2012 IEEE 26th International Parallel and Distributed Processing Symposium

A uGNI-based Asynchronous Message-driven Runtime System for Cray Supercomputers with Gemini Interconnect

Yanhua Sun, Gengbin Zheng, Laximant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois
{*sun51, gzheng, kale*}*@illinois.edu*

Terry R. Jones
Computer Science and Mathematics Division
Oak Ridge National Lab
Oak Ridge, Tennessee
trjones@ornl.gov

Ryan Olson
Cray Inc
ryan@cray.com

***Abstract*—Gemini, the network for the new Cray XE/XK systems, features low latency, high bandwidth and strong scalability. Its hardware support for remote direct memory access enables efficient implementation of the global address space programming languages. Although the user Generic Network Interface (uGNI) provides a low-level interface for Gemini with support to the message-passing programming model (MPI), it remains challenging to port alternative programming models with scalable performance.**

CHARM++ is an object-oriented message-driven programming model. Its applications have been shown to scale up to the full Jaguar Cray XT machine. In this paper, we present an implementation of this programming model on uGNI for the Cray XE/XK systems. Several techniques are presented to exploit the uGNI capabilites by reducing memory copy and registration overhead, taking advantage of the persistent communication, and improving intra-node communication. Our micro-benchmark results demonstrate that the uGNI-based runtime system outperforms the MPI-based implementation by up to 50% in terms of message latency. For communication intensive applications such as N-Queens, this implementation scales up to $15,360$ cores of a Cray XE6 machine and is 70% faster than the MPI-based implementation. In molecular dynamics application NAMD, the performance is also considerably improved by as much as 18%.

***Keywords*-Cray XE/XT, Gemini Interconnect, Asynchronous message-driven, Low Level Runtime System**

I. Introduction

Modern interconnection networks on parallel computers are complex systems. It is challenging to obtain good performance over the range of possible communication patterns that parallel applications exhibit. At the same time, there are multiple programming languages and models that are being developed as alternatives to the de facto industry-standard message passing interface (MPI), such as UPC [20], Chapel [3], X10 [4] and CHARM++ [9]. How should one go about attaining good performance for applications programmed in these alternative models?

In this paper, we focus on exploiting the performance of an asynchronous message-driven programming model CHARM++ and its runtime on Gemini. Gemini [1] is the new interconnect for Cray XE/XK systems; it is characterized by low latency, high bandwidth and strong scalability. A user-level Generic Network Interface (uGNI) [5] is provided as a low-level interface for Gemini, on top of which the Cray MPI is implemented. Gemini-based interconnects have been deployed in two petaflop supercomputers, Cielo at Los Alamos National Lab and Hopper at Lawrence Berkeley National Lab. Jaguar, one of the most powerful supercomputers, is being upgraded to a XK6 system with Gemini interconnect. The newly announced Blue Waters supercomputer, which will deliver sustained performance of 1 petaflop, will also be equipped with a Gemini interconnect. As Cray supercomputers with Gemini interconnects become one of the major platforms for large scale real-world science and engineering applications, it is important to port and scale existing runtime systems and parallel applications on the new interconnect.

One possible strategy is to implement the message-driven programming model on Cray's MPI implementation, which has been optimized for message passing using the low level features provided by the Gemini hardware. The advantage of this approach is the portability since MPI is supported ubiquitously on any supercomputing platform. In our previous work [11], we have demonstrated the success of this method. We have scaled a 100-Million-atom simulation up to 224,000 cores of the whole Jaguar machine using MPI-based CHARM++. However, we did observe some performance problems, for example, caused by prolonged MPI_Iprobe.

There are two reasons that make the MPI-based implementation inefficient. First, MPI supports synchronous communication and in-order message delivery as well as a large set of functionalities which are not required for implementing CHARM++, and these features potentially incur unnecessary overhead for CHARM++ on MPI. Second, the interaction patterns exhibited by CHARM++ programming model differ significantly from those in typical MPI applications. In the CHARM++ asynchronous message-driven programming model, communication does not require participation of both senders and receivers as in MPI. A sender initiates a send-like call with an outgoing message. When the message arrives on the receiver, it is queued for execution. This is the same as the active message model.

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.127

Moreover, CHARM++ manages its own memory allocation and de-allocation, while MPI implementation also internally manage its own memory. If CHARM++ is implemented on MPI, an extra memory copy between CHARM++ and MPI memory space may be needed.

Furthermore, when MPI is used as a layer above uGNI, it adds extra overhead to the software stack. As an experiment, we compared the one-way latency on a Gemini-based Cray machine using simple ping-pong benchmarks. The first data was obtained from the benchmark implemented directly on uGNI, the second from an implementation on MPI, and the third from a CHARM++ implementation based on MPI. The results are illustrated in Figure 1. It is clear that the MPI layer adds considerable overhead to the latency compared to the uGNI layer, and not surprisingly, CHARM++ runtime implemented on top of MPI performs even worse.

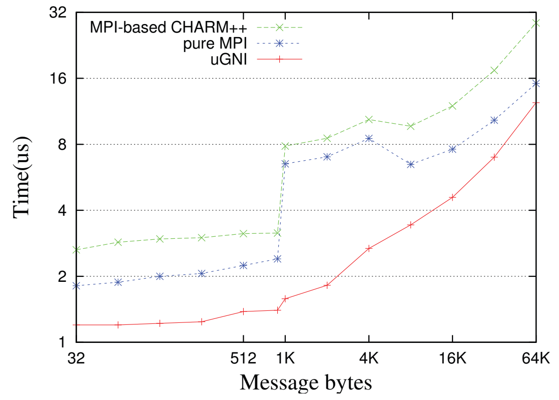


Figure 1. Comparison of ping-pong one-way latency in uGNI, MPI and MPI-based CHARM++

These factors motivated us to implement CHARM++ directly on uGNI, the lower level of the Cray network software stack. We first identified the functionality the CHARM++ runtime needs from from the low level communication library. We further defined an API for such required functionality in the form of a general and light-weight Low-level RunTime System (LRTS) interface to facilitate future porting efforts.

Remote direct memory access functions in uGNI require memory to be allocated and registered, which can be very expensive. To realize the potential of uGNI-based CHARM++ implementation, techniques are explored to take advantage of persistent communication, reduce memory copy and registration overhead, and improve intra-node communication. Although some of these optimization techniques are based on general ideas, they are deployed in the context of a message driven runtime system interfacing with the uGNI. The proposed techniques leverage the characteristics of the message driven programming model, e.g., message buffers are managed directly by the runtime, which allows the runtime to aggressively reuse the message buffers to avoid expensive memory operations. Our performance results on a Cray XE machine using several micro-benchmarks and real world applications demonstrate the success of this approach on uGNI. We believe that the LRTS interface and the optimization techniques are suitable to other target networks with remote memory access capabilities. It may also be useful for developers of other alternative programming models.

