Runtime Techniques to Enable a Highly-Scalable Global Address Space Model for Petascale Computing

Vinod Tipparaju · Edoardo Apra · Weikuan Yu · Xinyu Que · Jeffrey S. Vetter

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

V. Tipparaju (✉) · E. Apra · J. S. Vetter
Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge,
TN 37831, USA
e-mail: tipparajuv@gmail.com

E. Apra
e-mail: aprae@ornl.gov

J. S. Vetter
e-mail: vetter@ornl.gov

W. Yu · X. Que
Department of Computer Science, Auburn University, Auburn, AL 36849, USA
e-mail: wkyu@auburn.edu

🌀 Springer

technique that we have introduced in this paper. Our optimized ARMCI implementation improves scalability of both the Global Arrays programming model and a real-world chemistry application—NWChem—from small jobs up through 180,000 cores.

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) [35]. 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.

In this regard, Global Address Space (GAS) programming models - both the Partitioned and Asynchronous Partitioned Global Address Space (PGAS)—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, PGAS languages like Unified Parallel C (UPC) [36], Co-Array Fortran (CAF) [9], and GAS libraries such as Global Arrays (GA) Toolkit [13] have the unique ability to expose features, such as low-overhead communication or GAS support in the underlying hardware. Systems that lack one or more of these features typically result in poor performance.

Conceptually, GAS models do not differentiate between *local* and *remote* accesses. By contrast, 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* PGAS model, has emerged to add additional capabilities such as remote method invocations. IBM's X10 language [31] and Asynchronous Remote Methods (ARM)[32] in UPC have pioneered this new model.

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 [4] and Aggregate Remote Memory Copy Interface (ARMCI) [26], both of which are used in numerous GAS 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 GAS model—GA—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 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. The 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 GA library in Sect. 2; we discuss the validation benchmarks we used and the connection setup details in Sect. 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 Sect. 4 (this section also introduces flow intimation); and finally we discuss the achieved performance in the context of NWChem in Sect. 5, discuss related work in Sect. 6 and conclude with future steps in Sect. 7.

2 An Overview of Global Arrays

The 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 library 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. The ARMCI runtime system helps it to support blocking and non-blocking data transfers for contiguous, vector and strided data transfers. The model of execution in GA is multiple single-threaded processes per shared memory node, with all sharing done through the GA library via the ARMCI runtime.

The GA library relies on a message passing library for process setup and some collective communication operations. For this reason, the GA Toolkit [13], as distributed,

---

[1] Portals is the lowest level communication library available on the XT5.
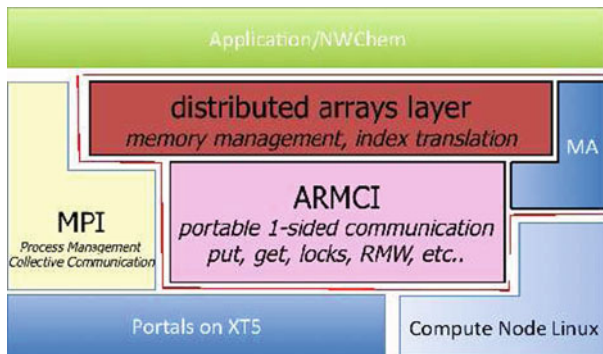
Fig. 1 The structure of the global arrays library on the Cray XT5

comes with a naive implementation of a message passing library called TCGMSG. One of the purposes of this light weight message passing library is to deal with situations when MPI and GA are unable to inter-operate (an example of this is the IBM SP system and the pre-Federation versions of the SP switch, which did not allow MPI and GA to be used concurrently). Unfortunately, TCGMSG is based on TCP/IP sockets making it unusable on the Cray XT5. We rely on the Cray provided MPI library on the XT5 system and work around the issues with utilizing MPI and GA together (as described in Sect. 4.2).

The structure of NWChem application, on top of the GA library, is shown in Fig. 1. The application (in our case NWChem) uses only the *GA interfaces*, a message passing wrapper (to initialize the message passing library), and the Memory Allocator (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 library: the *Distributed Array layer (DA)*, *ARMCI*, and the *MA*. MA provides simple interfaces to allocate "local" memory. We will further describe the DA layer, the GA interfaces, and ARMCI. The GA interfaces expose the distributed array layer to the application.

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 [25]. Three main categories of GA interfaces are of interest,

including array creation, one-sided, and data parallelization. All these interfaces have both C and Fortran bindings.

