Enabling a highly-scalable global address space model for petascale computing

Vinod Tipparaju[†] Edoardo Aprà[†] Weikuan Yu[‡] Jeffrey S. Vetter[†]

Computer Science & Mathematics[†]
Oak Ridge National Laboratory
Oak Ridge, TN 37831
{tipparajuv,aprae,vetter}@ornl.gov

Department of Computer Science [‡]
Auburn University
Auburn, AL 36849
wkyu@auburn.edu

ABSTRACT

Over the past decade, the trajectory to the petascale has been built on increased complexity and scale of the underlying parallel architectures. Meanwhile, software developers have struggled to provide tools that maintain the productivity of computational science teams using these new systems. In this regard, Global Address Space (GAS) programming models provide a straightforward and easy to use addressing model, which can lead to improved productivity. However, the scalability of GAS depends directly on the design and implementation of the runtime system on the target petascale distributed-memory architecture. In this paper, we describe the design, implementation, and optimization of the Aggregate Remote Memory Copy Interface (ARMCI) runtime library on the Cray XT5 2.3 PetaFLOPs computer at Oak Ridge National Laboratory. We optimized our implementation with the *flow intimation* technique that we have introduced in this paper. Our optimized ARMCI implementation improves scalability of both the Global Arrays (GA) programming model and a real-world chemistry application – NWChem – from small jobs up through 180,000 cores.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Performance

Keywords

Global Address Space, Global Arrays, ARMCI, XT5, NWChem, Flow Control, PGAS, GAS, GA

1. INTRODUCTION

Systems with unprecedented computational power are continuously pushing the frontier of high performance computing (HPC) [2]. Several sites have deployed systems that can perform 10^{15} floating point operations per second (petaflop): Cray XT5 (a.k.a. Jaguar) at the Oak Ridge National Laboratory (ORNL), IBM Cell-based system at Los Alamos National Laboratory (LANL), BlueGene/P at Forschungszentrum Juelich (FZJ). These facilities are used to solve important computational science problems in areas such as climate modeling, life sciences, and energy production. Yet many challenges in scientific productivity and application efficiency continue to plague these systems as they grow to unprecedented number of processes and complexity.

In this regard, GAS (Global Address Space) programming models - both the Partitioned and Asynchronous Partitioned Global Address Space - are being considered as an alternative model for programming these complex machines to improve productivity and application efficiency.

Briefly, a GAS model provides an abstraction that allows threads to access the remote memory of other nodes as if they were accessing local node memory using hardware shared memory. By virtue of the abstraction they provide, partitioned Global address space languages like Unified Parallel C (UPC) [3], Co-Array Fortran (CAF) [11], and Global Address Space libraries such as Global Arrays (GA) Toolkit [1] have the unique ability to expose features, such as low-overhead communication or global address space support in the underlying hardware. Systems that lack one or more of these features typically result in poor performance.

Conceptually, Global Address Space (GAS) models do not differentiate between *local* and *remote* accesses. By contrast, Partitioned Global Address Space (PGAS) is a category of GAS models that requires applications to explicitly distinguish between local and remote memory accesses, while providing simple mechanisms for reading, writing, and synchronizing remote memory. One benefit of this explicit separation is that the user is forced to consider and optimize the performance of remote memory access while leaving the optimization of local memory accesses to the compiler.

Recently, a slightly different category of PGAS model, termed *Asynchronous* Partitioned Global Address space model, has emerged to add additional capabilities such as remote method invocations. IBM's X10 language [5] and Asynchronous Remote Methods (ARM) [23] in UPC have pioneered this new model.

ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
CF'10, May 17–19, 2010, Bertinoro, Italy.
Copyright 2010 ACM 978-1-4503-0044-5/10/05 ...$10.00.

All the above mentioned GAS languages and libraries use the services of an underlying communication library (which we refer to as the GAS Runtime) for serving their communication needs. GAS languages normally use this runtime as a compilation target to do the data transfers on distributed memory architectures. They have a translation layer that translates a GAS access to a corresponding data transfer on the underlying system using the GAS runtime. Two example GAS Runtime libraries are GASNet [7] and ARMCI [18], both of which are used in numerous Global Address Space languages and libraries. In these runtime systems latency tolerating features such as non-blocking data transfers and message aggregation enable the GAS languages and libraries to obtain the best possible, close to the hardware, performance on clusters.

In this paper, we demonstrate the scalability of a specific Global Address Space model - Global Arrays - by designing and implementing a highly scalable port of its GAS runtime, ARMCI . This scalable GA/ARMCI ultimately enables the scaling of a real scientific application (the electronic structure methods of the chemistry computer code NWChem) to 180,000 cores on the 2.3 PetaFLOPs Cray XT5 at Oak Ridge National Laboratory. Our design and implementation of the Aggregate Remote Memory Copy Interface (ARMCI) on the Cray XT5 hardware uses the Portals communication layer[1]. To achieve this scalability, we introduce the concept of *flow intimation* – a unique and a useful technique that enables us to achieve performance at scale and yet use limited buffer space for one-sided communication. This end goal of performance at scale influenced every step of this project by aiming to efficiently exploit all of the system's hardware components: high-speed network, aggregate memory size, and multi-core components of the processing nodes of the Cray XT5.

The rest of the paper is structured as follows: we start with an overview of the structure of Global Arrays library in Section 2; we discuss the validation benchmarks we used and the connection setup details in Section 3; we describe the issues we faced in scaling this model with relation to the features of the physical network interconnect (Seastar2+) and the lowest level API to program it (Portals) in Section 4 (this section also introduces flow intimation); and finally we discuss the achieved performance in the context of NWChem in Section 5 and conclude with future steps in Section 6.

