

# A preliminary evaluation of the hardware acceleration of the Cray Gemini Interconnect for PGAS languages and comparison with MPI

Hongzhang Shan    Nicholas J. Wright  
John Shalf    Katherine Yelick  
CRD and NERSC, Lawrence Berkeley National  
Laboratory, Berkeley, CA 94720  
hshan, njwright, jshalf, kayelick@lbl.gov

Marcus Wagner    Nathan Wichmann  
Cray Inc. 380 Jackson Street, Suite 210, St. Paul, MN  
55101  
marcus, wichmann@cray.com

## Abstract

The Gemini interconnect on the Cray XE6 platform provides for lightweight remote direct memory access (RDMA) between nodes, which is useful for implementing partitioned global address space languages like UPC and Co-Array Fortran. In this paper, we perform a study of Gemini performance using a set of communication microbenchmarks and compare the performance of one-sided communication in PGAS languages with two-sided MPI. Our results demonstrate the performance benefits of the PGAS model on Gemini hardware, showing in what circumstances and by how much one-sided communication outperforms two-sided in terms of messaging rate, aggregate bandwidth, and computation and communication overlap capability. For example, for 8-byte and 2KB messages the one-sided messaging rate is 5 and 10 times greater respectively than the two-sided one. The study also reveals important information about how to optimize one-sided Gemini communication.

**Keywords** PGAS, CAF, UPC, MPI, Gemini, Hardware Acceleration, Performance, Message Rate, Overlap

## 1. Introduction

The classic parallel programming model, MPI, faces several new challenges on petaflop computing platforms, which are dominated by multicore-node architectures [9, 26]. These challenges include a reduced amount of memory per core, reduced memory and network bandwidth per core, and the inefficiency of using two-sided messages to handle a large amount of fine-grain communication. To address these challenges, researchers are starting to investigate other programming models to understand whether they could replace or used in combination with MPI.

Among these studied programming models, the Partitioned Global Address Space (PGAS) family of languages show great promise as the near-term alternative to MPI. Co-Array Fortran (CAF) [19] and Unified Parallel C (UPC) [5] are two representative examples of such languages. Compared with MPI, a big difference is that PGAS provides a global shared address space while controlling locality. This is designed to simplify programming as with this global shared space abstraction PGAS languages allow the ability to directly build distributed data structures that can be accessed throughout the machine. By integrating communication and synchronization into the language itself, PGAS languages allow the compiler or runtime system to distribute and schedule remote memory accesses in an optimal manner without the need for maintaining a global, uniform access view of memory on a distributed

memory system [1]. In contrast in MPI each process has only a private address space and the communication must be done explicitly. It is therefore difficult for the compiler to optimize communication and the MPI program developer has to be fully responsible for the communication optimization, and obtaining any possible overlap with computation.

In CAF, program instances are called *images* and global variables are allocated symmetrically across all images. An extended array index in square brackets is used to reference data access across different images. The subscripts for this extended array indices are the image rank starting from 1, indicating the physical location of the data. Accesses to these global data, called co-arrays, will result in remote memory accesses that are similar to remote memory put and get operations. In UPC, the global variables are defined by language keyword “shared”, and they can be accessed by all UPC threads. By default, the variables will be allocated on thread 0. However, UPC provides a block-cyclic method to distribute the data in a round-robin fashion across all threads.

Both CAF and UPC have been around for a decade or so. However these two languages still have not been widely adopted by user community; partly because of the lack of a developer environment, and partly because not enough convincing performance results have been presented to demonstrate they are superior and viable alternatives to the MPI programming model.

Hopper is a 1.28 PF peak Cray XE6 computing platform recently installed at NERSC. The defining feature of this platform is the custom interconnect, called Gemini, which provides a hardware accelerated global address space and allows remote direct memory access (RDMA) from any node to any other in the system. In this work, we will investigate what the effect of this special Gemini hardware support for global address space and one-sided messaging is upon the performance of the PGAS languages. We compare whether PGAS languages can outperform MPI, with an aim to determining if PGAS is a superior and viable alternative to MPI and under which circumstances. Note that we only compare PGAS performance with MPI two-sided messaging, because in our experience MPI one-sided functions are rarely used in scientific applications, which often leads to poor performance because of sub-optimal implementations.

The principle contributions of this paper are

- We translated several popular MPI benchmarks into PGAS languages to measure network bandwidth and messaging rate and facilitate comparisons. Our results show that in the bandwidth limit, with large messages, MPI and PGAS performance is identical. For medium-sized and small messages, the lower over-

head of the single-sided PGAS messages allows greater effective bandwidths to be achieved.

- We developed a version of the STREAM benchmark using CAF and showed that the RDMA operations executed by the Gemini allow very high STREAM copy bandwidths to be achieved. We also show that the STREAM benchmark can be optimized to overlap computation and communication, enabling substantial performance improvement over the naive version for the remaining operations (Scale, Add and Triad.)
- We developed an independent micro-benchmark to measure the capability to overlap computation and communication for PGAS languages and MPI. Our results show that with large messages almost complete overlap of computation and communication is possible with PGAS languages using Gemini whereas with MPI this is currently not possible.
- We translated the NAS FT benchmark into Co-Array Fortran and achieved up to  $2.8\times$  the performance of the original MPI version on 16K cores.

The paper is organized as follows. Related work is discussed in Section 2. Section 3 describes the experimental platforms. The messaging rate is examined in Section 4. As the number of cores on the future supercomputing platforms increases rapidly, the average message size will become smaller, therefore overall performance will become more sensitive to the performance of fine-grain communication thus the throughput of small messages is carefully studied in this section. The performance and optimization of the STREAM benchmark is discussed in Section 5. Section 6 examines the capability of PGAS and MPI to overlap computation and communication. In Section 7, we compare the performance differences of an MPI and a CAF version of the NAS FT benchmark. Finally, we summarize our conclusions and future work in Section 8.

