Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems

Howard Pritchard, Duncan Roweth, David Henseler, and Paul Cassella

Abstract—Cray has enhanced the Linux operating system with a Core Specialization (CoreSpec) feature that allows for differentiated use of the compute cores available on Cray XE compute nodes. With CoreSpec, most cores on a node are dedicated to running the parallel application while one or more cores are reserved for OS and service threads. The MPICH2 MPI implementation has been enhanced to make use of this CoreSpec feature to better support MPI independent progress. In this paper, we describe how the MPI implementation uses CoreSpec along with hardware features of the XE Gemini Network Interface to obtain overlap of MPI communication with computation for micro-benchmarks and applications.

Index Terms—MPI, CLE, core specialization, asynchronous progress

1 Introduction

The importance of overlapping computation with communication and independent progress in Message Passing (MPI) applications is well known (see, for example [5], [9], [15]), even if in practice, many MPI applications are not structured to take advantage of such capabilities. Many different approaches have been taken since MPI was first standardized to provide for this capability, including hardware–based approaches in which the network adapter itself handles much of the MPI protocol [3], hybrid approaches in which the network adapter and network adapter device driver together offload the MPI protocol from the application [4], host software–based approaches to assist RDMA–capable, but MPI–unaware, network adapters [10], [18], as well as more generalized host software–based approaches which take advantage of modern multi–core processors [11], [19].

The Cray XE Gemini RDMA–capable network adapter has features intended to assist in the implementation of effective host software–based approaches for providing independent progression of MPI and for allowing for overlap of communication with computation. To provide for more effective implementation of such host software–based approaches, Cray has also enhanced the Cray Linux Environment (CLE) Core Specialization feature to facilitate management of host processor resources needed for this approach. This paper describes the combination of Gemini hardware features, the CLE Core Specialization feature, and enhancements made to MPICH2 to realize this capability.

The rest of this paper is organized as follows. First, an overview of the Core Specialization feature is presented. Features of the Gemini network adapter that are significant for this work are described in Section 3. Section 4 describes the approach Cray has taken with the MPICH2 implementation of MPI to realize better support for independent progress and communication/computation overlap. In sec-

• *The authors are with Cray, Inc.*
E-mail: howardp,droweth,dah,cassella@cray.com

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

tion 5 results using a standard MPI overhead benchmark are presented, as well as results obtained for two MPI applications. The paper concludes with a discussion of future work planned for the Core Specialization feature in CLE, as well as improvements to MPICH2 and possible extensions to the GNI API to better support MPI independent progress.

2 Core Specialization Overview

Cray has enhanced the Linux kernel delivered as part of CLE to allow for dynamic partitioning of processors (cpus) on nodes of Cray XE/XK systems into two groups – one group dedicated to application threads, and the other dedicated to system services. This partitioning is selectable by the user at application launch time. Note this partitioning scheme does not prevent compute intensive applications from being able to use all available cpus on the node if so desired. A prototype implementation of this Core Specialization (*CoreSpec*) feature for the Cray XT is described in [13].

An initial goal for this *CoreSpec* feature was the reduction of the impact of system–related noise on the performance of noise–sensitive parallel applications. The basic idea is that by reserving one or more of the cpus on each node for system services daemons and kernel threads, noise sensitive applications can perform significantly better using the remaining cpus, which are now dedicated exclusively to the application processes. Note that in addition to system daemons and kernel threads, interrupts from the network adapter are also directed toward one or more of the cpus within the set of cpus reserved for the operating system. Significant improvement in the runtime for the POP ocean model application on a Cray XT system was demonstrated when using the prototype *CoreSpec* feature described in [13].

It was realized early on in the design of the Cray XE, that *CoreSpec* could also be used for other purposes. Unlike the earlier Cray XT systems, the Cray XE was introduced when the multi–core era of X86_64 processors was already in full swing. Indeed, except for a few special systems, all Cray XE compute nodes have at least 16 cpus per node, with the latest Interlagos–based systems having 32 cpus per node. The number of cpus per node is expected to continue to increase. However, the ability of most HPC applications to use all the cpus on the node effectively is unlikely to continue, partly owing to the decreasing amount of memory and memory bandwidth per cpu, as well as increased sharing of cpu resources with other cpus on the same compute unit[1]. These otherwise unused cpus are ideal for use by any threads responsible for progress of the MPI state engines of the application processes on the node.

The prototype implementation Cray XT version of *CoreSpec* was enhanced to support this new usage model. A capability was added to allow for a thread to inform *CoreSpec* to schedule it on the system services core(s). Threads that spend large amounts of time descheduled, waiting on interrupts from network devices are ideal candidates for scheduling on the system services cpus. The *job* kernel module package [17] was also enhanced with an extension to allow for MPI to create extra progress threads without interfering with, or knowing about, the placement policy requested by the application at job launch.