The remainder of the paper is organized as follows: Section II presents an overview of the Gemini and uGNI API. Section III describes the LRTS interface and its implementation on uGNI software stack. The optimization techniques to improve uGNI-based CHARM++ are presented in Section IV. Performance results are provided in Section V with both micro-benchmarks and real world applications. Finally, we discuss some related work in Section VI, and Section VII concludes the paper with future plans.

II. GEMINI AND UGNI API OVERVIEW

A. Gemini Overview

The Gemini interconnect [1] is the current state-of-the-art networking hardware used in the Cray XE and XK supercomputers. Using a system-on-chip design, the Gemini interconnect constructs a three-dimensional torus of connected nodes that has the capability of scaling beyond 100,000 multi-core nodes. The advantages of Gemini over the preceding Seastar2 interconnect are improved latencies and messaging rates, especially for small messages, and hardware support for one-sided messaging and atomic operations. The Gemini ASIC provides service for two AMD based Opteron nodes by connecting each node to one network interface controller (NIC) over a non-coherent HyperTransport(TM) 3 interface as shown in Figure 2. The data to/from the two NICs on the Gemini ASIC travel through the Netlink to a 48-port router which is capable of dynamically routing traffic on a packet-by-packet basis to fully utilize the links in the direction of the traffic.

There are two hardware features of the Gemini NIC for initiating network communication: the *Fast Memory Access* (FMA) unit and the *Block Transfer Engine* (BTE) unit. To achieve maximum performance, it is important for developers to properly utilize these two distinct components of the Gemini NIC.

FMA allows for direct OS-bypass enabling FMA communication to achieve the lowest latencies and highest message rates, especially for small messages. BTE transactions can achieve the best computation-communication overlap because the responsibility of the transaction is completely offloaded to the NIC. The BTE is most effective for messages greater than 4096 bytes. The crossover point between FMA and BTE for most application is between 2048 and 8192 bytes depending on the communication profile.

For users to interface with Gemini hardware, two sets of APIs are developed by Cray: User-level Generic Network Interface (uGNI) and Distributed Memory Applica-

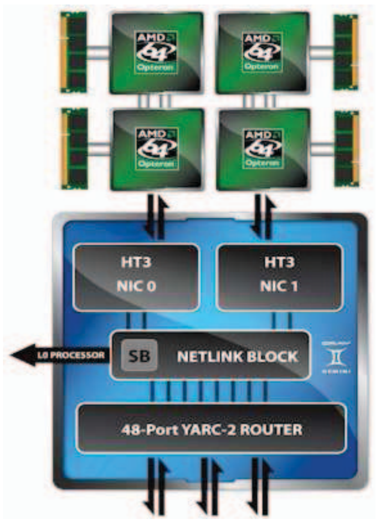


Figure 2. Gemini Hardware

tions (DMAPP). DMAPP is a communication library which supports a logically shared, distributed memory programming model. It is a good match for implementing parallel programming models such as SHMEM, and PGAS languages [20], [13]. On the other hand, uGNI is mainly designed for applications whose communication patterns are message-passing in nature. Therefore, we chose to target our message-driven machine layer on uGNI for the Cray XE network.

B. *uGNI API*

uGNI API defines a set of functions to transfer data using the FMA and BTE mechanisms described above. For small message communication, there are two types of messaging facilities available. They are GNI *Short Message* (SMSG) and *Message Queue* (MSGQ).

SMSG provides the best performance for short messages, but it comes at the expense of memory usage. It requires each peer-to-peer connection to create *mailboxes* for its both ends of the SMSG channel by allocating memory and registering it before the SMSG channel can be used to send messages. Therefore, the memory grows linearly with the number of peer-to-peer connections. MSGQ overcomes the above scalability issue due to memory cost, but at the expense of lower performance. Setup of MSGQs is done on a per-node rather than per-peer basis, so the memory only grows as the number of nodes in the job.

uGNI provides *Completion Queues* (CQ) as a light-weight event notification mechanism for applications. For example, an application may use CQ to track the progress of local FMA/BTE transactions, or to notify a remote node that data have been delivered to its memory.

While the uGNI API offers rich functionality for supporting communication, it is clear that there are many considerations in terms of performance, overhead and scalability issues to choose the best messaging facility for sending a message. This can be a challenge for implementing an efficient runtime system on uGNI.

The following functions are relevant to our implementation of the asynchronous message-driven runtime system. More details of the uGNI API can be found in [5].

- *GNI_CqCreate()*: It creates a Completion Queues (CQ), which can be linked to an end point, or a region of memory for incoming messages. The information of next event is returned by calling GNI_CqGetEvent to poll the CQ.
- *GNI_MemRegister()*, *GNI_MemDeregister()*: Memory registration and deregistration. In Gemini, memory can be used in communication only after the memory is registered.
- *GNI_SmsgSendWTag()*, *GNI_SmsgGetNextWTag()*: These functions are used to send and receive point-to-point short messages (SMSG).
- *GNI_PostFma()*: It executes a data transaction (PUT, GET, or AMO) by storing into the directly mapped FMA window to initiate a series of FMA requests.
- *GNI_PostRdma()*: It posts a descriptor to the RDMA queue for BTE to execute a data transaction. The descriptor contains all information about the transaction such as destination process, memory address, handler, etc.

III. Design of Charm++ Runtime on Gemini

In this section, we first describe the Charm++ runtime system and its low level runtime system (LRTS) interface for the portability layer. Next, a design of the uGNI-based LRTS is presented.

A. Charm++ *Programming Model*

Charm++ is a C++-based parallel programming system that implements a *message-driven migratable objects* programming model, supported by an adaptive runtime system [9]. The adaptive runtime system automates resource management and communication optimizations, and provides services such as automatic fault tolerance. Adaptive MPI [8] is an implementation of the message passing interface standard on top of the Charm++ runtime system.

Charm++ applications consist of C++ objects that are organized into indexed collections. Programmers are responsible for decomposing problems into collections of objects based on the domain knowledge, and letting the intelligent runtime manage these objects efficiently. In the Charm++ model, objects communicate via asynchronous method invocations similar to active messages. The runtime system automatically maps and balances these objects to processors, and optimizes the communication among them. Charm++ has been widely deployed on a large number of parallel platforms including IBM Blue Gene/P, Cray XT, commodity InfiniBand clusters and even Windows HPC clusters.

