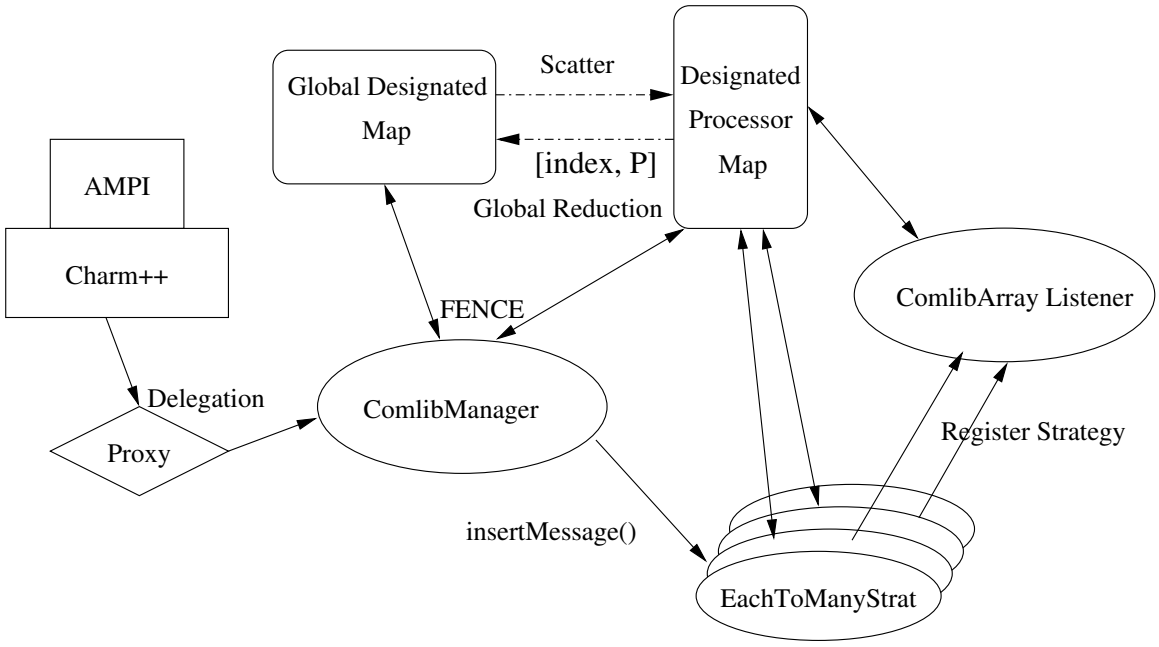


Figure 6.5: Fence Step in the Communication Framework

Destination migration can also be handled through designated processors. When an object migrates, all point-to-point and multicast messages for it are forwarded to its designated processor. The designated processor then passes the message on to the array manager, which knows where the object lives.

Array Section multicast trees are built on the designated processors of the participating objects in the section. So, each multicast message is sent to the set of designated processors for that section, which then send point-to-point messages to the array elements they are responsible for. When no migration occurs,

these messages are local messages.

At the fence step each processor contributes the pair $[index, MyPe]$ to a global collection. The client of this collection has the new designated processor map for each array element. The designated processor map is then sent to all the processors through a scatter. Processor only receives the designated processors of all objects it needs to communicate with. (This scatter is currently implemented through a broadcast). A schematic description of the fence step is shown in Figure 6.5.

This fence step in the communication framework is synchronized to occur just after load-balancing operation in the Charm runtime, because at the loadbalancing step several migrations are very likely. The framework also reconfigures all strategies and creates a new map of array elements and their designated-processors. The new strategies and the designated-processor map are then broadcast to all the processors.

6.5.1 Strategy Switching with Migration

Strategy switching in the presence of migration is a harder problem. The communication framework records processor level statistics for $[\delta, m, P]$. It is possible that the processor level statistics recorded is incorrect after object migration. This is because objects may now live on different processors changing δ and P.

For performance reasons, we do not record object-to-object communication, as that requires a costly hashtable access. We just record processor-to-processor statistics, which can be stored in a flat array, as the number of processors is much smaller than number of objects. This is another engineering trade-off we have considered. Fortunately, for applications like NAMD and CPAIMD the object to processor map of objects participating in collectives does not affect P, and δ much.

Chapter 7

Application Case Studies

In this chapter the present the performance improvements of three applications using the communication framework. Two of these, NAMD and CPAIMD, are critical applications for our research group. They have actually motivated several of the optimizations presented in this thesis. NAMD is a classical molecular dynamics program and CPAIMD is a quantum chemistry application. The radix-sort benchmark tests the all-to-all personalized communication strategies in the communication framework.

7.1 NAMD

NAMD is a parallel, object-oriented molecular dynamics program designed for high performance simulation of large bio-molecular systems [54]. NAMD employs the prioritized message-driven execution capabilities of the Charm++/Converse parallel runtime system, allowing excellent parallel scaling on both massively parallel supercomputers and commodity workstation clusters. NAMD has two critical collective communication operations that have to be optimized for it to scale well.

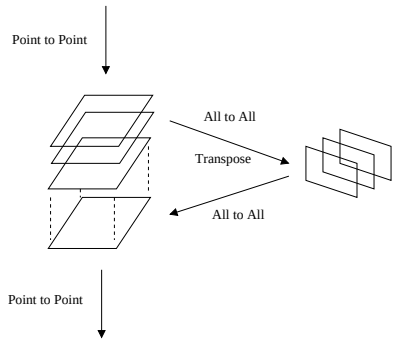


Figure 7.1: PME calculation in NAMD

The first is Particle Mesh Ewald (PME) computation, which involves a forward and a backward 3D FFT (Figure 7.1). One way to perform a 3D FFT, is by first performing a local 2D FFT on each processor on the Y and Z dimensions of the grid, and then redistributing the grid through a transpose for a final 1D FFT on the X dimension. The transpose would require an AAPC operation. If the grid is irregular then this would actually be an MMPC operation.

Processors	ApoA-1(ms)		ATPase(ms)		
	Direct	2-D Mesh	MPI	Direct	2-D Mesh
256	44.4	39.2	134.5	120.8	113.6
512	28.0	23.4	69.5	63.0	60.8
1024	26.8	20.3	39.3	38.6	35.8

Table 7.1: NAMD step time (ms)

We can use virtual topologies to optimize this AAPC (Chapters 3.1 and 3.3) operation. Table 7.1 shows the performance of NAMD on two molecules ApoA-1 and ATPase, with the 2D mesh and direct strategies. For ApoA-1 a $108 \times 108 \times 80$ grid is used and for ATPase it was a $192 \times 144 \times 144$ grid. The PME calculation involved a collective communication between the X planes and the Y planes. In our large processor runs, the number of processors involved in PME is $max(\#XPlanes, \#YPlanes)$, which is 108 for ApoA-1 and 192 for ATPase.

Table 7.1 also shows the performance of doing the all-to-all by making a call to MPI_Alltoall. NAMD carries out other force computations concurrently with PME. As MPI_Alltoall is a blocking call, it does not allow the application to take advantage of the low CPU overhead of the collective call. Hence, the 2-D mesh and direct strategies do better than MPI. As the size of messages exchanges are relatively small (about 600 bytes for ApoA-1 and 900 bytes for ATPase) 2-D mesh has a lower completion time and CPU overhead than the directly sending messages and hence does better. This lower CPU overhead of Mesh lets the more force computation to be overlapped with PME, resulting in lower step time.

The second collective operation is the many-to-many multicast of the coordinates by the cells of NAMD, to the processors where the forces and energies are computed. Atoms in NAMD are divided into a 3D grid of cells based on the cutoff radius. The interactions between these cells are calculated by compute objects which are distributed throughout the entire system.

Each cell multicasts the atom coordinates to 26 computes (atleast). For a large system like ATPase there could be 700 cells, which makes this many-to-many multicast a complex operation. As each cell only multi-

casts to a small set of neighbors, this collective operation is hard to optimize. We currently use a simple that implements multicast through point-to-point messages. The best way to optimize this operation is to build hardware multicast trees to multicast the messages in network hardware without processor involvement. This is idea will be presented in Chapter 8.

7.2 Scaling NAMD on large number of processors

A major issue we faced while scaling NAMD and other applications to large number of processors was that of stretched (prolonged) handlers for messages, also mentioned in [54]. We noticed that some processors had handlers lasting about 20-30 ms. Normally these handlers should take about 2-3ms to finish. We noticed stretches in handlers during a send operation and in the middle of the entry method itself. We believe these stretches were caused by a mis-tuned Elan library and operating system daemon interference. The subsequent section describes the mechanisms by which we overcame the stretching problem. (This research was done in collaboration with Gengbin Zheng and Chee Wai Lee [34, 35].)

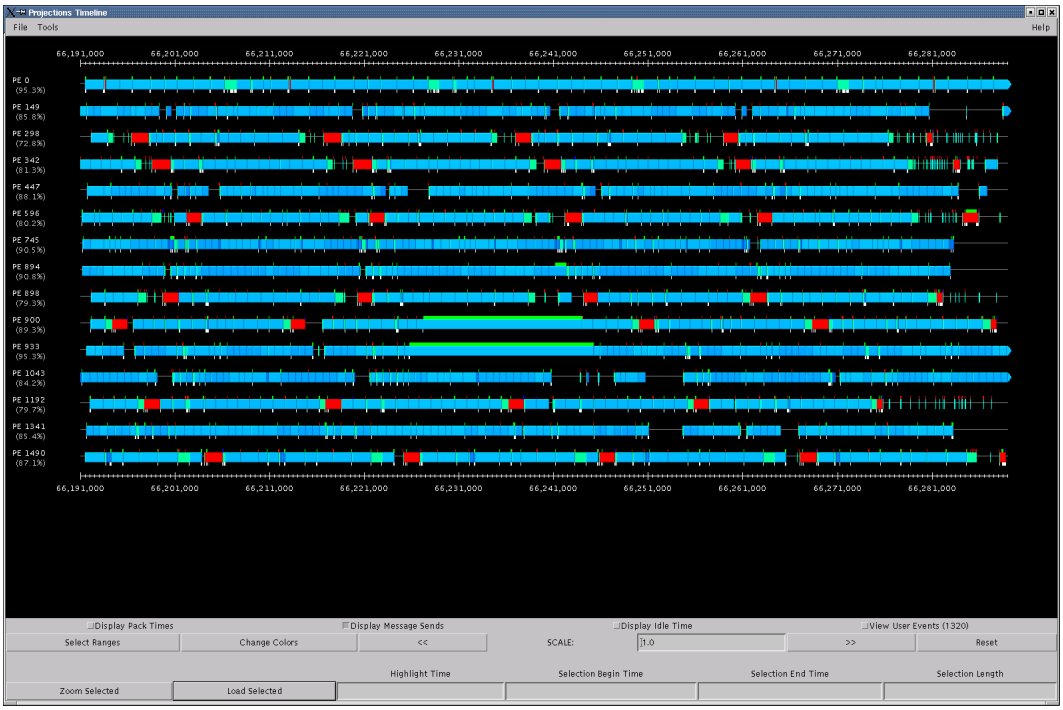


Figure 7.2: NAMD Cutoff Simulation on 1536 processors

Stretched Sends

The Converse runtime system only makes calls to elan_tportTxStart (equivalent of MPI_Isend in Elan) which should be a short call. From table 2.4 we know that the CPU overhead of ping-pong is just a few microseconds. However, the entry methods were blocked in the sends for tens of milliseconds.