Some of the functionality of *CoreSpec* described here could, in principal, be implemented using existing kernel interfaces available to user–space applications. However, in the context of complex, layered software, a more specialized kernel placement mechanism allows for simpler usage by various independent software packages which an application may be concurrently using.

3 Gemini Hardware Features for Supporting MPI Asynchronous Progress

The Cray XE Gemini network adapter [2] has several features to help support host software–based MPI independent progress mechanisms. The most important of these is a multi–channel

1. A compute unit refers to a group of cpus which share processor resources. An examples is the AMD Interlogos compute unit, in which two cpus share, among other resources, a floating point unit.

DMA engine. Transaction requests (TX descriptors) submitted to the DMA engine can be programmed to select for generation of a local interrupt at the originator of the transaction, generation of an interrupt at the target of the transaction, or both, when the transaction is complete. A TX descriptor may also be programmed to delay processing of subsequent TX descriptors in the same channel until the transaction is complete. Complete in this context means that for a RDMA write transaction, all data has been received at the target, and for a RDMA read transaction, all response data has been received back at the initiator's memory. In addition to the DMA engine, Gemini Completion Queues (CQ) can be configured for polling or blocking (interrupt–driven) mode. The Gemini also provides a low–overhead mechanism for a process operating in user–space to generate an interrupt and Completion Queue Event (CQE) at a remote node.

Some preliminary investigations of the overhead for using blocking CQs in conjunction with the low–overhead remote interrupt generation mechanism using the GNI [1] *GNI_PostCqWrite* and *GNI_CqWait* functions showed that when the processes were scheduled on the same cpu where the Gemini device driver (*kgni*) interrupt handler was run, a ping–pong latency of 3–4 μsecs was obtained. When the processes were run on cpus other than the one where the *kgni* interrupt handler was run, the latency increased to 7–8 μsec. These times were obtained on AMD Interlagos 2.3 GHz processors. The measured wake–up times, particularly when the process being made runnable by the *kgni* interrupt handler runs on the same cpu as the interrupt handler, were small enough to warrant pursuing an interrupt–driven approach to realizing MPI asynchronous progress.

4 Implementation of Progress Mechanism in MPICH2

4.1 Phase One

The MPICH2 for Cray XE systems utilizes the Nemesis CH3 channel [6]. A Nemesis Network module (Netmod) using the GNI interface was implemented for the Cray XE [14]. This initial GNI Netmod implementation did not provide an explicit mechanism for MPI asynchronous progress. The MPICH2 1.3.1 code base from Argonne was used for implementing changes in MPICH2 to support MPI asynchronous progress on Cray XE systems. This version of MPICH2 has effective support for MPI-2 MPI_THREAD_MULTIPLE. Although a global lock is used for protecting internal data structures, there are yield points within the MPI library where threads blocked waiting for completion of sends and receives yield the lock to allow other threads into the MPI progress engine.

A primitive asynchronous progress mechanism already exists in MPICH2, and is available when MPICH2 is configured for runtime selectable MPI-2 thread safety support. The method uses an *active* progress thread which posts a blocking MPI receive request on an MPICH2 internal communicator at job startup. The thread then goes into the MPI progress engine, periodically yielding a mutex lock to allow application threads to enter the MPI state engine. At job termination, the main thread in each MPI rank then sends the message to itself which matches the posted receive on which the progress thread is blocked. This causes the progress thread to return from the blocking receive, allowing MPI finalization to gracefully clean up any resources associated with the progress thread. This active progress thread approach only works satisfactorily if each MPI rank has an extra cpu for its progress thread. In addition to this problem, the extra thread adds significantly to MPI message latency, especially for short messages, since there is significant contention for the mutex lock. Note the progress thread is described as *active* because, except at points where it is trying to acquire a mutex lock, it is scheduled and running from the perspective of the kernel. Cray decided to take a different approach, utilizing the Gemini features mentioned above to avoid the use of *active* progress threads, and instead rely on progress threads that are only scheduled (woken up) when interrupts are received from the Gemini network interface indicating there is MPI message processing to be done.

There were several goals for the method used to enable MPI asynchronous progress in the MPICH2 library. One was to avoid to the greatest extent possible negatively impacting the message latency and message rate for short messages. Some degradation is unavoidable as long as a progress thread approach is used, owing to the need for mutex lock/unlocks in the path through MPI. Another goal was not to focus on optimizing only specific message delivery protocols, but to use the progress thread in a way that progresses the MPI state engine's non–cpu intensive tasks. For the first phase of this effort, another goal was to keep to a minimum the additional complexity needed to support the feature.