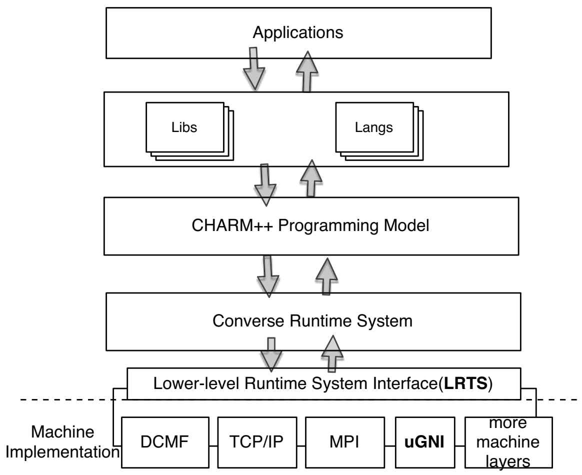


Figure 3. CHARM++ Software Architecture

A layered approach was taken in designing the CHARM++ runtime system, as illustrated in Figure 3. Underneath the CHARM++ software infrastructure, sets of lower-level runtime system interface are defined and needs to be implemented based on the native communication libraries. This interface provides basic message passing functions to the upper portability layer, Converse, which provides a common machine interface. It is clear that the performance of the underlying machine layers on specific hardware significantly influences the performance of CHARM++ applications. Other components of the CHARM++ software stack include the CHARM++ layer itself, which is implemented on top of the machine-independent layer of Converse. Various system and user libraries and applications are built on top of CHARM++. The layered approach facilitates machine porting, development and maintenance.

B. Lower-level Runtime System (LRTS) Interface

Converse is a component of the CHARM++ runtime system that provides a unified functionality across all machine layers to CHARM++. For portable and efficient implementations on a variety of parallel machines, this runtime system needs a minimum set of capabilities from the parallel machine, its communication infrastructure, and its node-level operating system. Therefore, it is desirable to separate machine specific components of Converse into a low-level runtime system, i.e., LRTS. Different machine-specific LRTS implementations can share common implementations such as collective operations (e.g. broadcast) to construct a full Converse layer.

For a supercomputer vendor, LRTS serves as a concise specification of the minimum requirements to implement CHARM++ software stack on their platform. This simplifies the work of porting the CHARM++ runtime to a new platform since the vendor only needs to implement the functions defined in the LRTS. These LRTS functions are classified into capabilities needed for communication, node-level OS interface (including memory allocation, virtual memory functions, topology information, and timers), support for user level threads, external communication, and fault tolerance.

Although many features including the machine timer, memory management and topology are specific to machine layers, the following three LRTS functions are essential to implement the upper level runtime system.

- *void LrtsInit(int argc, char **argv)*: This function is called at the beginning of the program, and is responsible to initialize any system specific variable/structure/device of the underlying hardware.
- *void LrtsSyncSend(unsigned int destPE, unsigned int size, void *msg)*: It sends *msg* of *size* bytes to processor destPE. It is a non-blocking call. The message is either sent immediately to network or buffered.
- *void LrtsNetworkEngine()*: This function is responsible for sending any pending messages and polling network messages. The incoming messages are delivered to the Converse layer to be executed.

C. Design of uGNI-based LRTS

As mentioned in Section II, two mechanisms are provided to transfer data: FMA and BTE. Both mechanisms support GET and PUT operations, which give us four options to perform a communication. The one-way latency achieved with these four options is depicted in Figure 4. It is clear that efficiently transferring message data requires the runtime to select the best mechanism based on the size of the message and the overhead associated with each method.

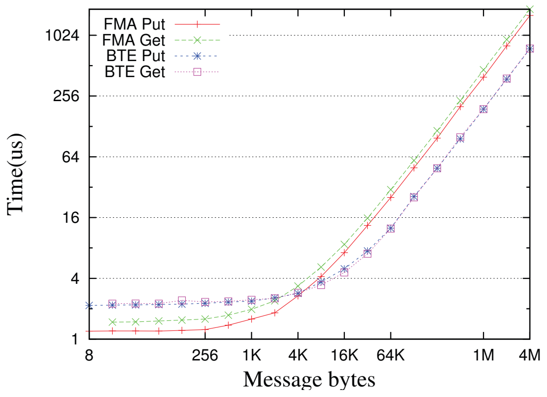
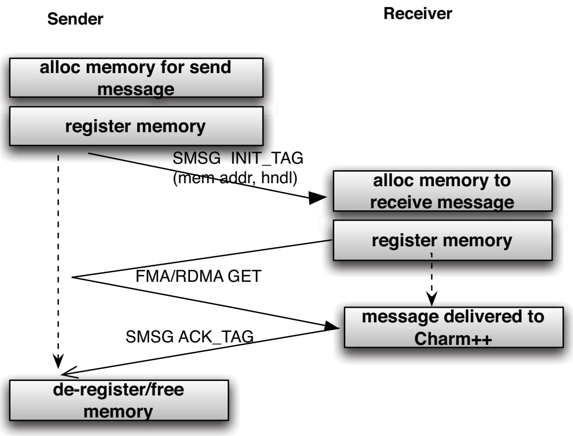


Figure 4. One way latency using FMA/RDMA Put/Get

In our design, small messages are directly sent using SMSG protocol. On the receiver side, dequeuing of incoming messages is done by polling the RX Completion Queue (CQ) associated with the SMSG mailbox. The runtime copies out the messages and hands off the messages to Converse. By default, the maximum SMSG message size is 1024 bytes. However, as the job size increases, this limit decreases to reduce the mailbox memory cost for each SMSG connection pair.



Sender
Receiver
alloc memory for send message
register memory
SMSG INIT_TAG (mem addr, hndl)
alloc memory to receive message
register memory
FMA/RDMA GET
message delivered to Charm++
SMSG ACK_TAG
de-register/free memory

Figure 5. Sending large messages using FMA/RDMA Get

When the message is larger than the size SMSG can deliver, either an FMA or BTE transaction is issued, depending on the message size. Notice that both FMA and BTE require the memory addresses and handles of the sender and receiver. However, in message driven systems, the receive buffer address is not available to the sender in common usage. Therefore, extra steps have to be taken before and after the transactions. First, memory information of one side must be sent to the other side to set up the transaction. After the transaction is finished, a completion event in uGNI is generated on the remote side, however, it does not return the exact memory address, which is needed for either freeing the sender memory (GET-based scheme) or delivering the message to the application (PUT-based scheme). Therefore, an acknowledgment message with detailed transaction information must be sent to the remote side. In our design, a GET-based RDMA scheme is used in conjunction with SMSG. The advantage of the GET-based scheme over the PUT-based scheme is that the PUT-based scheme requires one extra rendezvous message.

Figure 5 illustrates the process of sending a large message. A sender first allocates enough memory for the message and registers this memory. A small control message with INIT_TAG is constructed to contain information of memory address, memory handler and size. This small message is sent to the receiver using the SMSG protocol. When the receiver gets the control message, it allocates memory of the corresponding size and registers it. Now a FMA/BTE transaction is initiated depending on the message size. When resources are available to transfer the data, the transaction is processed. Once the transaction is done, the receiver sends another small message with ACK_TAG to the sender so that the sender can de-register and free the memory. Based on this scheme, the time cost of sending a large message can be described in Equation 1.

$$\begin{aligned} T_{cost} &= T_{sender} + T_{rdma} + T_{receiver} + 2 \times T_{smsg} \\ &= 2 \times (T_{malloc} + T_{register}) + T_{rdma} + 2 \times T_{smsg} \end{aligned} \tag{1}$$

In this equation, T_{rdma} and T_{smsg} are determined by the underlying hardware. $2 \times (T_{malloc} + T_{register})$ is the overhead of setting up the memory for the message.