The array creation interfaces result in the creation of data structures that are later used by the DA layer. Subsequently the 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, data structure and the allocation, and 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. For example a GA Get call first requires determination of data locality: the physical location of the data in the distributed/partitioned address space needs to be identified. Then indices corresponding to where the data is located (on the identified process) need to be found. When this information is available, multiple ARMCI non-blocking Get calls are made, one for each remote destination that holds a part of the data. After all the calls are issued, they are waited upon until completion. 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 Sect. 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 also 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 [9,28]. ARMCI offers an extensive set of functionality in the area of Remote Memory Access (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. 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 [5] 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 (GA, CAF, etc) requirements. Therefore the challenges are different for various systems [21,22,24,26,33]. 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

Our experiments were performed on two configurations. In the first configuration, is a Cray XT4. Each compute node contains a quad-core AMD Opteron 1354 (Budapest) processor running at 2.1 GHz, 8 GB of DDR2-667 memory, and a SeaStar2 router. The second configuration is the Cray XT5 Supercomputer. Each XT5 node has a two quad-core MD 2356 processors running at 2.3 GHz, 16 GB of DDR2-800 memory, and a SeaStar 2+ router. We designed a Get/Accumulate microbenchmark to run on two XT5 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
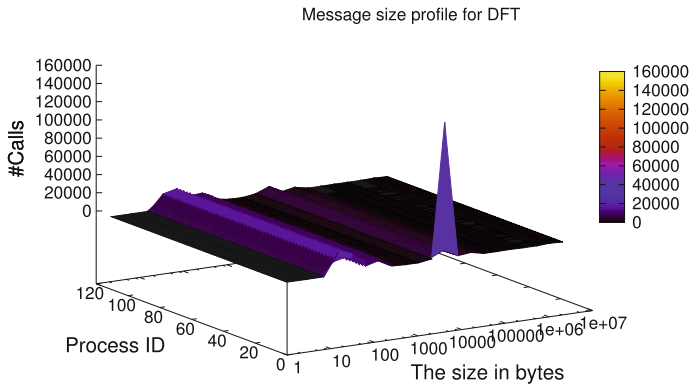
Message size profile for DFT

Fig. 2 Message profile of DFT: total messages per process divided in power of two bins

computation but, rather, the opposite. This microbenchmark is run for message sizes varying from 8 bytes to 128 KB.

In addition to this microbenchmark, the Density Functional Theory (DFT) module of NWChem is used to validate choices during each step of the design (DFT and its details are described in detail in Sect. 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 4 KB message size point in the graph represents all messages of size between 2048 bytes and 4095 bytes. Process 0 has a spike at the 4 KB point, it sends 150,000 more messages in the 2k–4k range than all other processes.

3.2 Connection Setup and Management

ARMCI typically relies on an external library for process setup. Cray XT5 has PMI (Process Management Interface) that is initialized by the job launcher (aprun) of Application Level Placement Scheduler (ALPS). 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 the network connection setup. This part of ARMCI is dependent on the support from the underlying network hardware and its communication software library. On networks such as InfiniBand, this has posed severe challenges in the past. Although InfiniBand developments such as the Extended Reliable Connection (XRC) and Shared Receive Queue (SRQ) [17] have alleviated this problem, they do not eliminate it. On the Cray XT5, the Portals communication library is the lowest level library available to program the Seastar2+ network interconnect. The Portals library does not require its users to maintain

connection state, 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 does not 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 memory for communication. We first started with a naive solution (described in [34]), translating all the above mentioned categories into multiple contiguous Portals calls.

Several techniques for transmitting non-contiguous data have been discussed in [33], all of them can be applied here. However, preliminary benchmarking (cf. Sect. 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 do not 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 GA 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 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 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.

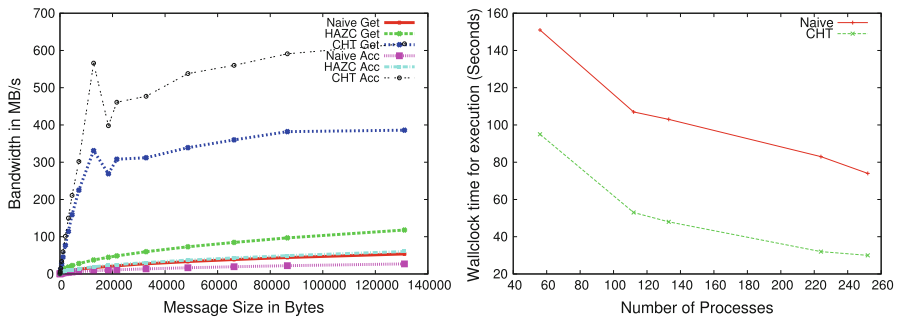[3] Lock can be implemented by using the Portals Ptl_GetPut call.