The first goal was met in two ways. A given rank's progress thread is only woken up when progress is needed on messages large enough to use the rendezvous (LMT) protocol [12]. For smaller messages, the progress thread is not involved. The second goal was realized by using the local/remote interrupt capability of the Gemini DMA engine in a manner that allows for both RDMA read, as well as cooperative RDMA write LMT protocols 4.1.2 to be handled by the progress threads. The third goal was realized by retaining significant parts of the existing MPICH2 progress thread infrastructure – and thus avoiding some of the complexities of more generalized on–load solutions [11], leveraging the relatively mature MPI–2 thread safety support in MPICH2, and using the Gemini DMA engine in a way that avoids the need for additional locks within the GNI Netmod itself. Also to keep things simple, no support was added to facilitate asynchronous progression of intra–node messages. The following sections detail changes to the GNI Netmod to support MPI asynchronous progress.

4.1.1 MPI Initialization

If the user has set environment variables indicating that the asynchronous progress mechanism should be used, MPICH2 early on in startup configures itself for MPI-2 thread support level MPI_THREAD_MULTIPLE, initializing mutexes and structures related to managing of thread private storage regions. As part of the GNI Netmod initialization, each MPI rank creates a *blocking* receive (RX) CQ, in addition to the CQs described in [14]. Each rank also creates a progress thread during the GNI Netmod initialization procedure. The *CoreSpec* related procedures described in Section 2 are taken to insure that the progress threads are schedulable only on one of the OS cpus if the *-r CoreSpec* option was specified on the *aprun* command line when the job was launched. The progress thread then enters a *GNI_CqWait* call where it is descheduled and blocks inside the kernel, waiting for interrupts from the Gemini network adapter. While blocked in the kernel, these helper threads consume no cpu cycles and don't interfere with the application threads.

4.1.2 Rendezvous (LMT) Protocol

In addition to changes in the GNI Netmod initialization procedure, the other major enhancements to MPICH2 were in the management of the rendezvous or LMT protocol. As described in [14], both RDMA read and cooperative RDMA write protocols are employed. For the RDMA read protocol, the sender rank should optimally be notified when the receiver has completed the RDMA read of the message data out of the sender's send buffer, in other words, when the receiver sends a Nemesis LMT *receive done* message. On the receiver side, the receiver needs to be notified when the sender sends a *ready to send (RTS)* message, as well as when the RDMA read request it posts to read the data out of the sender's send buffer has completed. Similarly for the RDMA write protocol, the sender rank needs to be notified when the receiver sends a *clear to send (CTS)* message and when the RDMA write has completed for delivering the message data from it's send buffer to the receiver's receive buffer. The receiver rank needs to be notified when a *RTS* message arrives from the sender, and when a *send done* is received from the sender. For long messages, multiple *CTS*, *RTS*, *send done*, and RDMA writes are required to deliver the message.

In order to allow for asynchronous progress for both transfer types, a generalized approach for waking up the progress threads at both the receiver and sender side is required. For both cases, the progress thread on either side

needs to be woken up when a relevant control message arrives. Rather than complicate the existing method for delivering the control messages, the control message itself is sent using the same approach as is taken when there is no progress thread. In addition to the control message, a remote interrupt message is also delivered to the target behind the control message using the *GNI_PostCqWrite* function. This has the effect of waking up the progress thread of the targeted rank once the LMT control message has been delivered. The progress thread then returns from *GNI_CqWait* and invokes the MPICH2 internal *MPID_nem_network_poll* function to progress the state engine and process the control message.

The other progress point to handle is how to receive notification that the Gemini DMA engine has completed processing a TX descriptor associated with a message using the LMT path. For phase one, to keep things simple, only a single RX CQ is used by each progress thread. This means, however, that the option to generate a local interrupt when the DMA engine has processed the TX descriptor is not available. Instead, for every TX descriptor posted to the DMA engine which is used to transfer data for an LMT message, a second TX descriptor is posted which does a 4–byte dummy write back to the initiator of the transaction. This second TX descriptor is programmed to generate a remote interrupt. Note that since the second descriptor is actually only doing a dummy write back to itself, the remote interrupt is being sent to the initiator of the DMA transaction. The first TX descriptor, the one that actually moves the data, is also marked with the GNI_RDMAMODE_FENCE rdma mode bit. Thus, the arrival of the remote interrupt back at the initiator means that the first TX descriptor is complete as defined in Section 3. This modification to the use of the DMA engine has the advantage that the progress thread only needs to block on a single RX CQ, and more importantly, that the processing of the first TX transaction's CQE is not delayed by the time it takes for the progress thread to wake up and process the CQE. The wake–up is purely optional. If the thread detects that an application thread is already in the MPI progress engine, it can just immediately go back into the *GNI_CqWait* call. The principal downside of this usage model is that there is a stall for every TX descriptor with the GNI_RDMAMODE_FENCE bit set. Since the fence blocks for all network responses to return before proceeding to the next TX descriptor, this stall can be significant, particularly for distant target nodes and in cases where the network is heavily congested.

