Evaluating NIC Hardware Requirements to Achieve High Message Rate PGAS Support on Multi-Core Processors

Keith D. Underwood [*]
Sandia National Laboratories [†]
P.O. Box 5800
MS-1319
Albuquerque, NM 87185-1319
kdunder@sandia.gov

Michael J. Levenhagen
Sandia National Laboratories
P.O. Box 5800
MS-1319
Albuquerque, NM 87185-1319
mjleven@sandia.gov

Ron Brightwell
Sandia National Laboratories
P.O. Box 5800
MS-1319
Albuquerque, NM 87185-1319
rbbrigh@sandia.gov

ABSTRACT

Partitioned global address space (PGAS) programming models have been identified as one of the few viable approaches for dealing with emerging many-core systems. These models tend to generate many small messages, which requires specific support from the network interface hardware to enable efficient execution. In the past, Cray included E-registers on the Cray T3E to support the SHMEM API; however, with the advent of multi-core processors, the balance of computation to communication capabilities has shifted toward computation. This paper explores the message rates that are achievable with multi-core processors and simplified PGAS support on a more conventional network interface. For message rate tests, we find that simple network interface hardware is more than sufficient. We also find that even typical data distributions, such as cyclic or block-cyclic, do not need specialized hardware support. Finally, we assess the impact of such support on the well known RandomAccess benchmark.

1. INTRODUCTION

In striving for higher productivity, numerous partitioned global address space (PGAS) languages have been proposed, including Co-Array Fortran (CAF) [24], Unified Parallel C (UPC) [9], and the languages recently proposed for the DARPA High Productivity Computing Systems (HPCS) program[10, 7, 31]. Each purports to facilitate the transition into the trans-petascale regime by leveraging PGAS capabilities. Indeed, some argue that the only viable approach to programming the next generation of machines is to leverage a global address space model. However, to actually deliver improvements in productivity, these languages must deliver high performance at reasonable levels of effort.

Unfortunately, the PGAS model tends to express communications in extremely fine-grained accesses. Every access of remote memory can easily (and frequently does) become an 8-byte network transaction. This type of network usage differs dramatically from what is typically seen in MPI applications. Current high-end network interfaces tend to incur 1 to 2 microseconds of latency[21, 27, 28] for an MPI message. Partially because of the two-sided semantics of MPI, they are able to sustain only one to three million MPI messages per second (per core) in the best case scenarios[27, 28, 21][1]. This modest level of performance — particularly the high latency and high overhead — would force programmers using a PGAS model to program as if they were using MPI. While it is likely that achieving the highest performance will always require some attention to data layout and remote access patterns, if applications are forced to use only large block transfers, then PGAS languages will fail to deliver an improvement in productivity.

Of course, many systems have attempted to bridge this gap in performance for the PGAS model. Notably, Cray, Inc. has typically designed products that directly support PGAS semantics. For example, Cray's vector processors (e.g. the Cray X1[11]) tend to provide a large native address space and a sufficient number of outstanding loads and stores to make remote accesses efficient. In contrast, the highly successful Cray T3E[1] leveraged commodity microprocessors augmented with a novel mechanism called E-registers[30] to provide global address space capabilities. Indeed, the T3E is the archetype of modern expectations for PGAS support on a commodity microprocessor platform. One of the key features of the T3E was the *centrifuge* operation combined with the E-register accesses[30]. The centrifuge swizzles bits from an array that is mapped into linear virtual addresses on each core to generate the appropriate PE and offset to access the physical array that is distributed across the machine. The T3E provided robust support for the common data distributions leveraged by the PGAS model; however, much has changed since the T3E was designed more than a decade ago.

Perhaps the most dramatic change over this time has been the continuing shift in the balance between computation and communication. While one of the initial concerns with the T3E was the ability to issue enough transactions to keep the network busy [30], the number of operations a processor can perform in the time required to transfer a network word has constantly increased. Similarly, although network bandwidth has grown more slowly than processor performance, the rate of increase in network bandwidth has far outstripped the rate of improvement in network latency. Finally, the system interface environment has changed dramatically

[†]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

[*]Keith is now with Intel Corporation.

[1]The older InfiniPath adapter from Qlogic and the recently announced ConnectX adapter from Mellanox claim to sustain 11 million and 25 million messages per second, respectively, when using 8 cores

as well. Modern systems are shifting to point-to-point processor interconnects (e.g. HyperTransport), and processors have added write-combining buffers for I/O space that allow the intelligent aggregation of transfers that cross into the network interface. This significant change in the landscape since E-registers were designed [2] has provided motivation for exploring what could be achieved with minimal hardware support. Thus, this paper examines the impact of using minimal hardware support in the context of modern systems. We find that while a single processor core has insufficient issue bandwidth to drive a network at full rate, two or four cores is all that is required to saturate the network (depending on the workload). Indeed, cyclic and block cyclic data distributions can achieve over 80% of the network's theoretical limit (when accounting for overheads) with only two cores driving the network. Even with the additional work per access required by the HPCC RandomAccess benchmark[19], two cores are still able to achieve 63% of the network limit while four cores can saturate the network.

The next section describes related work on hardware support for PGAS as well as relevant network programming interfaces. Section 3 discusses a minimal set of hardware, and our methodology is detailed in Section 4. Microbenchmark results and analysis are provided in Sections 5 and 6, respectively. Section 7 presents results for the HPCC RandomAccess benchmark. Conclusions are presented in Section 8 followed by future work in Section 9.

2. RELATED WORK

Efficient network hardware support for small messages has been an active area of research for a number of years. While our research is focused on providing hardware support specifically for the PGAS model on multi-core distributed memory systems — more specifically efficient one-sided network transfers — there is a large body of previous work on techniques to optimize small message data movement for parallel and distributed systems.

Many different approaches have been taken to address the performance and overhead issues associated with doing small network transfers in distributed memory systems. A popular approach has been to extend processor- and bus-level interconnect technologies to support communication over greater distances. This method was used for technologies like the Scalable Coherent Interface (SCI) [15], which was an extension of the FutureBus technology developed in the late 1980's. This technique has had several different incarnations over the last twenty-plus years, and has most recently appeared in Advanced Switching for PCI Express [20]. A distinguishing characteristic of these approaches is support for memory coherence across the entire system. Most of these technologies were developed to provide coherency for a global address space rather than the more limited coherency model that is required to support PGAS implementations. The aforementioned Cray T3E E-registers and the network interface for the Intel TeraFLOPS machine [8] are examples of hardware specifically designed to support PGAS-style coherency on a large distributed memory parallel machine.