Fig. 3 Naive implementation vs. helper thread for non-contiguous get and accumulate (*left*) and NWChem DFT Benchmark (*right*)

– Natural Serialization: operations such as Accumulate merely require the guarantee of the commutative property [23]. 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 messages 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, Fig. 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 [23]). This is also a benchmark that selects the most suitable method for doing non-contiguous data transfers from the ones described in [33].
– The second, Fig. 3 (right), is a DFT run as described in Sect. 3.1. The DFT benchmark executes in much shorter time when using CHT.

For the first test, in addition to the Naive and the CHT methods, the Host-Assisted Zero-Copy (HAZC) method is also shown in Fig. 3 (left). In HAZC method, non-contiguous calls are converted into Portals network scatter/gather calls. Despite some hardware support for non-contiguous data transfers, the HAZC method does not demonstrate better performance than the CHT method (which packs and unpacks the data and hence has more host involvement). Figure 4 demonstrates the reason behind this. Figure 4 compares the middle ground that the HAZC method offers to an optimal scenario where the network is directly able to transmit well structured data types. In this figure a 3D section of an array is being transmitted to a 3D destination. The top line shows the HAZC method where the non-contiguous data is converted to I/O vectors (the size of the I/O vector can be very large). Multiple transmissions of such I/O vectors might be required depending on the network packet size (and the
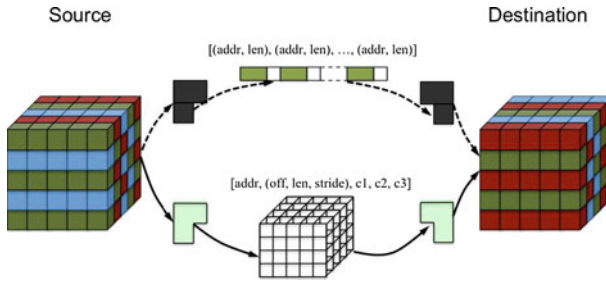
Fig. 4 Packing/unpacking and efficient data movement for non-contiguous data: HAZC vs. direct network support

number of I/O vectors it can hold). The line in the bottom half of the figure, in contrast, shows a scenario where the network interconnect offers direct support for transfer of different shapes of non-contiguous data. In this case, a single message can carry a compact representation of the entire 3D block. It is important to note 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 do not utilize any additional Seastar hardware support and are implemented in software).

Clearly, the helper thread approach is more suitable for both the microbenchmark (Accumulate and non-contiguous data transfers) and the application benchmark (DFT). Many 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) messages 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. When application is launched by Cray ALPS (discussed in Sect. 3.2), there is an option to assign core-affinity to the launched processes via a parameter to the `aprun` command. This allows the user to set the affinity of their application tasks to physical CPU cores. However, the CHT is launched inside the ARMCI library and shares the core with the process launching it. A single CHT (as in our case) serves all the application tasks with-in a given node. Hence, 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). As an alternative, constantly moving the helper thread between CPU cores is expensive, it impacts the cache usage and incurs latency that is associated with rescheduling. 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. The rescheduling is done
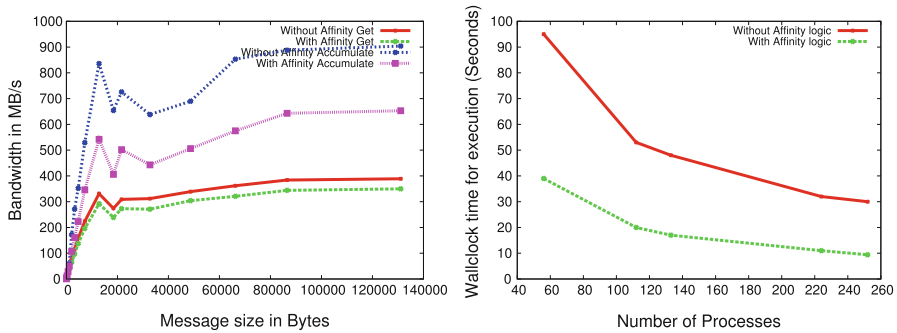
Fig. 5 With and without thread affinity logic: non-contiguous get and accumulate (*left*) and NWChem DFT Benchmark (*right*)

in a round robin fashion between all available cores. The glibc `sched_setaffin-ity` call was used to change the CHT affinity. For most of the NWChem runs on the Cray XT5, this is not required for the reason described below.