This phase one feature is available in the MPICH2 packaged as part of the Message Passing Toolkit (MPT) 5.4 release.

4.2 Changes to MPICH2 - Phase Two

The main goal with the phase two portion of the asynchronous progress feature is improving the mechanism for progression of DMA transactions by removing the use of the GNI_RDMAMODE_FENCE bit from the TX transactions which move message data, as well as dispensing with the dummy TX transaction used to generate an interrupt at the initiator. By removing the need to use the GNI_RDMAMODE_FENCE bit and the extra TX descriptor, three significant performance advantages are realized, at least in theory:

1) Since the fence mode is no longer required, multiple BTE channels can be used by the application,
2) The stalls introduced into the DMA engine TX descriptor pipeline are eliminated,
3) The number of TX descriptors that need to be processed is cut in half.

In order to generate local interrupts upon completion of a TX descriptor associated with a message using the LMT path, an additional blocking TX CQ must be created during the GNI Netmod initialization procedure. The progress threads now have to block on two CQs, which can be accomplished using the *GNI_CqVectorWaitEvent* function. In addition to these changes, a more complex CQE management system is required, since now the progress threads will need to store CQEs recovered from the new blocking TX CQ in a way that both the progress threads, as well as any application threads that are in the

MPI progress engine, can process these CQEs. Currently in the phase two implementation, a linked list of CQEs, protected by a mutex, is used for storing these CQEs. When a progress thread is woken up due to a CQE on the blocking TX CQ, the dequeued CQE is added to this linked list. If the application thread is within the MPI progress engine already, the asynchronous progress thread simply goes back to blocking in the kernel by again calling *GNI_CqVectorWaitEvent*. Application threads already in the MPI progress engine then pick up the CQE off of this linked list and process them in a manner analogous to that used to process CQE's returned from the existing non–blocking TX CQ.

Phase two is still very much a work in progress. As will be seen in the results section, the need for the progress thread to explicitly dequeue CQE's from the blocking TX CQ has a significant impact on MPI performance if the application threads do not have sufficient work to completely overlap the communication time.

5 RESULTS

5.1 Micro–benchmark Results

The CPU overheads of sending and receiving MPI messages were measured with the Sandia SMB test [16] using the *post–work–wait* method described in [7] where the overhead is defined to be:

> ...the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.

Application availability is defined to be the fraction of total transfer time that the application is free to perform non-MPI related work. In terms of the Sandia micro–benchmark and the following figures:

$$overhead = iter\_t - work\_t \quad (1)$$

and

$$availability = 1 - \frac{iter\_t - work\_t}{base\_t} \quad (2)$$

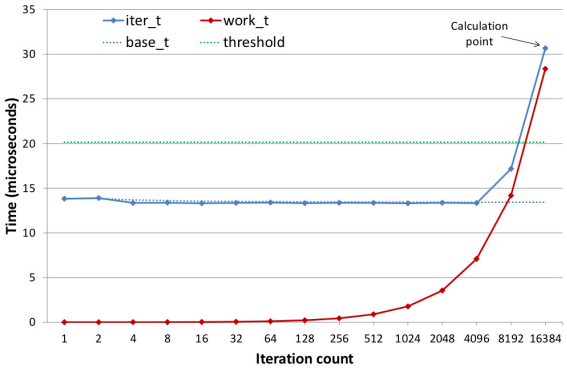

Fig. 1. Calculating sender availability on 16K messages. Communication overhead is low; CPU availability is measured at 83%.

The test measures the time to complete a non-blocking *MPI_Isend* (or *MPI_Irecv*) of a given size, some number of iterations of a work loop and then an *MPI_Wait*. The test is repeated for increasing numbers of iterations of the work loop until the total time taken (*iter_t*) exceeds the time for the communication step alone (*base_t*) by some threshold. Examples of this calculation are illustrated for 16 Kbyte transfers in Figures 1 and 2 in which sender availability is measured to be 83% and receiver availability 20%.

The SMB test repeats this measurement reporting sender and receiver CPU availability for increasing message size (see Figure 3). The results shown in the figure are baseline values without MPI asynchronous progression enabled.

The characteristics of each of the three MPI protocols are clear. The small message case requires CPU time for each message on both the sender and receiver. For intermediate size messages where the LMT read protocol is used, sender availability increases as the receiver fetches data independently. The receiver however consumes increasing amounts of CPU time as the message size increases. The large message LMT cooperative put protocol provides no overlap of computation and communication, sender and receiver must both be in MPI calls to make progress.