Traditional attached high-performance network technologies have also been enhanced to better support small one-sided network transfers. Most current high-performance networks, such as Myrinet [3], Quadrics [25], and InfiniBand [16] support programmed I/O transfers where the processor writes directly to the network interface to initiate the transfer of small messages. This method avoids the significant overhead of programming a DMA engine to do the transfer. The Quadrics QSNet-II hardware also has a Short Transaction Engine (STEN) [29], which takes a sequence of writes into network interface memory and formats them directly into network transactions, bypassing the processor on the network interface. This approach is greatly enhanced by processor buses that employ write-combining to allow for the transfer to the network interface to occur in a single bus transaction. Recently, the InfiniPath [13] network has demonstrated significant increases in latency performance and message rate for small messages. It is able to achieve these rates by using programmed I/O to stream messages into network interface memory where custom logic formats messages and then streams them out to the network. This approach of bypassing DMA engines on the send-side and avoiding the overheads of an embedded processor on the network interface has demonstrated several benefits for small message performance.

While our work here does not specifically target application programming interfaces for PGAS-style network transfers, it is important to understand how the capabilities and semantics of the underlying network are exposed to the application level, either directly through library routines or indirectly through a compiler. The majority of the network programming interfaces for current high-performance attached networks are designed to optimize bulk transfers for two-sided messaging architectures like MPI.

One of the original programming interfaces aimed at efficient handling short messages for distributed multiprocessor systems was Virtual Memory Mapped Communication [2], which mapped an area of remote memory into a process' local address space. This technique allowed for loads and stores to this mapped region to be translated into remote operations by the network interface. The Cray SHMEM [12] API was developed specifically to support application level use of E-registers. This API has also been supported on several shared memory platforms and some commodity networks, most notably Quadrics. The Aggregate Remote Memory Copy Interface (ARMCI) [22] was designed specifically to support Global Arrays [23], a PGAS model library. It has many features in common with Cray SHMEM, including support for non-contiguous data transfers and remote atomic memory operations. The GASNet [4] programming interface was designed to be a compiler target for the Berkeley UPC compiler and runtime system. It was designed to support efficient small message transfers and is heavily influenced by previous work on Active Messages [34]. We have chosen to implement a subset of the Cray SHMEM API for our experiments in this study.

3. MINIMAL HARDWARE

The proposed hardware is designed to interact with systems in a modern context. Specifically, it assumes that the network interface connects to the processor through a high-performance, packetized I/O interface like HyperTransport (HT)[3]. Furthermore, it is assumed that the processor will have a series of write-combining buffers to aggregate accesses over that interface. A basic block diagram is shown in Figure 1.

3.1 Transmit

Figure 1 illustrates a 3D router, an interface to the host processor, and four blocks to implement support for PGAS. For the transmit side, commands are pushed down into a 4 KB command queue, with linear addressing to leverage the write-combining resources on the host processor. Each process (core, in this case) needs a private command queue mapped into its address space to allow user-level access. *Free space* updates are pushed back into the user

[2]Not to mention the fact that E-registers are patented

[3]For this particular design study, PCI-Express would be equivalent; however, it is likely that numerous other aspects of PGAS support, such as remote atomic operations, would be better served by tighter integration with the processor

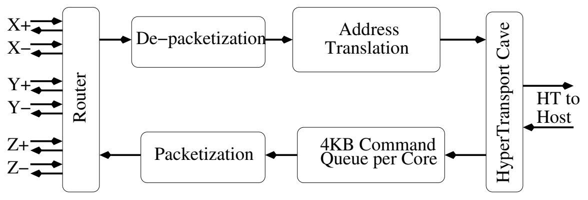


Figure 1: The NIC block diagram

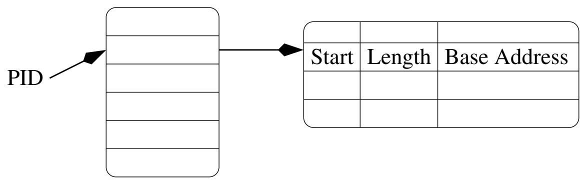


Figure 2: Minimal translation hardware

address space for every 512 bytes consumed. This is required to minimize the update traffic across the HT interface, while preventing the bandwidth throttling that can be induced by a buffer that is too small or updates that are too infrequent.

The contents of a minimal command are shown in Table 1. The operation field is set to be large enough to support a range of operations. The element size maps to the SHMEM operation sizes (1 to 128 bytes), but maps well to the general needs of PGAS languages. The node identifier (NID) is sized to support a system up to 64K nodes and the process identifier (PID) is mapped to the size of a Linux PID. The stride is smaller than the stride required by SHMEM, but larger strides can be readily supported by multiple native commands. On the transmit side, the translation from virtual (PE) to physical (NID/PID) is currently algorithmic, on the host[4].

Commands written to the NIC are translated to packets for the network. The NIC adds multiple trusted fields[5] to the message for inspection at the remote side. An 8-byte header is added in addition to 16 bytes containing the source NID, source PID, user identification (UID), and job identification (JID). While not strictly necessary, these fields are typically added in real networks to provide authentication at the remote node for a wide range of protocols. The data transferred over the network is shown in Table 2 and includes a total of 36 bytes of overhead on the data.

3.2 Receive

At the receive side, authentication would occur based on the trusted fields in the packet. The commands for the target would then be extracted from the packet, with the destination virtual address and PID being used for address translation. In a system like the Cray XT3, the Catamount O/S[17] uses a linear virtual to physical mapping that would enable a simple translation to a physical address. Because the simulator does not model an operating system, this approach was used for this work. Thus, the minimal set of hardware required is a table indexed by PID that selects an associative structure that is used to map the virtual address into an application segment (e.g. stack, heap, etc.), which provides a base address. This is illustrated in Figure 2, where the second structure indexed has a start and length that are compared to determine what region the incoming request's virtual address falls in. This region then provides a base address that is used for translation.

[4]A real system would include a translation mechanism (either on the host or on the NIC), but that is independent of the other PGAS support mechanisms.

[5]The trusted fields would be initialized on the NIC through a system call at application initialization.