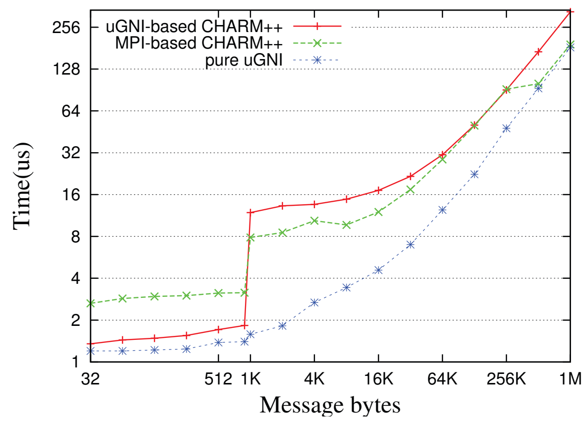


Figure 6. Comparison of on-way latency in uGNI, MPI-based CHARM++ and the initial version of the uGNI-based CHARM++

Based on the above design and implementation, we measured the one-way latency in uGNI-based CHARM++ and compared it to the pure MPI and uGNI. The result is illustrated in Figure 6. For messages smaller than the maximum size that SMSG can deliver (around 1000 bytes), uGNI-based CHARM++ performs quite well, with performance close to the latency obtained by uGNI. The small overhead is due to CHARM++ overhead. MPI-based CHARM++, however, performs the worst mainly due to the additional MPI overhead and the inefficiency of using MPI to implement the message-driven execution model. For large messages, however, uGNI-based CHARM++ performs even worse than the MPI-based CHARM++. In the next section, we will present techniques to improve large-message performance.

IV. Optimizing Performance

In this section, we focus on improving the communication performance, especially for messages that are larger than SMSG can deliver. Techniques are presented to exploit persistent communication, reduce memory copy and registration overhead, and optimize intra-node communication.

A. *Persistent Messages*

In many scientific applications, communication with a fixed pattern is repeated in time steps or loops of a parallel computation. In different iterations, the communicating processors and communication arguments including data size remains persistent. In such a situation, it may be possible to optimize the communication by reusing the memory for messages to avoid memory allocation operations, and by using efficient one-sided communication. For example, MPI standard provides persistent communication requests by creating a persistent communication request once and, then, repeatedly using the request to initiate and complete messages.

CHARM++ LRTS defines a simpler persistent communication API which only requires the sender for setting up the communication channel, as the allocation of the receive buffer is controlled by the runtime itself.

- *PersistentHandle LrtsCreatePersistent(int destPE, int maxBytes)*: Sender initiates the setting up of persistent communication with processor *destPE* and returns a persistent communication handler. A buffer of size *maxBytes* is allocated in the destination processor.
- *void LrtsSendPersistentMsg(PersistentHandle hdl, int destPE, unsigned int size, void *msg)*: This call sends *msg* of *size* bytes to *destNode* using the persistent communication handle *hdl*.

In Gemini, only those large messages beyond what SMSG can deliver can benefit from using the persistent messages, since SMSG already provides the best performance for small messages. The process of a persistent communication is depicted in Figure 7(a). Comparing with Figure 5, persistent messages eliminate the overhead of memory allocation, registration and de-registration, which may be expensive [21]. Moreover, because the memory buffer on the receiver is persistent and known to the sender, the sender can directly put its message data into the persistent buffer, which saves one control message that was needed to exchange message buffer information. Therefore, the one-way latency is reduced to:

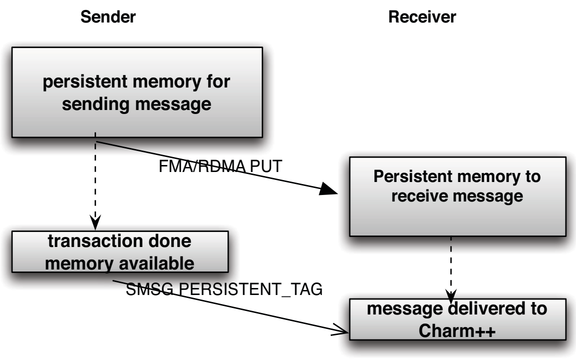
$$T_{cost} = T_{rdma} + T_{smsg}$$

Figure 8(a) compares the single message transfer time with and without using persistent message. As can be seen, persistent message greatly reduces the latency of messages using the large message RDMA protocol.

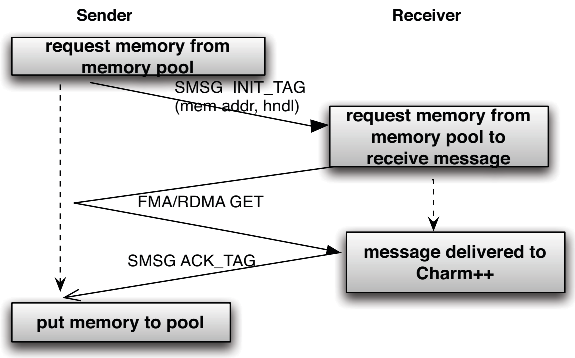
B. Memory Pool

Persistent communication eliminates the memory allocation and registration overhead during send and receive, but it requires explicitly setting up the communication channel, which is not always feasible. A more general optimization using a memory pool is described next.

As we have seen in Figure 6, the performance of our original uGNI-based CHARM++ is worse than the MPI implementation. One reason is that MPI uses the registration cache (uDREG) [17] to reduce the overhead of memory registration for large messages. However, uDREG has its own overhead and pitfalls [21]. Furthermore, while the MPI standard specifically demands user-supplied buffers, the CHARM++ runtime takes full control of the memory allocation and de-allocation for messages due to the nature of its the message driven programming model. Taking advantage of this fact, we can exploit the use of a memory pool aggressively by pre-allocating and registering a relatively large amount of memory, and explicitly managing it for CHARM++ messages. When messages are allocated or freed, the runtime allocates or frees memory from the memory pool, instead of calling system malloc/free. Since the entire memory pool is pre-registered, there is no additional registration cost for each message. In the case when the memory pool overflows, it can be dynamically expanded.