Inspecting the Elan library source (and also working with Quadrics), we found that stretching during the send operation was a side effect of the Elan software's implementation of MPI message ordering. MPI message ordering requires that messages between two processors be ordered. Incidentally, Charm++ does not require such ordering.

To implement this ordering, the Elan system made a processor block on an elan_tportTxStart if the rendezvous of the previous message had not been acknowledged. So in the presence of a hot-spot in the network, all processors that sent the hot-spot a message would freeze. This could cascade leading to long stretches of even tens of milliseconds.

We reported this to Quadrics, and obtained a fix for this problem. This involved recompiling the Elan software library, after enabling the *ONE_QXD_PER_PROCESSOR* flag. Now, a message send only blocks if there is an unacknowledged message to *its* destination. The runtime system keeps a list of processors with unacknowledged messages and buffers future messages to them until all those messages have been acknowledged. This problem has been fixed in version 1.4 of the Elan software.

OS Daemon Stretches

Fixing the Elan software did not completely eliminate stretches. When applications used four processors per node, some handlers still experienced stretches. NAMD simulation of the ATPase system takes about 12ms on 3,000 processors. This time step is very close to the 10ms time quanta of the operating system. So if on any of the 3,000 processors a file system daemon is scheduled, NAMD step time could become 22ms.

Petrini et al. [13] have studied this issue of operating system interference in great detail. They present substantial performance gains for the SAGE application on ASCI-Q (a QsNet-Alpha system similar to Lemieux) after certain file system daemons were shutdown.

We did not have control over the machine to do the system level experiments carried out by Petrini et al. However, we were still able to reduce and mitigate the impact of such interference: First, NAMD uses a reduction in every step to compute the total energies. However, with Charm++, it was able to use an asyn-

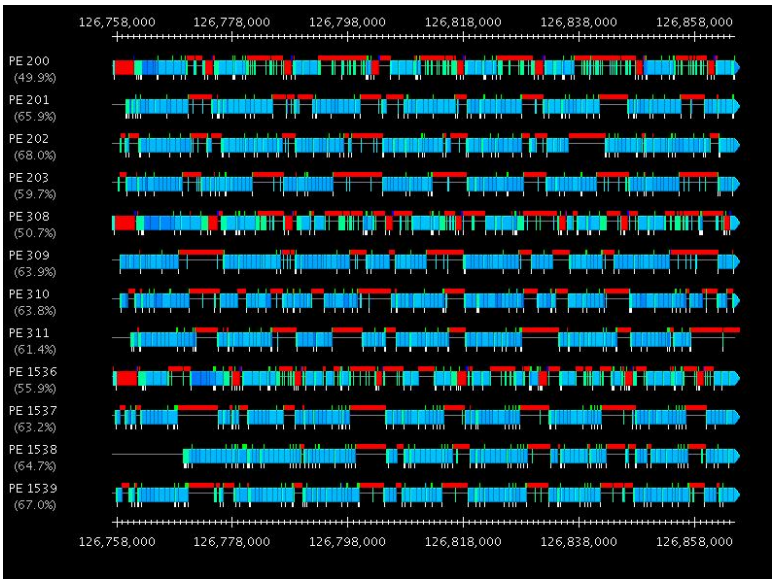


Figure 7.3: NAMD With Blocking Receives on 3000 Processors

chronous reduction, whereby the next time-step doesn't have to wait for the completion of the reduction. This gives the processors that were lagging behind due to a stretch an opportunity to catch up (figure 7.2). When a processor becomes idle, the *receive module* in the Converse communication layer *sleeps* on a receive, instead of busy-waiting. This enables the operating system to schedule daemons while the processor is sleeping. On receiving a message, there is an interrupt from the network interface which awakens the sleeping process.

The new timeline is presented in figure 7.3. Notice the red superscripted rectangles, which imply that a processor is blocked on a receive.

Blocking receives are based on interrupts and hence have overheads. The Elan library gives the option of polling the network interface for n μs before going to sleep. Setting the environment variable LIBELAN_WAITTYPE to n achieves this. NAMD performance on 3000 processors of Lemieux was best with $n = 5$. This was before daemons were shutdown on Lemieux. NAMD still achieved a 1.04 TF peak performance and a 12ms time step, by just the use of blocking receives.

Most of the daemons (recommended in [13]) have been shutdown on the compute nodes of Lemieux. But a full system run uses head and I/O nodes which still run some of these daemons. Table 7.2 shows a more recent performance of NAMD on 2912 processors for different values of n. Observe that now NAMD performance is best for n in the range of a few hundred, which implies that there is lesser OS interference

Poll Time (n)(μs)	Processors	Step Time (ms)
100	2912	11.3
200	2912	11.2
500	2912	11.0

Table 7.2: NAMD with blocking receives

after the daemons were shut down.

7.3 CPAIMD

Car-Parinello *ab initio* molecular dynamics (CPAIMD) ([20], [50], [6], [67]) can be used to study key chemical and biological processes. Moreover it is a simulation methodology that also can be employed in material science, solid-state physics and chemistry. CPAIMD methodology numerically solves Newton's equations using forces derived from electronic structure calculations performed "on the fly" as the simulation proceeds. This technique can revolutionize a host of technological problems including molecular electronics and enzyme catalysis.

We at the parallel programming laboratory have designed and developed our parallel implementation of the Car-Parinello calculation. It was developed by Ramkumar, Yan-shi, Vikas Mehta and Sameer Kumar in collaboration with Professors Glenn Martyna (IBM Research) and Mark Tuckerman (New York University).

We shall first briefly describe the CPAIMD calculation and then show the performance of several optimizations through the communication framework. Figure 7.4 shows the parallel structure of the Car-Parinello calculation. Here the roman numerals show the phases of computation. It involves several simultaneous 3-D FFTs in phase I to transform the electron wave functions (represented as state files) from Fourier space to real space. Once the states have been transformed from Fourier-space to real-space several reductions are performed in phase II to generate a density matrix.

Phases III and IV compute the gradient correction and the exchange correlation energy of the density matrix [70]. In phases V,VI the new densities are first multicast to all the states and then it is used to compute forces through inverse 3d-FFTs. Phases VII and VIII normalize the states for the next iteration of minimization, through ortho-normalization.

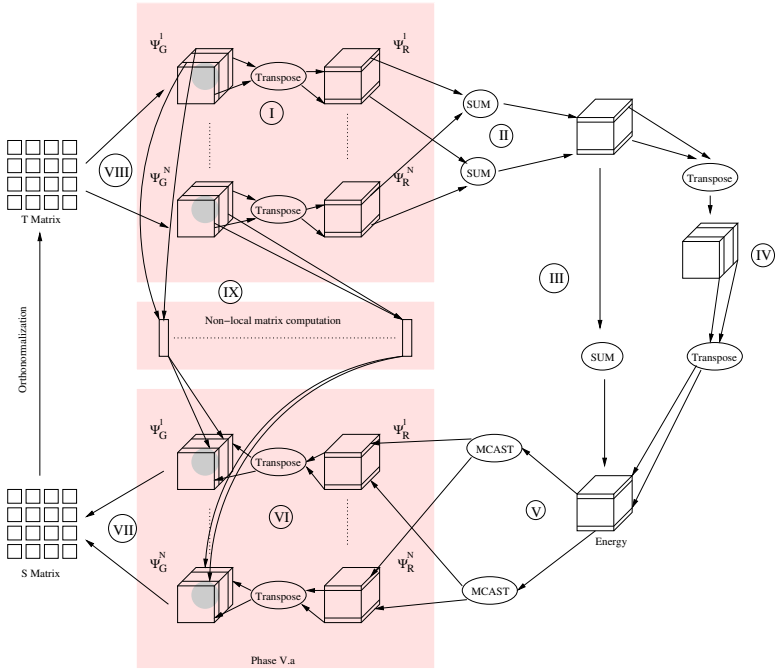


Figure 7.4: Parallel Structure of the CPAIMD calculation

7.3.1 Communication Optimizations

CPAIMD is a very communication intensive problem. Several communication optimizations were necessary to scale CPAIMD to thousands of processors. This application is an ideal case-study for the communication framework. In this section we illustrate some of these optimizations.

The hardest problem in CPAIMD was that of multiple simultaneous FFTs in Phases I and VI. For example a 32 water molecule system has 128 states. Each state is a 3-D Grid of points stored as planes in the 3-D Grid. The states in the 32 water system have 100 planes. The 3D-FFT operation involves a transpose which requires all-to-all communication. Two mapping are possible: i) all planes of the same state are placed close to each other, ii) planes with the same index across all states are placed on the same processor.

Our experiments show that state mapping i has good performance upto 128 processors and mapping-ii has better performance on larger number of processors. The main reason for this is the multicast operation in phase V. With mapping-i this operation is more like an all-to-all multicast operation and with mapping-ii it is a multicast to a small number of nodes.

With state mapping-i FFTs are either local or are performed on nearby processors and hence do not become a performance impediment. However in mapping-ii the FFTs transpose message will go across the

system to far-away nodes. To improve the performance of 3D-FFTs with mapping-ii we had to optimize a many-to-many communication problem with a degree between 25-100. The key idea to optimize this operation is to overlap computation with communication. As soon as a transpose message is received it is processed and copied into the plane array. This processing can be pipelined with the message sends.