A Linux based O/S would require significantly more complexity in the address mapping scheme; however, this is independent of the support for PGAS and is an issue for all NICs. In either case, a translated address is used to generate an HT request that places the data directly into application memory.

3.3 Completion Notification

Another required aspect of support for PGAS languages is completion notification. The transmitter must be capable of determining when all outstanding messages have completed to support operations such as `shmem_quiet()`. The hardware at the receiver generates an acknowledgment so that the transmitter can determine when all outstanding requests have completed. The transmitter's NIC keeps a simple count that is incremented when a message is sent and decremented when an acknowledgment is received so that a `shmem_quiet()` is simply a blocking call to the NIC.

3.4 Interaction with MPI

Nothing about the proposed hardware is mutually exclusive to an efficient MPI implementation. Indeed, the command structure on the NIC can be shared and demultiplexed at the NIC. Similarly demultiplexing would be needed at the receiver, but such demultiplexing is a single cycle, easily pipelined operation.

3.5 Trade-offs

This proposal pushes support from hardware to the processor and makes three trade-offs relative to the E-register approach[30]. Our hypothesis is that these trade-offs simplify the network interface implementation without significantly impacting performance. The first trade-off is increased processing time per network transaction. In this case, we can consider dual- and quad-core Opterons and recognize that the typical *centrifuge* operation [30] ranges from four to ten integer instructions for a multi-gigahertz processor with multiple cores capable of issuing three or more instructions per cycle.

The second trade-off is the need to write additional data across the I/O bus to provide the information to the NIC that is implied in the address accessed with a command in the E-register case. This, however, is a false comparison in a processor model that is dependent on write-combining for sustaining high performance. A single core in our proposed approach writes 24 bytes (command and data) per 8-byte message with multiple message commands being aggregated into a single 64-byte HyperTransport (HT) packet containing an overhead of 8 bytes. The E-register approach, because it implies some information in the *address* accessed on the NIC, would frequently make accesses that interfered with the write-combining hardware and would cause each HT packet to contain only one 16-byte transfer (target address and data) for an 8-byte message. Thus, where our proposed approach would average 27 bytes (including HT overhead) per 8-byte message, the E-register approach would average 24 bytes (including HT overhead) — a relatively small 12.5% savings.

The third trade-off is to place returns from *get* operations in host memory instead of E-register structures on the NIC. Given a system like the Cray XT3, which can incur a round trip time of more than 5 microseconds across an unloaded machine just due to routing delays, over 2048 E-registers would be needed to keep the network pipe full for get-based traffic. This also implies that the application would have 2048 concurrent get operations available. This points to a future where get-based traffic is less and less usable for fine-grained accesses, resulting in the need to move the responses to host memory. It does, however, add an additional 4 bytes to initiate get commands between the processor and the NIC.

Table 1: Commands for PGAS support on network interface

Field	Bits	Comment
Operation	5	Puts, Gets, Atomics
Element Size	4	Size of data element in bytes
Num. Elements	16	Number of elements to transfer
Dest. NID	16	Up to 64K nodes in system
Dest. PID	16	Support Linux process ID
Dest. Address	38	Up to 256 GB per node
Dest. Stride	16	Larger strides use multiple commands
Total (for Put Commands)	111	16 bytes
Local Address	38	For Gets only
Total (for Get Commands)	149	20 bytes

Table 2: Message contents

Field	Bytes	Comment
Header	8	Matches Cray XT3 network
hline Command	16	All of the command is needed at the target
Source NID	2	Trusted source fields used for authentication
Source PID	2	
Source UID	4	
Source Job ID	4	
Data	Variable	

Table 3: Processor Parameters

Parameter	CPU
Clock Frequency	2 GHz
Fetch Queue	4
Issue Width	8
Commit Width	4
RUU Size	64
Integer Units	4
Memory Ports	3
L1 (Size/Assoc.)	64KB/2
L2	1MB/16
ISA	PowerPC
Main Memory Bandwidth	6.4 GB/s peak
Main Memory latency	140-160 cyc.
System I/O	HyperTransport
I/O Bandwidth	2.3 GB/s/dir sustainable
I/O Latency	250 ns

Table 4: Network Parameters

Topology	3D Torus
Clock Frequency	500 MHz
Link Bandwidth (Peak)	4 GB/s/dir
Router Latency	50 ns
Link Overhead	15%
Router Arbitration	Round-Robin
Router VCs	4

4. METHODOLOGY

This work leveraged the Structural Simulation Toolkit (SST) developed by Sandia National Laboratories and the University of Notre Dame[33]. SST provides a hybrid discrete event and cycle-driven simulation infrastructure that enables the coarse-grained integration of components modeled at a very detailed level. SST integrates the SimpleScalar[6] processor simulation to provide detailed processor models. SST enhances the processor model with a more robust memory model, including the ability to memory map I/O devices. In addition, SST models a HyperTransport I/O interface and has been used to model the Cray XT3 supercomputer — including a robust model of the network interface[33].

We replaced the XT3 network interface with a model to study hardware support for SHMEM. Tables 3 and 4 present the salient properties of the processor and network models. The peak sustainable rate of the HyperTransport (HT) interface on the XT3 is approximately 2.3 GB/s per direction, before accounting for the "header flit" of each HyperTransport packet. The one-way latency across the HyperTransport interface is approximately 250 ns, which is remarkably high, but does match the actual hardware performance. The processor runs at 2 GHz and the newtork interface at 500 MHz. Both match the points validated in previous work [33].

The router-to-router links have a peak of 4.8 GB/s per direction; however, to simplify the implementation, the router model assumes that the links are synchronous to the router core (8 bytes wide, 500 MHz) and accounts for the difference in peak bandwidth by adjusting the overhead. Whereas the XT3 links have a 24% protocol overhead, this model only adds 15% overhead to the individual links. Finally, the virtual channel architecture (two virtual channel classes each having two virtual channels) is modeled along with the round-robin arbitration variant.

Each of the benchmarks was implemented using a limited version of the Cray SHMEM API. Only the put and barrier functions were implemented for this study[6]. As an example, the function:

```
shmem_int_put(int *target, int *source, size_t len, int pe)
```

copies `len` integers from the address `source` on the local PE[7] to the address `target` on the remote PE designated `pe`. Each of the put functions pushes the required 16-byte command to the network interface followed by the data from the source to the network interface. On the target PE, the network interface initiates transac-

[6]Since the Cray SHMEM get functions are all blocking, they are generally less useful in modern systems.