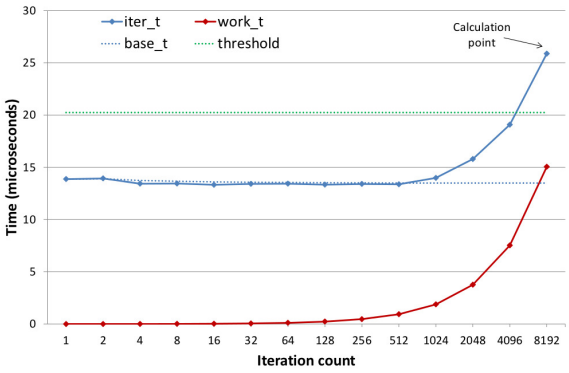

Fig. 2. Calculating receiver availability on 16K messages. Communication overhead is high; CPU availability is measured at 20%.

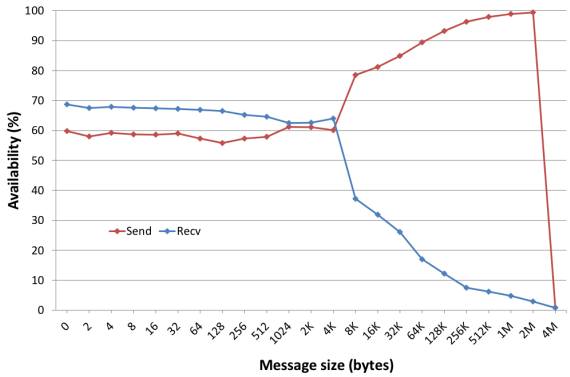

Fig. 3. Application availability as a function of message size measured using the SMB overhead test with one process per node. Availability is shown for the sender (red) and the receiver (blue). Progression disabled.

These results are in line with expectations in that the user process must be in an MPI call in order to perform MPI matching or to advance the underlying GNI protocol. This restriction is not an issue when the application is making MPI calls at a high rate, but poor overlap on intermediate and large transfers has a negative impact on performance of some applications.

In Figure 4 we repeat the measurements using the phase one library. At small message sizes performance is much as before. Receive availability dips as we switch to rendezvous protocol at message sizes of 8K bytes and above. With progression enabled both send and receive side availability then increase steadily with message size.

In Figure 5 we compare the phase one and phase two implementations, focusing on intermediate size messages. The phase one implementation shows poor receive side availability for message sizes between 8K and 10K.

We can lower the threshold at which we switch to the rendezvous protocol. The effects of making this change are illustrated in Figure 6 below. We show send and receive size availability for intermediate message sizes with progression enabled and disabled. Receiver availability increases markedly once progression is enabled, and from there continues to increase gradually with message size.

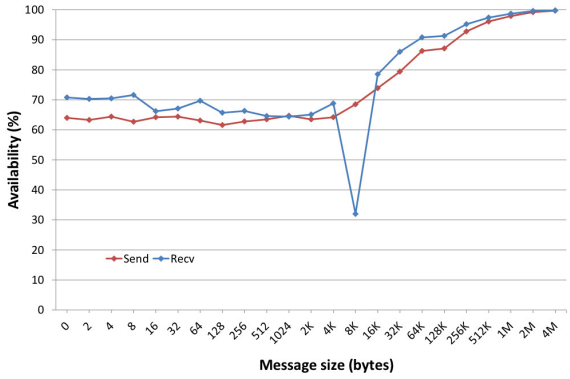

Fig. 4. Updated availability plot. Send and receive side availability are shown as a function of message size. Progression enabled, phase one.

In Figure 7 we show the impact on latency of enabling asynchronous progression. There is a jump in latency as we switch from eager to rendezvous protocol. By default the library makes this change at 8K bytes so as to minimize the effect. The jump in latency is more pronounced when asynchronous progression is enabled. Latency also increases with the number of processes per node. This is a result of the Gemini device only supporting a single user in-band interrupt. All progress threads must be woken when an interrupt is delivered.

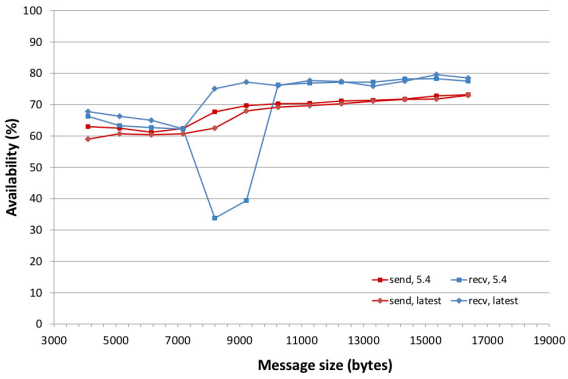

Fig. 5. Comparison of CPU availability at intermediate message sizes for phase one and phase two implementations.

