Opportunities and Challenges of Modern Communication Architectures: Case Study with QsNet

Sameer Kumar, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
{skumar2, kale}@cs.uiuc.edu

Abstract

We describe our efforts to scale message driven applications to a large number of processors on an Alpha cluster interconnected by QsNet. The clustering technology QsNet has a network interface with a communication co-processor. The presence of the co-processor minimizes main processor participation in message passing. We show the advantages of the communication co-processor for message driven applications. To scale fine-grained message driven applications to a large number of processors we had to overcome several hardware, software and operating system hindrances. We describe them in detail and present solutions for them. We use NAMD, a molecular dynamics program, as a case study for many of our performance optimizations.

1 Introduction

QsNet [12, 13] is a high bandwidth low latency clustering technology from Quadrics[15], that has been widely deployed. Several of the top 500 machines are built upon it. Pittsburgh's Lemieux [11], interconnected by QsNet, is an Alpha cluster with 750 Compaq ES45 nodes (3000 processors) and a peak performance of over 6 TF. In this paper, we present the challenges and opportunities we identified during our efforts to scale message driven applications to a large number of processors of Lemieux.

Along with providing low latency and high bandwidth for messages, QsNet has a programmable network interface called *Elan*. The Elan network interface has a communication co-processor and a remote DMA engine. The communication co-processor relieves the main processor from most of the work while the message is in flight.

In our past work, through simulation studies, we had shown [3] that *message-driven execution* can exploit such communication co-processors much more effectively than traditional MPI-style message-passing. In this paper (section 4) we substantiate this assertion with application benchmark data using Charm++ [7], our asynchronous message driven system.

Processor virtualization is the key concept in Charm++: the computation is divided into a large number of chunks (virtual processors or VPs) by the programmer, that are assigned to the processors by an adaptive runtime system. As a result, Charm++ supports dynamic load balancing [5] (by migrating VPs), automatic check-pointing, automatic out-of-core execution, ability to change the number of processors used by the application [8], optimized collective communications [9], etc. From the point of view of this paper, processor-virtualization leads to message-driven execution: since there are multiple objects on a processor, there must be a scheduler that decides which one of them executes next. It schedules an object when there are messages available for that object (thus the scheduling is driven by messages). This allows for an adaptive overlap of communication and computation.

We have developed a machine layer for Charm++ (described in Section 3) that can scale to a large number of processors on Lemieux, as demonstrated by applications like NAMD [14] and CPAIMD [18]. NAMD is a molecular dynamics simulation program, used routinely by biophysicists, that has over 1 TF of peak performance on Lemieux (A paper on NAMD was awarded the Gordon Bell award at SC'02). NAMD is completely written in Charm++. CPAIMD is a quantum chemistry application that has impressive performance on Lemieux.

NAMD is a challenging application because the amount of computation to be parallelized in each time step is relatively small — each time-step finishes in around 11 ms on 3000 processors. To achieve high performance in this fine-grained situation, we had to overcome several challenges with the QsNet-Alpha system. They were related to hardware deficiencies of the QsNet network interface, software issues in implementation of Elan libraries, and interference from the OS. In this paper, we describe these issues and the

techniques we used to overcome them.

We begin with an overview and basic performance characteristics of QsNet.

2 Performance Evaluation of QsNet

QsNet is based on two building blocks, a programmable network interface called *Elan* and a low latency high bandwidth switch called *Elite*. The Elan network interface has a SPARC processor that can run four threads. These threads can be used to implement a high level message passing system. The Elan network interface also has a DMA engine which is capable of accessing remote memory. The DMA and the co-processor together minimize main processor participation in communication operations. This results in a short MPI message latency of about $4.7\mu s$. The Elite switches can support a bandwidth of 400 Million Bytes/s (or 381 MB/s) each way, though protocol overheads bring this bandwidth down to about 335 Million Bytes/s (319 MB/s) [13] each way. (1 MB = 1048576 bytes).

In this section, we evaluate the performance of message latency and network bandwidth of QsNet, specifically with respect to message driven execution.

2.1 Message latency

The Charm++/Converse [6] runtime system has a scheduler which executes handlers on message arrival (Section 3). These handlers may execute user level code and take milliseconds to finish. Control is returned to the scheduler only after the handler has finished executing. So all arriving messages should have receives posted for them, or else unexpected handling of messages would lead to loss of performance. The Elan NIC receives an unexpected message (a message for which there is no receive posted) by first allocating a memory buffer and then receiving the message in that buffer. When the runtime posts a receive for that message, there may be an additional copying overhead too. So the communication performance worsens when incoming messages do not have receives posted for them, as demonstrated by Table 1. Here the cost of a memory copy from the unexpected buffer to the user buffer is clearly evident.