The streaming communication strategy can be used to optimize this operation. The choice of the bucket is critical here. With a large bucket size few messages will be sent reducing communication overhead but pipelining of messages with computation will not be achieved. A short bucket would lead to good pipelining but suffer from a high communication overhead.

Processors	Bucket Size	CPAIMD step time (ms)
512	1	737
512	5	687
512	10	677
512	20	685

Table 7.3: Performance of streaming with bucket size on Lemieux

Table 7.3 shows the performance of the streaming strategy with different bucket sizes. The best performance is achieved with a bucket size of 10 which achieves a good communication performance and overlap of computation and communication.

Multicast: The multicast operation is phase V multicasts a large message to many processors. With mapping-i it actually sends the message to all the processors in the system leading to an all-to-all multicast operation. We used th Quadrics optimizations for all-to-all multicast presented in Chapter 4 to optimize this operation and the performance improvements are presented in Table 7.4.

Processors	Message Size (KB)	Multicast Optimization	CPAIMD step time (ms)
128	165	Pt-to-pt from Main memory	2067
128	165	Pt-to-pt from Elan memory	1268

Table 7.4: Performance of multicast optimizations on Lemieux

7.4 Radix Sort

The radix sort benchmark is a sorting program which sorts a random set of 64 bit integers, which is useful in operations such as load balancing using space-filling curves. The initial list is generated by a uniform

random number generator. The program goes through four similar stages. In each stage the processors divide the local data among 65,536 buckets based on the appropriate set of 16 bits in the 64 bit integers. The total bucket count is globally computed through a reduction and each processor is assigned a set of buckets in a bucket map which is broadcast to all the processors. All the processors then send the data to their destination processors based on the bucket map. This permutation step involves an AAPC and has the most communication complexity. Radix sort is therefore a classic example of AAPC. The performance of Radix sort with the 2-D mesh and direct strategies on 1024 processors is shown in Table 7.5. Here, N is the number of integers per processor. The table also shows the approximate size of message exchanged between the processors in the all-to-all operation. In Section 3.1, we showed that the combining strategies do better than the direct strategy for messages smaller than 1KB (Figure 3.5). But in Table 7.5 2-D mesh strategy does better than direct even for messages as large as 8KB. We believe this is because of the lower CPU overhead of the 2-D mesh strategy (Figure 3.7).

N (ints per proc)	Message Size (b)	2-D Mesh	Direct
10k	200	1.63	9.94
100k	900	2.10	11.3
500k	4000	4.37	16.2
1m	8000	7.5	18.7

Table 7.5: Sort Completion Time (sec) on 1024 processors

Chapter 8

Supporting Collectives in Network Hardware

Collective communication is a critical communication operation involving all or a large number of processors in the system. The previous chapters have described processor based collective-communication optimizations that send several point-to-point messages. For example, Chapter 3 describes optimizations for all-to-all communication that combine messages and send them along a virtual topology. This message combining happens on processors. Another example is the broadcast operation, that can be implemented as point-to-point messages sent along a spanning tree rooted at the source. The root processor sends messages to its children. As the children receive message, they send those messages to all their children. This scheme has $log(P)$ phases of point-to-point messages. In each of these phases, messages would typically go $log(P)$ hops on a tree network, making the actual total number of phases of the software collective $O(log^2(P))$.

Software collective optimization schemes have several other problems. For short messages, the broadcast completion time is dominated by the CPU and the network interface controller (NIC) overheads of sending the messages. Large messages sent by the several children may contend for the same communication channels. Software contention avoidance schemes may have to use barriers to keep these messages synchronized [37]. Good collective performance also requires that all intermediate processors immediately process and forward the incoming message. Performance is affected if one of the intermediate processors is running an operating system daemon [13], which can delay the collective operation. Moreover, with message driven execution [29] and asynchronous collectives [30] it is possible that the remote processor is busy doing other work and cannot process the message immediately delaying the broadcast completion.

For the above reasons, collective communication support is necessary in communication hardware. One

of the approaches studied in literature implements collectives in the network interface. This can reduce the CPU overhead of sending messages as the processors are less involved in the collective operation. This scheme is also unaffected by operating system daemon issues. Performance improvement of network interface reductions has been presented by Panda et. al. [46]. However, performance of such optimizations can be limited by the slow NIC hardware. Such collectives still exchange several point-to-point messages. So the broadcast overhead would still be $O(log^2(P))$. Similar to processor based schemes, large messages from different NICs could also contend with each other.

Hence, we believe that collective communication should be supported in the switching network. Both multicasts and reductions should be supported in the switches. Here, the broadcast overhead is just $O(log(P))$. On parallel systems with thousands of nodes, switch collectives will make a significant difference.

Some current clustering interconnects like Quadrics [57] QsNet [52] and Mellanox [45] Infiniband [1] have multicast support in their switches. But multicast performance is restrictive as these switches have input-queued architectures [43]. For example, in Quadrics QsNet only consecutive ports can be multicast to. Input queuing architectures require complex centralized arbitration to achieve high utilization, and are not a natural match for multicast [56, 44, 43]. Popular interconnects today also do not have reduction support in their switche architectures.

In this chapter, a switch-based solution to optimize multicasts and reductions is presented. We propose an output queuing architecture with crosspoint buffering, to achieve higher performance with multicast operations. We also show that output-queued architectures have better performance with point-to-point messages. In the past output queuing architectures have been less popular because they require higher internal bandwidth and more memory. But with current ASIC technology, it is possible to build crosspoint-buffered output-queued switches. A brief intuition showing the feasibility of output-queued routers is presented in the next section.

Our solution derives from existing literature and further extends it. The architecture supports efficient multicasts and reductions, as shown by the performance results in Section 8.3. With basic multicast and reduction support in switches other collectives like barrier, all-reduce and all-gather can be easily implemented in the network hardware. For example, all-reduce can be implemented as a reduce followed by a broadcast.

We evaluate our switch architecture with several point-to-point and collective benchmarks, which eval-

uate throughput and latency of collectives on output-queued routers. We simulate independent switches and networks of switches. To support collectives in the network a spanning tree has to be built on the network topology, which is topology specific. Here, we assume a fat-tree topology [41, 53]. Fat-trees are a popular network topology used by several interconnects like Quadrics QsNet [52], Infiniband [1], IBM SP networks. Fat-tree networks have high bisection bandwidth and can be scaled to thousands of nodes. We also present schemes to build collective spanning trees on fat-tree networks, and the performance of collectives using those spanning trees. Our scheme conserves routing table entries, as only one tree is needed to multicast data to a group with any leaf as the source.

We also present the network throughput and latency when several collectives happen simultaneously. Applications like NAMD [54] and CPAIMD [70] need multiple such simultaneous collectives. The advantages of hardware collectives is shown through a synthetic benchmark that emulates the collectives in NAMD.

8.1 Router Architecture

Several input and output queuing architectures have been proposed for high performance interconnect switches. Input queuing (IQ) schemes allow simpler data flow but require centralized arbitration to achieve high utilization. IQ routers also suffer from head of line blocking which restricts their throughput. Using multiple virtual channels and smart buffer management improves the performance of input-queued routers [60, 64, 19]. *Virtual output queuing* [44] (VOQ) can fully utilize the switch. Here each input queue has reserved buffer space for every output queue. Virtual output queuing also has a centralized arbiter and requires $O(K^2)$ buffer space, where K is the number of ports.

We believe that switch design should have efficient support for multicasts and combines. Input queuing (IQ) and virtual output queuing (VOQ) do not handle multicasts efficiently as they need centralized arbitration [56, 43]. VOQ can achieve full utilization for multicast if every input port has $(2^K - 1)$ queues [56, 43] in a KXK switch, one for every possible subset of output ports. As this requires a tremendous amount of memory, IQ multicast scheduling algorithms use heuristics. Performance can sometimes be severely affected if there is contention for outputs by different multicasts [43].

Two schemes have been proposed to handle multicasts in IQ routers [43, 56, 44], (i) No fanout splitting, and (ii) fanout splitting. Here *fanout* refers to the number of multicast destination ports. In *no-fanout-*

splitting, a multicast packet is only sent out if all destination ports are available in that arbitration cycle. The crossbar is used only once, but no-fanout-splitting may require several arbitration cycles to send the packet out and free the input buffer for that packet. No-fanout-splitting is good for multicasts with small fanouts.

In the *fanout-splitting* scheme, a multicast packet is sent to all output ports that are available in that arbitration cycle. Here the multicast packet uses the crossbar bandwidth for several cycles. The maximum achievable utilization for multicast is presented in [43], which is far from full utilization for many traffic patterns. IQ multicast schemes can also have deadlocks in a network of switches.

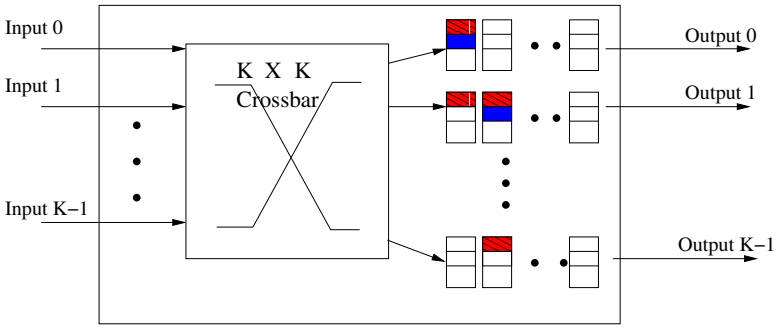


Figure 8.1: Output Queued Router Design

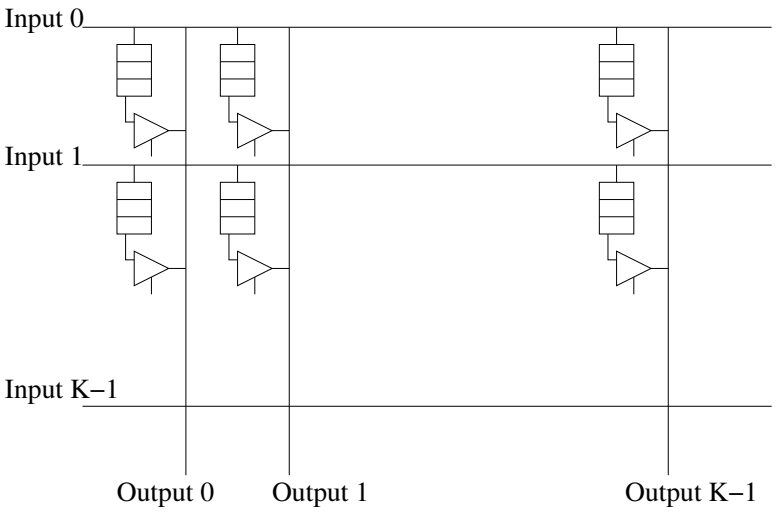


Figure 8.2: Crosspoint Buffering flow control

In this Chapter, we show the effectiveness of output queuing for hardware collectives. Packets in output queuing are buffered on the output ports of a switch before being sent out. Output queuing has distributed arbitration where each output decides which packet to send independent of other outputs. Figure 8.1 shows

an output-queued router with buffers at the outputs. This architecture is less commonly used as it requires more internal speedup to let input ports talk to several output ports simultaneously. With current ASIC technology however, it is possible to build output queuing switches. We now present an intuition to demonstrate the feasibility of such routers.