[7]The SHMEM PE (processing element) is analogous to the MPI rank.

```
/* Perform loop iterations */
for (i = 0; i < loop; i++) {
  int dest =0;
  /* Send window_size messages to each node */
  for (j = 0; j < window_size; j++) {
    dest = num_cores() + coreNum;
    /* Loop over remote nodes */
    for ( k = 0; k < numNids; k++ ) {
      /* target nid is loop variable */
      int nid = k;
      /* PE is matching core on remote nid */
      int pe = nid * num_cores() + coreNum;
      far_rbuf = calcRBuf(pe);
      shmem_int_put( far_rbuf, sbuf, size>>2, pe );
    }
  }
  /* Wait for completion */
  wait_for_response();
}
```

(a)

```
/* Perform loop iterations */
for (i = 0; i < loop; i++) {
  int dest =0;
  /* Send window_size messages to each node */
  for (j = 0; j < window_size; j++) {
    sbuf[(size>>2)-1] = j+1;
    dest = num_cores() + coreNum;
    /* loop over all nodes */
    for ( k = 0; k < numNids; k++ ) {
      /* nid from an indirection vector */
      int nid = destList[j*numNids+k];
      /* Target PE is matching core on remote node */
      int pe = nid * num_cores() + coreNum;
      /* Calculate target remote address */
      far_rbuf = calcRBuf(pe);
      shmem_int_put( far_rbuf, sbuf, size>>2, pe );
    }
  }
  /* Wait for completion */
  wait_for_response();
}
```

(b)

Figure 3: Benchmark code for One-to-N (a) cyclic and (b) indirect distributions

tions across the HT interface to place the data at the target location. The `shmem_barrier_all()` routine was implemented using the simulator's built-in barrier routine with an added overhead of $2\mu s$, which is a conservative estimate of the performance of a real barrier on a system designed to support a PGAS model.

5. MICROBENCHMARKS

Many of the issues regarding hardware support for the PGAS model can be readily explored using microbenchmarks. Unlike MPI, where it is extremely difficult to make microbenchmarks representative of application behavior[32], PGAS-style messaging is relatively simple. There are no message matching semantics (with the associated lists to traverse), no message ordering semantics, and, thus, relatively few ways to "cheat" with a benchmark. This section presents several variations on benchmarks to assess the performance of our simplified PGAS hardware support.

5.1 Description

The first three microbenchmarks used in this study mirror their counterparts for MPI. A ping-pong latency benchmark *puts* a single item of a given size to a remote node. Upon arrival, the remote node initiates a put operation to the sending node. The time from the initiation of the first message to the receipt of the response is measured, and the one-way latency is assumed to be half of this value. Similarly, the streaming message benchmark closely matches the Ohio State University (OSU) streaming message benchmark for MPI [18] and the bidirectional bandwidth benchmark is simply an analogue of the ping-pong latency benchmark using simultaneous bidirectional transfers.

Three variants of the OSU streaming bandwidth benchmark were also created to mimic accesses to various data distributions within

```
/* Total data to send */
int loopSize = (window_size * numNodes * size) - size;
/* Perform loop iterations */
for (i = 0; i < loop; i++) {
  /* Loop over all data - increment by data sent*/
  for (j = 0; j < loopSize; j += size ) {
    /* Extract nid from loop variable */
    int destNid = (j >> nidShift) & nodeMask;                  *
    int destPe = destNid * numCores + coreNum;
        /* target address also based on loop variable */
    int *targAddr = (int*) (targBuf + j);
    shmem_int_put( targAddr, srcBuf, size>>2, destPe );
  }

  wait_for_response();
}
```

Figure 4: Benchmark code for One-to-N block-cyclic distribution

a typical PGAS model. In Figure 3(a), the benchmark mimics a cyclic data distribution. For a given size of transfer, the benchmark performs several loop iterations that consist of sending `window_size` items to each of `numNids` nodes, with the loop over the NIDs (equivalent to socket number) being the inner loop. Note that the destination PE is chosen as the PE corresponding to the remote core on a given socket that matches the sending core on the local socket. Thus, on a dual- or quad-core node, there are two or four PEs sending data.

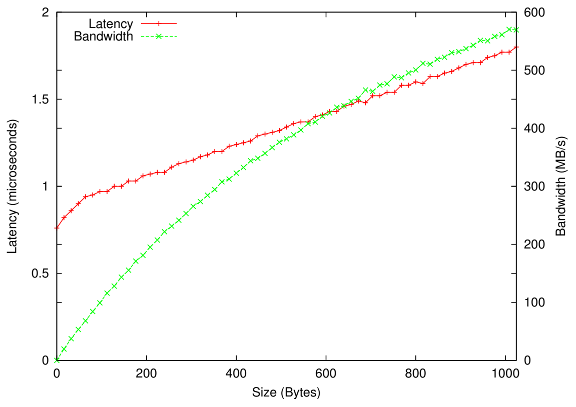
The streaming test using an indirection vector in Figure 3(b) very closely matches Figure 3(a). The only change is that the target PE is selected by using the loop variables to index into an indirection vector. Finally, to mimic a block-cyclic distribution, it is necessary to somewhat change the loop structure, as shown in Figure 4. The benchmark loops over all data to transfer and extracts the target PE and the target address from the loop variable.

5.2 Results

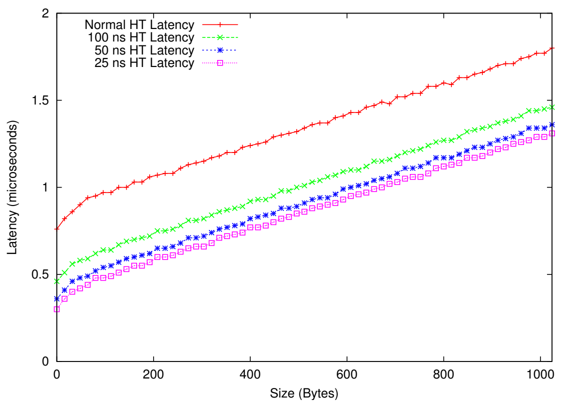
We begin with simple ping-pong latency and bandwidth in Figure 5(a). Relative to MPI on the same simulated platform[33], latency is reduced by a factor of five; however, this is not a fair comparison, since we have switched from using DMA on every message to using programmed I/O. Relative to a comparable interconnect, such as InfiniPath, using programmed I/O and HT [5, 13], latency is still reduced by almost a factor of two, due to the elimination of various MPI overheads.