Considering the 8-core Jaguar Cray system at ORNL as example, it 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 the application's computation for codes that use fewer 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 5 shows both the runs. For the DFT benchmark here (Fig. 5 (right)), the advantage of balancing the CHT load by changing affinity can be clearly seen from the figure. The microbenchmark (Fig. 5 (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 [21,2]. 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
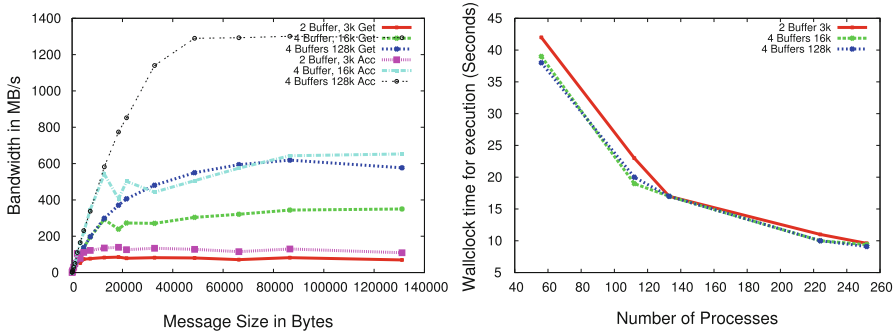
Fig. 6 Number and size of client buffers: non-contiguous Get and Accumulate (*left*) and NWChem DFT Benchmark (*right*)

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 impacts of flow control and the size and number of buffers.

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 (150 K 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, Fig. 6 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 Fig. 6 (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.

4.4.2 Techniques for Buffer Management

It can be seen in Fig. 6 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 results in about 72 GB of buffer space per node. The challenge here is to minimize the CHT buffer space and yet not compromise performance. There are several techniques proposed in literature for doing the buffer management.

Request-Allocate This techniques that relies on on-demand allocation starts off with minimal buffer size sufficient to send header messages and "allocates" buffer space as necessary. This technique requires a response sent for every request and incurs an additional round trip latency for every message sent. In addition to this, memory allocation may fail and hence this case needs to be handled as well.

Request-Release In our prior work [24] on the Quadrics network, we adopted a strategy that registers a request for a buffer with the CHT. Once CHT has buffer space available, it sends a response and the client proceeds to utilize the allocated buffer and upon completion of the task that requires the buffer, releases it. The advantage of this buffer management in comparison to *Request-Allocate* techniques is that this does not have memory allocation in the critical path of a message transmission. However, this techniques still incurs an additional round-trip latency for every message. This cost may amount to a small percent of a large message transmission but has significant impact on small messages.

Credit Based This popular technique utilizes a combination of credits and request-release for buffer allotment at the CHT. The CHT represents available buffer space with credits. The clients tracks the credits it has available and embeds requests for more credits in the messages it sends. The CHT piggybacks the acknowledgments it sends to the clients with the response to the credit request. This is a slight optimization over the Request *Request-Release* as in that the credit requests are embedded into the message transmission/response logic.

None of these techniques are sufficient to both limit the buffer space and deliver performance. Also, none of these schemes carry and utilize the information from the programming model in deciding how to manage buffers or prepare for upcoming data transfers. Our solution to this is described by the flow control and the flow intimation techniques as detailed below.

4.4.3 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. In ARMCI, the `ARMCI_Fence` and `ARMCI_AllFence` [4] operation are mechanisms provided to the user to complete one-sided transactions and bring the memory to a *known* state. Introduction of CHT complicates the implementation of the fence operations. Typically, when an RMA mechanism provided by the network is utilized to transmit data directly (without ARMCI involvement at the remote end in moving data to its destination memory location), any *completion* mechanism, as provided by the network, may be utilized to implement the fence operations. For example, on the Cray XT5,

---

[4] Fence is mechanism to ensure remote completion of all initiated one-sided operations.

the Portals Messaging API provides a mechanism to chose the type of events that are generated both at the source and the destination when a message is transmitted—these events can be utilized to check for remote completion of a data transfer.