Msg. Size(b)	Converse	Conv. Unexpected Msgs.
1024	17.3	22.8
4096	29.5	46.9
16384	72.1	144.9

Table 1. Converse Latency (μs)

However, posting several receives may also negatively affect the performance of the network interface. To study the affect of preposted receives, we ran the pingpong benchmark (both in MPI and Converse) with preposted receives. For MPI, these receives were posted with a tag that was *different* from the one used for the pingpong. The results of this experiment are presented in Table 2. Here, each latency value is half the round trip time of pingpong. The least message latency for Converse is $6\mu s$. It is a little more than $4.7\mu s$ for MPI. The additional overhead for Converse is due to timer calls and scheduling overheads in the runtime system.

Msg. Size(b)	#Recvs.	Converse	MPI	Conv. (SHM)
16	1	6.02	4.69	6.5
16	5	7.27	5.66	8.17
16	9	8.34	6.93	9.24
16	17	10.9	10.9	11.5
16	33	18.5	17.8	19.3
64	1	7.27	5.96	7.63
64	5	8.28	7.23	9.54
64	9	9.48	8.20	10.4
64	17	12.0	12.3	12.9
64	33	19.7	19	20.6
256	1	9.89	8.89	10.4
256	5	11.1	10.0	12.5
256	9	12.2	11.2	13.4
256	17	15.3	15.4	16
256	33	22.9	22.7	23.7

Table 2. Latency (μs) vs No. of recvs. posted

As the number of receives posted increases from 1 to 33, the 16 byte message latency increases from $6\mu s$ to $18.5\mu s$ for Converse and from $4.7\mu s$ to $17.5\mu s$ for MPI. We believe that the increase in message latency is because the NIC loops over a list of posted receives on message arrival. The Elan NIC processor has a clock speed of 100MHz and traversing a list can be expensive. Moreover, for a large number of receives posted the NIC also starts experiencing cache misses. This further increases the message latency. The cache performance of the NIC is shown in Table 3. These results have been obtained from performance counters in the Elan NIC [17, 16]. The results indicate the number of cache misses for 150K messages of sizes between 16 and 256 bytes. Hence there is a tradeoff between posting more receives and running the risk of unexpected message handling.

In order to improve the performance of intra-node communication we use the Elan shared memory library. However, using this library further affects the message latency because a lock is needed while receiving a message. Message latencies with shared memory enabled are also shown in Table 2. The MPI results presented in Table 2 should actually be compared with these Converse latencies.

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

Table 3. Receives Posted vs Cache Misses

Msg. 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 4. Converse Latency vs CPU Overhead

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 pingpong with each processor exchanging messages with the corresponding processor in the remote node. Table 4 [1] shows the message latency for PPN=1 and PPN=4. Here 9 receives posted by each processor. Observe that the message latency is much higher when PPN=4, $17\mu s$ for a 16 byte message compared with $9.5\mu s$ for PPN=1. Along with node contention by the 4 processors, the latency is further increased because there are 36 receives 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 4 presents the CPU overhead of the pingpong 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. Notice from Table 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 into the network interface and then sent from there, thus incurring a higher CPU overhead.

[1] The latency reported in Table 4 is slightly more than that reported in Table 2. This is because of the additional timer overhead of computing the CPU overhead and idle time in the pingpong benchmark.

Msg. 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 5. Converse with two way traffic

PPN=1	Main-Main	Elan-Elan	Elan-Main
One Way Traffic	290	319	305
Two Way Traffic	128	319	305

Table 6. Elan Node Bandwidth (MB/s)

Normally parallel applications tend to have bi-directional traffic. So we computed the CPU overheads and latencies with bi-directional traffic. The results are presented in Table 5. For applications that have bi-directional traffic, 4 processors per node and many irrelevant receives posted, the latency for short messages could be 27 μs. This latency is significant. Section 5.1 describes mechanisms to manage this large latency.