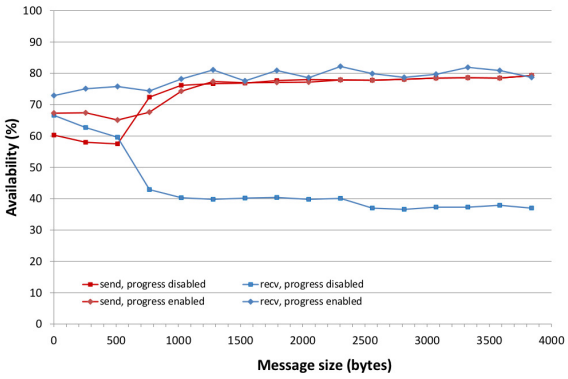

Fig. 6. CPU availability for intermediate message sizes. In these measurements the small message eager protocol is used for transfers of up to 512 bytes and rendezvous for all larger sizes.

The standard rendezvous algorithm performs message matching and then initiates a block transfer to move the bulk data. Our results show this working well for large message sizes, but pushing up latencies at small sizes. An alternative approach is use an FMA get at intermediate sizes rather than a block transfer. This reduces the latency, especially with large numbers of processes per node, but it also reduces CPU availability. In Figure 8 we show the impact on bandwidth of enabling progression at 1K and 8K message sizes. The eager protocol delivers good bandwidth at intermediate bandwidth, but CPU availability is relatively poor and main memory bandwidth is wasted copying from system buffers to user space. The rendezvous protocol with thread based progression delivers high bandwidth and high CPU availability at large message sizes, but reduced bandwidth at intermediate sizes. These effects will diminish as we reduce the overhead costs of thread based progression. The combination of the overhead plots and latency/bandwidth charts can be seen as a measure of success for progress mechanisms. *CoreSpec* was observed to significantly reduce the variability in the SMB benchmark measurements.

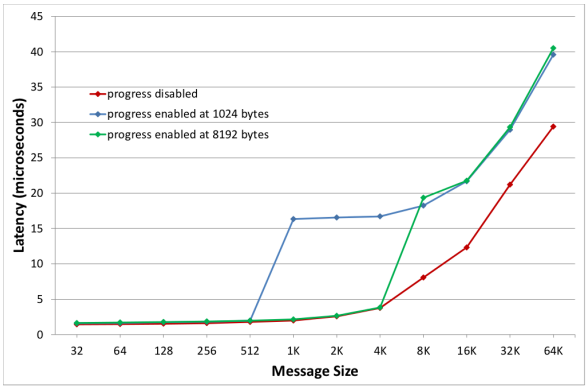

Fig. 7. Latency measurements with 4 processes per node. In these measurements the small message eager protocol is used for transfers of up to 512 bytes and rendezvous for all larger sizes.

5.2 S3D Application Results

S3D is a massively parallel direct numerical solver (DNS) for the full compressible Navier-Stokes, total energy, species and mass continuity equations coupled with detailed chemistry [8]. It is based on a high-order accurate, non-dissipative numerical scheme solved on a three-dimensional structured Cartesian mesh. S3D's performance has been optimized for increased grid size, more simulation time steps, and more species equations. These are critical to the scientific goals of turbulent combustion simulations in that they help achieve higher

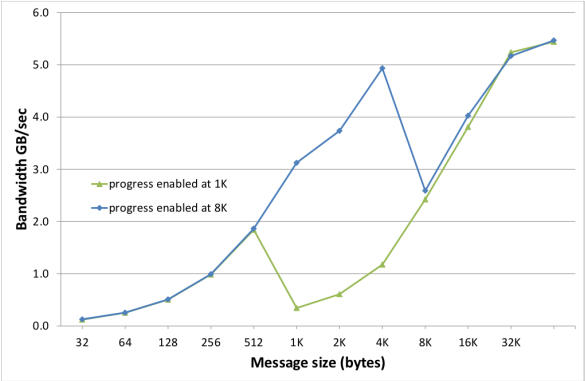

Fig. 8. Bandwidth measurements with 4 processes per node. In these measurements the small message eager protocol is used for transfers of up to 512 bytes (green) or 4K bytes (blue). Rendezvous protocol with progression enabled is used for all larger sizes.

TABLE 1
Profile of S3D without progression.

Time%	Time(secs)	–	Calls	Total
100.0%	333.55	–	2022296	Total
56.7%	189.06	–	605408	USER
38.3%	127.90	–	869027	MPI
23.7%	78.91	–	61200	mpi_waitall_
11.8%	39.43	–	6960	mpi_wait_
2.4%	7.938	–	6960	mpi_wait_
4.1%	13.73	–	546009	OMP

Reynolds numbers, better statistics through larger ensembles, more complete temporal development of a turbulent flame, and the simulation of fuels with greater chemical complexity. In addition, ORNL and Cray have spent the past year converting S3D into a hybrid MPI/OpenMP application capable of using MPI for inter–node data exchange, while using OpenMP within the node.