The CHT in ARMCI completes several of the ARMCI operations on the remote end. Hence merely utilizing a network provided mechanism for remote completion is not sufficient. A combination of CHT acknowledgments and credit is used to implement the fence operations. When using the client-CHT buffers for transmitting non-contiguous data and accumulate, the achievable performance is dependent on the number of buffers and their size.

4.4.4 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 [21]).

In scenarios such as these, the 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 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 does not add to the communication delay.

Figure 7 is very similar to Fig. 6, 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 Fig. 7, 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.
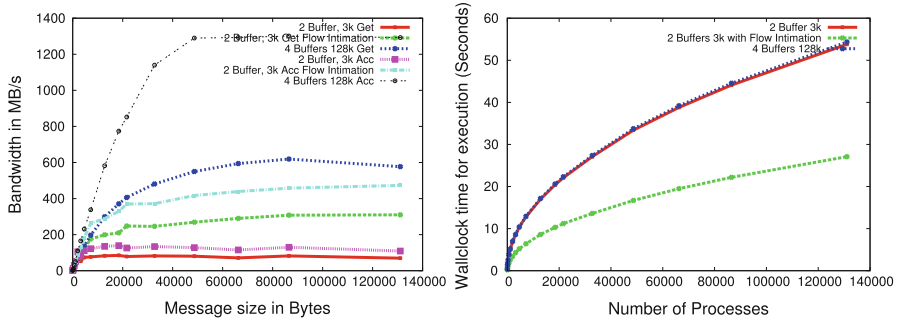
Fig. 7 Flow Intimation: non-contiguous Get and Accumulate (*left*) and NWChem DFT Benchmark (*right*)

In case of the microbenchmark the bandwidth was improved by three fold, the application benchmark performed as well as the large buffer case. Since GA is a library based GAS model and ARMCI is not a compilation target for GA, 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 [8] or Rice CAF [9]) 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" [11] 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 (siosi7 represents a specific molecule). The distributed-data algorithm adopted in this computational kernel makes extensive use of two GA calls: `ga_get` and `ga_acc`. The `ga_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 Eq. (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.
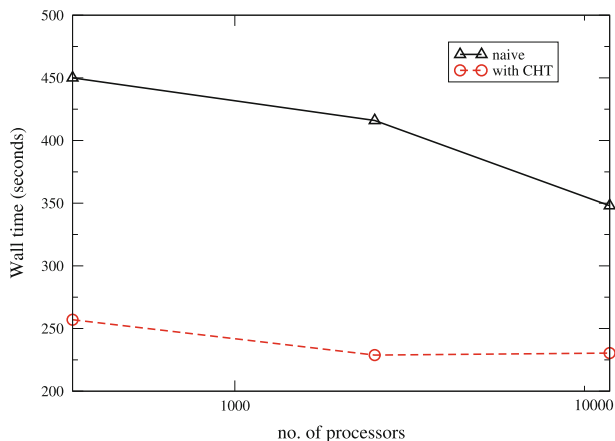
Fig. 8 Walltime to solution for the DFT siosi8 benchmark (7,108 basis functions). The timing includes the complete calculation to the convergence of LDA wavefunction

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

The charge density function $\rho(r_q)$ can be used to compute the Exchange-Correlation potential $V_{XC}[\rho(r_q)]$. In the next step, the Exchange-Correlation potential $V_{XC}[\rho(r_q)]$, combined with Gaussian functions $\chi_\lambda(r_q)$ 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 Eq. (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, on the order of 4 KB, 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 Sect. 4, 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 Fig. 8 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 2 K are shown. Beyond that, the wall
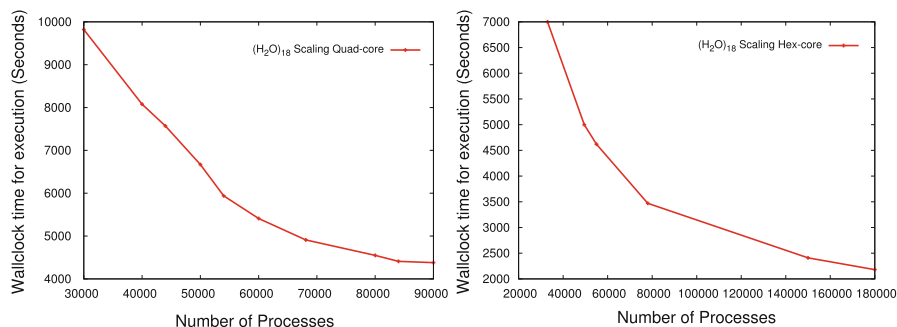
Fig. 9 Walltime to solution for $(H_2O)_{18}$ CCSD(T) benchmark on XT5