2. AN OVERVIEW OF GLOBAL ARRAYS

The Global Arrays (GA) library provides an efficient and portable GAS styled shared memory programming interface for distributed memory computers. Each process in a parallel program can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays, without the need for explicit cooperation by other processes. GA is a unique GAS model that provides explicit functionality to realize the difference between local and non-local data accesses, supports asynchronous data accesses, provides interfaces that translate to remote procedure calls, and naturally supports load-balancing. GA is equipped with the ARMCI runtime system to support blocking and non-blocking data transfers for contiguous, vector and strided data transfers. The model of execution in GA is multiple

[1]Portals is the lowest level communication library available on the XT5

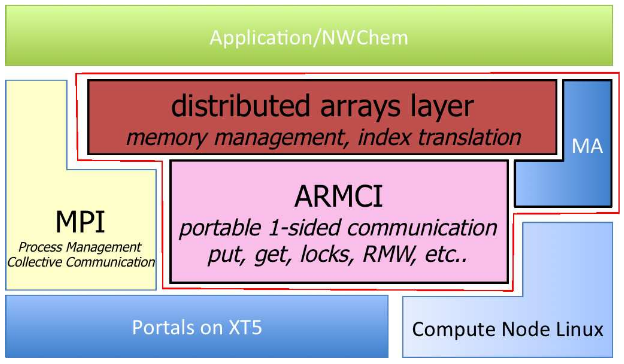


Figure 1: The Structure of the Global Arrays library on the Cray XT5

single-threaded processes per shared memory node, with all sharing done through the GA library via ARMCI.

The structure of GA is shown in Figure 1. The application (in our case NWChem) uses only the ***GA interfaces***, a message passing wrapper (to initialize the message passing library), and the MA layer. The rest of the elements seen in the figure are not exposed to the user application. The highlighted area in the figure shows the primary components of GA: the ***Distributed Array layer (DA)***, ***ARMCI***, and the ***Memory Allocator (MA)***. MA provides simple interfaces to allocate "local" memory. We will further describe the DA layer, the GA programming interfaces, and ARMCI.

2.1 Distributed Array (DA)

DA is the layer in GA that realizes the virtually shared memory access and translates it to actual process/virtual-address information. A simple shared memory style access to a section of a GA data array can translate to multiple blocks of physically distributed data. This is the layer that gives the GA operations the information about the actual location of the data. Such translation subsequently results in calls to one-sided ARMCI calls.

2.2 Programming Interfaces in GA

GA provides a plethora of interfaces that operate on the array abstractions. Most of the interfaces are described in [15]. There are three main categories of GA interfaces of interest here: array creation, one-sided, and data parallel. All the GA interfaces have both C and Fortran bindings.

The array creation interfaces result in the creation of data structures that are later used by the Distributed Array layer. Subsequently ARMCI memory allocation interface is used to allocate the actual memory for the array. An example of 2D-array creation interface in Fortran: `logical function ga_create(type, dim1, dim2, array_name, chunk1, chunk2, g_a)`. The memory allocation, the data structure and the allocation, and their sizes need to be handled carefully.

The GA one-sided operations, after the necessary index translation using the DA layer, result in calls to the ARMCI one-sided API. Access to a GA segment via a one-sided operation may result in multiple non-blocking ARMCI function calls based on the distribution of physical array. Very efficient, low latency, non-blocking calls are important for GA. With the number of ARMCI calls made in a typical NWChem run (discussed in section 5), even sub-micro second saving in each call collectively amounts to a noticeable performance difference in the application. An example of a GA one-sided operation to get a section of a remote array into a local buffer is: `subroutine ga_get(g_a, ilo, ihi, jlo, jhi, buf, ld)`.

GA data parallel operations are collective in nature, and may translate into several ARMCI one-sided and atomic function calls, simultaneously, across all the involved processes. An example of a data parallel operation to scale and add two arrays `g_a` and `g_b` into a third array `g_c` in Fortran is: `subroutine ga_add(alpha, g_a, beta, g_b, g_c)`. Since several ARMCI function calls may be made simultaneous at the scale of the entire system, controlling the flow of these messages is a critical problem to address.

GA is optimized to overlap intra-node data transfers in shared memory and inter-node data transfers using non-blocking ARMCI calls.

2.3 The ARMCI Runtime System

GA uses ARMCI as the primary communication layer. Neither GA nor ARMCI can work without a message-passing library and elements of the execution environment (job control, process creation, interaction with the resource manager). ARMCI, in addition to being the underlying communication interface for GA, has been used to implement other communication libraries and compilers [11, 21]. ARMCI offers an extensive set of functionality in the area of RMA communication: 1) data transfer operations (Get, Put Accumulate); 2) atomic operations; 3) memory management and synchronization operations; and 4) locks. Communication in most of the non-collective GA operations is implemented as one or more ARMCI communication operations.

ARMCI supports blocking and non-blocking versions of contiguous, strided and vector data transfer operations along with Read-Modify-Write operations. ARMCI uses the fastest available mechanism underneath to transmit data. For example, it uses shared memory with-in the node and, on the Cray XT5 system, uses Portals library for inter-node communication, as will be detailed in this presentation. ARMCI provides collective memory allocation interfaces, which allocate communicatable memory[2]. On the Cray XT5 system, the Portals library can be used to transmit data in a one-sided fashion from any address in the processes virtual memory address space, as long as a Memory Descriptor [8] representing that memory is posted to the portals layer. Before we describe the design of one-sided communication in ARMCI that enabled GA and NWChem to run at petascale, we briefly describe the validation benchmarks and discuss the network connection details that were required to setup one-sided communication in ARMCI.

3. VALIDATION BENCHMARKS AND CONNECTION SETUP

Strategies for designing ARMCI interfaces on various networks have always been based on benchmark performance and user (Global Arrays, Co-Array Fortran, etc) requirements. Therefore the challenges are different for

[2]Memory is communicatable when all the necessary steps required by the communication library in order to send messages to this memory are performed at its allocation

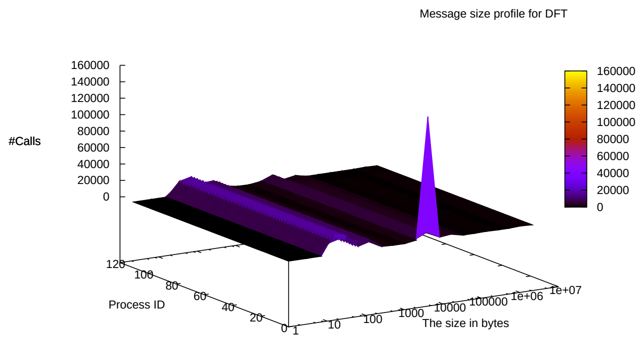


Figure 2: Message Profile of DFT: Total messages per process divided in power of two bins

various systems [17–20, 25]. The size of targeted systems for earlier efforts was hundreds or thousands of cores at the most. Issues like memory usage for connection setup and duplication of this between the message passing library and ARMCI were not considered as bottlenecks due to the small number of processing cores. Design decisions were focused on performance of each individual functionality and microbenchmarks.