Sender
Receiver
persistent memory for sending message
FMA/RDMA PUT
Persistent memory to receive message
transaction done memory available
SMSG PERSISTENT_TAG
message delivered to Charm++

(a) Send persistent messages


Sender
Receiver
request memory from memory pool
SMSG INIT_TAG (mem addr, hndl)
request memory from memory pool to receive message
FMA/RDMA GET
SMSG ACK_TAG
message delivered to Charm++
put memory to pool

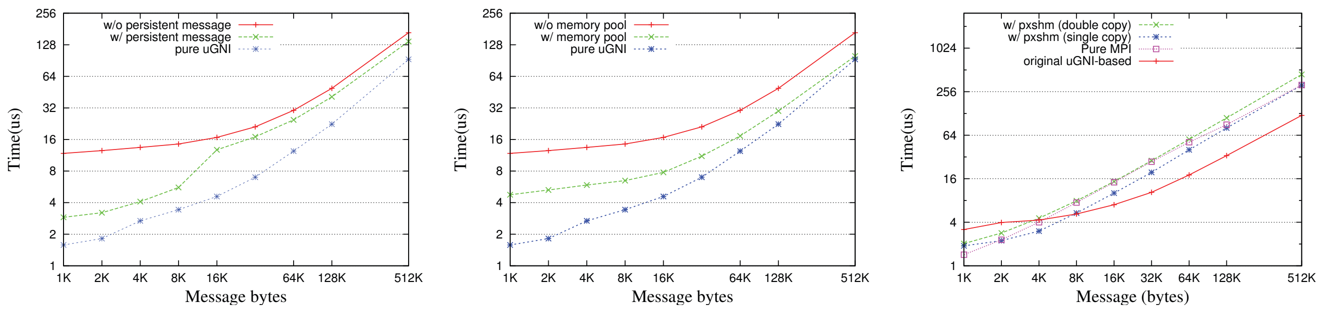
(b) Send messages using memory pool

Figure 7. Optimization with persistent message and memory pool

After using the memory pool, the process of transferring a message is depicted in Figure 7(b). Compared to equation 1, T_{malloc} and $T_{register}$ are now eliminated due to the use of memory pool. Therefore, the time to transfer a message is reduced to the following:

$$T_{cost} = 2 \times (T_{mempool}) + T_{rdma} + 2 \times T_{smsg}$$

Figure 8(b) shows the performance improvement of sending single messages using the memory pool. Comparing with the original uGNI-based CHARM++ implementation without using a memory pool, the latency is significantly



(a) w/ and w/o persistent message (b) w/ and w/o memory pool (c) w/ and w/o intra-node optimization

Figure 8. Comparison of message latency in MPI-based and uGNI-based CHARM++

reduced by 50%. For relatively small messages, the latency difference between the uGNI-based CHARM++ and pure uGNI is around $2.5us$, which includes runtime overhead and one small message latency caused by the control message required in the send protocol. As the message size increases, the message latency in uGNI-based CHARM++ with memory pool gets quite close to that in pure uGNI.

C. Intra-node Communication

The Cray XE6 system has 24 cores per node. Intra-node communication is crucial to the overall performance. One can choose to let the runtime send outgoing messages through the uGNI communication library regardless of whether it is an intra-node or inter-node message. This implementation is quite efficient in a pingpong test, as shown in the first curve of Figure 8(c). However, in the scenario when there are lots of intra-node and inter-node messages, the uGNI hardware can be a bottleneck and may cause contention.

MPI implementations usually offer a user-space double-copy based intra-node communication strategy, which is very efficient for small messages. For large messages, some MPI implementations (including Cray MPI) utilize the XPMEM [12] driver to do single copy, which supports single copy message transfers between two processes within the same host. During job startup, MPI uses the XPMEM driver (via the xpmem kernel module) to map memory from one MPI process to another, which allows each process to directly access memory from the address space of another process. Enabling single copy transfers may result in better performance; however, it may introduce additional synchronization points, which can reduce application performance in some cases.

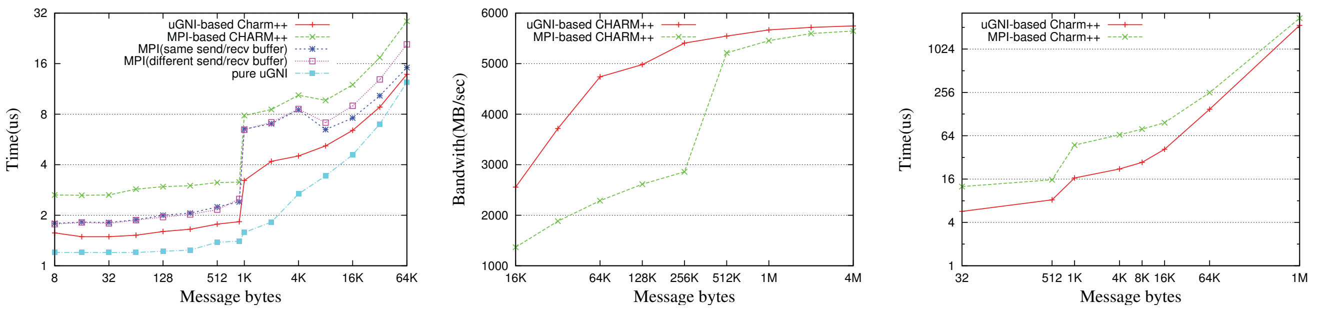
Compared with MPI, the CHARM++ runtime manages the message buffer completely. Messages are delivered from runtime to the application without copying. This model is flexible for the runtime to adopt intra-node optimizations without having to rely on the special XPMEM driver, which is only available on certain platforms.

CHARM++ initially adopted a similar user-space double-copy intra-node communication scheme (pxshm). It is based on POSIX shared memory, an efficient means of passing data between processes. In this scheme, at start-up time, each core on a node sets up a POSIX shared memory for transferring data between processes on the same node. A sender first allocates a memory region in the shared memory with the receiver, and then copies its data from user-space to the shared memory, and notifies the receiver about the incoming message. On the receiver side, the network progress engine periodically checks any incoming message in the shared memory. When a message arrives, the receiver copies the message out of the shared memory. In this scheme, the sender and receiver work in a producer-consumer like pattern on the shared memory; a lock or memory fence is needed.

This scheme has the advantage of simplicity. After the message is copied out of the shared memory, the shared memory region is immediately marked free for receiving messages. However, for the same reason as in MPI, this scheme is not efficient for large messages due to double copies.

The single copy optimization in CHARM++ is a sender side copy scheme. The sender allocates a message without the runtime knowing where the messages will be sent. If the message is actually sent as an intra-node message, it is copied to the shared memory. On the receiver side, the received message in the shared memory is delivered directly to the user program without copying.

Figure 8(c) compares intra-node pingpong performance in MPI with that in uGNI-based CHARM++ using the single/double copy optimization. With double copy in the pxshm-based scheme, CHARM++ achieves very close performance to MPI for message size below 16KB, however, it performs worse beyond that. With the single copy optimization, overall CHARM++ achieves better performance than MPI. As explained earlier, although uGNI performance is superior than our implementation, one should not use uGNI for intra-node communication since this interferes with uGNI handling inter-node communication.



(a) Latency comparison (b) Bandwidth comparison (c) Latency comparison in one-to-all benchmark

Figure 9. Comparison of message latency and bandwidth running on MPI-based and uGNI-based CHARM++

V. Performance

As demonstrated above, the uGNI-based CHARM++ runtime with various optimization techniques enables better point-to-point communication performance on various sizes of messages than the MPI-based CHARM++ implementation. This section compares the uGNI-based CHARM++ performance with the MPI-based CHARM++ using two micro-benchmarks and two real world applications. All the following benchmark programs and applications are written in CHARM++, but linked with either MPI- or uGNI-based message-driven runtime for comparison. The flexibility provided by the LRTS interface allows the application to change its underlying LRTS implementation transparently.