2.2 Bandwidth

The Elite network can support an application bandwidth of 319 MB/s in each direction, which is only achievable if messages are sent from Elan memory. PCI contention brings the main memory bandwidth down to about 290 MB/s. Further when processors are simultaneously sending and receiving, this bi-directional traffic brings the bandwidth down 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 6 shows the achievable network bandwidth for different placements of the sources and destinations of the messages. Notice that sending messages from Elan memory is faster. To send messages from Elan memory, the message would have to be copied into Elan memory. We can use DMA to copy a message into Elan memory, with a bandwidth of about 305MB/s. This memory copy overhead can nullify the advantage of sending the message from Elan memory, except in specified situations (Section 5.2).

Bandwidth to distant nodes: We also observed that the Elite network bandwidth drops to far away nodes. This has also been reported in [2]. Table 7 shows the worst case bandwidth as a function of the size of the fat tree. The bandwidth is the lowest when messages go to the highest level of

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

Table 7. Elan Node Bandwidth (MB/s)

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. Hence large messages will have several packets, acknowledgments to which may be delayed if the nodes are far away [2].

3 Converse Machine Layer

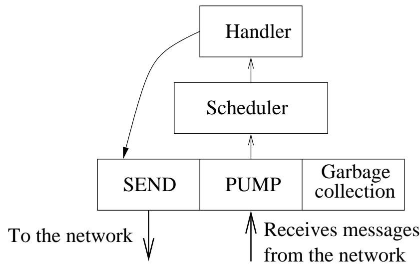


Figure 1. Converse System

Converse [6], the portability layer of Charm++, provides basic point-to-point communication. Converse also provides efficient collectives, and has been designed to be inter-operable with many languages. Prioritized message driven execution is enabled by the Converse's message-driven scheduler. The user program registers handlers with the Converse system. Each message carries the id of the destination handler. When the message is received on the destination processor, the Converse scheduler invokes the correct handler for that message. The *Converse machine interface* (CMI) currently implements point to point messaging, broadcasts and multicasts. On Lemieux (and QsNet) Converse is implemented on top of the Elan Tport [16, 17] tagged message passing interface. The CMI has three main components, (i) *Send module* which sends user messages, (ii) *Pump module* which pulls messages of the network by posting receives and periodically polling their handles and (iii) *Garbage collection module* which frees up user buffers after the message has been sent. Figure 1 shows the block diagram of the Converse runtime system.

The *send module* makes calls to the Elan Tport library to send user messages. There are three types of messages, identified by three different tags [2]. First are the short messages (sizes 4KB or below), that are sent eager by calling elan_tportTxSend. The next range of messages are the intermediate sized messages (less than 64KB), that are sent by calling elan_tportTxSend with a different tag. We need two different types of eager messages to optimize memory allocation on the receiver. Large messages are sent by first sending (as a small message) a header that contains the size and address of the message. The destination then RDMAs the message off the sender's memory.

The *pump module* posts receives for both short and intermediate sized messages. These receives are posted in an optimized circular queue, where only one receive for each tag is polled for completion. If the message received is a header for a large message, elan_get is used to pull it off the sender's memory. After receiving the message the pump module passes the message to the converse scheduler. The scheduler either queues the message or calls the appropriate message handler.

We have seen in the previous section that posting receives increases the message latency between nodes. But the pump module is only called between handler invocations. User handlers can take several milliseconds to finish. Hence several receives may need to be posted, or else messages may require unexpected message handling. Hence there will be a trade-off which can only be optimized in an application specific manner. For NAMD, this optimal number is about 10 posted receives.

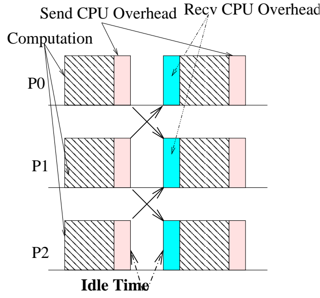
The *garbage collection* module polls the Elan system for the completion of the send operations. After the messages have been sent, the the garbage collection module frees up all buffers allocated to the messages.

4 Advantages of Co-Processor

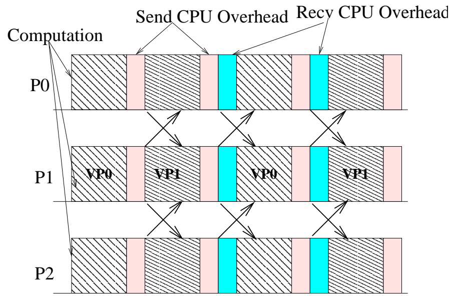
The presence of the co-processor in Elan makes the CPU overhead a small fraction of the total communication time. For example, a 4KB message takes 51 μs to send (Table 5), but the CPU overhead is only about 13 μs. In principle, the remaining time can be used for other computation. The advantage of this overlapping is demonstrated by a 3d 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.