Feasibility of the Output Queuing architecture: Suppose we plan to build an Infiniband 4X switch with a bandwidth of 10Gbps per port. We would also like to support 20m cables or 200ns of round trip time (RTT) . Hence, we would need atleast 250 bytes of memory at each crosspoint. It is usually good to have two or four RTTs for good switch performance. For an 8 port switch the total memory requirement is about 64KB which is easily available in modern ASICs. For a 32 port switch we need 512KB to 1MB of buffer space. With current ASIC technology this should still be possible.

Popular output queuing routers in the past have used shared buffers between output ports [60]. Such shared buffer schemes have limited scalability with respect to link bandwidth and number of ports. We use crosspoint buffering in our router architecture to make the router support high bandwidth links efficiently. Cross-point buffering guarantees that there is a reserved buffer for each pair of input and output ports. A graphic description of cross-point buffering is shown by Figure 8.2. Each input port has some reserved memory on every output port. Hence the total buffering required is $O(P^2)$. Packets arriving on input ports are immediately sent to the crosspoint determined by the destination output queue. Our output queuing router with cross point buffering is similar to the SAFC scheme presented in [64]. But [64] only presents the point-to-point performance on one switch. We are mainly concerned with multicast and reduction performance on one switch, and on networks of switches.

We use virtual cut through routing and credit based flow control [5] between switches. Each switch keeps track of the buffer space available in the next switch. Packets are only sent out if buffer space is guaranteed on every port of the next switch. With crosspoint buffering this implies that all crosspoints for the current input port should have buffer space available for this packet. The flow control is implemented through a credit counter. This counter is initially set to the maximum buffer space at each crosspoint and as packets are sent out it is decremented. When ever the next switch dispatches a packet it sends back the credits to receive more packets.

We show in the next few sections that multicasts and reductions are efficient and easy to implement in such an output-queued architecture.

8.1.1 Multicast

Our credit based flow control scheme ensures that when a packet is sent out buffer space is available on all cross-points corresponding to this input port. So for every multicast buffer space will be available on every output port. On arrival, the multicast packet is immediately sent to all the ports determined by the destination address. The multicast packet only uses the crossbar once. Flow-control credits for this multicast packet are only sent back after all multicast packets have been sent out. Hence this scheme can achieve full throughput and also avoid deadlock issues of input queuing schemes.

8.1.2 Reduction

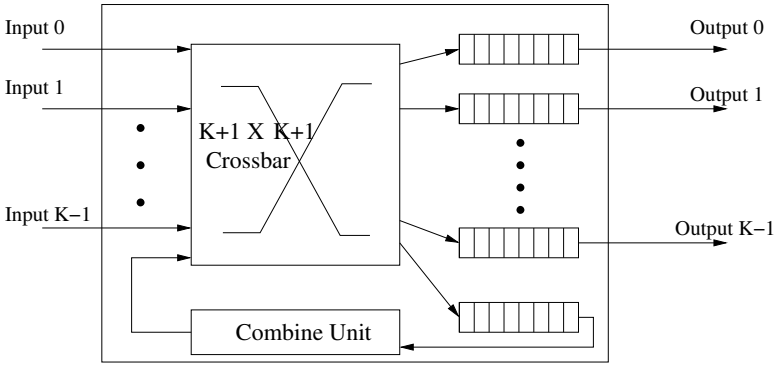


Figure 8.3: Switch Design with Combine Unit

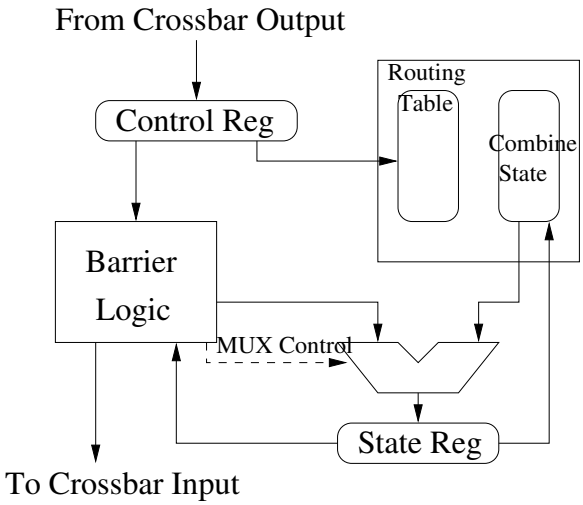


Figure 8.4: Combine unit architecture in the Output-Queued Router

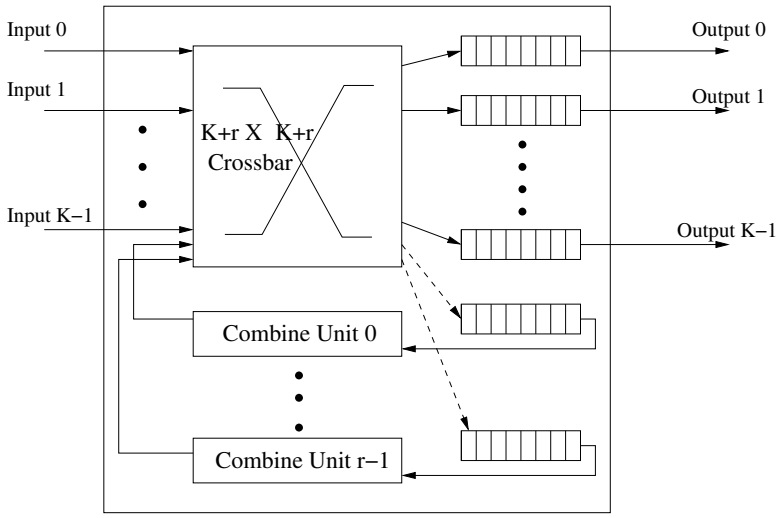


Figure 8.5: Switch with r combine units

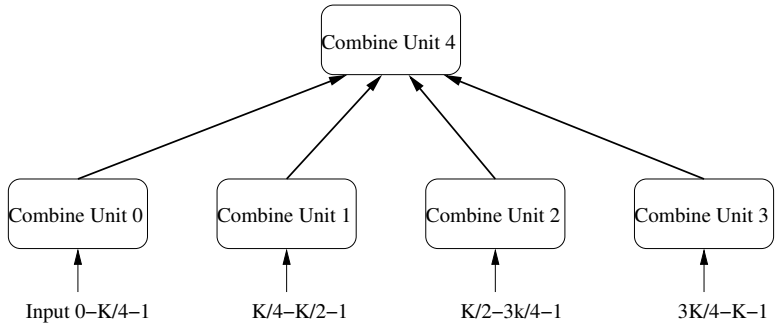


Figure 8.6: Combine units organized in a tree ($r = 5$)

Our design also supports the *Combine* operation which can be used to support reductions and barriers in hardware. We extend the barrier combine unit presented in [59] to perform reductions. The combine unit receives packets from the crossbar output and performs reductions. Every reduction has access to local state. For example, in the global sum operation the local state can store the current partial sum. For a global array sum, the local state could be an array of floating point numbers. This local state is updated by the combine unit whenever a reduction packet arrives. After all reduction packets have been processed, the combine unit sends a reduction packet back into the crossbar to be sent to the parent switch in the spanning tree.

The combine unit connects from the output port through a feedback to an input port in the switch, as demonstrated in Figure 8.3. The combine unit behaves like any other output port in the switch. Reduction packets arriving on input ports of the switch are buffered at the output port connected to the combine unit

before being processed. The architecture of the combine unit is shown in Figure 8.4.

It can take a few cycles to receive reduction packets, as the entire packet is needed to detect errors. (We do not explicitly simulate errors but we do model the delays.) The header of the packet is stored in the control register. The combine logic uses the address in the packet header to lookup the routing table for the local state of the current reduction. In the following cycles the logic unit computes and updates the local state based on the data from the packet.

For short reductions and barriers it may be possible to pipeline packet arrival and computation to process one packet every cycle [59]. But for larger reductions involving several data points, the combine unit may stall on each combine operation. In switches with a large number of ports, a single combine unit will become a point of contention. As ASIC speeds are much slower than custom designed CPU speeds, this may hamper the overall efficiency of the global reduction operation.

Figure 8.5 shows the switch architecture with 'r' combine units. The combine units are organized as a tree with $r - 1$ leaves and one parent (Figure 8.6). The leaves process the reduction packets from a subset of ports and pass their partial result to the root of the tree. Such a hierarchical design scales to more number of ports as several combines at the leaves can happen simultaneously. In Section 8.3.2, we show that with $r << K$ we can achieve good performance. Hence, the reduction units are only a small additional overhead.

8.2 Building a collective spanning tree

Spanning trees are essential to support collectives in the network hardware. These spanning trees can be directed trees where packets only travel in one direction on each hop. A broadcast with one source needs such a tree. If the network time to do a broadcast does not depend on the root of the spanning tree, we can also build undirected spanning trees for broadcasts. Here any leaf of the tree can do a broadcast with the same overhead. It is possible to build such a tree in a fat-tree network. The time to do a broadcast would be $O(log(P))$ independent of which leaf has sent the broadcast message. Our switch design has support for undirected spanning trees. Such undirected trees save routing table memory as any leaf can send messages. With directed trees [59] each sender would require a separate tree.