While the latency was reduced dramatically, it is not as low as might be expected. For these experiments, we used the model of HT built to simulate the Sandia Red Storm (Cray XT3) system[33]. The latency of the HT implementation accounts for nearly 70% (500 ns) of the total latency, with the crossing of two routers contributing another 14% (100 ns). Figure 5(b) considers the impact of reducing the HT latency, and indicates that, in the extreme case where the implementation on the NIC is as good as the coherent implementation found in an Opteron (25 ns for one crossing), the one-way latency can be reduced to 300 ns.

In turn, Figure 6 presents the bidirectional bandwidth achievable with the simplified PGAS hardware. The primary point illustrated here is that a set of processor cores can readily contribute enough data to keep the network pipe full. An interesting note, however, is that a single core can only fill half of the network pipe. This is an artifact of the memory copy rate. The processor modeled does not support SIMD instructions. Like the Opteron processor it approximates, if wide (SIMD) loads are not used for data movement, it cannot achieve more than approximately 2 GB/s of memory copy bandwidth. In discussions with processor vendors, it has become clear that this is a trend of the future: a single core will not be able to sustain the full memory bandwidth available on a socket — even for a memory copy. Transferring data in the PGAS model is the equivalent of a memory copy where the target is simply the local network interface.



(a)



(b)

Figure 5: (a) Ping-pong latency and bandwidth; (b) Ping-pong latency with reduced HT latency

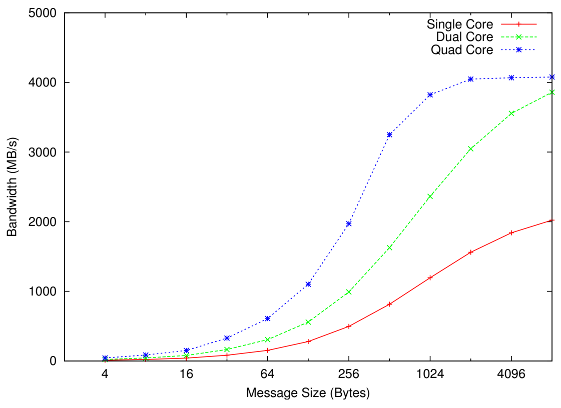


Figure 6: Bidirectional bandwidth

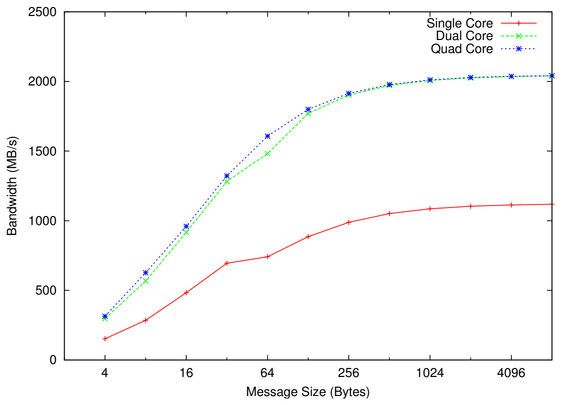


Figure 7: Unidirectional streaming bandwidth

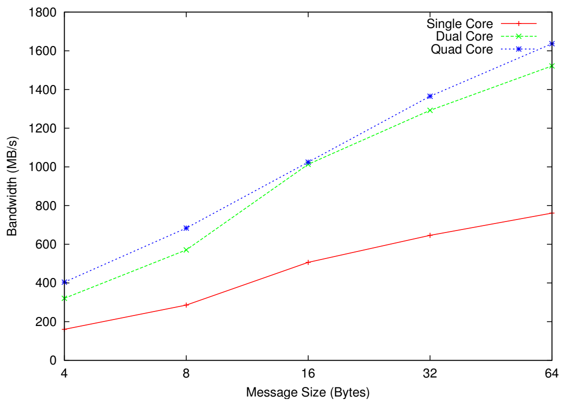


Figure 8: One-to-N unidirectional streaming bandwidth

Figure 7 presents the small message streaming bandwidth between two nodes. Without the overheads imposed by MPI, the oft-touted $N^{\frac{1}{2}}$ number is a mere 16 bytes. For the peak bandwidths, we again see the limitations of a single core without wide (SIMD) loads and stores for data movement; however, when streaming messages between a pair of nodes, the computations are simple enough that two cores are sufficient to compute all of the remote addresses and do the work necessary to fill the network link. Indeed, two cores sustain 90% of the rate achievable by four cores, and modern quad-core systems are still limited to approximately the same I/O interface (HT in this case) rates as are seen on dual-core systems.

While a streaming test provides some insight into the performance of the PGAS hardware support, applications are not typically going to stream data to a single node with much regularity. The next most complicated operation that an application can perform is to write data to an array that is striped across many (in this case all) nodes. Figure 8 presents the performance when messages are striped across all nodes in the system[8]. The data in Figure 8 represents the performance that could be expected when writing linearly to an array of objects of size *M* where the array is distributed cyclically across all of the nodes. Here, the work per transfer is increased so that two cores are only able to generate requests fast enough to sustain 80% of the full rate, while four cores can still easily saturate the network.

A second common data distribution in the PGAS model is a block-cyclic distribution. A block-cyclic distribution places several contiguous items from a single array on a node; thus, the calculation of