Profiles of this hybrid version of S3D running on Cray XE6 show MPI traffic dominated by non-blocking communications , with large amounts of time spent in wait functions.(see Table 1). Table 2 shows the corresponding message size profile. The use of large non-blocking messages makes S3D a good candidate for asynchronous progression.

TABLE 2
MPI message profile for S3D

MPI Msg Size	Msg. Count
< 16	1795
$16 - 256$	27
$256 - 4K$	100806
$4K - 64K$	2
$64K - 1M$	298681

TABLE 3
S3D Time Step Summary

# Application Threads	Progression disabled	Progression enabled
14	4.77	3.93
15	4.68	4.05
16	4.59	4.06

In Table 3 we show the S3D runtime in seconds per time step with 14, 15, and 16 threads per node. Our first set of measurements was performed with progression disabled. In the second set progression (phase one) was enabled, with 2, 1, or 0 cores dedicated to progression using the *CoreSpec* method. In the final measurement, with 16 application threads and progression enabled the interrupts generated by the progress mechanism will cause application threads to be descheduled. All tests were run on 64 nodes of a Cray XK system using AMD Interlagos processors. Best performance was optioned using 14 application threads using two cpus for the progress thread. A 14% reduction in overall runtime was achieved by enabling progression. The benefits of core specialization were relatively small for this hybrid MPI/OpenMP application, 3% of the 14% gain.

5.3 MILC7 Application Results

The MIMD Lattice Computation (MILC) code (version 7) is a set of codes developed by the MIMD Lattice Computation Collaboration for doing simulations of four dimensional SU(3) lattice gauge theory on MIMD parallel machines. Profiling of this application showed that for certain types of calculations, the MPI message pattern was dominated by small (2-4 KB) to medium size (9-16 KB) messages

TABLE 4
MPI message profile for S3D

MPI Msg Size	Msg. Count
< 8	16
$513 - 1K$	283736
$1K - 2K$	98304
$2K - 4K$	1167192
$4K - 8K$	35926
$8K - 16K$	948208
$16K - 32K$	107202

TABLE 5
MILC Run Time Summary(secs)

# Run Type	4096 ranks	8192 ranks
No progression	2165	1168
Progression (phase 1)	2121	1072
Progression (phase 2)	3782	2138
Progression (phase 1) no reserved cores	3560	2210
Progression (phase 1) reserve core but no corespec	2930	2070

at 8192 ranks (see Table 4). The application's use of MPI point to point messaging appears to lend itself to potential overlap of communication with computation. For purposes of investigating the effectiveness of the thread–based asynchronous progression mechanism, Cray benchmarking modified the application to gather timing data for the main MPI message gather/scatter kernel within the application. A *Asqtad R algorithm* test case was run with a lattice dimension of 64x64x64x144. Unlike the S3D application, MILC is a pure MPI code and is typically run with one MPI rank per cpu.

The application was run four different ways on a Cray XE6 using AMD Interlagos processors. Runs were made without MPI progression, with MPI progression using the phase one method and one cpu per node reserved by *CoreSpec*, with MPI progression using the phase two method and one cpu per node reserved by *CoreSpec*, and one set of runs with MPI progress using the phase one method but no cpus reserved for the progress threads. Table 5 gives the runtime in seconds for these different runtime settings. With *CoreSpec* reserving 1 cpu per node, the 8192 rank job requires 264 nodes as opposed to 256 nodes when running without *CoreSpec*. The modest runtime improvements when using the phase one progression mechanism correlate closely with the reduction in the average and maximum MPI wait time part of the gather/scatter operation. For example, at 8192 ranks, the average *MPI_Wait* time in the application's gather_wait function fell from 309 seconds to 270 seconds, and the maximum time fell from 719 seconds to 504 seconds using the phase one progress method. The modest reduction in the average wait time when using the phase one progress mechanism, and the much higher wait times when the phase two mechanism indicates that at least some of the ranks do not have sufficient work to fully mask the communication time. The message sizes are in the range where latency dominates, particularly when running a flat MPI application with many ranks per node. The bad performance of the phase two method for this application is most likely explained by the fact that the progress thread must wake up and dequeue CQEs before the main thread can make progress on completing a message. Since the application does not have sufficient computation to fully mask the communication time, the rank processes themselves have, in many cases, probably already entered the *MPI_Wait* function well before the message has been delivered. Thus the full overhead of the need to wakeup the progress threads in order to dequeue CQES from the TX CQ, plus contention for mutex locks, is encountered.