The routing table has a bit vector of destination ports for each collective address, as opposed to a parent and a list of children. For a multicast operation, packets are sent to all ports except the port on which the packet arrived on.

We implement combines as follows: suppose a routing table destination bit-vector has k outputs set, then the combine manager would process k-1 reduction packets and send the current partial result to the remaining port on which it did not receive a packet.

Both multicasts and combines use the same routing table entries. The tag in the packet determines whether the operation is a multicast, barrier, reduction etc.

8.2.1 Fat-tree Networks

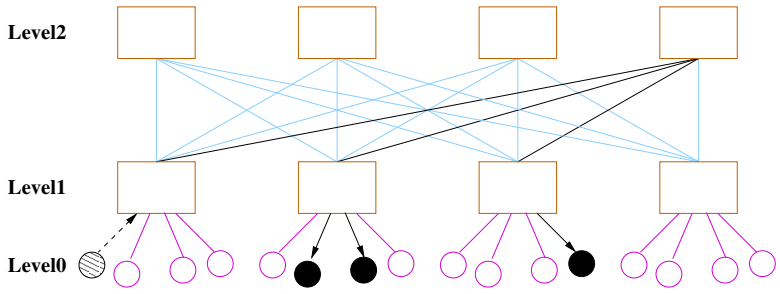


Figure 8.7: Fat-tree with 16 nodes

In this section, we describe our design to build collective spanning trees on network topologies. We take fat-trees as an example of an interconnect topology. Fat-trees are generalizations of k-ary n-trees [53]. Figure 8.7 shows a 4-ary 2-tree network. Routing in a k-ary n-tree has two phases, (i) the upward phase where a packet is sent to any of the lowest common ancestors of the source and the destination, (ii) the downward phase where the packet is routed from this ancestor to the destination through a fixed path.

This scheme can be extended to support collectives as follows: a multicast packet is sent to one of the lowest common ancestors of all the nodes from where it is routed to all the destinations in the tree. The advantage of using one common ancestor for all the nodes is that the spanning tree can be used by any leaf to do a multicast.

Collective tree algorithms for the Quadrics QsNet are presented in [11]. Here several trees are built to support hardware multicast on a discontinuous set of nodes. This is because the Quadrics network can only multicast to a contiguous set of nodes. Our switch architecture places no such constraints. We propose schemes to build several spanning trees to support multiple simultaneous multicasts and reductions.

Figure 8.7 illustrates a simple collective tree building algorithm. In the figure, the port $K-1$ (upper right corner port in the switch) is used to go up to the lowest common ancestor of all the nodes. Routes from

this ancestor to all the destinations constitute the spanning tree. This simple scheme would lead to a load-imbalance among the top level switches when several multicast trees need to be built. An more effective tree building algorithm is presented below :

```
buildTree(id, destlist, swlist, tlist, up)
id : the switch id of the current switch
destlist : list of processor destinations
swlist : list of previous switches
tlist : list of treeInfos, where each
treeInfo contains the list of output ports
at that switch
up : boolean flag that shows direction

begin
   swlist.insert(id);
   //Need to go further up
   if(!inHighestLevel(id, destlist) && up) {
      parent = leastLoadedParent(id);
      buildTree(parent,destlist,swlist,
                tlist, true)
   }

   //Going down in the fat-tree
   for count : 0 -> numPorts/2 - 1
     if(child[count] routesto destlist) {
        tlist[my_pos].insert(count);
        buildTree(child[count],destlist,
                  swlist,tlist, false);
     }
end
```

Here *leastLoadedParent()* gets the least loaded parent for the current switch. The load of the switch is determined by the number of multicast trees passing through that switch. This algorithm minimizes contention on the upward path of the packet. It also load-balances the routing memory required for each

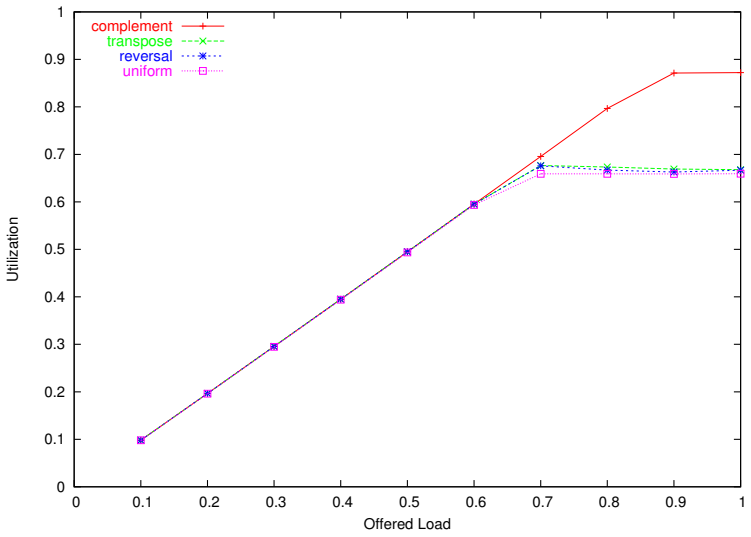


Figure 8.8: Throughput on a 256 node network with 8 port switches and 2 packet buffers

collective operation by choosing switches with fewer collective trees passing through them.

8.3 Network simulation

We simulated switches with the above architecture using POSE [72] which is a parallel event driven simulation language. We simulated 8 port and 32 port routers in a fat-tree topology with adaptive routing. Table 8.1 shows the parameters of our simulation. These parameters are derived from Infiniband 4X interconnects.

Parameter	Value
Bandwidth	10 Gbps
Packet Size	256 bytes
Channel Delay	20 ns
Switch Delay	90ns
Switch Ports	32
ASIC Speed	250 Mhz
NIC Send Overhead	1300 ns
NIC Recv. Overhead	1300ns

Table 8.1: Simulation Parameters

We first present the throughput and packet latency of point-to-point communication using the well known communication patterns *transpose, bit reversal, complement and uniform*. We simulated a 256 node fat-tree network with 8 port and 32 port output queuing switches. We also varied the amount of buffer space

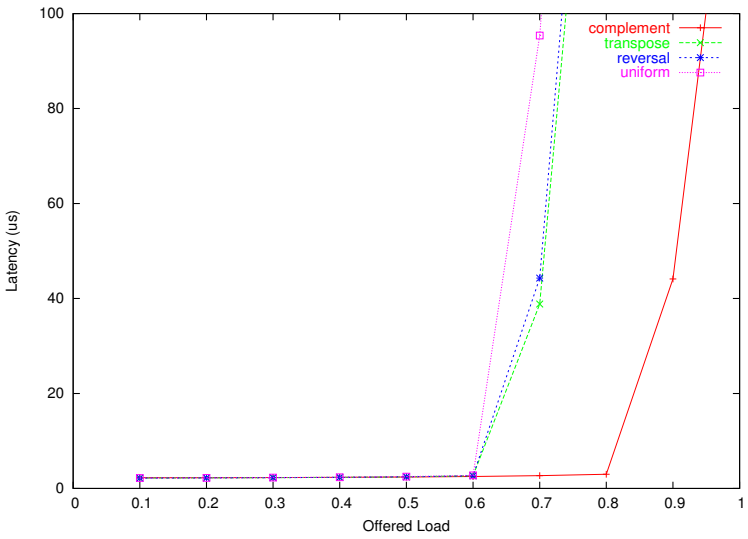


Figure 8.9: Latency on a 256 node network with 8 port switches and 2 packet buffers

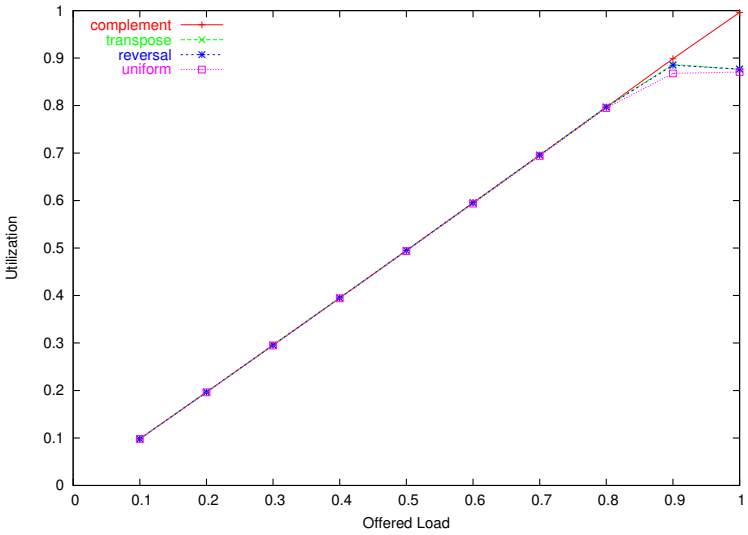


Figure 8.10: Throughput on a 256 node network with 8 port switches and 4 packet buffers

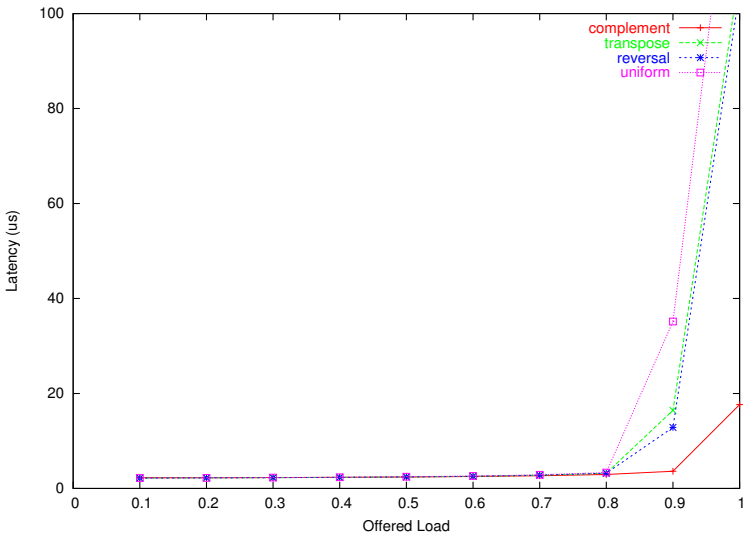


Figure 8.11: Latency on a 256 node network with 8 port switches and 4 packet buffers