[8]32 sockets for the case presented here

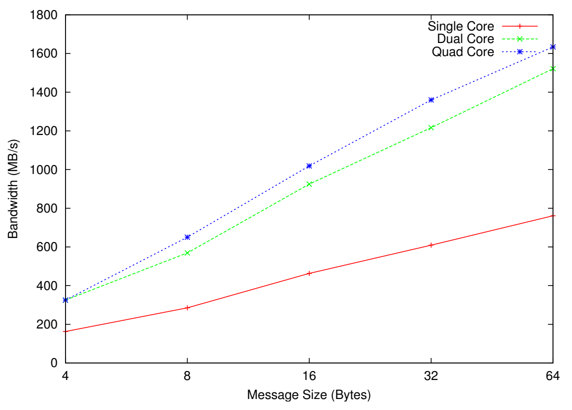


Figure 9: One-to-N block unidirectional streaming bandwidth

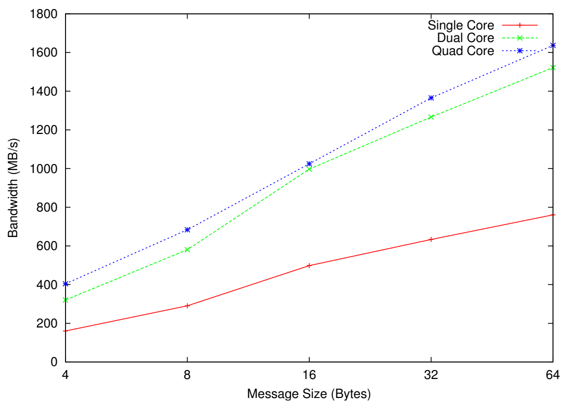


Figure 10: One-to-N random unidirectional streaming bandwidth

the destination PE occurs less often. The block streaming test results in Figure 9 indicate that two cores can sustain 87% of the rate that four cores can sustain, and four cores can saturate the network. The third access pattern considered is vector indirection. Iterative sparse solvers, for example, often access remote data through an indirection vector. This adds a local memory access for every remote access and changes the work balance somewhat; however, accesses to the indirection vector have perfect spatial locality[9]. Thus, using the indirection vector is equivalent in performance to striped accesses (see Figure 10). That is, two cores can keep the interconnect 80% saturated, while four cores can saturate it.

6. ANALYSIS

Given the hardware design in Section 3 and the performance seen in Section 5, it is worthwhile to consider the constraints on peak transfer rates. The processors' injection rate, the I/O bandwidth, and the network link bandwidth all place upper bounds on the level of performance that can be achieved. In the Cray XT3 design, HT provides 3.2 GB/s of peak bandwidth per direction; however, the actual sustained bandwidth is 2.3 GB/s. When the 8-byte overhead per 64-byte data transfer is accounted for, this becomes just over 2 GB/s. Thus, the peak data rate for a message of N bytes using the design from Section 3 is bounded by:

$$HyperTransportBW \leq \frac{N}{N+16} \times \frac{64}{72} \times 2.3GB/s \qquad (1)$$

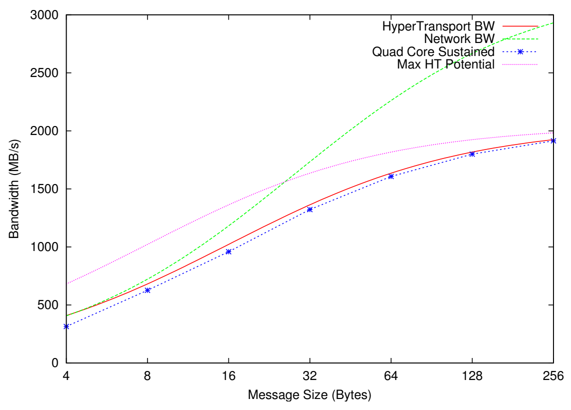


Figure 11: Comparison of actual to theoretical peak

where there is 16 bytes of overhead for a given command and 8 bytes of overhead on 64-byte HyperTransport packets.
The router-to-router links also use 64-byte packets with 8 bytes of overhead on each one. In addition, the message contains 28 bytes of overhead and the links themselves add 5 bytes of overhead to every 16 bytes to provide link-level flow control and retransmission. Fortunately, the network link provides 4.8 GB/s of bandwidth per direction yielding a network bandwidth bounded by:

$$NetBW \leq \frac{N}{N+28} \times \frac{64}{72} \times \frac{16}{21} \times 4.8GB/s \qquad (2)$$

The maximum possible bandwidth achievable with HT is given in Equation 3. Here we assume that each command only needs to add an 8-byte address to each transfer and that the commands plus data leverage write-combining to the maximum extent possible.

$$MaxPotentialHyperTransportBW \leq \frac{N}{N+8} \times \frac{64}{72} \times 2.3GB/s \quad (3)$$

The three performance limits are compared to the actual streaming bandwidth using four cores[10] in Figure 11. The constraints of the network links and the constraints of our proposed hardware support are virtually identical for 4- or 8-byte transfers. Even at 16 bytes, minimizing the HT overhead only adds about 10% to the performance as the network links become the primary constraint. Since 8-byte (double precision floating-point) and 16-byte (complex double precision floating-point) transfers are the "typical" data item sizes, it is unclear that this is a sufficient advantage to make more complicated hardware desirable.
A second viewpoint on the analysis is to consider the message rate and its relationship to the performance of the processor and I/O interface. The potential message rate for HT and the network link is related to the size of the message (with overhead adding a multiplier) and the bandwidth:

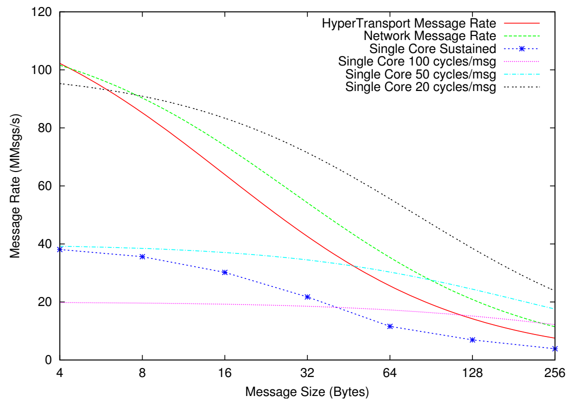
$$HyperTransportMessageRate \leq \frac{2.3GB/s}{(N+16)\times\frac{72}{64}} \qquad (4)$$

$$NetMessageRate \leq \frac{4.8GB/s}{(N+28)\times\frac{72}{64}\times\frac{21}{16}} \qquad (5)$$

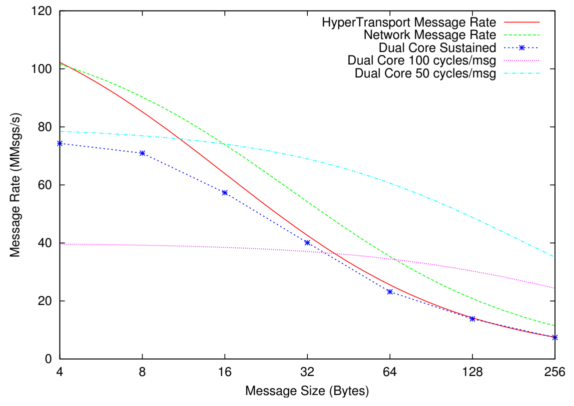
At the same time, messages certainly cannot be transferred faster

[9]Applications typically access the indirection vector with linear stride-1 accesses, which cause random appearing remote accesses

[10]All of the streaming variants achieved approximately equivalent peak performance at four cores.