In traditional MPI style of programming, this program would be written as computation followed by communication with no overlapping. The communication operation involves the exchange of 6 messages. Figure 2(a) demostrates this style of programming, though only two messages are

[2]For applications is written in Charm++ or AMPI [4], the runtime system hides Elan message tags and is free to use the entire Elan tag space



(a) Without Virtualization



(b) With Virtualization

Figure 2. Timeline for neighbor-exchange

shown. Here, the shaded rectangles show computation and the solid rectangles show the CPU overhead of communication (both send and receive overheads are shown). Observe that there is idle time between the computations. This scenario does not effectively utilize the low CPU overhead of Elan network interface.

In Charm++, with processor virtualization, the above program can have multiple virtual processors in each processor. After a virtual processor has sent its messages and is waiting for reply messages, another virtual processor can compute (Figure 2(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. Thus, effectively overlapping communication with computation. Also 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 or a short delay.

Table 8 shows the performance of the 3D 6-pt stencil benchmark on Lemieux (these results are based on our AMPI framework [4]). 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 8. Observe that NVP=8 performs better as it makes better use of the communication co-processor. Moreover, observe 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 larger number of processors, leading to shorter messages and a smaller difference between the total latency and CPU overhead.

Processors	NVP=1	NVP=8
8	193	105
64	16.4	12.7
512	6.5	5.7

Table 8. Time (ms): 3D stencil Computation of size 240^3 on Lemieux

5 QsNet Challenges

We next describe techniques we used to overcome the three main challenges posed by QsNet, namely (i) significant message latency, (ii) low memory to memory bandwidth, (ii) unexpected stretches in handlers.

5.1 Handling Latency: Message Combining

The message latency, with 4 processors per node and two way traffic, is about $27\mu s$ (Section 2.1). This high latency can restrict the scaling of some collective communication operations, e.g. all-to-all communication. We use message combining as our approach to handle this high latency. Message combining has been presented in literature before. We quote here a specific example from our past work [9].

We used message combining to optimize the performance of the NAMD Particle Mesh Ewald step. This step requires a 3d FFT which is implemented as 2d FFTs over planes followed by a transpose and then 1d FFTs. The transpose requires an all-to-all personalized communication operation. In [9] we have described virtual topologies and message combining to optimize all-to-all personalized communication for short messages. Table 9 (taken from our past work [9]) shows the advantage of virtual topologies in NAMD. Here APoA-1 and ATPase are bio-molecules being simulated. Observe that the Mesh strategy performs best.

The mesh strategy sends messages along a 2D-Mesh topology. In the first round of the mesh strategy, processors send messages along their rows. In the second round these messages are combined and sent to processors along the columns. So in the mesh topology, each processor only sends $2\sqrt{P}$ messages, as opposed to P messages in direct

all-to-all communication. Hence mesh has the best performance in Table 9.

APoA-1 uses a 108X108X80 sized grid while ATPase uses a 192X144X144 sized grid for the 3d FFT (PME) operation. Hence the all-to-all operation occurs over 108 processors for APoA-1 and 192 processors for ATPase. In our current implementation of NAMD, these PME virtual processors are distributed with as few of them on a node as possible. This gives the all-to-all operation the maximum available node bandwidth.

As shown in Table 9, the performance gain with the mesh strategy is best for 256 processors. But, since only 108(ApoA-1) or 192(ATPase) processors are being used for the all-to-all operation, the performance gain is not significant on 1024 processors.

Processors	ApoA-1(ms)		ATPase(ms)		
	Mesh	Direct	Mesh	Direct	MPI
256	39.2	44.4	113.6	120.8	134.5
512	23.4	28	60.8	63.0	69.5
1024	20.3	26.8	35.8	38.6	39.3

Table 9. NAMD step completion time (ms)

5.2 Better bandwidth for multicast

The send module in the CMI (Section 3) implements a multicast (CmiListSend) by first copying the message to the network interface and then send the message from there. In Section 2.2, we showed that sending messages from Elan memory more than doubles the node bandwidth, as compared with sending messages from main memory. The k-prefix strategy[3] for all-to-all broadcast(AAB) [10] sends messages from Elan memory by making calls to CmiListSend. The performance of the k-prefix strategy is presented in Table 10 (also taken from [10]). It outperforms Lemieux MPI which only sends messages from main memory.