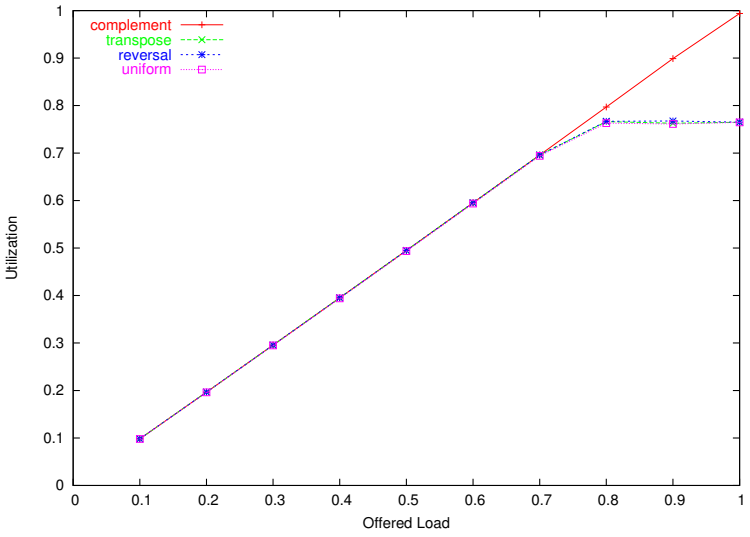


Figure 8.12: Throughput on a 256 node network with 32 port switches and 2 packet buffers

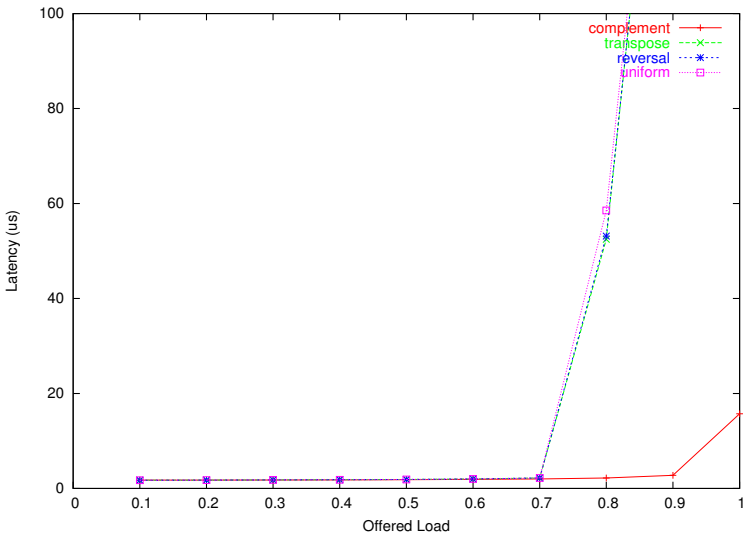


Figure 8.13: Latency on a 256 node network with 32 port switches and 2 packet buffers

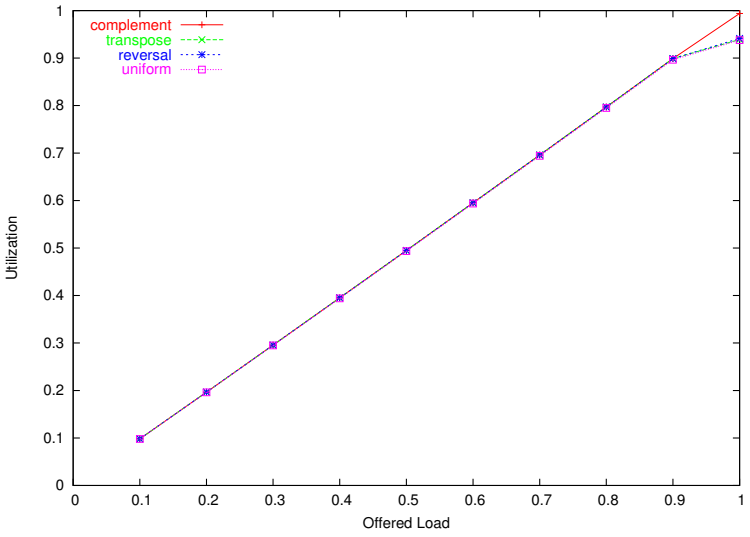


Figure 8.14: Throughput on a 256 node network with 32 port switches and 4 packet buffers

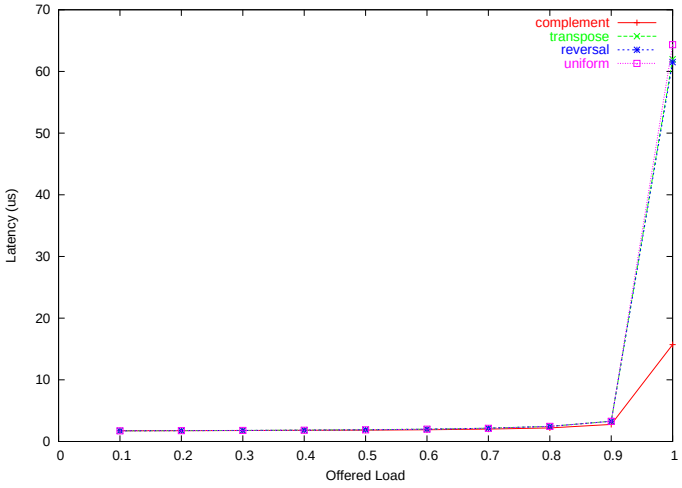


Figure 8.15: Latency on a 256 node network with 32 port switches and 4 packet buffers

at each crosspoint in the switch.

Figures 8.8, 8.9, 8.10 and 8.11 show the throughput and response times with 8 port switches. Figures 8.12, 8.13, 8.14 and 8.15 show the performance of 32 port switches.

Performance is best with 32 ports and 4 packets for each crosspoint. With 32 port switches the fat-tree has 32 switches organized in two levels. Since complement is contention free [53, 22] its throughput is 100% at full load. Uniform, Transpose and Reversal also have good throughput of about 93%. This high throughput is due to output queuing, adaptive routing [4] in fat-trees and the fact that there are only two levels or 3 points of contention in the entire network. Response times are also good for Complement and Uniform and only blow up for Transpose and Reversal for load factors greater than 0.9.

A performance evaluation of 8 port input-queued routers and a 256 node fat-tree networks is presented in [53]. Our output queuing routers perform better for all permutations with close to full throughput.

8.3.1 Multicast Performance

The performance of multicast on an 8 port switch is presented in Figure 8.16. Here each port sends to a packet to random destinations and with an average fanout of 4. Packets are generated on each port with a Poisson distribution with a mean inversely proportional to the load factor.

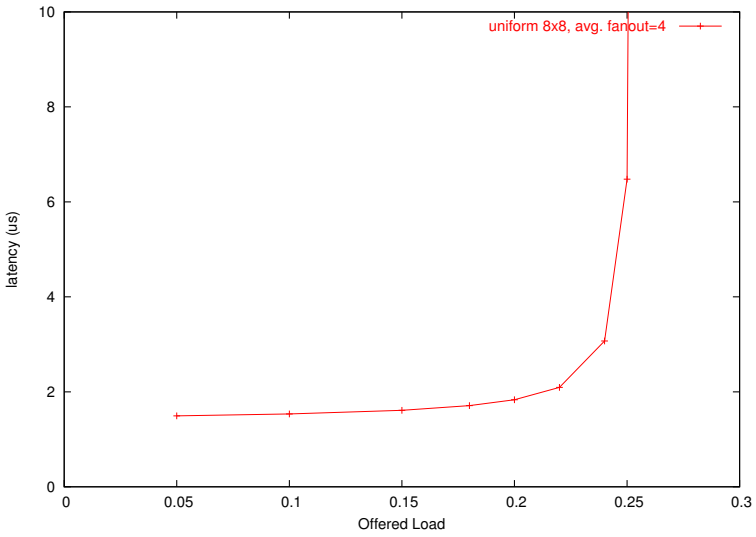


Figure 8.16: Response time for multicast traffic on an 8X8 switch with an average fanout of 4

As the mean fanout of the multicast is four, performance saturates at a load factor close to 0.25. Infact, this is the maximum achievable throughput with a fanout of 4. With only two nodes sending data the performance of multicast saturates at a load factor of 0.8, as shown in Figure 8.17. These results are better than the performance of virtual output-queued routers, presented in [56], where for un-correlated traffic with fanout of 4 the performance for a 2X8 switch is 0.65 and for an 8X8 switch it is 0.22.

Figure 8.18 shows the multicast latency for a 256 node fat-tree network. Here each node sends a multicast packet to a random set of destinations with an average fanout of 8. It can be seen that the latency is stable for load-factors under 0.125, showing the effectiveness of our scheme on a network of switches.

8.3.2 Reduction Performance

The simulated performance of a reduction is shown in Figure 8.19 for a fat-tree network with 256 nodes. With only one reduction a network with 8 port switches performs better than a network with 32 port switches for message sizes greater than 64 bytes. The 32 port performance degrades with increasing message size because of the stalls in the reduction pipeline for large packets (Section 8.1.2).