(a)



(b)

Figure 12: Comparison of actual to theoretical peak message rate for (a) single- and (b) dual-core systems

than the processor can issue them. Issuing a message through a library like SHMEM incurs a certain amount of overhead. For example, the SHMEM library will frequently lead to a function call. Within that function, it is necessary to parse the arguments and handle the options. For example, the length argument of a `shmem_int_put()` can commonly involve an unrolled loop to perform the transfer. The message rate that a given processor socket running at 2 GHz can inject is also governed by:

$$SocketMessageRate \leq cores \times \frac{2GigaCycles/s}{Ncycles/msg} \qquad (6)$$

Figures 12 and 13 compare the actual sustained message rate with limits imposed by the network and processor. For the processor comparison, constant overheads of N cycles, the number of cycles required to inject a message beyond the data transfer time, are shown for multiple interesting values of N. In addition, for each 4 bytes of data, an additional cycle is added to account for the extra store instruction on the 32-bit architecture modeled. In the single- and dual-core scenarios, it is clear that messages require approximately 50 cycles/message (see the 4- and 8-byte message cases). In the quad-core case, the bandwidth begins to saturate — even at 8-byte messages. Four-byte messages similarly saturate the bus because accesses to the command queue have to be 8-byte aligned. Thus, 4-byte messages require just as much I/O bandwidth as 8-byte messages.

Another interesting note from Figures 12 and 13 is the cross-over point for the various message overheads. For single-core processors, only 20 cycles of overhead can be allowed to achieve full message rate on 4- or 8-byte messages; however, even 100 cycles of overhead is acceptable for 128-byte messages. With quad-core processors, even 100 cycles of overhead per message is acceptable as the I/O interface saturates first.

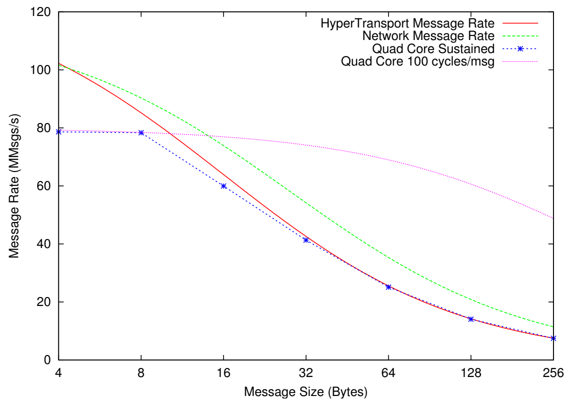


Figure 13: Comparison of actual to theoretical peak message rate for quad-core systems

7. RANDOM ACCESS PERFORMANCE

The HPC Challenge (HPCC) RandomAccess benchmark[19], also known as GUPS (GigaUpdates Per Second), was designed to test the performance of small, randomly distributed memory accesses. While numerous "optimizations" of the benchmark have been performed over time[26, 14], remaining in the spirit of the benchmark requires using small messages to PEs that are randomly distributed over the machine.

Figure 14 shows the core loop from our implementation of HPCC RandomAccess using the Cray SHMEM API[12]. For each random update generated, the local PE sends that update to the appropriate remote PE. The update arrives in a per-process buffer. After all updates have been generated and transmitted, the code enters a barrier to insure that all others have completed their update generation. After the barrier, each process scans the local buffers for updates that have arrived[11]. A second barrier ensures that all processes have completed their local updates before beginning the next iteration. Note that to simplify the code, all updates go to the network. This is not a particular performance limitation as the code scales such that very few updates are local. The table on each node is 128 MB, which is sufficient to insure that caching has a minimal effect on the performance. Finally, due to the limitations of the simulator, we limit the number of iterations and the number of updates to below what is allowed by the actual HPCC RandomAccess benchmark. Both limits actually put this implementation at a disadvantage relative to other implementations.

Figure 15 presents the performance achieved using this SHMEM variant of the HPCC RandomAccess benchmark. The x-axis is the number of *sockets*, rather than the total number of PEs, since the network interface is a per-socket resource. As the results in Section 5 would suggest, two cores per socket perform twice as well as one core per socket. Four cores per socket have a consistent 25% advantage over two cores per socket. This indicates that even though the amount of work per update has increased somewhat, dual-core processors can still drive 80% of the network bandwidth and quad-core processors can saturate the network.

[11]The random number generator is such that the update is *never* zero.

```
for (k = 0; k < 100; k++) {
  for (i = 0; i < 512; i++) {
    /* Generate random update */
    ran = (ran << 1) ^
      (ran < ZERO64B ? POLY : ZERO64B);

    /* Extract PE */
    PE = (mask & ran) >> loglocal;

    /* Send to remote buffer
         * maintained per processor
         */
    shmem_int_put(&updates[me][count[PE]],
      &ran, 2, PE);

    ++count[PE];
  }

  /* Clear remote update count */
  for (i = 0; i < nprocs; i++)
    count[i] = 0;

  /* wait for everyone to finish */
  shmem_barrier_all();

  /* Loop over per processor buffers */
  for (i = 0; i < nprocs; i++) {
    j=0;
    while (datum = updates[i][j++]) {
      index = datum & nlocalm1;
      table[index] ^= datum;
      updates[i][j] = 0;
    }
  }

  /* wait for everyone to finish */
  shmem_barrier_all();
}
```

Figure 14: RandomAccess code

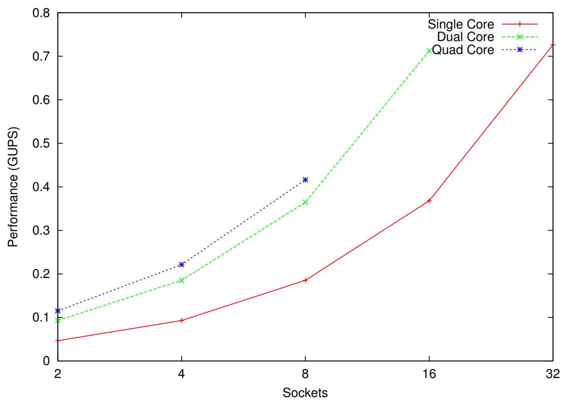


Figure 15: RandomAccess performance

8. CONCLUSIONS

This paper explores whether the computational capabilities provided by modern, multiple issue, out-of-order, multi-core CPUs is sufficient to perform the address computation functionality to support PGAS style accesses. Specifically, we hypothesized that the processors could compute remote addresses for various addressing styles, including cyclic, block cyclic, and indirect remote addressing without direct support from hardware constructs such as E-registers[30]. We test this hypothesis through the use of targeted microbenchmarks. The results indicate that four cores is always sufficient to fully saturate the network and that two cores are frequently sufficient to saturate the network — even for small message sizes. In addition, we explore the HPCC RandomAccess benchmark and find that four cores are sufficient even in cases where the processor has somewhat more work to do — in this case generating random numbers and updating tables.