Systems today have hundreds of thousands of cores and systems with millions of cores are not too far in the future. For example, the Jaguar Cray XT5 at ORNL has over 200,000 processing cores. For such large systems it becomes imperative to consider the impact on the applications for every design choice. Choices made during connection setup also need thorough consideration. We will briefly describe the validation benchmarks used throughout our design before discussing the connection setup.

3.1 Validation Benchmark

We designed a Get/Accumulate microbenchmark to run on two nodes (16 cores). All but one process in the benchmark are computing. The process 0 (which is on node 0) does a one-sided, non-contiguous, ARMCI Get and a one-sided ARMCI Accumulate to each of the eight processes on node 1. The objective of this microbenchmark is not to measure the impact of communication on computation but, rather, the vice-versa. This microbenchmark is run for messages sizes varying from 8 bytes to 128KB.

In addition to this microbenchmark, the DFT module of NWChem is used to validate the choice during each step of the design (DFT and its details are described in detail in Section 5). The characteristics of message sizes and the corresponding computation/communication intervals of DFT reflect that of some of the other NWChem modules. Figure 2 shows the sizes and the frequency of different messages of a 128-process run. The profile measures the Exchange-Correlation (XC) kernel of DFT, including approximately 8 million total ARMCI Put, Get and Accumulate calls. During the measurement, bins are formed with power-of-two message sizes, the number of messages are counted into these bins based on the size. For example, the 4KB message size point in the graph represents all messages of size between 2048 bytes and 4095 bytes.

Process 0 has a spike at the 4KB point, it sends 150,000 more messages in the 2k-4k range than all other processes.

3.2 Connection Setup and Management

Because of the design of the Cray provided software stack on the XT5, connection setup and management, which is otherwise a very challenging issues for ARMCI, is almost a no-op on the Cray XT5. Application Level Placement Scheduler (ALPS) is the Cray supported mechanism for placing and launching applications, ARMCI uses ALPS for process startup and placement on the Cray XT5 system.

One of the first steps in the initialization of ARMCI is network connection setup. This part of ARMCI and is dependent on the support from the underlying network hardware and its communication software library. On the Cray XT5, the Portals communication library is the lowest level library available to program the Seastar2+ network interconnect. The Portals library doesn't require its users to maintain connection state. Just a Network ID (NID) and Process ID (PID) pair is sufficient to send a message to any process in the system. Even this NID/PID pair doesn't need to be exclusively stored by the user (in this case ARMCI). An array of NID/PIDs is created in shared memory and a pointer to this shared location may be obtained. This eliminates the need for redundant copies of the NID/PID information needed for subsequent communication to the processes in the rest of the system. Note that despite the fact that NID/PID are merely two integers, on a system with hundreds of thousands of cores, just a copy of this on each process amounts to a lot of memory that would otherwise be available to applications.

To support the connection-less Portals interfaces, the Cray Seastar2+ allows for 256 simultaneous message streams. When additional streams need to be initiated (or in case of resource exhaustion), the Cray BEER (Basic End to End Reliability) protocol does the necessary flow control and handles reliability completely transparent to the user. This makes it much easier to write the network connection setup code for the Portals communication layer. We believe "hiding" (e.g. Seastar) or eliminating the need to maintain pair-wise connection state at the user level is essential to the scaling of PGAS runtime systems.

4. DESIGN OF ARMCI ONE-SIDED COMMUNICATION ON XT5

All contiguous ARMCI Put/Get interfaces were directly implemented on top of the Portals Put and Get calls. The three categories of one-sided calls in ARMCI to be considered during the design are: a) non-contiguous ARMCI Put/Get; b)accumulate; c) Read-Modify-Write(RMW); and d) Lock/Unlock. In addition, there are the collective memory allocation operations to prepare communicatable memory. We first started with a naive solution (described in [24]), translating all the above mentioned categories into multiple contiguous portals calls.

Several techniques for transmitting non-contiguous data have been discussed in Tipparaju et al. [25], all of them can be applied here. However, preliminary benchmarking (cf. Section 4.2) demonstrated that the server-based technique was ideal for non-contiguous and atomic operations on the XT5. In this technique, a communication helper thread is spawned on each node. One-sided messages that don't directly have a corresponding portals call are packed and sent to the communication helper thread. The helper thread receives, unpacks and processes the messages on behalf of all the processes on the node. For the rest of this discussion, all the application processes are referred to as clients and the Communication Helper Thread is referred to as CHT. Before discussing CHT, we first describe how CHT can access the memory of any client on the node.

4.1 Collective Memory Allocation

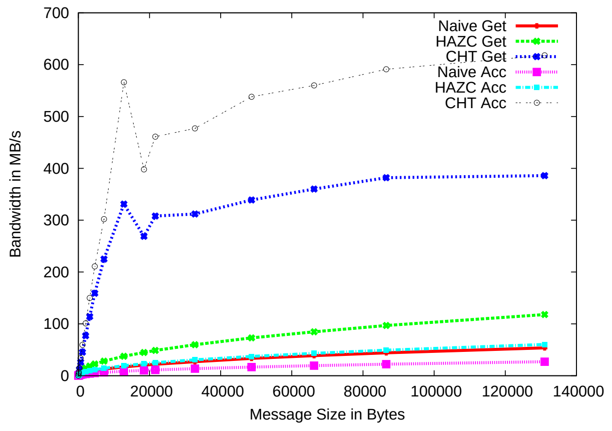
The collective memory allocator in ARMCI is used to allocate memory for global arrays of data. One restriction of ARMCI library is that ARMCI one-sided calls may only access remote memory when it is allocated with an ARMCI memory allocator. Both collective and non-collective versions of the memory allocator are provided in ARMCI. System V shared memory is allocated on the Cray XT5 by the ARMCI memory allocators. This enables intra-node data movement through shared memory. It also makes the entire memory available for remote communication on a node accessible by any client on the same node, including CHT.

4.2 Communication Helper Thread (CHT)

The CHT in ARMCI (also referred to as the Data Server) has traditionally been spawned either as a duplicated process (using `fork()` system call) or as a thread by using the pthreads library. However, neither of these methods are suitable for Cray XT5. The child process must not inherit the parent's stack as this contains "network identity" information for the portals library to identify a process. This network identity, which seeps into multiple network data structures in the portals library, prevents CHT from initializing or using the Portals library. To address this problem, we used the `clone()` system call to start CHT with a clean stack. This enables CHT to initialize its own Portals communication including the network identity, and use it for sending and receiving messages.