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 petaclass 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 [16], 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 [29] 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[12]. What distinguishes the benchmark numbers reported here is the unprecedented scale of the calculations and floating-point performance achieved. 5.1 version of NWChem [6] was utilized for Benchmarks.

We used the $(H_2O)_{18}$ water cluster with a modified cc-pvtz [10] basis set for a total of 918 basis functions. This benchmark was run on the Jaguar XT5 at ORNL. Figure 9 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 Fig. 9 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 $(H_2O)_{18}$ and 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 Prior Related Work

To our knowledge, when this work was performed, NWChem was the first GAS model based application to have scaled to petascale. In fact, NWChem was on of the few applications (even among those that were entirely MPI based) to even scale to the petascale level.

Section 2 discusses that NWChem uses GA one-sided operations and shows a figure describing how a GA one-sided call translates to multiple ARMCI communication calls. Performance and scalability of NWChem, via its usage of GA, is entirely dependent on how ARMCI implements communication protocols. There are several major contributors to the field of scalable GAS runtime libraries. The notable of these are the ARMCI runtime group at Pacific Northwest National Laboratory, Argonne National Laboratory and Oak Ridge National Laboratory and the GASNet developers at Berkeley National Laboratory. Although several research groups have explored various implementations of MPI-2 one-sided API, the applications of MPI-2 one-sided have not yet been demonstrated at the scale of 100s of 1,000s of processes.

One of the first steps in enabling a scalable GAS solution was to utilize user-space communication in a GAS Runtime. The most significant subsequent step has to do with working around several system and network-specific constraints in the design of efficient communication protocols on several platforms. To deliver good performance despite the constraints. ARMCI and GASNet both implement and switch between many communication protocols based on a few system/network-specific parameters (e.g. buffer size). One of the biggest platform-specific constraints is the requirement by network protocols such as GM and Infiniband to have the memory they send-from/receive-to to be registered. Unfortunately, for many systems on which these network cards are used, the cost of registration/deregistration is very high and the amount of memory that can be registered/locked is limited. This constraint has a profound impact on the implementation strategies of user-level RMA on this network. ARMCI uses three techniques to address the requirement for registered memory: (1) copying data via preallocated registered memory buffers (this is the host-based/buffered technique); (2) providing on-demand dynamic memory registration and deregistration (or lazy deregistration) as a part of the data transfer; and (3) providing the user with a memory-allocation interface that allocates registered memory underneath. Early work in this area includes the port of ARMCI on GM [20] and the work by Bell et al. [3] discussing hybrid memory registration/deregistration strategy for GASNet. This hybrid memory registration strategy has been utilized even in ARMCI.

Even when the buffers are registered, noncontiguous operations still pose a challenge. The noncontiguous chunks can utilize either the underlying network interface

that supports the transfer of non-contiguous data or the data is packed into preallocated buffers and sent to a buffer on the remote side, where the unpacking is done and data is copied into the destination by the data server thread. Sect. 4 mentions the pipelined protocol, this is an additional optimization to the above discussed copy-pack-transmit-unpack-copy protocol. It pipelines the copy, pack, transmit, unpack and copy steps involved so that some of the cost of the copying and packing may be hidden. Tipparaju et al. discuss this pipelined protocol in [21]. The memory registration schemes and pipelining played an essential role in scaling the applications that used these GAS runtimes to 100s of cores.

Subsequently, ARMCI team worked with Quadrics supercomputer company [30] to design a scalable available buffer fetching protocol and a scalable non-contiguous data transfer protocol [24]. This lead to the first ever 1,000+ processor run of NWChem. The Berkeley UPC team, utilizing GASNet demonstrated that one-sided communication helps bandwidth limited problems for Infiniband and Quadrics clusters.

Arrival of Blue Gene /L supercomputers allowed applications to scale to 100s of Teraflops. Messaging layer available on Blue Gene machines, starting from the Blue Gene Messaging Layer (BGML), followed by Deep Computing Messaging Framework [19], were designed to direct port GAS/PGAS runtimes to run in a scalable way. Both ARMCI and GASNet were ported to run on Blue Gene machines [7,18,27].