## 2. Related Work

The related work can be divided into two categories based on the programming models used: CAF and UPC. For CAF, Mellor-Crummey et al. have proposed Co-array Fortran 2.0 [14]. Their CAF2.0 compiler uses a source-to-source translator to convert CAF code into Fortran 90 (F90) with calls to a low-level one-sided communication library such as GASNet. The CAF programs have been converted from their MPI counterparts and some optimizations have been made to improve the CAF performance. The applications include MG, CG, BT, and SP from the NAS parallel benchmark suite and the Sweep3D neutron transport benchmark. The codes have been tested on several small clusters which consist of different processors and networks. The CAF programs show nearly equal or slightly better performance than their MPI counterparts [6, 7]. Recently, the CAF2.0 version of High Performance Computing Challenge Benchmark suite has been developed and tested on a Cray XT4 platform [10]. They have scaled several benchmark applications to 4096 CPU cores and shown CAF2.0 is a viable PGAS programming model for scalable parallel computing.

Barrett [3] studied different Co-array Fortran implementations of Finite Differencing Methods on Cray X1 and found that CAF exhibits better performance for smaller grid sizes and similar results for larger ones. Numrich et al. [20] developed a Co-array enabled Multigrid solver focusing primarily on programmability aspects. Bala et al. [2] demonstrated performance improvements over MPI in a molecular dynamics application on a legacy HPC platform (Cray T3E). In addition, parallel linear algebra kernels for tensors [18] and matrices [22] benefit from a Co-array based one-sided communication model due to a raised level of abstraction with little or no loss of performance over MPI.

Comparing the performance of UPC and MPI have been the subject of many papers [4, 8, 11, 13, 16, 23]. Mallon et al. [13] evaluated the performance of MPI, OpenMP, and UPC on a machine with 142 HP Integrity rx7640 nodes interconnected via InfiniBand. The authors claim that MPI is the best choice to use on multicore platforms, as it takes the highest advantage of data locality. El-Ghazawi and Cantonnet [8] discussed UPC performance and potential advantage using NPB applications. With proper hand tuning and optimized collective libraries, UPC delivered comparable performance to MPI. Shan [23] and Jin [11] also compared the performance of NPB on several different platforms and similar conclusions were drawn. Nishtala and other researchers [4, 16] discussed the scaling behavior and better performance of NAS FT for UPC on several different platforms using the Berkeley UPC compiler with GASNET communication system. The UPC version is developed from scratch to take advantage of the lower overhead of the one-sided communication and the overlap of communication and computation.

The principle difference of our study is that it is performed on a platform with the Cray Gemini interconnect which provides hardware acceleration for one-sided communication, which is advantageous for the performance of PGAS languages. In this study, we examine how this special hardware support affects the performance of PGAS languages as compared to MPI.

## 3. Experimental Platform

### 3.1 Hopper

A majority of our work has been performed on a Cray XE6 platform, called Hopper, which is located at NERSC and consists of 6,384 dual-socket nodes each with 32GB of memory. Each socket within a node contains an AMD “Magny-Cours” processor at 2.1 GHz with 12 cores. Each Magny-Cours package is itself a MCM (Multi-Chip Module) containing two hex-core dies connected via hyper-transport. (See Fig. 1.) Each die has its own memory controller that is connected to two 4-GB DIMMS. This means each node can effectively be viewed as having four chips and there are large potential performance penalties for crossing the NUMA domains. Every pair of nodes is connected via hypertransport to a Cray Gemini network chip, which collectively form a  $17\times 8\times 24$  3-D torus. (See Fig. 2.)

In this work we used the Cray compiler version 7.4.0 which provides support for Co-Array Fortran (CAF) and Unified Parallel C (UPC). The Cray compiler supports different kinds of compiler directives to control synchronization points and different communication approaches. The most significant one for this work is the `#PRAGMA PGAS DEFER SYNC` pragma which instructs the compiler to make the PGAS communication event on the following source code line non-blocking, thus allowing the potential for overlap of computation and communication. Note that this is not analogous to overlap of communication and computation with MPI. In that case the aim is to allow both the sender and receiver can perform computation whilst the message is in flight. In this case the thread or image being put to (or got from) is not directly involved in the single-sided communication event and is already able to compute while it is being communicated with.

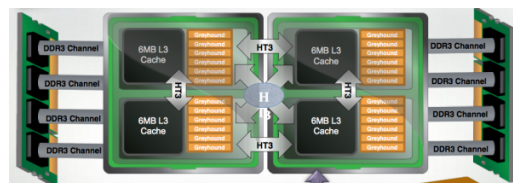


Figure 1. The node architecture of Hopper.

### 3.1.1 Gemini

The defining feature of the Cray XE6 architecture is the Gemini interconnect, which provides a global address space. There are two mechanisms to transfer the internode messages using one-sided communication with Gemini. The first is uses Fast Memory Access (FMA) and the second uses the Block Transfer Engine (BTE). The FMA transport mechanism involves the CPU, has low latency and more than one transfer can be active at the same time. Transfers using the BTE are performed by the Gemini network chip, asynchronously with CPU so that the communication and computation can overlap. The FMA mechanism enables native processor stores to a memory-mapped window to initiate network transaction directly from user mode. For PUTs, the store contains data to be stored at the remote destination. For GETs, the store contains a control word that indicates where the result of the GET should be deposited in local memory [12]. The BTE allows direct memory transfers between different nodes. Transmit descriptors are used to specify both the local and remote memory addresses of the transfers. The descriptors have to be processed via a hardware transmit queue which does not have direct access in user mode. In general, FMA is used to transfer short messages and BTE for long messages. The point at which this transition occurs is controlled by the `MPICH_GNI_RDMA_THRESHOLD` environment variable for MPI and by `PGAS_OFFLOAD_THRESHOLD` for PGAS languages.

All the results shown here using two nodes are for two nodes connected to different Gemini's, unless otherwise mentioned.