The lowest ranked client in every node clones to create the dedicated thread CHT. This thread is then used in receiving non-contiguous messages, accumulate messages, read-modify-write, and lock operations. A CHT receives messages for all the clients on the same node. It waits for messages from the clients. CHT does so by waiting on events generated from any communication to its portal. If a CHT were to poll on this event queue, valuable CPU resources may be wasted in polling. The Cray Portals library allows for an alternative. To avoid polling, we set the environment variable `CRAY_PORTALS_USE_BLOCKING_POLL` from CHT to make the thread block when no messages are in the queue and leave the compute core for the client to do their computations. This logic is simple and yet has several characteristics that reduce the pressure on the network.

- Fair communication scheduling: this is probably the biggest advantage. Each source that needs to send a message via CHT can only send n messages asynchronously (n determined based on the total process count to limit the server buffer memory). It can only send the $n+1$th message when one of the previous n has been acknowledged. The first advantage is that even when the server has thousands of messages to be processed in the queue, n messages can still be sent by each client asynchronously. The second advantage



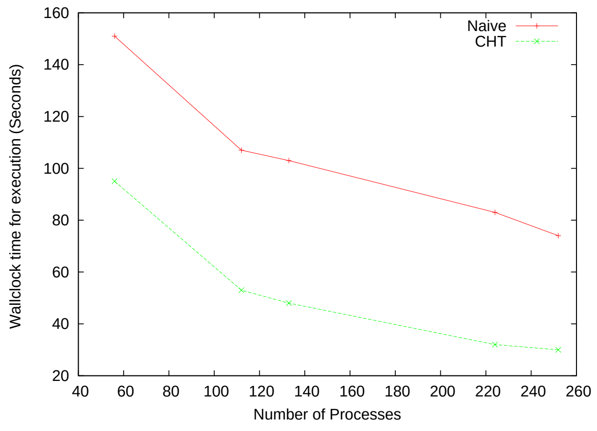


Figure 3: Naive implementation vs. Helper thread for non-contiguous Get and Accumulate (left) and NWChem DFT Benchmark (right)

is that if a client has to send a very large number of messages to CHT, it prevents other clients from starving for CHT's processing of its messages.

- Many to one case: when pathological scenarios (such as all clients in the system trying to increment one value) arise, CHT is able to process them in a sequential and fair manner. For example, in a general case, when a network is used to implement a lock functionality[3], every failed attempt from every client to acquire a lock results in one additional network message. On the other hand, when using CHT, a lock request is only sent once and the request stays on at CHT until it succeeds.

- Natural Serialization: operations such as Accumulate merely require the guarantee of the commutative property [16]. Hence they can be implemented without a need for explicit locks if multiple simultaneous accumulates can be serialized. CHT, with its network message queues, provides the serialization at no additional cost (such as the cost of a lock). When there are many threads serving as CHTs, each thread still provides natural serialization for the client processes it supports (communication between clients served by a CHT is via a CHT).

- Cache Reuse: All incoming message that CHT processes have the potential to be reused by the application running on the same CPU socket. Note that we haven't measured the percentage of cache reuse for our application.

We have two different tests that evaluate the benefits from CHT:

- The first, Figure 3 (left), is a microbenchmark that measures the bandwidth of Accumulate and non-contiguous Get. It compares the difference between using CHT vs. converting the non-contiguous data transfer to multiple portals calls (these two methods implement Accumulate as described in [16]). This is also a benchmark that selects the most suitable method from the ones described in [25]. In addition to the Naive and the CHT method, the Host-Assisted Zero-Copy (HAZC) method is also shown in Figure 3 (left). In this method, non-contiguous calls are converted into portals network scatter/gather calls. It is important to note that that the HAZC numbers in the graph are not much better than our naive approach as both of them are implemented in software (the portals vector calls don't utilize any additional Seastar hardware support and are implemented in software).

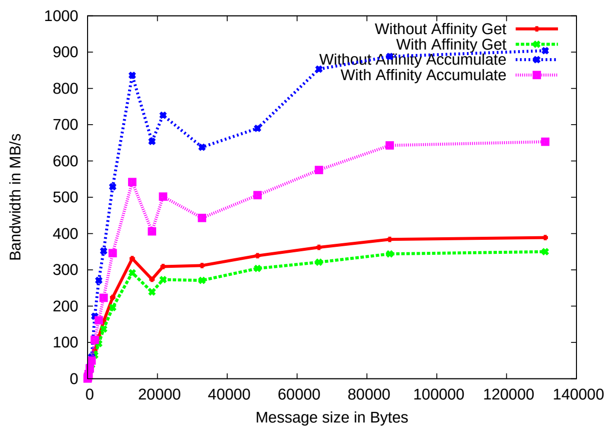
- The second, Figure 3 (right), is a DFT run as described in Section 3.1. The DFT benchmark executes in much shorter time when using CHT.

[3]that can be implemented by using the Portals Ptl_GetPut call

Clearly, the helper thread approach is more suitable for both the microbenchmark (Accumulate and non-contiguous data transfers) and the application benchmark (DFT). Many of such helper threads may be spawned for larger SMP- and Many-core architectures. In our case, with the 8-core XT5 node, one helper thread was sufficient. Contiguous Get and Put messages are transmitted directly through the portals calls without the involvement of CHT while non-contiguous, accumulate, lock, and Read-Modify-Write(RMW) operations are sent to CHT to be processed on behalf of all the processes on the remote node.

4.3 Core Affinity

CHT shares the CPU core with the client that spawns it, in our case, this is the lowest ranked application process. However, the scheduling of CHT needs thorough consideration. Leaving it to run on the same core as the client causes imbalance in computations because CHT also processes *accumulate* calls (accumulate, in addition to data movement, also requires computation). Rapidly moving the helper thread between CPU cores is expensive, it impacts the cache usage and has latency associated with rescheduling. Hence we adopt a formula that considers the following logic to reschedule the helper thread: `n*(8*compute_bytes + communication_bytes)>Threshold`. If this boolean expression is true, the CHT thread is rescheduled. However, for most of the NWChem runs on the Cray XT5, this is not required for the reason described below. The glibc `sched_setaffinity` call was used to change the CHT affinity.