Multiple combine units enhance the performance of 32 port switches. Reduction completion time with 32 port switches and 5 combine units is shown in Figure 8.19. (Reductions on fat-tree networks use only one port in the upward path and a maximum of K/2 ports. Hence the effective number of combine units is

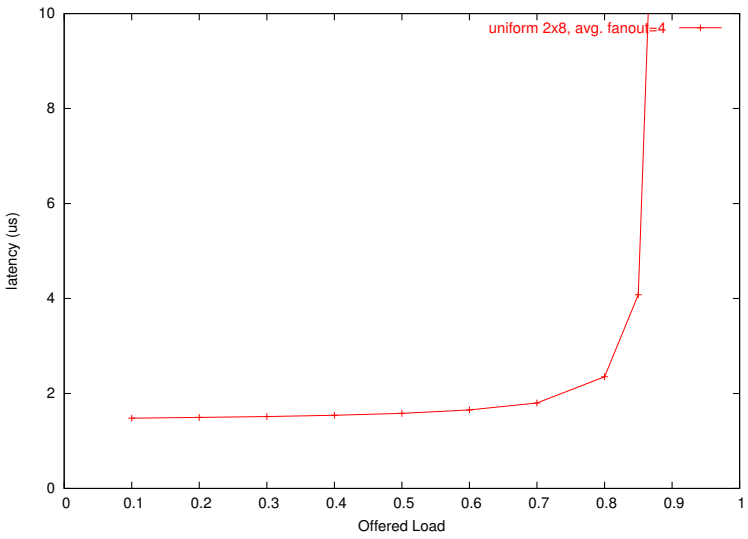


Figure 8.17: Response time for multicast traffic on a 2X8 switch with an average fanout of 4

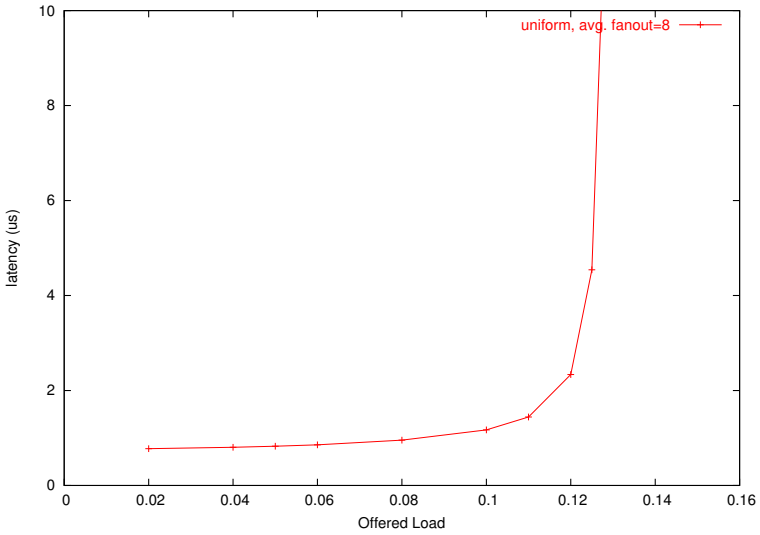


Figure 8.18: Multicast response time on a 256 node fat-tree network with an average fanout of 8

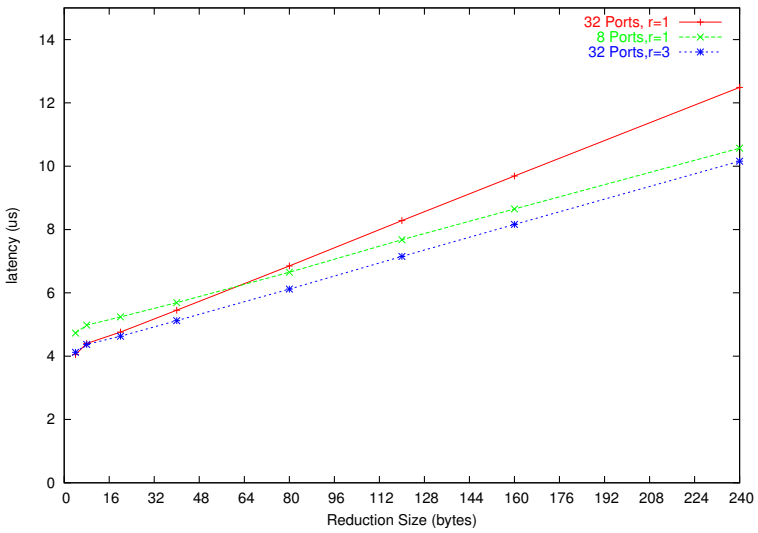


Figure 8.19: Reduction Time on 256 nodes

actually 3.) Notice this performance is good even with large reductions. This shows that a small number of reduction units can achieve good performance for large messages.

8.4 Synthetic MD benchmark

In this section, we present the advantages of having hardware collective support in the network. We present the performance of a synthetic benchmark that emulates our molecular dynamics application NAMD. Processors in NAMD multicast coordinates to a small subset of processors which compute forces on those atoms and return results back to the source processor. In the synthetic benchmark, $P/16$ processors multicast data to random destinations with an average fanout of 16. In the benchmark on 256 nodes, 16 nodes send multicast messages with an average fanout of 16. Here fanout represents the number of destination nodes of a multicast.

Figures 8.20 and 8.21 show the performance of this synthetic benchmark with hardware multicast and multicast with point-to-point messages on 256 nodes. The figures clearly show the advantage of hardware multicast. As the network with 8 ports has more levels of switches and hence more points of contention, hardware multicast has more performance gains. On parallel systems with thousands of nodes even with 32 port switches there will be several levels and more contention for switch outputs. We believe that performance gains of hardware collectives on such large systems are indicated in the 8 port plots.

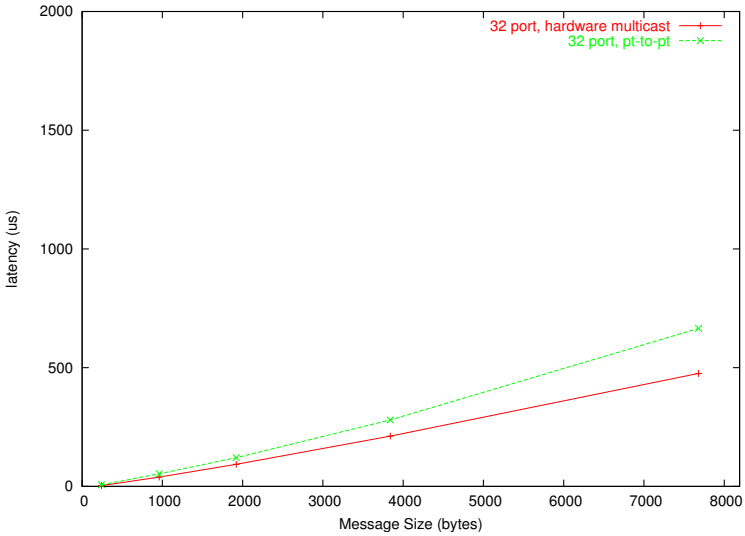


Figure 8.20: Comparison of hardware multicast and pt-to-pt messages for several small simultaneous multicasts of average fanout 16

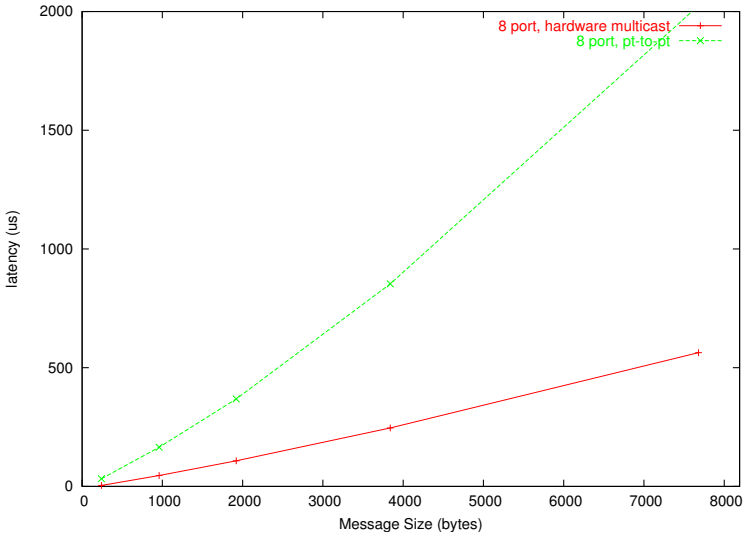


Figure 8.21: Comparison of hardware multicast and pt-to-pt messages for several small simultaneous multicasts of average fanout 16

Chapter 9

Summary and future work

This thesis described strategies to optimize communication in parallel applications. We presented three types of optimizations,

1. Low level techniques that optimize the runtime system to a vendor communication API, specifically taking advantage of network interfaces with co-processors.

2. Development of new collective communication algorithms that scale with good performance.

3. An object based adaptive communication framework that implements smart communication strategies and can switch them at runtime. This swith is based on the network architecture and dynamic application patterns observed through instrumentation.

The QsNet communication layer of the Charm runtime system implements many of the ideas presented in this thesis. The well known bio-molecular modeling program NAMD implemented using this machine layer on top of QsNet was awarded the Gordon bell award at SC'02 and later scaled to 1TF of peak performance.

The object based communication framework has dynamic strategy switching capabilities, as it can choose the best strategy based on the communication patterns in the application, using an analytical model to predict the performance of the different communication strategies. Strategy switching is demonstrated for all-to-all communication in this thesis with synthetic benchmarks.

This thesis also emphasized the importance of the CPU involvement in collective communication. On a large number of processors, all-to-all operations can take several milli-seconds to finish. With a co-processor in the network interface, the CPU is relieved from communication and can compute while the messages are in flight. Hence the all-to-all operation can be (and should be) overlapped with computation.

The future explorations for this thesis include development of several other dynamic learning schemes. For example, a streaming learner could choose between streaming, mesh streaming and direct message sending. The decision could be based on how many messages objects send, to how many different processors they send to etc.

More accurate prediction of all-to-all communication could be made through a network simulation that models the contention in the network too. Such a network simulator could be tied into the communication framework and be invoked to make predictions at the fence step.

It will be interesting to see how the strategies do on a large configuration of BlueGene/L. New strategies may also have to be developed for the 3d-torous BlueGene/L topology.

In this thesis, we extended the strategies for all-to-all communication to uniform many-to-many communication with short messages. We also studied a selected set of non-uniform many-to-many patterns. A generalized framework that works well for all on a large set of many-to-many patterns would be useful. Moreover, with large messages, the contention free permutations could also be applied to many-to-many communication. However, in this case all processors may not be able to send data in every phase. We leave design of such strategies to future work.

We also handled object migrations through forwarding. The array maps were only reset at fence steps. Strategies could update these maps in a distributed manner at a faster rate than the periodic fence steps. Such schemes will be valuable, if they add no overhead to the common scenario where migrations are infrequent.

References

[1] InfiniBand Architecture Specification Release 1.0. InfiniBand Trade Association, Portland, Ore., 2000.