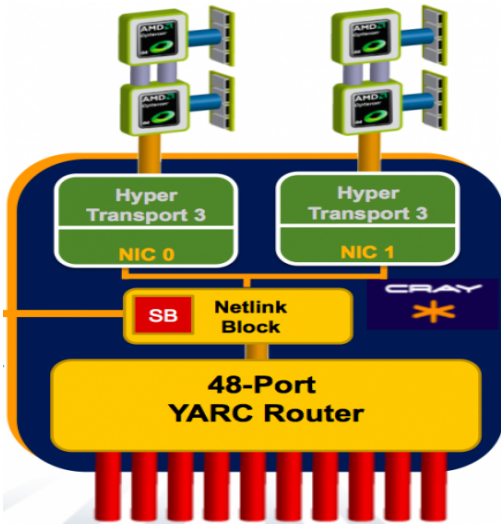


Figure 2. The Gemini interconnect on Hopper.

### 3.2 Franklin

For comparison purposes, we also performed some of the tests on a Cray XT4 platform, called Franklin, which is also located at NERSC. Each node consists of a quad-core AMD Budapest 2.3GHz processor and 8 GB DDR3 800 MHz memory. The nodes are connected through a proprietary SeaStar2 interconnect which is designed to optimize MPI performance by handling the handshaking protocol needed by MPI. As with Hopper the interconnect is a 3D-torus but in this case each node represents a distinct point on the torus.

## 4. Messaging Rate

Messaging rate is an important performance metric for measuring the viability of the interconnects on HPC platforms, especially for

PGAS programming languages [25]. Also, as we look towards exascale architectures, where because of memory constraints problems are likely to be strong-scaled, messaging rate is likely to become a more and more important performance metric.

### 4.1 Implementation

The MPI version is obtained from OSU Micro benchmark suite [15] and slightly changed to use a different buffer for each iteration. In the MPI version, a process sends a series of same size messages to its partner using nonblocking `MPI_Isend`. The number of messages in is determined by a variable called “window size” and our experiments show that setting the window size to 64 is large enough to achieve converged performance. After the partner has received all the messages, it will send an acknowledgement back to the sender. In the CAF implementation, the nonblocking `MPI_Isend` is substituted by a loop with direct load/store assignment (corresponding to one-sided put operation). However for each loop iteration the starting address is incremented by one so that the data sent is not contiguous between loop iterations. This prevents the compiler collapsing the loop into one put. In order to ensure that non-blocking communication was used the delayed synchronization compiler directive `pgas defer_sync` was used. Then, at the end, a synchronization is called to ensure all data have been received.

### 4.2 Performance

The codes are executed using two sets of processes, one on each node. The messaging rate between two nodes using 1, 6, and 24 communicating pairs per node for CAF and MPI are shown in Fig. 3 and 4 respectively. (The two nodes have a 1-hop network distance.) For small messages, MPI achieves the best performance when 6 pairs are used, a rate of 9 million messages per second. Using 24 pairs, the message rate drops slightly.<sup>1</sup> On the contrary, the message rates of CAF for small messages increase steadily with the number of communicating pairs used. The best performance is obtained when 24 pairs are used, which is about 4.7 times better than the best MPI message rate. CAF clearly shows much better scalability for small messages.

We also measured the messaging rate using get instead of put for CAF. In the bandwidth limit, as one might expect, the get and put performance is identical. However, for small messages put performs significantly better than get. The messaging rates with 8-byte messages are 1.46, 1.64, and 3.46× greater than get using one, six, and 24 pairs respectively.

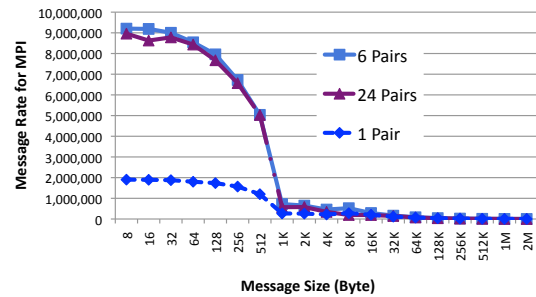
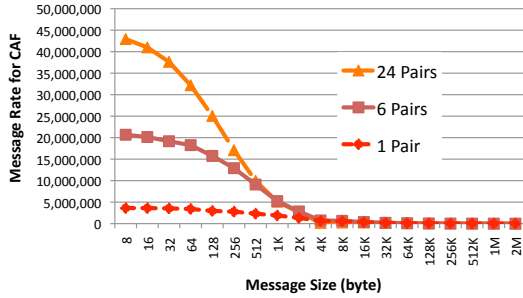


Figure 3. The messaging rate for MPI using 1, 6, and 24 pairs per node.

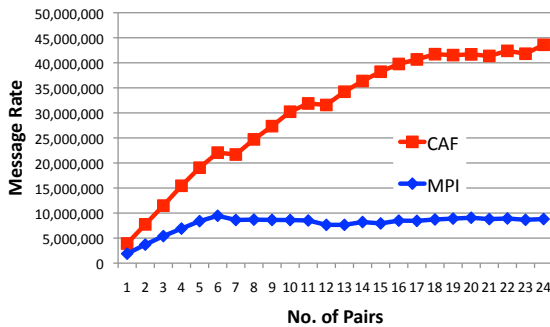
<sup>1</sup> Note that in order to achieve this result with 24 pairs the environment variable `MPICH_GNI_MBOX_PLACEMENT` was changed to “nic”. With it set to the default value the performance for 24 pairs is approximately equal to that for one pair.



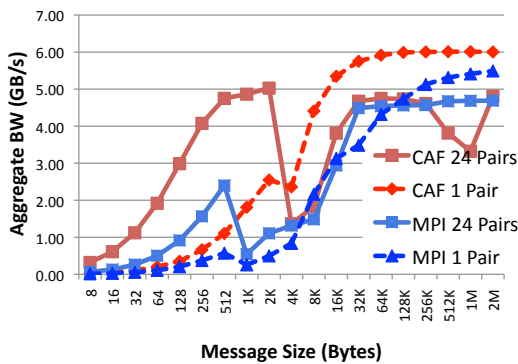
**Figure 4.** The messaging rate for CAF using 1, 6, and 24 pairs per node.