The times in the final row of Table 5, when compared with the times in the 2 row, shows that *CoreSpec* leads to significantly better results for the phase one progress mechanism than can be obtained simply be running a reduced number of MPI ranks per node and leaving an extra cpu available.

6 Conclusions and Future Work

The thread–based progression mechanism appears to have promise for use with applications on Cray XE and future architectures. The results presented in this paper show that

the phase one approach can be used with the class of MPI applications which tend to use larger messages in the 10–100 KB range. Hybrid MPI/OpenMP applications which tend to have larger messages and fewer intra–node messages are the most likely beneficiaries of this progress mechanism.

The preliminary results for the phase two approach have demonstrated that improved processor availability is realized for smaller messages when using the specialized MPI overlap test, although other micro–benchmarks and applications show a need to process more efficiently CQEs dequeued from CQs configured as blocking. Extensions to the GNI API to allow for more efficient CQE processing for such cases are currently being investigated.

Cray is also enhancing the *CoreSpec* mechanism to target compute units with hyper–thread support. In many cases, HPC applications cannot efficiently use all of the hyper–threads that a compute unit can support, yet these unused CPUs should work very well for MPI progress threads.

In addition to software improvements to the thread–based progress mechanism, follow–on products to the Cray XE6 will have additional features including many more in–band interrupts and lower–overhead access to the DMA engine, that will improve the performance of host thread–based MPI asynchronous progression techniques.

ACKNOWLEDGMENTS

The authors would like to thank John Levesque (Cray, Inc.) and Steve Whalen (Cray, Inc.) for the their help with the S3D and MILC applications.

REFERENCES

[1] *Cray Software Document S-2446-3103:Using the GNI and DMAPP APIs*, March 2011.

[2] R. Alverson, D. Roweth, and L. Kaplan. The Gemini System Interconnect. *High-Performance Interconnects, Symposium on*, 0:83–87, 2010.

[3] J. Beecroft, D. Addison, D. Hewson, M. McLaren, D. Roweth, F. Petrini, and J. Nieplocha. QsNetII: Defining High-Performance Network Design. *IEEE Micro*, 25(4):34–47, July 2005.

[4] R. Brightwell, K. Predretti, K. Underwood, and T. Hudson. Seastar Interconnect: Balanced Bandwidth for Scalable Performance. *Micro, IEEE*, 26(3):41–57, May-June 2006.

[5] R. Brightwell, R. Riesen, K. D. Underwood, R. Brightwell, R. Riesen, and K. D. Underwood. Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications. *Int. J. High Perform. Comput. Appl*, 19:103–117, 2005.

[6] D. Buntinas, G. Mercier, and W. Gropp. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. In *CCGRID'06*, pages 521–530, 2006.

[7] D. Doerfler and R. Brightwell. Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 331–338, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] E. R. Hawkes, R. Sankaran, J. C. Sutherland, and J. H. Chen. Direct Numerical Simulation of Turbulent Combustion: Fundamental Insights Towards Predictive Models. *Journal of Physics: Conference Series*, 16(1), 2005.

[9] J. W. III and S. Bova. Where's the Overlap? - An Analysis of Popular MPI Implementations, 1999.

[10] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman, and D. K. Panda. Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 185–193, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] P. Lai, P. Balaji, R. Thakur, and D. K. Panda. ProOnE: a General-purpose Protocol Onload Engine for Multi- and Many-core Architectures. *Computer Science - R&D*, pages 133–142, 2009.

[12] MPICH2-Nemesis. Nemesis Network Module API. wiki.mcs.anl.gov/mpich2/index.php/Nemesis_Network_Module_API.

[13] S. Oral, F. Wang, D. Dillow, R. Miller, G. Shipman, D. Maxwell, D. Henseler, J. Becklehimer, and J. Larkin. Reducing Application Runtime Variability on Jaguar XT5. In *Proceedings of Cray User Group*, 2010.

[14] H. Pritchard, I. Gorodetsky, and D. Buntinas. A uGNI-based MPICH2 Nemesis Network Module for the Cray XE. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 110–119, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications. *SC Conference*, 2006.

[16] Sandia Micro-Benchmark (SMB) Suite. www.cs.sandia.gov/smb/.

[17] SGI. Process Aggregates (PAGG). oss.sgi.com/projects/pagg.

[18] S. Sur, H.-W.Jin, L. Chai, and D. K. Panda. RDMA Read Based Rendezvous Protocol for MPI over Infiniband: Design Alternatives and Benefits. In *Symposium on PPOPP, March*, 2006.

[19] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A Multithreaded Communication Engine for Multicore Archi-

tectures. In *International Parallel and Distributed Processing(IPDPS)*, 2008.