The initial incarnations of the Cray XT series (the Cray XT3 system) ran Catamount [15], a specialized high-performance kernel, that did not support pthreads. Several GAS runtimes were ported to still run, in an inefficient way on the XT3 [34,14]. The ARMCI XT3 port lead NWChem to scale to 10k cores.

The work described in this paper describes the techniques utilized to scale the ARMCI GAS runtime to 100k+ cores.

7 Conclusions

We have examined a variety of issues in designing a scalable ARMCI communication layer for the 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. 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 CAF.

References

1. Apra, E., Harrison, R.J., de Jong, W., Rendell, A., Tipparaju, V., Xantheas, S., Olsen, R.: Liquid water: obtaining the right answer for the right reasons. In: Supercomputing, 2009. SC'09. Proceedings of the ACM/IEEE SC 2009 Conference (2009)
2. Barrett, B.W., Shipman, G.M., Lumsdaine, A.: Analysis of implementation options for MPI-2 one-sided. In: Proceedings, Euro PVM/MPI, Paris, France (2007)
3. Bell, C., Bonachea, D.: A new DMA registration strategy for pinning-based high performance networks. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IPDPS'03, IEEE Computer Society, Washington, DC, USA, p. 198.1. http://dl.acm.org/citation.cfm?id=838237.838681 (2003)
4. Bonachea, D.: GASNet specification, v1.1. Tech. rep., Berkeley, CA, USA (2002)
5. Brightwell, R., Riesen, R., Lawry, B., Maccabe, A.: Portals 3.0: protocol building blocks for low overhead communication. In: Parallel and Distributed Processing Symposium, Proceedings International, IPDPS 2002, Abstracts and CD-ROM, pp. 164–173 (2002)
6. Bylaska, E., et al.: NWChem, a computational chemistry package for parallel computers, version 5.1 (2007)
7. Blocksome, M., Archer, C., Inglett, T., McCarthy, P., Mundy, M., Ratterman, J., Sidelnik, A., Smith, B., Almási, G., Castaños, J., Lieber, D., Moreira, J., Krishnamoorthy, S., Tipparaju, V., Nieplocha, J.: Design and implementation of a one-sided communication interface for the IBM eServer Blue Gene supercomputer. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC'06. ACM, New York, NY, USA, http://doi.acm.org/10.1145/1188455.1188580 (2006)
8. Chapel language specifications, v0.780. http://chapel.cs.washington.edu/spec-0.780.pdf (2006)
9. Dotsenko, Y., Coarfa, C., Mellor-Crummey, J.: A multi-platform co-array Fortran compiler. In: Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on, pp. 29–40. doi:10.1109/PACT.2004.1342539 (2004)
10. Dunning, T.H.: Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen. J. Chem. Phys. **90**(2), 1007–1023 (1989). doi:10.1063/1.456153, http://link.aip.org/link/?JCP/90/1007/1
11. Dunning, T.H., Peterson, K.A., Woon, D.E., Wilson, A.K.: Quantifying quantum chemistry. In: American Conference on Theoretical Chemistry, unpublished (1999)
12. de Jong, W.A., Krishnamoorthy, S.: Private communication (2008)
13. GA: Global Arrays Toolkit. http://www.emsl.pnl.gov/docs/global (2010)