Fig. 5 shows in detail the performance differences for 8 byte messages between MPI and CAF using from 1 to 24 pairs between two nodes. The peak performance of MPI increases up until when 6 pairs are used. After that point, the performance no longer improves. On the contrary, the performance of CAF keeps going up as more pairs are used and is several times better than MPI when 24 pairs are used. The increase with adding more and more communicating pairs is not linear though, for example the sixth pair achieves almost half of the overall aggregate rate. The principle difference between MPI and CAF is that CAF is using the low-level single-sided messaging, whereas MPI is using two-sided messaging.

The CAF rate drops slightly going from 6 to 7 and 11 to 12 pairs. Presumably in these cases the extra amount of requests produced by adding one core is offset by the increased contention introduced.



**Figure 5.** The scaling behavior of MPI and CAF between two nodes for 8-byte message.



**Figure 6.** The corresponding bandwidth of MPI and CAF for message rates in Fig. 3 and 4.

The corresponding bandwidths for the message rates shown in Fig. 3 and 4 are shown in Fig. 6. The CAF performance increases much faster with increasing message size. The main performance difference between CAF and MPI occurs for message sizes in the middle of the range where the benefits of single-sided messaging are mostly strongly felt. For a 512 byte message, using 24 pairs, the CAF performance has reached 4.75 GB/s, which is much higher than the corresponding MPI bandwidth of 2.4 GB/s. This result indicates that for PGAS languages, by using more frequent medium-sized messages instead of the large bulk transfers by explicit message passing, better performance can be achieved due to increased messaging rate and less network contention. This is in agreement with a recent study of the GTS fusion application on a Cray XE6 [21].

For very small messages aggregation is still necessary. As shown in Fig. 4, the messaging rate for 8-byte and 16-byte messages is very close, thus using 16-byte messages will achieve almost double the bandwidth of 8-byte messages. Even so, because of the increased messaging rate, in the latency limit the PGAS effective bandwidth for 8-bytes messages is about  $4.5\times$  the MPI one.

As the message size increases there is a performance drop for MPI when the message size reaches 1KB. This is the threshold value for switching from using FMA to using the BTE for transferring internode MPI message data. The startup cost for the BTE is the reason for the sudden performance drop, which cannot be amortized well for such small message sizes.

There is also a performance drop for CAF when the message size reaches 4096 bytes, which is the threshold in CAF to switch from the FMA mechanism to using the BTE. We also note that as more communicating pairs are used, the phenomenon becomes more explicit, which is simply because the BTE processes requests through the kernel and therefore sequentially which means the startup cost will be accumulated as more communicating pairs are used.

Using one pair the highest bandwidth for CAF is around 6GB/s, which is close to the peak injection bandwidth to the Gemini interconnect from a node. Using 24 pairs this is reduced to around 5 GB/s, presumably because of contention for resources. The highest bandwidth for MPI is achieved using 1 pair and is a little lower than the CAF result, around 5.5GB/s.

### 4.3 Performance on Cray XT4 with SeaStar Interconnect

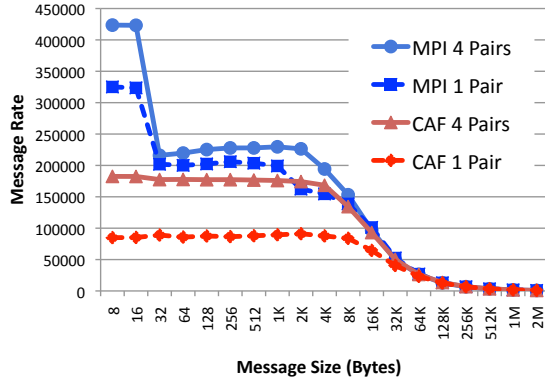
In order to better understand the performance benefits of the Gemini interconnect for PGAS languages, we examined the message rates on Franklin, a Cray XT4 platform, with a custom SeaStar interconnect. The SeaStar interconnect does not support a global shared address space. Instead it was designed to optimize the MPI performance, managing the handshaking protocol.

The XT4 messaging rates for MPI and CAF for one and four pairs over two nodes are shown in Fig. 7 and the corresponding aggregate bandwidths are displayed in Fig. 8. For large messages, both MPI and CAF deliver very similar performance and the network can be saturated easily by using 1 pair only. For the smallest messages, especially 8 and 16 bytes, contrary to the Hopper results, MPI performs much better than CAF. Across all the rest of the message size range the differences in performance between MPI and CAF are less significant, and the noticeable differences that were present for Gemini are not longer present, due to the absence of hardware acceleration for one-sided messaging.

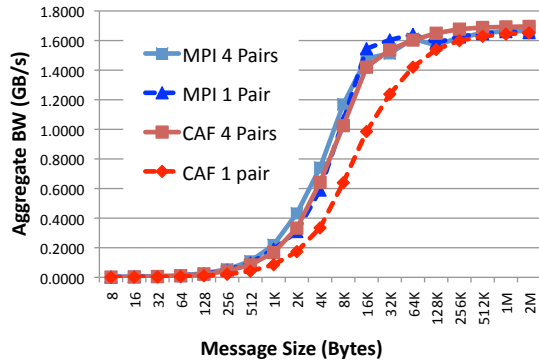
The absolute message rate of CAF and MPI for 8 bytes and 2M bytes messages on SeaStar and Gemini Interconnects are shown in Table 1. They are measured using 1 communication pair. For 8-byte messages, on SeaStar, it's the MPI that achieves the best performance while on Gemini, it's CAF. From SeaStar to Gemini, the

**Table 1.** The message rate for 8B and 2MB messages using 1 pair on SeaStar and Gemini Interconnects

	SeaStar			Gemini			Gemini/SeaStar	
	CAF	MPI	CAF/MPI	CAF	MPI	CAF/MPI	CAF	MPI
8B	86,347	325,053	0.27	3,625,000	1,903,663	1.90	41.98	5.86
512B	87,850	203,633	0.43	2,315,000	1,201,215	1.93	26.35	5.90
2MB	846	847	1.00	3,074	2,812	1.09	3.63	3.32



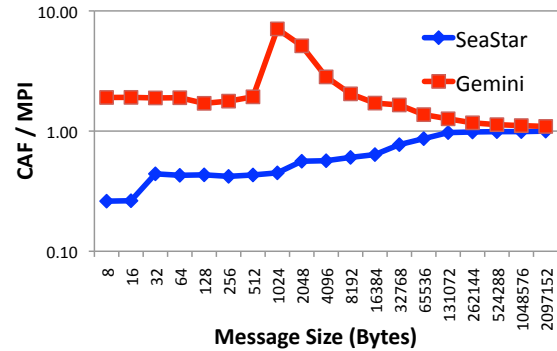
**Figure 7.** The message rates for MPI and CAF on XT4 for 1 and 4 pairs communicating per node.