5.3 Stretched Handlers

Another major issue we faced while scaling NAMD and other applications to large number of processors was that of *stretched* (delayed) entry methods (handlers for messages) [14]. Figure 3 shows the timeline of NAMD on 1536

[3]The k-prefix strategy is an all-to-all broadcast optimization strategy which uses two contention free permutations namely, prefix send (p exchanges messages with $pXORi$ in step i) and ring (p sends messages to $(p+1) \mod P$). In this scheme, the cluster of P nodes is partitioned into sub-clusters of size k. Prefix send is used for the AAB operation within sub-clusters while ring is used across the sub-clusters. This hybrid nature minimizes data exchange with far away nodes, enabling the strategy to scale to 256 nodes. The details of this strategy are presented in [10]

Nodes	Native MPI (MB/s)	k-Prefix (MB/s)
64	113	266
128	99	260
256	95	256

Table 10. AAB each-way node bandwidth

processors. Each grey rectangle on the processor time line shows the execution of a handler in NAMD. Observe that processors 900 (timeline 6 from the top) and 933 (timeline 7) have handlers that last about 20-30 ms. This is clearly shown by the long superscript bar (colored in light grey) on top of the handler. Normally these handlers should take about 2-3ms to finish, as shown by the remaining rectangles in Figure 3. Both the stretched handlers here block on a send operation, indicated by the superscript bar on top of the entry method. Observe that the other superscript bars are just dots. We also noticed other stretches in the middle of entry methods (not shown in the figure). We believe these stretches were caused by a mis-tuned Elan library and operating system daemon interference. We now mention the mechanisms by which we overcame this stretching problem.

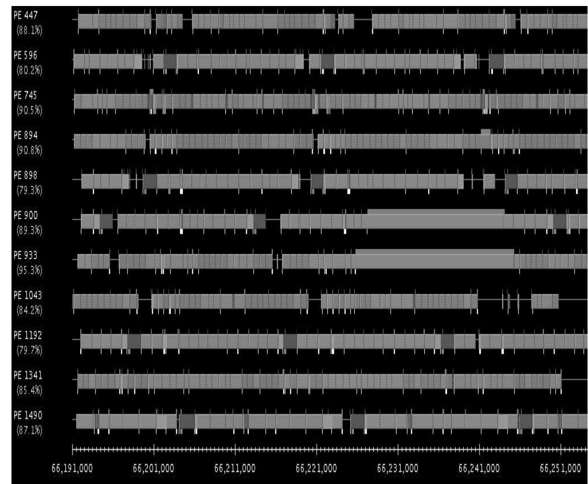


Figure 3. NAMD Run on 1536 processors

5.3.1 Stretched Sends

The Converse runtime system (Section 3) only makes calls to elan_tportTxStart (equivalent of MPI_Isend in Elan) which should be a short call. From Table 4 we know that the CPU overhead of pingpong is just a few microseconds. But the entry methods were blocked in the sends for tens of milliseconds.

On looking at the Elan library source (and also working with Quadrics), we found that this was a side effect of 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.

In order to implement this ordering, the Elan system made a processor block on an elan_tportTxStart if the rendezvous of any previous message had not been acknowledged, irrespective of the destination of that previous message. So in the presence of a hot-spot in the network, all processors that sent a message to the hot-spot 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 $ONE_QXD_PER_PROCESSOR$. Now, a message send only blocks if the previous rendezvous to *its* destination is unacknowledged. CMI keeps a list of processors with unacknowledged rendezvous, and buffers future messages to them till those rendezvous have been acknowledged. Thus, eliminating the stretched sends. This problem has been fixed in version 1.4 of Elan software.

5.3.2 OS Daemon Stretches

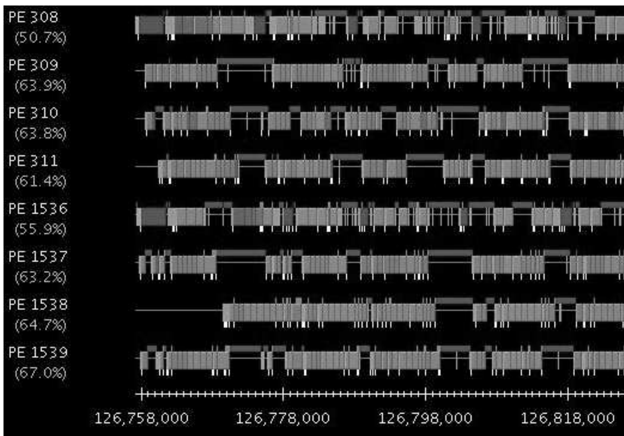


Figure 4. NAMD on 3000 processors

Fixing the Elan software did not completely eliminate stretches if the application used four processors per node. NAMD simulation of the ATPase system takes about 11ms on 3000 processors. This time step is very close to the 10ms time quanta of the operating system. So if on any of the 3000 processors a file system daemon is scheduled, NAMD step time could become 21ms.