All the experiments are done on Hopper, a peta-flop Cray XE6 system at National Energy Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory. The whole system has 6,384 nodes with 153,216 cores, 217TB of memory and 2 Petabytes of online disk storage. Each node consists of 2 twelve-core AMD "MagnyCours" processors and 32 GB DDR3 1333 MHz memory. All the compute nodes are connected via a "3D torus" Cray Gemini Interconnect.

A. Message latency and bandwidth

In the first micro-benchmark, we compare the one-way latency performance among uGNI, MPI, and CHARM++. In the benchmark, for each iteration, processor 0 sends a message of a certain size to processor 1 on a different node. Processor 1 receives the message and sends the same message back to processor 0. The average one-way latency is calculated after measuring a thousand iterations. In this benchmark, the message buffer is reused; that is, when a processor receives a message, it sends the same message to the other processor. This is to eliminate the overhead of memory allocation and registration so that the latency measured by this benchmark is as close as the best performance the runtime system can achieve.

Three versions of this benchmark are written: the first is written in native uGNI API; the second is written in MPI with two variants, one with same send/recv buffer, the other with different buffers; and the third is written in CHARM++ which can be linked either with a MPI-based or uGNI-based runtime for comparison. Figure 9(a) illustrates the latencies for various sizes of messages using these benchmarks.

For small messages ranging from 8 bytes to 1024 bytes, both uGNI-based CHARM++ runtime and MPI use the SMSG protocol. It can be seen from the Figure 9(a) that the uGNI-based CHARM++ runtime system obtains a latency as low as $1.6us$ for an 8-byte message, which is close to the case with the pure uGNI ($1.2us$) and pure MPI. It is much better than the case of MPI-based CHARM++. This is mainly due to the reduction in runtime overhead caused by both CHARM++ runtime and MPI library.

Notice that between 512 bytes and 1024 bytes, while the pure uGNI latency is slowly increasing, there is a sudden jump in both cases of pure MPI and uGNI-based CHARM++. It is due to the runtimes switching to FMA/BTE read protocol, where the latency is now composes of a one-way SMSG latency and FMA/BTE message transfer time. After 1024 bytes, both MPI and MPI-based CHARM++ runtime switch to an eager message protocol while uGNI-based CHARM++ adopts FMA/BTE read protocol for messages allocated from memory pool. The figure shows that uGNI-based CHARM++ outperforms both pure MPI and MPI-based CHARM++ systems. This is due to the use of memory pool technique in the uGNI-based CHARM++ runtime, which helps to avoid the extra memory copy between the user space and runtime as MPI does in its implementation.

After 8192 bytes, the performance of the MPI benchmark that uses same send and recv buffer and the uGNI-based CHARM++ become close because MPI switches to rendezvous protocol for large messages, in which case MPI sends message directly from the sender buffer to the receiver without memory copy. Note that in the case of MPI benchmarks, if a same user buffer is used in both sending and receiving, the MPI performance is much better than the case of using different buffers. Since using different buffers is the use case in the implementation of the MPI-based CHARM++, its performance is worse than the uGNI-based CHARM++.

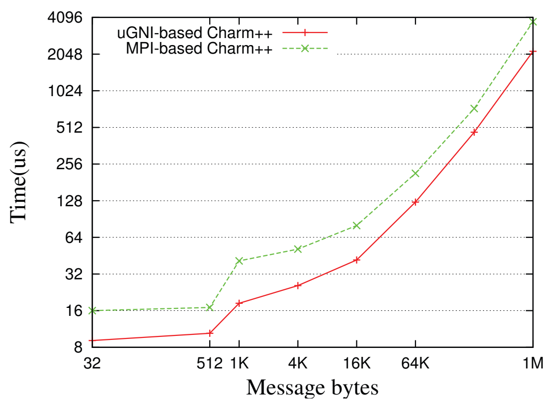


Figure 10. Comparison of kNeighbor performance on uGNI-based and MPI-based charm++

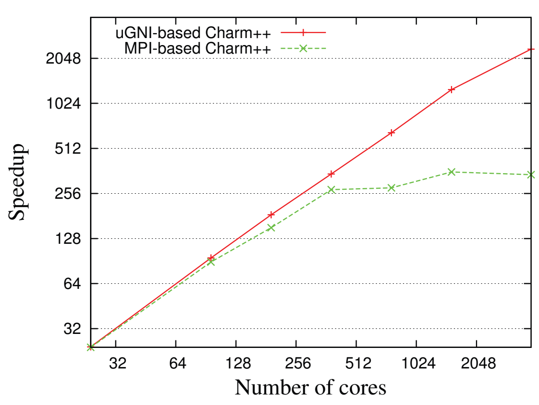


Figure 11. Strong scaling results of 17-Queens with uGNI-based and MPI-based CHARM++

Figure 9(b) shows the bandwidth achieved with messages of sizes larger than $16K$ bytes. The gap between the uGNI-based and the MPI-based CHARM++ for message sizes smaller than $1M$ bytes is mainly due to the overhead of the MPI layer in the MPI-based implementation. As the message sizes increase, the performance becomes similar.

While the previous ping-pong benchmark is useful, it does not represent the real scenarios in applications. This motivated us to the next benchmark which is a one-to-all benchmark. In this benchmark, processor 0 sends a message to one core on each remote node, and each destination core sends an ack message back. The results of running this benchmark on 16 nodes of Hopper are shown in Figure 9(c). As expected, for small messages, uGNI-based CHARM++ outperforms MPI-based CHARM++ by a large margin. While message sizes increase, the gap closes. The large difference for small messages is due to the difference in how much CPU-time used in different implementations.

B. kNeighbor

An even more realistic scenario is when all processors send and receive messages concurrently that possibly leads to network contention. We use a synthetic kNeighbor benchmark for this purpose. In this benchmark, each core sends messages to its k left and k right neighbors in a ring virtual topology. When each core receives all the $2 * k$ messages, it proceeds to the next iteration. We measure the total time for sending $2 * k$ messages and receiving $2 * k$ ping-back messages. When one message is received, the same message buffer is used to send the ack back to the sender.

We tested 3 cores on 3 different nodes doing 1-Neighbor communication. In each round-trip iteration, each core sends a total of 4 messages and receives 4 messages. Figure 10 compares the performance of the uGNI-based and MPI-based CHARM++. The latency on uGNI-based CHARM++ is only half of that on the MPI-based CHARM++ even for $1M$ byte message, even though it has similar one-way latency in our ping-pong test. This is because in MPI-based CHARM++, once a MPI_IProbe returns true, the progress engine calls blocking MPI_Recv to receive the large message, which prevents the progress engine from doing any other work. However, in uGNI-based CHARM++, the progress engine is free to continue working when the underlying BTE is receiving message.