[2] Vikram S. Adve, Rajive Bagrodia, James C. Browne, Ewa Deelman, Aditya Dube, Elias N. Houstis, John R. Rice, Rizos Sakellariou, David J. Sundaram-Stukel, Patricia J. Teller, and Mary K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Transactions on Software Engineering*, 26:1027–1048, November 2000.

[3] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[4] Y. Aydogan, C. B. Stunkel, C. Aykanat, and B. Abali. Adaptive source routing in multistage interconnection networks. In *Proceedings of the International Parallel Processing Symposium*, pages 258–267, 1996.

[5] Blackwell, T. Chang, K. Kung, H.T., and Lin. D. Credit-based flow control for ATM networks. In *Proc. of the First Annual Conference on Telecommunications R&D in Massachu setts*, 1994.

[6] P.E. Bloechl. *Phys. Rev. B*, 50:17953, (1994).

[7] S. Bokhari. Multiphase complete exchange: a theoretical analysis. *IEEE Trans. on Computers*, 45(2), February 1996.

[8] Sayantan Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[9] Ming-Syan Chen, Jeng-Chun Chen, and Philip S. Yu. On general results for all-to-all broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 7(4), 1996.

[10] Christina Christara, Xiaoliang Ding, and Ken Jackson. An efficient transposition algorithm for distributed memory clusters. In *13th Annual International Symposium on High Performance Computing Systems and Applications*, 1999.

[11] Salvador Coll, Jos Duato, Fabrizio Petrini, and Francisco J. Mora. Scalable hardware-based multicast trees. In *Supercomputing 2003*, Phoenix, AZ, November 2003.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, San Diego, CA, May 1993.

[13] Scott Pakin Darren J. Kerbyson, Fabrizio Petrini. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Supercomputing 2003*, November 2003.

[14] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CHARM (4.5) programming language manual*, 1997.

[15] Vassilios V. Dimakopoulos and Nikitas J. Dimopoulos. A theory for total exchange in multidimensional interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):639–649, 1998.

[16] Vassilis V. Dimakopoulos and Nikitas J. Dimopoulos. Communications in binary fat trees. In *International Conference on Parallel and Distributed Computing Systems*, September 1995.

[17] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *International Parallel Processing Symposium*, April 1997.

[18] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-Fast Resource Management. In *Supercomputing 2002*, Baltimore, MD, November 2002.

[19] Mike Galles. The sgi spider chip. In *Proceedings of Hot Interconnects IV*, pages 141–146, 1996.

[20] G. Galli and M. Parrinello. *Ab-initio* molecular dynamics: Principles and practical inplementation. *Computer simulation in chemical physics, NATO ASI Series C*, 397:261, 1993.

[21] A. Gursoy. *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance*. PhD thesis, University of Illinois at Urbana-Champaign, June 1994. Also, Technical Report UIUCDCS-R-94-1852.

[22] Steve Heller. Congestion-free routing on the cm-5 data router. *LNCS*, 853:176–184, 1994.

[23] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[24] Demmel J., Dongarra J., Eijkhout V., Fuentes E., Petitet A., Vuduc R., Whaley R. C., and Yelick K. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93:293– 312, February 2005.

[25] Matt Jacunski, P. Sadayappan, and D. K. Panda. All-to-all broadcast on switch-based clusters of workstations. In *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1999.

[26] Ben H. H. Juurlink, P. S. Rao, and Jop F. Sibeyn. Worm-hole gossiping on meshes. In *Euro-Par, Vol. I*, pages 361–369, 1996.

[27] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[28] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[29] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[30] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[31] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[32] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[33] Laxmikant V. Kalé, Sameer Kumar, and Jayant DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[34] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[35] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, number to appear, 2005.

[36] Sameer Kumar and L. V. Kale. Opportunities and Challenges of Modern Communication Architectures: Case Study with QsNet. Technical Report 03-15, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003.

[37] Sameer Kumar and L. V. Kale. Scaling collective multicast on fat-tree networks. In *ICPADS*, Newport Beach, CA, July 2004.

[38] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings, 1994.

[39] Chi Chung Lam, C.-H. Huang, and P. Sadayappan. Optimal algorithms for all-to-all personalized communication on rings and two dimensional tori. *Journal of Parallel and Distributed Computing*, 43(1):3–13, 1997.

[40] Orion Sky Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[41] C. Leiserson. Fat-trees: Universal networks for hardware efficient supercomputing. *IEEE Transactions on Computers*, 35(10):892–901, 1985.

[42] Lemieux. http://www.psc.edu/machines/tcs/lemieux.html.

[43] M. A. Marsan, A Bianco, P. Giaccone, E. Leonardi, and F. Neri. On the throughput of input-queued cell-based switches with multicast traffic. In *Proceedings of IEEE Infocom*, 2001.

[44] Nick McKeown, Martin Izzard, Adisak Mekkittikul, and William Ellersick an d Mark Horowitz. Tiny Tera: A packet switch core. *IEEE Micro*, 17(1):26–33, /1997.

[45] Mellanox inc. http://www.mellanox.com.

[46] A. Moody, J. Fernandez, F. Petrini, and D. K. Panda. Scalable nic-based reduction on large-scale clusters. In *Supercomputing 2003*, Phoenix, AZ, November 2003.

[47] Csaba Andras Moritz and Matthew Frank. Logpc: Modeling network contention in message-passing programs. In *Measurement and Modeling of Computer Systems*, pages 254–263, 1998.

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

[49] Scott Pakin and Avneesh Pant. VMI 2.0: A dynamically reconfigurable messaging layer for availability, usability, and management. In *The 8th International Symposium on High Performance Computer Architecture (HPCA-8), Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, Massachusetts, February 2002.

[50] M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos. *Rev. Mod. Phys.*, 64:1045, 1992.

[51] F. Petrini, Wu chun Feng, S. Hoisie, A.and Coll, and E. Frachtenberg. The quadrics network: high-performance clustering technology. *IEEE Micro*, 22(1):46 –57, 2002.

[52] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Performance Evaluation of the Quadrics Interconnection Network. *Cluster Computing*, 6(2):125–142, April 2003.

[53] Fabrizio Petrini and Marco Vanneschi. K-ary N-trees: High performance networks for massively parallel architectures. Technical Report TR-95-18, 15, 1995.

[54] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[55] Ravi Ponnusamy, Rajeev Thakur, Alok Chourdary, and Geoffrey Fox. Scheduling Regular and Irregular Communication Patterns on the CM-5. In *Supercomputing*, pages 394–402, 1992.

[56] Balaji Prabhakar, Nick McKeown, and Ritesh Ahuja. Multicast scheduling for input-queued switches. *IEEE Journal of Selected Areas in Communications*, 15(5):855–866, 1997.

[57] Quadrics ltd. http://www.quadrics.com.

[58] D Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Sixth Distributed Memory Computing Conference*, pages 398–403, 1991.

[59] R. Sivaram, C. Stunkel, and D. Panda. A reliable hardware barrier synchronization scheme. In *Proceedings of IPPS*, pages 274–280, 1997.

[60] Rajeev Sivaram, Craig B. Stunkel, and Dhabaleswar K. Panda. HIPIQS: A high-performance switch architecture using input queuing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):275–289, 2002.

[61] Y. J. Suh and S. Yalamanchili. All-to-all communication with minimum start-up costs in 2d and 3d tori. *IEEE Transactions on Parallel and Distributed Systems*, 9(5), 1998.

[62] N. S. Sundar, D. N. Jayasimha, Dhabaleswar K. Panda, and P. Sadayappan. Hybrid algorithms for complete exchange in 2d meshes. In *International Conference on Supercomputing*, pages 181–188, 1996.

[63] A. Tam and C. Wang. Efficient scheduling of complete exchange on clusters. In *ISCA 13th International Conference On Parallel And Distributed Computing Systems*, August 2000.

[64] Yuval Tamir and Gregory L. Frazier. High performance multiqueue buffers for vlsi communication switches. In *Proceedings of 15th International Symposium on Computer Architecture (ISCA)*, pages 343–354, 1988.

[65] Thakur and Choudhary. All-to-all communication on meshes with wormhole routing. In *IPPS: 8th International Parallel Processing Symposium*. IEEE Computer Society Press, 1994.

[66] Gunawan T.S. and Cai W. Performance Analysis of a Myrinet-Based Cluster. *Cluster Computing*, 6:299–313, October 2003.

[67] M. E. Tuckerman. Ab initio molecular dynamics: Basic concepts, current trends and novel applications. *J. Phys. Condensed Matter*, 14:R1297, 2002.

[68] Tungsten IA32 Cluster. http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/XeonCluster/.

[69] Turing cluster. http://turing.cs.uiuc.edu/.

[70] Ramkumar Vadali, L. V. Kale, Glenn Martyna, and Mark Tuckerman. Scalable parallelization of ab initio molecular dynamics. Technical report, UIUC, Dept. of Computer Science, 2003.

[71] M. S. Warren and J. K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing 92*, November 1992.

[72] Terry Wilmarth and L. V. Kalé. Pose: Getting over grainsize in parallel discrete event simulation. In *2004 International Conference on Parallel Processing*, pages 12–19, August 2004.

[73] Yuanyuan Yang and Jianchao Wang. Efficient all-to-all broadcast in all-port mesh and torus networks. In *The Fifth International Symposium on High Performance Computer Architecture*, 1999.

[74] Yuanyuan Yang and Jianchao Wang. Near-optimal all-to-all broadcast in multidimensional all-port meshes and tori. *IEEE Transactions on Parallel and Distributed Systems*, 13(2), 2002.

[75] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004.

Author's Biography

Sameer Kumar was born and grew up in a small town in central India. He received the B. Tech. degree in Computer Science from Indian Institute of Technology Madras, India in 1999.

Sameer earned an MS degree in Computer Science from the University of Illinois at Urbana Champaign in 2001. His Master's thesis was on the design of an *Adaptive Job Scheduling for Timeshared Parallel Machines*.

On completion of the MS program, Sameer started working on communication optimizations by first designing a machine layer for the Charm runtime system on top of Quadrics QsNet at the Pittsburgh's Lemieux machine. A paper co-authored by him on scaling the molecular dynamics program NAMD was one of the winners of the Gordon Bell award in SC2002, using this machine layer. Sameer later worked on the communication optimization framework, which became the core of his thesis.

On completion of his PhD, Sameer will join the IBM T. J. Watson Research Center on a post doctoral position.