14. GASNet Portals Conduit Docs. http://gasnet.cs.berkeley.edu/dist/portals-conduit/README (2008)
15. Kelly, S.M., Brightwell, R.: Software architecture of the light weight kernel, Catamount. In: In Cray user group, pp. 16–19 (2005)
16. Kobayashi, R., Rendell, A.P.: A direct coupled cluster algorithm for massively parallel computers. Chem. Phys. Lett. **265**(1–2), 1–11 (1997). doi:10.1016/S0009-2614(96)01387-5
17. Koop, M., Sridhar, J., Panda, D.: Scalable MPI design over InfiniBand using eXtended reliable connection. In: Cluster Computing, 2008 IEEE International Conference on, pp. 203–212. doi:10.1109/CLUSTR.2008.4663773 (2008)
18. Krishnan, M., Nieplocha, J., Blocksome, M., Smith, B.: Evaluation of remote memory access communication on the IBM Blue Gene/P Supercomputer. In: Parallel Processing—Workshops, 2008. ICPP-W '08. International Conference on, pp. 109–115. doi:10.1109/ICPP-W.2008.34 (2008)
19. Kumar, S., Dozsa, G., Almasi, G., Heidelberger, P., Chen, D., Giampapa, M.E., Michael, B., Faraj, A., Parker, J., Ratterman, J., Smith, B., Archer, C.J.: The deep computing messaging framework: generalized scalable message passing on the Blue Gene/P Supercomputer. In: Proceedings of the 22nd Annual International Conference on Supercomputing, ICS'08. ACM, New York, NY, USA, pp. 94–103. http://doi.acm.org/10.1145/1375527.1375544 (2008)
20. Nieplocha, J., Ju, J., Apra, E.: One-sided Communication on the Myrinet-based SMP Clusters using the GM Message-Passing Library. In: In Proceedings of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS 01 (2001)
21. Nieplocha, J., Tipparaju, V., Saify, A., Panda, D.: 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, pp. 164–173 (2002a)
22. Nieplocha, J., Tipparaju, V., Saify, A., Panda, D.: Protocols and strategies for optimizing performance of remote memory operations on clusters. In: In: Proceedings Workshop Communication Architecture for Clusters (CAC02) of IPDPS'02, Ft (2002b)
23. Nieplocha, J., Tipparaju, V., Apra, E.: 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, IEEE Computer Society, Washington, DC, USA, p. 215.2. doi:10.1109/IPDPS.2005.96 (2005a)
24. Nieplocha, J., Tipparaju, V., Krishnan, M.: Optimizing strided remote memory access operations on the quadrics QsNetII network interconnect. In: HPCASIA'05: Proceedings of the 8th International Conference on High-Performance Computing in Asia-Pacific Region, IEEE Computer Society, Washington, DC, USA, p. 28. doi:10.1109/HPCASIA.2005.62 (2005b)
25. Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., Apra, E.: Advances, applications and performance of the global arrays shared memory programming toolkit. Int. J. High Perform. Comput. Appl. **20**(2), 203–231 (2006a) http://hpc.sagepub.com/cgi/content/abstract/20/2/203
26. Nieplocha, J., Tipparaju, V., Krishnan, M., Panda, D.K.: High performance remote memory access communication: the ARMCI approach. Int. J. High Perform. Comput. Appl. **20**(2), 233–253 (2006b) http://hpc.sagepub.com/cgi/content/abstract/20/2/233
27. Nishtala, R., Hargrove, P., Bonachea, D., Yelick, K.: Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. In: Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pp. 1–12. doi:10.1109/IPDPS.2009.5161076 (2009)
28. Parzyszek, K.: Generalized portable shmem library for high performance computing. PhD thesis, Ames, IA, USA, co-Major Professor-Kendall, Ricky A. and Co-Major Professor-Lutz, Robyn R. (2003)
29. Pollack, L., Windus, T.L., de Jong, W.A., Dixon, D.A.: Thermodynamic properties of the C5, C6, and C8 n-alkanes from ab initio electronic structure theory. J. Phys. Chem. A **109**(31), 6934–6938 (2005). doi:10.1021/jp044564r, http://pubs.acs.org/doi/abs/10.1021/jp044564r, http://pubs.acs.org/doi/pdf/10.1021/jp044564r
30. Quadrics Supercomputer Company. http://en.wikipedia.org/wiki/Quadrics (1996)
31. Report on experimental language X10. http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf (2008)
32. Shet, A., Tipparaju, V., Harrison, R.: Asynchronous programming in UPC: a case study and potential for improvement. In: Workshop on Asynchrony in the PGAS Programming Model Collocated with ICS 2009 (2009)
33. Tipparaju, V., Santhanaraman, G., Nieplocha, J., Panda, D.K.: Host-assisted zero-copy remote memory access communication on InfiniBand. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, p. 31. doi:10.1109/IPDPS.2004.1302943 (2004)

34. Tipparaju, V., Kot, A., Nieplocha, J., Bruggencate, M., Chrisochoides, N.: Evaluation of remote memory access communication on the Cray XT3. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pp. 1–7. doi:10.1109/IPDPS.2007.370478 (2007)
35. Top500 list. http://www.top500.org (2010)
36. UPC specifications, v1.2. http://www.gwu.edu/~upc/publications/LBNL-59208.pdf (2005)