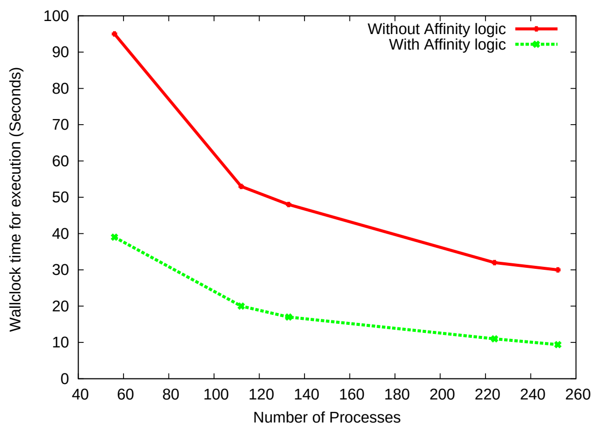
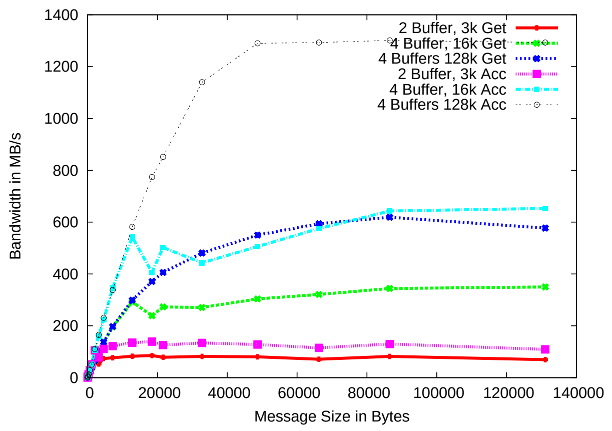


Figure 4: With and without thread affinity logic: non-contiguous Get and Accumulate (left) and NWChem DFT Benchmark (right)



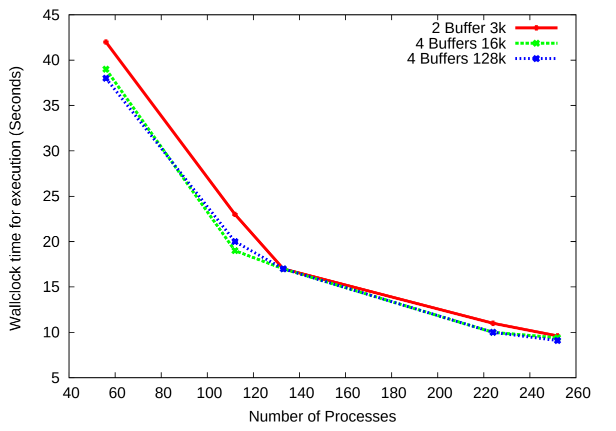
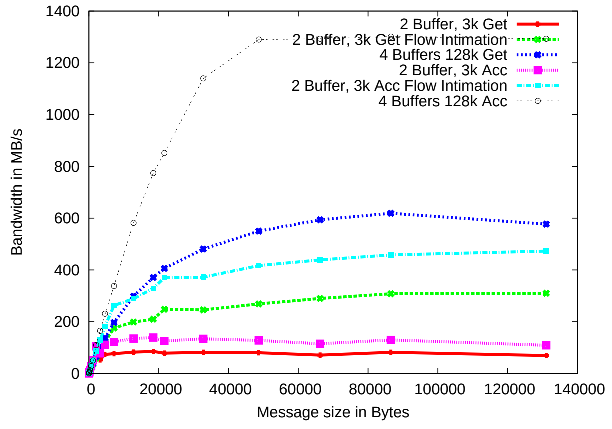


Figure 5: Number and size of client buffers: non-contiguous Get and Accumulate (left) and NWChem DFT Benchmark (right)

The 8-core Jaguar Cray XT5 system at ORNL has about 75 GigaFlops floating point compute capacity per node, 25 GB/s shared bandwidth to memory and 3GB/s unidirectional shared bandwidth to network. This gap between the memory bandwidth and floating point performance can be utilized to our advantage. CHT checks to see if the application has been spawned on all the cores on the XT5 node. If not, CHT sets its affinity to any unused core on the node. Instead of contending for memory bandwidth with all eight clients and one CHT, NWChem may also run 7 compute processes and let the CHT utilize the remaining CPU core. There are some modules of NWChem that benefit from running on 7 of the 8 compute cores on the XT5 node. When only 7 out of the 8 CPU cores are used for computation, the helper thread is automatically scheduled to the free core. The affinity logic helps balance the CHT load better for codes that use all 8 cores on the XT5 node and prevent interference with applications computation for codes that use less than all 8 available cores on the XT5 node. We ran both the microbenchmark and DFT application kernel to compare the runs with and without affinity. Figure 4 shows both the runs. For the DFT benchmark here (Figure 4 (right)), the advantage of balancing the CHT load by changing affinity can be clearly seen from the figure. The microbenchmark (Figure 4 (left)) however shows the opposite behavior. This is due to the tight loop in which the measurements are taken. The cost of moving this thread around reflects directly on the tight communication loop so does the cache footprint. Given the benefit to the application benchmark, the affinity switching logic has been made a configurable option.

4.4 Buffers and Flow Control

Buffer management and flow control are critical to the performance for messages transmitted via CHT. Several techniques have been proposed for buffer management [6,19]. Here we describe the details of buffer management and its flow control. We propose a scalable technique referred to as *flow intimation*, to achieve performance with a few buffers. There are two kinds of buffers in use when utilizing CHT for message transmission: client buffers and CHT buffers. Client keeps a few (static number) of buffers to send data to CHT. CHT has to allow for at least one unacknowledged message per client and hence requires at least as many buffers as the number of clients. We discuss the performance



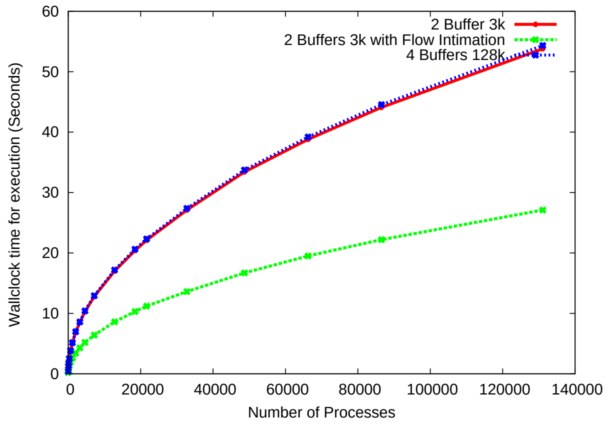


Figure 6: Flow Intimation: non-contiguous Get and Accumulate (left) and NWChem DFT Benchmark (right)

impacts of the size and number of buffers, as well as that of flow control.