We do find, however, that generating a message requires 50 cycles of overhead for a 2 GHz processor (25 ns of processor time) on average. This is more expensive than strictly necessary for a “remote store”; however, when considering the latency of a remote store relative to the remarkable amount of processing available on a node, this seems like a reasonable trade-off.

9. FUTURE WORK

While high message rate, low overhead communications are a critical component for supporting the PGAS model well, there are numerous other network interface features that are desirable for supporting the full semantics of the PGAS model. For example, we plan to explore support for remote atomic operations as well as lightweight synchronization primitives. Another area for exploration will be the translation from virtual (PE-based) to physical (network ID-based) addressing when transitioning from a SHMEM call to the physical network.

10. REFERENCES

[1] E. Anderson, J. Brooks, C. Grassl, and S. Scott. Performance of the Cray T3E multiprocessor. In *1997 ACM/IEEE Supercomputing Conference (SC’97)*, November 1997.

[2] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. Felten. Virtual memory mapped network interface for the SHRIMP multicomputer. In *21st Annual International Symposium on Computer Architecture*, pages 142–153, Chicago, Illinois, USA, Apr. 1994.

[3] N. J. Boden, D. Cohen, R. E. F. A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[4] D. Bonachea. Gasnet specification, v1.1. Technical Report UCB/CSD-02-1207, October 2002.

[5] R. Brightwell, D. Doerfler, and K. D. Underwood. A preliminary analysis of the InfiniPath and XD1 network interfaces. In *20th International Parallel and Distributed Processing Symposium (IPDPS’06) Workshop on Communication Architectures for Clusters*, April 2006.

[6] D. Burger and T. Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[7] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade high productivity language. In *Ninth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52–60, April 2004.

[8] J. Carbonaro and F. Verhoorn. Cavallino: The Teraflops

router and NIC. In *Fourth IEEE Symposium on High-Performance Interconnects (HotI'96)*, August 1996.

[9] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, May 1999.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Twentieth ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 519–538, October 2005.

[11] Cray, Inc. Cray X1E supercomputer. http://www.cray.com/products/systems/x1.

[12] Cray Research, Inc. *SHMEM Technical Note for C, SG-2516 2.3*, October 1994.

[13] L. Dickman, G. Lindahl, D. Olson, J. Rubin, and J. Broughton. PathScale InfiniPath: A first look. In *Proceedings of the 13th Symposium on High Performance Interconnects (HOTI'05)*, August 2005.

[14] R. Garg and Y. Sabharwal. Software routing and aggregation of messages to optimize the performance of the HPCC Randomaccess benchmark. In *2006 ACM/IEEE International Conference for High-Performance Computing, Networking, Storage, and Analysis (SC'06)*, November 2006.

[15] H. Hellwagner and A. Reinefeld, editors. *SCI: Scalable Coherent Interface: Architecture and xo Software for High-Performance Compute Clusters*, volume 1734 of *Lecture Notes in Computer Science*. Springer, 1999.

[16] Infiniband Trade Association. *http://www.infinibandta.org*, 1999.

[17] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, May 2005.

[18] J. Liu and D. K. Panda. Implementing efficient and scalable flow control schemes in MPI over InfiniBand. In *2004 Workshop on Communication Architecture for Clusters (CAC'04)*, April 2004.

[19] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, R. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC challenge benchmark suite, March 2005. http://icl.cs.utk.edu/hpcc/pubs/index.html.

[20] D. Mayhew and V. Krishnan. PCI Express and Advanced Switching: Evolutionary path to building next generation interconnects. In *Eleventh IEEE Symposium on High-Performance Interconnects (HotI'04)*, August 2004.

[21] Mellanox, Inc. New Mellanox ConnectX IB adapters unleash multi-core processor performance. http://www.mellanox.com/news/press_releases/pr_032607.php.

[22] J. Nieplocha and B. Carpenter. *ARMCI: A Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems*, volume 1586, pages 533–546. Springer, 1999.

[23] J. Nieplocha and R. Harrison. Shared-memory programming in metacomputing environments: The Global Array approach. *The Journal of Supercomputing*, 11:119–136, 1997.

[24] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[25] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[26] S. Plimpton, R. Brightwell, C. Vaughan, K. Underwood, and M. Davis. A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark. In *2006 IEEE International Conference on Cluster Computing*, September 2006.

[27] QLogic, Inc. InfiniPath interconnect performance. http://www.pathscale.com/infinipath-perf.html.

[28] Quadrics, Inc. QSNet-II performance results. http://www.quadrics.com/.

[29] D. Roweth and A. Pittman. Optimised global reduction on QsNet-II. In *Thirteenth IEEE Symposium on High-Performance Interconnects (HotI'05)*, August 2005.

[30] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Seventh ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[31] G. L. Steele, Jr. Parallel programming and code selection in Fortress. In *Eleventh ACM Symposium on Principles and Practice of Parallel Programming*, March 2006.

[32] K. Underwood. Challenges and issues in benchmarking MPI. In B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI Users' Group Meeting, Bonn, Germany, September 2006 Proceedings*, volume 4192 of *Lecture Notes in Computer Science*, pages 339–346. Springer-Verlag, 2006.

[33] K. D. Underwood, M. Levenhagen, and A. Rodrigues. Simulating Red Storm: Challenges and successes in building a system simulation. In *21st International Parallel and Distributed Processing Symposium (IPDPS'07)*, March 2007.

[34] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual International Symposium on Computer Architecture*, pages 256–266, May 1992.