Petrini et al. [1] 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 have been shutdown.

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

Table 11. NAMD with blocking receives

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 with two mechanisms. First, NAMD uses a reduction in every step to compute the total energies. With Charm++, it was able to use an asynchronous reduction, whereby the next timestep 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. Second, when a processor becomes idle, the *pump module* in the Converse machine layer *blocks* 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 4, where there are no stretched entry methods. Observe the dark grey superscripted rectangles on top of the black idle time, which implies 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 blocking. 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.

Most of the daemons (recommended in [1]) 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 11 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 after the daemons were shut down.

6 Summary and Future Work

QsNet is an excellent interconnect that has increased the performance of parallel applications considerably, as demonstrated by NAMD. It includes a communication coprocessor with remote and local DMA capabilities, which off-loads the task of communication from the main processor. We have demonstrated how a message-driven system like Charm++ (or Adaptive MPI) can take better advan-

tage of such co-processors, as compared with traditional compute-communicate-compute style used in many MPI applications.

To achieve good performance we had to overcome the challenges posed by QsNet. Under no-load conditions the communication latencies are as low as $4.7\mu s$ for short messages. But, in real-application contexts when multiple processors on each node are used and bi-directional communication ensues, these latencies are much larger. However, since the CPU overhead is still low in these contexts, a message-driven application can utilize the latencies for overlapping useful computation without additional programmer effort. Such relatively large latencies for short messages also motivated optimizations we implemented for message-combining with virtual topologies, especially for collective operations. We showed how we were able to use the higher bandwidth attained when sending from the co-processor's memory to optimize the all-to-all broadcast operation. We also discussed the problems faced when scaling applications with relatively fine-grained parallelism: stretched communication operations and OS interference. These were handled by changing communication protocols in the Elan library, allowing OS to use the processor when idle (by calling "sleep" instead of busy-waiting for a message) and use of asynchronous reductions (which is natural in message-driven programming).

Our current implementation of the runtime system on QsNet extensively uses tagged message passing. We believe that using one sided communication even for short and intermediate sized messages would improve the communication performance of applications, as it will reduce the processing load on the NIC. This is particularly useful when application communication patterns are persistent and buffers can be reserved on destinations for future messages. But, for this scheme to work, the application will have to guarantee that the buffers on the destinations are used before the next message arrives in that buffer. This new scheme is under investigation.

In summary, the capabilities offered by modern co-processors can significantly improve performance, but fully exploiting them requires sophisticated handling of communication operations. Further, message-driven execution and adaptive runtime systems can achieve high performance, without increasing programming complexity in this context by automatically adjusting to runtime conditions.

Acknowledgments We would like to thank David O'Neal (PSC) and David Addison (Quadrics) for their assistance. This work was supported by the National Institutes of Health (NIH PHS 5 P41 RR05969-04) and the National Science Foundation (NSF NFS 0103645, NSF CTR 0121357).

References

[1] S. P. 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.

[2] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll. STORM: Lightning-Fast Resource Management. In *Supercomputing 2002*, Baltimore, MD, November 2002.

[3] 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.

[4] C. Huang, O. 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.

[5] L. V. Kale, M. Bhandarkar, and R. 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.

[6] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

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

[8] L. V. Kalé, S. Kumar, and J. 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.

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

[10] S. Kumar and L. V. Kale. Scaling collective multicast on fat-tree networks. Technical Report 03-11, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003.

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

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

[13] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Performance Evaluation of the Quadrics Interconnection Network. To appear, Journal of Cluster Computing, 2002.

[14] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[15] Quadrics Ltd. http://www.quadrics.com.

[16] Quadrics Ltd. Elan Programming Manual. 1999.

[17] Quadrics Ltd. Elan Reference Manual. 1999.

[18] R. Vadali, L. V. Kale, G. Martyna, and M. Tuckerman. Scalable parallelization of ab initio molecular dynamics. Technical report, UIUC, Dept. of Computer Science, 2003.