4.4.1 *Number and Size of Buffers*

For a scalable solution, CHT bounds the value of the total memory it uses for receiving client messages. The upper bound on the individual buffer size is based on the number of processes that are being used. Although there isn't an accepted or recommended size of the total memory to use inside a GAS runtime, considering our largest configuration (150K process run), we limit our memory usage to under 5% of XT5 node memory. By limiting the total amount of memory consumed by the GAS runtime, we are able to scale the application successfully by allowing it to use most of the available memory on the system. However, Figure 5 clearly shows that in the Get/Accumulate microbenchmark, more buffers of larger size show significant increase in bandwidth, in some cases the bandwidth was doubled. In Figure 5 (left), the microbenchmark shows three different cases for Get and Accumulate one-sided calls. The label in the figure indicates how many buffers are available at the CHT for each client and the size of the buffer. For example, `2 Buffer 4K` means CHT has 2 Buffer per client and the size of each such buffer is 4 KB. It can be seen in the figure that the 4 buffers each of 128 KB per client case does much better in the microbenchmark (because all the messages under 128 KB just fit in the buffer and splitting into multiple buffers is not necessary). But for 150,000 clients, this computes to about 72GB of buffer space per node. The challenge here is to minimize the CHT buffer space and yet not compromise performance. This is addressed by the flow control and the flow intimation techniques as detailed below.

4.4.2 *Flow Control*

For every message the client sends from the client buffer to the CHT buffer, CHT sends an acknowledgment back to the client. Before sending the acknowledgment back to the client, the CHT checks to see if it can coalesce the acknowledgments. Coalescing logic is only enabled when there are at least four buffers per client. The acknowledgment both synchronizes the client-CHT buffers and does flow control. In addition to this, all the contiguous Put and Get messages that are not transmitted via the buffers or CHT have a simple source-credit based flow control. A combination of CHT acknowledgments and credit is used to implement the Fence[4] operation. When using the client-CHT buffers for transmitting non-contiguous data and accumulate, the performance depends on the number of buffers and their size.

4.4.3 *Flow Intimation*

To achieve the effect of using many large buffers and yet not increase the overall CHT buffer usage to over 5% of total memory per node, we propose and implement the *Flow Intimation* technique. The GAS Runtime message transmission logic is aware of whether an individual message is first among many. Some scenarios of this awareness include:

- transmissions of multidimensional strided data involve the parsing of data structure through a recursive packing routine, hence some information about subsequent transmissions is available
- any large accumulate or vector message that needs to be broken down into smaller blocks for transmission (based on buffer size) knows the total number of upcoming blocks
- any one-sided call that needs to fence before sending the message (this is typically done for correctness, for example, to ensure message ordering).

As an example, to transmit a 128 KB non-contiguous block with 16 KB buffers, the message is broken down into 16 KB chunks, and individual chunk is packed and transmitted in a pipelined fashion with flow control (described in [19]).

In scenarios such as these, client gives CHT an *intimation* of upcoming flow and continues transmission as usual. When CHT receives a message tagged with this intimation, it checks to see if additional buffers (spare buffers) can be used for the transfers it has been intimated about. It then tags the flow control response with offset and the number of these spare buffers available. In its regular handling of the flow control response, if the client sees the acknowledgment tag

[4]fence is mechanism to check for remote completion of all initiated one-sided operations

to its intimation, it immediately sends the rest of the buffers based on the CHT response and expects just one aggregated flow control response. Notice that flow intimation requires just four additional long integers to be sent along with the messages already being transmitted. Hence the lack of buffers at CHT doesn't add any more delay to the communication.

Figure 6 is very similar to Figure 5, but it includes the intimation logic. The amount of memory that can be used for implementing this logic is flexible. For the microbenchmark and application benchmarks in Figure 6, the size of the spare buffer was 32 KB. Using a larger spare buffer (larger than 32 KB) will certainly benefit the microbenchmark (left) however, the impact on the application benchmark was negligible. Hence with the same, two 3-KB buffers per client, and merely 4 additional spare 32-KB buffers for all the clients, flow intimation technique demonstrated significant benefit. In case of the microbenchmark the bandwidth was improved by three fold, the application benchmark performed as well as the large buffer case. Since Global Arrays is a library based GAS model and ARMCI is not a compilation target for Global Arrays, the use of flow intimation here is limited to non-contiguous data transfers and accumulate operations. However, for scenarios where ARMCI is used as a compilation target (for example, in Chapel language [4] or Rice Co-Array Fortran [11]) there are many more opportunities to utilize this technique.

5. PERFORMANCE AND SCALING WITH A SCIENTIFIC APPLICATION

In this section, we report results from two most widely used electronic structure methods in NWChem: Density Functional Theory (DFT) and Coupled Cluster (CC). DFT is the workhorse of electronic structure for its balance between computational cost and accuracy (1998 Nobel prize in Chemistry), whereas the more expensive CC method, in its CCSD(T)[5] incarnation, is labeled as the "gold standard" [13] because of its remarkable accuracy. We have used DFT primarily to measure the benefit of our CHT design, and CC primarily to verify that the CHT design with flow intimation has accomplished the goal of scaling an application to petascale on GA.

5.1 Density Functional Theory (DFT)

The DFT siosi7 benchmark reported below computes the matrix elements of the Local-Density Approximation (LDA) Exchange-Correlation potential on molecular fragments for a total of 3554 basis functions. The distributed-data algorithm adopted in this computational kernel makes extensive use of two GA calls: `ga_get` and `ga_acc`. The `get` operation is used to fetch patches of the Density matrix $D_{\mu\nu}$ to compute the charge density function $\rho(r_q)$ on a set of grid points following Equation 1

$$\rho(r_q) += \sum_{\mu\nu} D_{\mu\nu}\chi_\mu(r_q)\chi_\nu(r_q) \tag{1}$$

[5]CCSD(T) is one of several CC methods that estimate the effect of electron correlations by considering single, double and triple excitations; single and double excitations are fully computed with a self-consistent approach, while the contribution of the triple excitations is computed perturbatively

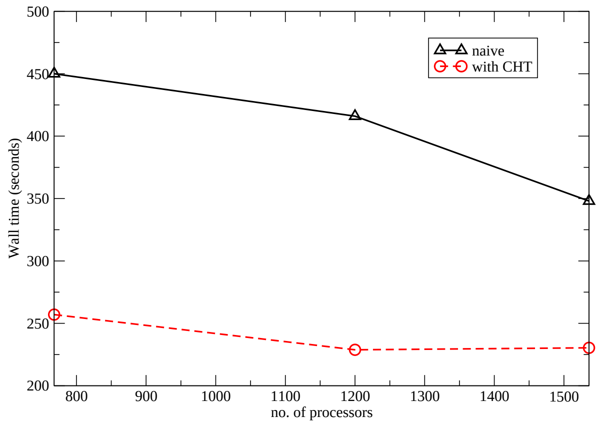


Figure 7: Walltime to solution for the DFT siosi8 benchmark (7108 basis functions). The timing includes the complete calculation to the convergence of LDA wavefunction.