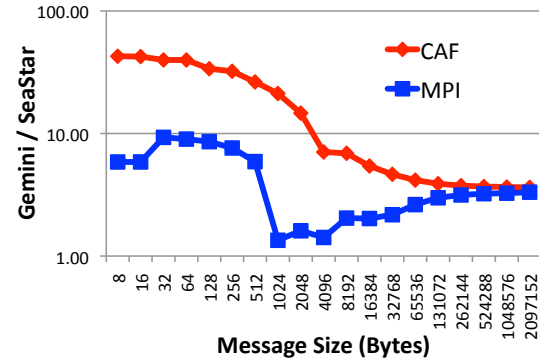
**Figure 8.** The bandwidth corresponding to the Message Rate Measurements for MPI and CAF on XT4 for one and four communicating pairs per node.

MPI performance has been improved about 6 times while the CAF performance have been increased over 40 times. The results clearly demonstrated the critical importance of the hardware support to programming models and languages. For 2MB messages, the performance will be bound by the network bandwidth. On SeaStar, both MPI and CAF deliver similar performance while on Gemini, CAF performs slightly better. As one might expect in the bandwidth limit the importance of the hardware support is diminished.

The performance differences between the XE6 and XT4 are shown in figures 9 and 10. Fig. 9 shows the ratio of the CAF to MPI performance for the two systems as a function of message size and illustrates the benefits of the hardware acceleration for PGAS. In the best case, for 1024 byte messages, CAF is almost 10 times faster than MPI on the XE6, whereas on the XT4 CAF is never faster. Fig.10 shows the Gemini/Seastar (XE6/XT4)performance ratio for CAF and MPI. Apart from the fluctuations in the performance caused by the change in protocol the MPI performance ratio is



**Figure 9.** Ratio of the CAF performance to the MPI for the Gemini (XE6) and SeaStar (XT4) based machines.



**Figure 10.** Ratio of the Gemini (XE6) to SeaStar (XT4) performance for CAF and MPI.

always significantly less than the CAF one. For both machines, the performance differences between MPI and CAF are not present in the bandwidth limit.

## 5. STREAM

STREAM is a benchmark that has been used extensively to measure memory bandwidth [24]. It measures the performance for four array operations: Copy, Scale, Add, and Triad. In this section, we created a CAF version of STREAM. Instead of measuring the memory bandwidth, we focus on communication. Compared with Message Rate benchmarks, it involves computation as well as communication and measures the compilers ability to coalesce off-node memory accesses for optimal performance.

### 5.1 Implementation

Table 2 lists how the four local array operations are adapted into communication in our CAF implementation. Similar to the Message Rate benchmark, each CAF image has a peer to communicate with it and the communication is done via get-style operations.

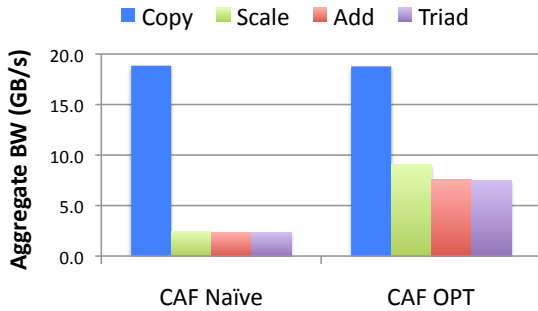
**Table 2.** STREAM Operations in CAF.

	Sequential	CAF
Copy	$c(1:n) = a(1:n)$	$c(1:n) = a(1:n)[peer]$
Scale	$b(1:n) = scalar * c(1:n)$	$b(1:n) = scalar * c(1:n)[peer]$
Add	$c(1:n) = a(1:n) + b(1:n)$	$c(1:n) = a(1:n)[peer] + b(1:n)[peer]$
Triad	$a(1:n) = b(1:n) + scalar * c(1:n)$	$a(1:n) = b(1:n)[peer] + scalar * c(1:n)[peer]$

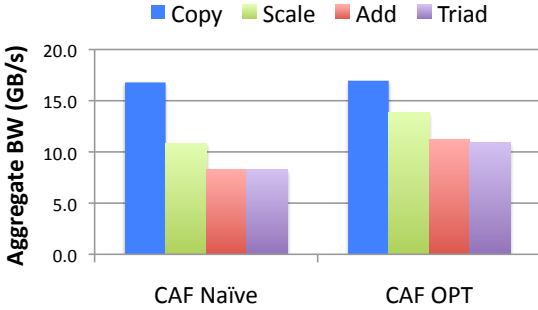
These experiments are performed using two nodes, with either one or 24 communicating pairs per node. (The communication is bi-directional.)

## 5.2 Performance

The performance of these four array operations using two different nodes are shown in Fig. 11 (labeled as CAF). (Each array is 4 Million double-precision words long - about 32 MB.)