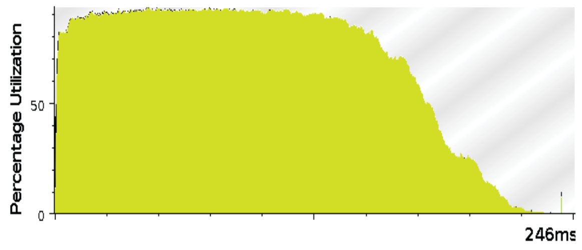
C. N-Queens

N-Queens problem is a classical backtracking search problem, in which N queens must be placed on an $N \times N$ chess board so that no queens are in the same rows, columns and diagonals. In this benchmark, we aim to find all solutions. A task-based parallelization scheme is used, wherein each task is responsible for the exploration of some states and spawn new tasks if necessary. After a new task is dynamically created, it is randomly assigned to a processor. The grain size of each task is controlled by a user-defined threshold. For example, the threshold of 6 to a 17-Queens problem means that only the first 6 queens are treated as parallel tasks, while the problem of placing the remaining 11 queens is done sequentially. Increasing the threshold decreases the grain size and increases the parallelism. However, it also causes more overhead due to communication and scheduling in runtime system. In this problem, the size of messages are quite small (around 88 bytes), but the number of messages is large.

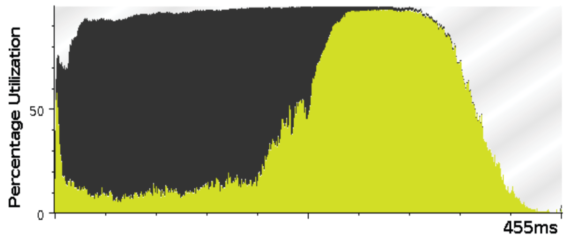
We used an N-Queens implementation based on the state space search framework described in [19]. We demonstrated how improving communication and reducing runtime overhead in the uGNI-based CHARM++ improves its performance. Figure 11 compares the speedup of 17-Queens problem running on uGNI-based and MPI-based CHARM++. With uGNI-based CHARM++, the 17-Queens problem keeps scaling almost perfectly up to 3840 cores using threshold of 7 (which is optimal). In comparison, with MPI-based CHARM++ implementation, the same program stops scaling

Queens	14	15	16	17	18	19
Cores(uGNI-CHARM++)	256	480	1536	3840	7680	15360
Cores(MPI-CHARM++)	48	120	384	1536	3840	7680
Time(sec)uGNI-CHARM++	0.005	0.007	0.014	0.029	0.09	0.33
Time(sec)MPI-CHARM++	0.02	0.03	0.056	0.19	0.35	1.42

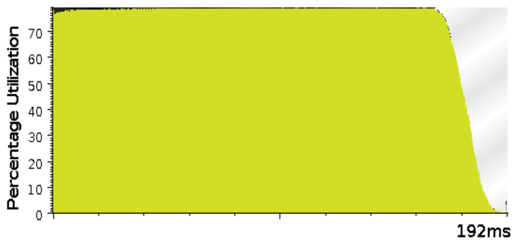
Table I
BEST PERFORMANCE ON DIFFERENT NUMBER OF CORES FOR VARIOUS QUEENS PROBLEMS WITH UGNI- AND MPI-BASED CHARM++



(a) 17-Queens of threshold 6 on MPI-based CHARM++



(b) 17-Queens of threshold 7 on MPI-based CHARM++



(c) 17-Queens of threshold 7 on uGNI-based CHARM++

Figure 12. Time profile of running 17-Queens on 384 cores using MPI- and uGNI-based CHARM++

at around 384 cores using threshold of 6. Using threshold of 7 only makes the MPI case worse as explained next.

Figure 12 compares the sum of the CPU utilization on all 384 cores as it varies over time for three 17-Queens cases with different thresholds on MPI- and uGNI-based CHARM++. These figures are generated by the performance analysis tool *Projections* [10]. The yellow portion stands for the useful computation, while idle is shown in white, and runtime overhead is depicted in black. In the case of MPI-based CHARM++ (Figure 12(a)), with threshold of 6, the program generates a total of $15K$ messages. The long tail is caused by load imbalance at the end, where only some processors still have tasks. In the case of the uGNI-based CHARM++, using higher threshold of 7, it achieves best performance due to more messages (about $123K$) and more finer-grain tasks, which results in better load balance even at the end of the execution. For the case of MPI, however, when we increase the threshold to 7, as shown in Figure 12(b), the performance turned out to be even worse due to the large communication overhead shown in the black portion.

Table I compares for a number of queens problems, the maximum cores and the corresponding performance that uGNI-based and MPI-based CHARM++ scales to. It is clear that for the same N-Queens problem, uGNI-based Charm++ scales to more cores with much less time. In particular, the 19-Queens problem on the uGNI-based CHARM++ scales up to $15,360$ cores using 70% less execution time than the MPI-based CHARM++.

D. NAMD

NAMD [16], [2] is a scalable parallel application for molecular dynamics simulations written in CHARM++. It is widely used for the simulation of biomolecules to understand their structures. The dynamic measurement-based load balancing framework in CHARM++ is deployed in NAMD for balancing computation across processors. Objects migrate between processors periodically according to load balancing decisions that explicitly takes communication into account. Thanks to the dynamic load balancing, and asynchronous communication which allows dynamic overlapping of communication and computation, NAMD is built tolerant to message latency to a certain extent.

One challenge in NAMD is how to improve performance when the communication-to-computation ratio is high, e.g., when the number of atoms on each core is low. Moreover, calculating both short-range and long-range forces (PME) every step imposes more challenges because of the global communication involved in the PME calculation. Compared to the small messages in the N-Queens problem above, the message sizes in NAMD is typically ranged from $1K$ to $16K$ bytes. In the following experiments, we run NAMD with long-range force calculation every step, and compare the performance of uGNI- and MPI-based CHARM++ to demonstrate the benefit of the uGNI-based implementation.

We first evaluate the strong scaling of NAMD on Hopper using both uGNI-based and MPI-based CHARM++. We ran NAMD with the standard molecular system used by biophysicists for benchmarking, Apolipoprotein-A1 (ApoA1), a 92,224 atom benchmark. The results are shown in Table II. As we can see, uGNI-based NAMD outperforms the MPI-based NAMD in all cases by about 10%.