The charge density function $\rho(r)$ can be used to compute the Exchange-Correlation potential $V_{XC}[\rho(r)]$. In the next step, the Exchange-Correlation potential $V_{XC}[\rho(r)]$, combined with Gaussian functions $\chi_\lambda(r)$ and integration weight w, is then used for the evaluation of the Exchange correlation matrix element $F_{\lambda\rho}$ by means of numerical quadratures according to Equation 2

$$F_{\lambda\rho} += \sum_q w_q \chi_\lambda(r_q) V_{XC}[\rho(r_q)]\chi_\rho(r_q) \tag{2}$$

The resulting matrix element $F_{\lambda\rho}$ is then stored into a distributed global array by means of the GA `accumulate` operation. The messages used in this implementation correspond both to the patches of the Density matrix $D_{\mu\nu}$ that are fetched, and to the patches of the XC matrix $F_{\lambda\rho}$ that are stored. As described earlier, their size is fairly small, in the order of 4KB, therefore resulting in an algorithm that is more sensitive to the network latency than to the bandwidth. As shown in the DFT scaling figures in the previous pages, our optimized ARMCI implementation on the 8-core Cray XT5 has enabled this critical NWChem algorithm to achieve very good efficiency. Larger tests cases (using larger molecules with a correspondingly higher number of basis functions) will result in even better scalability at larger processor counts.

In Figure 7 we report the walltime for the complete calculation of the LDA wavefunction with the DFT siosi8 benchmark; this benchmark is slightly large than siosi7 as reported above (7108 vs. 3554 basis functions, respectively). Note that in the previous DFT figures only the time to compute the XC kernel is reported. The most striking point in this figure is the larger effect in performance improvements when CHT is used for the `get` operations. Process counts up to 2K are shown. Beyond that, the wall clock time stays flat primarily due to DFT's latency sensitivity and the small array sizes.

5.2 CCSD(T)

As stated above, the CCSD(T) is more expensive than the DFT methods (its cost roughly scales as N^7, while DFT scales as $N - N^3$, where N is the number of basis functions). Therefore it is a natural candidate for demonstrating peta-class performance once an efficient parallel implementation is in place. We reported performance measurements by using as the base the parallel implementation of CCSD(T) in NWChem of Kobayashi and Rendell [14], which was designed to effectively utilize massively parallel processors and to minimize the use of I/O resources.

Previous CCSD(T) runs with the same NWChem implementation achieved a performance of 6.3 TFlops using 1,400 processors [22] on a cluster of Itanium2 processor with a Quadrics QsnetII network, while more recent runs at PNNL utilized 14,000 processors on an InfiniBand network of Opteron processors [10]. What distinguishes the benchmark numbers reported here is the unprecedented scale of the calculations and floating-point performance achieved. We run a series of benchmark with the 5.1 version of NWChem [9].

We used the $(H_2O)_{18}$ water cluster with a modified cc-pvtz [12] basis set for a total of 918 basis functions. This benchmark was run on the Jaguar XT5 at ORNL. Figure 8 shows the walltime for $(H_2O)_{18}$ for different processor runs. The Jaguar supercomputer used for these tests was recently upgraded to Hex-core from Quad-core increasing the number of cores per node from 8 to 12. In the graph on the left side of Figure 8 we show the scaling for up to 90,000 cores on the Quad-core Jaguar (before the upgrade). All 8 cores on each node were used for computation. The last data point at 90,000 processes reached a sustained 64-bit floating-point performance of 358 TFlops. In the same figure, the graph on the right side of the figure shows the scaling of $(H_2O)_{18}$ after the upgrade. In this case, only 10 of the 12 cores per node were used for computation. One core was exclusively left for CHT utilization while the core 0 on socket 0 was left unused (the reason for this had to do with the amount of OS activity that was measured on this core and is outside the scope of this paper). The last data point at 180,000 processes reached a sustained 64-bit floating-point performance of 718 TFlops.

The above accomplished scaling with DFT and $(H_2O)_{18}$ the unprecedented double precision floating point performance with $(H_2O)_{18}$ have adequately validated the effectiveness of our ARMCI Runtime design and exemplified the impact of GAS models for the Cray XT5.

6. CONCLUSIONS

We have examined a variety of issues in designing a scalable ARMCI communication layer for the Global Arrays (GA) programming model. Accordingly, we have exploited many ideas used across the GAS community so far to implement a petascale-ready GAS runtime. We demonstrated a complete, successful, petascale GAS runtime solution for the GA GAS model. We have also demonstrated the scaling of a real scientific application that uses the GA model. This solution is applicable to other GAS runtime and hence can potentially be lent as a good precedent to their scalability.

In achieving our goal to enable a highly scalable GA model for Jaguar, we have dealt with various issues, such as connection setup and management, the CPU core affinity, buffer management, and the flow control of buffers. Particularly, we have found that flow control is critical to the GA model because of the one-sided communication they advocate and use. Buffering is necessary to implement some functionality that network cards and their communication interfaces lack. The buffer space for communication needs to be limited for memory intensive applications. At the scale of hundreds of thousands and millions of cores, smarter ways to utilize buffer space for communication and deliver performance are necessary. We have designed a Flow Intimation technique to reduce the burden on the network, control the number of total messages in flight, reduce contention, and yet be able to deliver good performance. Our implementation of Flow Intimation demonstrated the success with 180,000 cores. We believe issues addressed here at petascale, such as those on buffering and flow control, can offer an exemplary perspective on what it will take to support a GAS model beyond petascale.