**Figure 11.** The STREAM bandwidth between two nodes using 1 pair.



**Figure 12.** The STREAM bandwidth between two nodes using 24 pairs.

Firstly, focusing our attention of the first set of bars that represent the results for the naïve CAF implementation, we notice that the Copy bandwidth is much higher than the peak injection bandwidths ( $2 * 6$  GB/s). This is related with the bandwidth calculation formula. In STREAM, the sizes of both the source array  $a$  and destination array  $c$  are included in the computation of the bandwidth. However, when the BTE is used only the source array  $a$  needs to be accessed across the interconnect and destination array  $c$  is directly overwritten in place using an RDMA style operation, thus the total amount of data moved is overestimated by a factor of two. If we divide the STREAM copy bandwidth by two, the result falls below the peak injection bandwidth. If we replace the  $peer$  in our CAF code to  $this\_image()$ , i.e., using local memory operations to replace

the communication, all four operations deliver almost identical performance, albeit with a slightly reduced bandwidth as RDMA operations will no longer be used.

For other three operations the whole arrays have been divided into 2048-byte blocks. For each block, in addition to calling `pgas_memget_nb` to fetch the data, they also need to be routed to CPU to perform additional computations on them, leading to the lower reported bandwidth. There are two problems with the above approach generated by the compiler. First, using 2048-byte messages is not the optimal option to saturate the network bandwidth using one core on each node. From Fig. 6, we find that for such message size, using 1 pair, the corresponding bandwidth is below than 3 GB/s, far from the 6GB/s peak. Secondly, there is no overlap between the data transfer and computations.

To address these two problems, we perform two optimizations .

First, we use a larger block size, then we overlap the computation and communication. For this purpose, we divide the array into blocks, and overlap the communication and computation through pipelining. The changed code for Scale is shown as following:

```

b(1:block) = c(1:block)[peer]
DO i = 1, nblock-1
  !dir$ pgas defer_sync
  b(i*block+1 : (i+1)*block) =
    c(i*block+1 : (i+1)*block)[peer]
  b((i-1)*block+1 : i*block) =
    scalar * b((i-1)*block+1 : i*block)
  call sync_memory();
ENDDO
i = nblock
b((i-1)*block+1 : i*block) =
  scalar * b((i-1)*block+1 : i*block)

```

One can imagine using similar schemes for Add and Triad, using an extra buffer array of size `block`. The overlapped results are also shown in Fig. 11 labelled as CAF OPT, which is obtained when the block size is 256K double precision words. The performances of all three operations, Scale, Add, and Triad, improves substantially, more than three times better than the naïve results.

Fig. 12 shows the STREAM performance using 24 pairs (two full nodes) to communicate. The performance of Copy is slightly lower than that of using 1 pair, probably due to greater contention for resources, as all of the Gemini traffic is routed through die 0 as that is the one with the direct HT link. These results are consistent with Fig. 6. Using 24 pairs, the Scale, Sum, and Triad operations perform much better than using 1 pair. Using more pairs significantly improves the aggregate network bandwidth. The difference between the performance of Copy and other three operations also becomes smaller, especially the Scale operation.

## 5.3 Effects of Compiler Optimization Flags

To further study the effects of the compiler optimizations upon performance, we examine the bandwidth differences for STREAM between two different optimization flags “-O0” and “-O2” (the default case). The results are reported in Table 3 for the naïve CAF version using 1 pair between two nodes. With “-O2” optimization option, the performance is significantly better than that with “-O0”, with

hundreds of times better bandwidth. With “-O0” flags, the compiler does not perform any optimization, each element of the array is remotely fetched as a single word. This illustrates the importance of the compiler in achieving good performance for a PGAS code and its ability to recognize communication patterns. In the case that the compiler does not recognize the communication pattern involved correctly the performance will be reduced to the “-O0” case. For example, we also developed a put version of STREAM. (For Scale, the code becomes “b(1:n)[peer]=scalar\*c(1:n)” and so on.) In this case the compiler could not recognize the communication pattern correctly and generated code to fetch one word at a time. Applying the pipelining and communication-computation overlap strategies outlined above lead to the performance of the put version being within 5% of the get ones for both the one and 24 tasks per node cases.

**Table 3.** The Performance Effects of Compiler Flags on STREAM results (GB/s)

	-O0	-O2	Ratio
Copy	0.0193	17.78	921
Scale	0.0190	2.50	131
Add	0.0148	2.31	156
Triad	0.0148	2.32	156

## 6. Computation and Communication Overlap

One important technique to tolerate the cost of communication is to overlap it with local computation. To measure the overlap capability of different programming models using Gemini, a synthetic micro-benchmark has been developed.

### 6.1 Implementation

The micro benchmark has two input parameters, one is the communication message size (S), another is the time ratio of computation to communication (R). The communication time is first measured based on the input message size and is used to determine the loop length of the computational kernel

```
for (i = 0; i < Length; i++) {
    temp += buf1[i] * buf2[i] }
```

so that Computation Time = Ratio \* Communication Time.

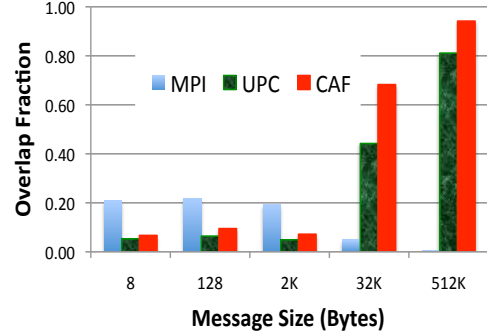
The arrays used for the communication kernel and the arrays used for the computation kernel are independent. Therefore, ideally, the computation and communication can be completely overlapped. The MPI version uses the nonblocking MPI\_Isend, MPI\_Irecv, and MPI\_Wait functions. In UPC, the nonblocking call `upc_memput_nb` is used. In CAF, get statements are used with the compiler directive, `pgas_defer_sync`, as above in the STREAM example. In principle, such statements can also be used in UPC, however, we found that this approach does not work for UPC using the Cray compiler currently.

### 6.2 Performance

A metric called `overlapped_fraction` is computed using following formula:

$$overlapped\_fraction = 1 - \left( \frac{T_{TotalRunningTime} - \max(T_{Comp}, T_{Comm})}{\min(T_{Comp}, T_{Comm})} \right)$$

where  $T_{comp}$  is the computation time and  $T_{comm}$  is the communication time. In the case that the runtime is equal to the maximum of the separate measurements of computation and communication



**Figure 13.** The overlap capability of MPI, UPC, and CAF.

the overlap is perfect. This fraction represents the amount of work that it was not possible to overlap.