Weak scaling is also evaluated by showing the compu-

Number of Cores	2	12	48	120	240	480	1920	3840
MPI-based CHARM++	987	172	45.1	20.2	10.8	6.2	3.3	3.06
uGNI-based CHARM++	979	168	38.2	16.7	8.8	5.1	2.7	2.78

Table II
APOA1 TIME(MS/STEP) RUNNING NAMD ON MPI-BASED AND UGNI-BASED CHARM++

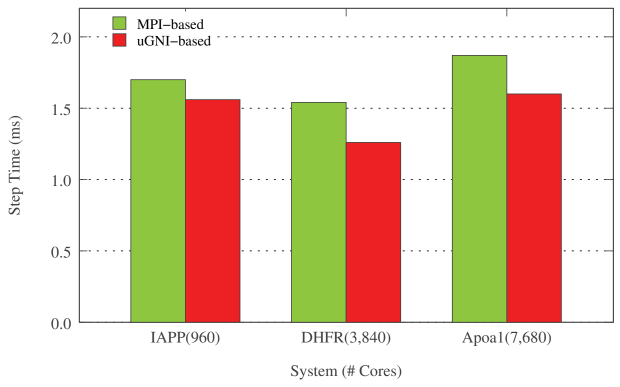


Figure 13. NAMD weak scaling results

tation time varying as a function of processor count for a fixed problem size per processor. This is to study the performance of NAMD for various simulation sizes that grow proportionally to the number of processors. Figure 13 shows the results of the weak scaling tests on Hopper using several relatively small benchmarks using both uGNI-based and MPI-based CHARM++. The benchmarks used are IAPP (5570 atoms), DHFR (23,558 atoms) and Apoa1 (92,224 atoms). We ran PME every time step and compared the benchmark time between the uGNI-based and MPI-based implementations. We observed about 10% improvement on IAPP and Apoa1 by using the new uGNI machine layer. As much as 18% improvement is achieved on DHFR system. Considering the fact that the timestep was already as low as 1.5 ms/step, a 10% improvement by using uGNI-based CHARM++ is an encouraging result.

VI. RELATED WORK

The importance of MPI has ensured a variety of research approaches focused on improving performance. Many times, these optimizations involve compiler support. Such approaches are typically tightly-coupled to a specific target network, and may involve strategies to improve resource utilization such as reusing communication buffers; an example is OMPI [15]. Friedley and Lumsdaine describe a compiler approach to transforming MPI calls to a one-sided communication model [7]. They report a 40% increase over MPI by using a simpler one-sided communication model.

Gravel is a communication library that provides several message passing protocols to provide better communication-computation overlap; this is combined with compiler analysis and transformations for improvements [6]. In a similar vein, Small and Yuan proposed multiple protocols to improve performance [18].

Partitioned Global Address Space (PGAS) languages like UPC [20], Global Arrays [13], and Co-Array FORTRAN [14] are optimized to use one-sided communication protocols like we have developed.

Finally, Cray's Gemini interconnect, the network we utilized for our measurements in this paper, is described in [1].

The work presented in this paper is different from existing work in that it does not rely on compiler transformations. In addition, it benefits from its incorporation into the CHARM++ runtime system. We believe our techniques to be general in nature and suitable to other target networks with one-sided capabilities. Our approach differs from the above PGAS languages in its object-based virtualization strategy which has been demonstrated to provide helpful adaptive load balancing and fault tolerance optimizations. Further, the message-driven model in CHARM++ runtime is not as well-supported by the RDMA interfaces of uGNI as the PGAS languages.

VII. CONCLUSION AND FUTURE WORK

Cray Gemini interconnect as the network for the new Cray XE/XK systems is characterized by low latency, high bandwidth and strong scalability, thanks to the powerful yet flexible Generic Network Interface (GNI). With the Cray XE/XK supercomputers becoming one of the major platforms for running large scale parallel applications, this paper tackles the challenging problem of designing an asynchronous message-driven runtime system, CHARM++ on uGNI, and scaling its applications including NAMD. Several approaches are explored to optimize the communication to best utilize the Gemini capability. These include reducing memory copy and registration overhead using memory pool; exploiting persistent communication for one-sided communication; and improving intra-node communication via POSIX shared memory. These optimization techniques leverage the message driven programming model where message buffers are managed by the runtime itself. Our micro-benchmarks show that this run-time system outperforms MPI-based implementation by up to 50% on a Cray XE machine. For an irregular program N-Queens, the uGNI-based implementation scales up to $15,360$ cores using 70% less execution time than MPI-based implementation. In NAMD, a full-fledged molecular dynamics application, the performance is significantly improved by as much as 18% in a fine-grain parallel run. To the best of our knowledge, the uGNI-based CHARM++ runtime is the first public third-party runtime ported on uGNI, which achieves scalability up to tens of thousands of cores on a Cray XE machine.

Although optimized, the intra-node communication via POSIX shared memory is still quite slow due to memory copy. We plan to investigate the SMP mode of CHARM++ (as described in [11]) on uGNI to further optimize the intra-node communication.

ACKNOWLEDGMENTS

This work was supported in part by a NIH Grant PHS 5 P41 RR05969-04, by NSF grant OCI-0725070 for Blue Waters, and by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign. Major funding was provided by the HPC Colony Project, an effort supported by the DOE Office of Advanced Scientific Computing Research. This work used machine resources from NERSC (Hopper) through an ERCAP award to the HPC Colony Project.

REFERENCES

[1] R. Alverson, D. Roweth, and L. Kaplan. The gemini system interconnect. In *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, 2011.

[2] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[3] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[5] Cray Inc. *Using the GNI and DMAPP APIs*, 2010. http://docs.cray.com/books/S-2446-3103/S-2446-3103.pdf.

[6] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. Mpi-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 316–325, New York, NY, USA, 2009. ACM.

[7] A. Friedley and A. Lumsdaine. Communication optimization beyond mpi. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2018 –2021, may 2011.

[8] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance Evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[9] L. V. Kale and G. Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[10] L. V. Kale, G. Zheng, C. W. Lee, and S. 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*, volume 22, pages 347–358, February 2006.

[11] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C.Phillips, and C. Harrison. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[12] S. Moreaud, B. Goglin, D. Goodell, and R. Namyst. Optimizing MPI Communication within large Multicore nodes with Kernel assistance. In IEEE, editor, *Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2010*, page 7 p., Atlanta, États-Unis, Apr. 2010.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996. 10.1007/BF00130708.

[14] R. W. Numrich and J. Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24:4–17, August 2005.

[15] H. Ogawa and S. Matsuoka. Ompi: Optimizing mpi programs using partial evaluation. *SC Conference*, 0:37, 1996.

[16] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[17] H. Pritchard, I. Gorodetsky, and D. Buntinas. A ugni-based mpich2 nemesis network module for the cray xe. 6960:110–119, 2011.

[18] M. Small and X. Yuan. Maximizing mpi point-to-point communication performance on rdma-enabled clusters with customized protocols. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 306–315, New York, NY, USA, 2009. ACM.

[19] Y. Sun, G. Zheng, P. Jetley, and L. V. Kalé. ParSSSE: An Adaptive Parallel State Space Search Engine. *Parallel Processing Letters*, 21(3):319–338, September 2011.

[20] T. S. Tarek El-Ghazawi, William Carlson and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[21] P. Wyckoff and J. Wu. Memory registration caching correctness. *Cluster Computing and the Grid, IEEE International Symposium on*, 2:1008–1015, 2005.