In future, we look forward to further optimization of the GA model on the Cray XT5 system, such as Jaguar, or better scalability and scientific productivity. We also plan to study the applicability of ARMCI to enable other GAS models such as UPC and Co-Array Fortran.

7. ACKNOWLEDGMENTS

This work was funded by National Center for Computational Sciences, and in part by a UT-Battelle grant (UT-B-4000087151) to Auburn University. This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This work reflects over a decade of research and development in ARMCI, Global Arrays and NWChem. Several people and several research ideas have contributed to ultimately scaling these libraries and applications to petascale today. Jarek Nieplocha had led the research and development of Global Arrays toolkit for over a decade. We thank all the contributors to these libraries. We thank all the people at Cray who gave valuable suggestions during this work, particularly Kitrick Sheets for his help in addressing several issues we faced along the way.

8. REFERENCES

[1] Global arrays toolkit. http://www.emsl.pnl.gov/docs/global.

[2] Top500 list. http://www.top500.org.

[3] Upc specifications, v1.2. http://www.gwu.edu/ upc/publications/LBNL-59208.pdf.

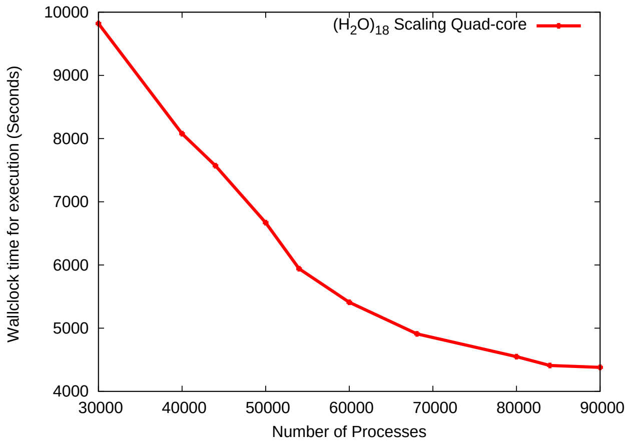
[4] Chapel language specifications, v0.780, 2006. http://chapel.cs.washington.edu/spec-0.780.pdf.

[5] Report on experimental language X10, 2008. http://dist.codehaus.org/x10/documentation /languagespec/x10-170.pdf.

[6] B. W. Barrett, G. M. Shipman, and A. Lumsdaine. Analysis of implementation options for mpi-2 one-sided. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[7] D. Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[8] R. Brightwell, R. Riesen, B. Lawry, and A. Maccabe. Portals 3.0: protocol building blocks for low overhead communication. *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 164–173, 2002.



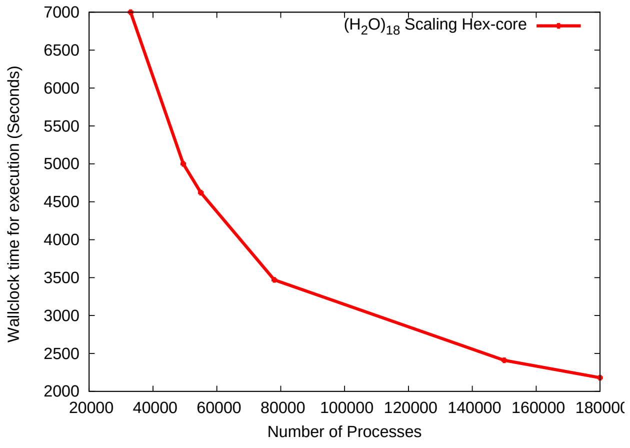


Figure 8: Walltime to solution for $(H_2O)_{18}$ CCSD(T) benchmark on XT5

[9] E. Bylaska and et al. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 5.1*, 2007.

[10] W. A. de Jong and S. Krishnamoorthy. private communication, 2008.

[11] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multi-platform co-array fortran compiler. In *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, pages 29–40, Sept.-3 Oct. 2004.

[12] T. H. Dunning. Gaussian basis sets for use in correlated molecular calculations. i. the atoms boron through neon and hydrogen. *The Journal of Chemical Physics*, 90(2):1007–1023, 1989.

[13] T. H. Dunning, K. A. Peterson, D. E. Woon, and A. K. Wilson. Quantifying quantum chemistry. In *American Conference on Theoretical Chemistry*, 1999. unpublished.

[14] R. Kobayashi and A. P. Rendell. A direct coupled cluster algorithm for massively parallel computers. *Chemical Physics Letters*, 265(1-2):1 – 11, 1997.

[15] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[16] J. Nieplocha, V. Tipparaju, and E. Apra. An evaluation of two implementation strategies for optimizing one-sided atomic reduction. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 9*, page 215.2, Washington, DC, USA, 2005. IEEE Computer Society.

[17] J. Nieplocha, V. Tipparaju, and M. Krishnan. Optimizing strided remote memory access operations on the quadrics qsnetii network interconnect. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 28, Washington, DC, USA, 2005. IEEE Computer Society.

[18] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High Performance Remote Memory Access Communication: The Armci Approach. *International Journal of High Performance Computing Applications*, 20(2):233–253, 2006.

[19] J. Nieplocha, V. Tipparaju, A. Saify, and D. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 164–173, 2002.

[20] J. Nieplocha, V. Tipparaju, A. Saify, and D. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. In *In: Proc. Workshop Communication Architecture for Clusters (CAC02) of IPDPS '02, Ft*, 2002.

[21] K. Parzyszek. *Generalized portable shmem library for high performance computing*. PhD thesis, Ames, IA, USA, 2003. Co-Major Professor-Kendall,, Ricky A. and Co-Major Professor-Lutz,, Robyn R.

[22] L. Pollack, T. L. Windus, W. A. de Jong, and D. A. Dixon. Thermodynamic properties of the c5, c6, and c8 n-alkanes from ab initio electronic structure theory. *The Journal of Physical Chemistry A*, 109(31):6934–6938, 2005.

[23] A. Shet, V. Tipparaju, and R. Harrison. Asynchronous programming in upc: A case study and potential for improvement. In *Workshop on Asynchrony in the PGAS Programming Model Collocated with ICS 2009*, Sept. 2009.

[24] V. Tipparaju, A. Kot, J. Nieplocha, M. Bruggencate, and N. Chrisochoides. Evaluation of remote memory access communication on the cray xt3. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–7, March 2007.

[25] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda. Host-assisted zero-copy remote memory access communication on infiniband. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 31–, April 2004.