Fig. 13 shows the results for MPI, UPC, and CAF using two cores, one on each node. The computation time to communication time ratio is set as 1. Several message sizes are tested, ranging from 8 to 512 KB bytes. For MPI, we observe around 20% overlap fraction for messages up to 2 KB. Beyond that point, the overlapped fraction goes down dramatically and the overlap almost completely disappears when the message size reaches 512 KB. UPC and CAF show the opposite overlap capability to MPI as a function of message size. For small messages, the overlap fraction is only around 5%. However, for large messages, such as 512KB, the overlap fraction can reach above 80%. CAF performs even better than UPC, for message size 512KB, over 90% of the communication time is overlapped with computation time. Presumably the difference between the CAF and UPC results is due to the slightly different mechanisms used in each case, as described above.

The higher overlap capability of UPC and CAF for large messages is related to the BTE message transfer mechanism. The BTE is part of the Gemini, and works asynchronously with respect to the CPU.

The poor overlap capability of MPI is related with the handshaking protocol needed between the sender and the receiver in MPI programming model, which may consume a lot of CPU cycles. The Gemini has no special hardware support for this, unlike SeaStar interconnect, and therefore it is much more difficult for MPI to overlap the communication and computation. (We note that a software based mechanism for this is in development at Cray currently.)

For smaller messages using PGAS, FMA is used to transfer the message data. FMA needs CPU involvement to initiate the transfer activity leading to lower overlap capability. Setting the input parameter R higher (i.e. more local computation time), could improve the overlapped fraction.

The above computational kernel involves a lot of data access to memory; it is a STREAM-like loop. We also developed a computation intensive kernel with reduced memory activity that worked only on data in cache and obtained similar results, which indicates we are not subject to contention for memory bandwidth between the CPU and the BTE, at least for these experiments.

## 7. NAS FT

In this section, we examine the performance differences between a MPI and a CAF version of a popular benchmark application, NAS FT. The NAS FT benchmark solves partial differential equations using Fast Fourier Transform (FFT) method. The MPI version is obtained directly from NP3.3 benchmark suite [17]. The CAF version is converted from the MPI version by replacing the dom-

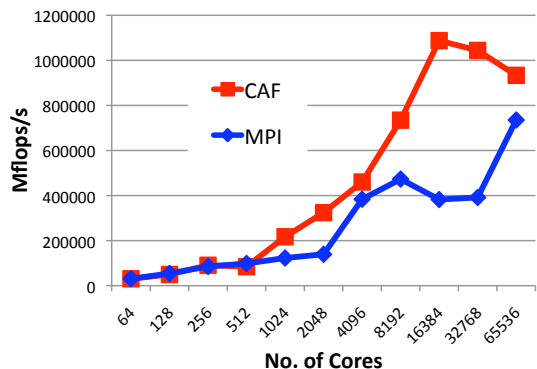
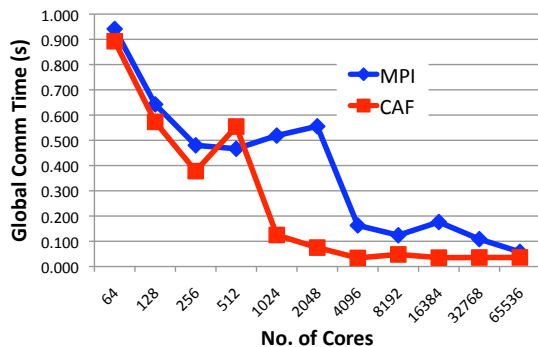
**Table 4.** The message sizes for NAS FT Class B (BYTE)

	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
transpose 1	131072	32768	8192	4096	2048	1024	512	256	128	64	32
transpose 2				524288	131072	32768	8192	2048	512	128	32

inant MPI call, an MPI\_Alltoall, with a CAF implementation. We also added some necessary synchronizations, and changed the corresponding data array to a co-array. The implementation of the CAF alltoall communication uses a round-robin communication pattern.

The performance in terms of Mflops for the Class B problem size is shown in Fig. 14 for up to 64K cores. The 3D grid size is  $512 \times 256 \times 256$  in X, Y, and Z direction individually. When the total number of processes (nprocs) is less or equal to grid size in Z direction (256 for Class B), a 1-D partitioning scheme is used and the grid will be partitioned among the cores along the Z direction. When the total number of processes becomes greater than the grid size in the Z direction, 2-D partitioning will be used and in addition to partitioning the grid along Z direction, the grid will also be partitioned in Y direction. The corresponding process grid is  $256 * (\text{nprocs}/256)$  and two new sub-communicators will be created. Therefore, there will be two transposes under 2-D partition, one for each new communicator. Table 4 shows the alltoall message size for different number of processes.

The overall performance comparison between MPI and CAF in terms of mflops is shown in Fig. 14 and the corresponding communication times are shown in Fig. 15.

**Figure 14.** The performance of NAS FT for Class B for the CAF and MPI versions.**Figure 15.** The global communication time of NAS FT for Class B in CAF and MPI.

Up to 256 tasks, MPI and CAF deliver similar performance. This is for two reasons. Firstly, for these three cases, 1-D partition-

ing is used, and the message size is large, in the regime where CAF and MPI performance is almost the same. This is clearly shown in Fig. 15 which compares the alltoall communication times in CAF to those of the MPI version. Secondly, local computation and local transpose time dominate the runtime at these core counts and affect the overall performance much more than global communication time. For 512 processes, we notice that the communication time for CAF has a sudden jump. This is because the message size for the first global alltoall communication is 4 KB bytes, which is the switch threshold from using FMA to the BTE. We have seen this phenomena in our earlier micro benchmarks. If we change the PGAS\_OFFLOAD\_THRESHOLD to 1MB, the performance of CAF is significantly improved at 512 cores and the performance is no longer anomalous. For MPI a similar effect occurs at 2048 cores, as shown in Fig. 15, where we can see that MPI has the highest global communication time at this core count.

For all other core counts, CAF performs significantly better than MPI as the message sizes become smaller. The best performance is obtained by using 16K cores, at which CAF is about 2.8 times faster than MPI. However, the performance gap shrinks when 64K cores are used. As shown in Fig. 15, from 16K to 64K cores, the global communication time for CAF is relatively stable while for MPI, it continues to drop but maintains higher than corresponding CAF time. Again this is due to a change in communication protocol, for messages smaller than 128 bytes, MPI uses a store and forward protocol instead of the default, as it has better performance. At 65536 cores both transposes involve messages below this limit, and hence the improved algorithm is used. Experiments increasing the threshold of the cutoff for the change in algorithm to 1 KB bytes at 16384 and 32768 cores show performance improvements of almost 50% for the MPI version.

## 8. Summary and Conclusions

Compared with the popular MPI programming model, PGAS languages provide substantial ease of programming, and the ability to construct globally accessible data structures. However, they still have not been widely adopted by user community today. This is mainly due to lack of the direct hardware support, a mature developer environment and lack of convincing performance results that they are superior to MPI.

In this work we evaluated the performance of PGAS languages on a Cray XE6 high-performance computing platform for which the Gemini interconnect provides direct support for a globally addressable memory and hardware-accelerated one-sided messaging. We examined the performance in terms of bandwidth, message rate, and capability to overlap computation with communication. The results demonstrated that with this special hardware acceleration, PGAS languages can outperform MPI, especially for messages a few KB in size, and therefore provide a viable alternative. However, they also show that simply swapping MPI calls for equivalent PGAS constructs may not necessarily be the optimal path forward for achieving good performance with PGAS, as the performance in the bandwidth limit is identical to that of MPI. Codes may need to be modified to send smaller messages more frequently than one would with MPI in order to achieve the greatest benefit from using PGAS languages. Our future work will focus on converting exist-



tent scientific applications into PGAS codes and study their performance on Hopper.

## 9. Acknowledgements

We would like to thank Ryan Olson of Cray for useful comments and suggestions. All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

## References

- [1] S. Alam, W. Sawyer, T. Stitt, N. Stringfellow, and A. Tineo. Evaluation of productivity and performance characteristics of CCE CAF and UPC compilers. In *CUG 2010, Edinburgh, Scotland*, May 2010.
- [2] P. Bala, T. Clark, and S. L. Ridgway. Application of pfortran and co-array fortran in the parallelization of the gromos96 molecular dynamics module. In *Scientific Programming* 9:61-68, January 2001.
- [3] R. Barrett. Co-array fortran experiences with finite differencing methods. In *The 48th Cray User Group meeting, Lugano, Italy*, May 2006.
- [4] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *20th International Parallel and Distributed Processing Symposium IPDPS*, April 2006.
- [5] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. In *Tech. Rep. CCS-TR-99-157 May*, May 1999.
- [6] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. M. Crummey. Co-Array fortran performance and potential: An NPB experimental study. In *In Proc. of the 16th Intl. Workshop on Languages and Compilers for Parallel Computing*, 2003.
- [7] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. Experiences with sweep3d implementations in co-array fortran. In *The Journal of Supercomputing*, 36:101-121, May 2006.
- [8] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: A npb experimental study. In *In Supercomputing*, 2002.
- [9] AI Geist. Sustained petascale: The next MPI challenge. In *EuroPVM/MPI*, October 2007.
- [10] G. Jin, J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and C. Yang. Implementation and performance evaluation of the hpc challenge benchmarks in coarray fortran 2.0. In *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Anchorage, AK. May 16-20, 2011*.
- [11] Haoqiang Jin, Robert Hood, and Piyush Mehrota. A practical study of UPC with the NAS parallel benchmarks. In *Partitioned Global Address Space Languages*, Oct., 2009.
- [12] L. Kaplan, M. T. Bruggencate, T. Johnson, S. Vormwald, D. Knaak, K. McMahon, J. Godfrey, and I. Gorodetsky. Softwares one-sided challenge. In *Cray User Group Meeting (CUG) 2011*, May 2011.
- [13] D. A. Mallon, G. L. Taboada, C. Teijeiro, J. Tourino, B. B. Fraguera, Andres Gomez, Ramon Doallo, and J. Carlos Mourino. Performance evaluation of MPI, UPC, and OpenMP on multicore architectures. In *Euro PVM/MPI 2009*, Sept 7-10, 2009.
- [14] J. Mellor-Crummey, L. Adhianto, W. N. Scherer III, and G. Jin. A new vision for coarray fortran. In *In Proceedings of the 3rd Conference on Partitioned Global Address Space Programming Models, PGAS '09, pages 5:1-5:9, New York, NY, USA, 2009*.
- [15] Osu micro-benchmark. <http://mvapich.cse.ohio-state.edu/benchmarks/>.
- [16] R. Nishtala, P. Hargrove, D. Bonachea, and K. Yelick. Scaling communication-intensive applications on bluegene/p using one-sided communication and overlap. In *23rd International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.
- [17] NAS Parallel Benchmarks. <http://www.nas.nasa.gov/Resources/Software/npb.html>.
- [18] R. W. Numrich. Parallel numerical algorithms based on tensor notation and co-array fortran syntax. In *Parallel Computing*, 31:588-607, June 2005.
- [19] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. In *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 131, August 1998.
- [20] R. W. Numrich, J. Reid, and K. Kieun. Writing a multigrid solver using co-array fortran. In *In Proceedings of the 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems, PARA '98, pages 390-399, London, UK, 1998*.
- [21] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, and A. Koniges. Multithreaded global address space communication techniques for gyrokinetic fusion applications on ultra-scale platforms. In *SC2011, to appear*, November 2011.
- [22] J Reid. Co-array fortran for full and sparse matrices. In *In Proceedings of the 6th International Conference on Applied Parallel Computing Advanced Scientific Computing, PARA '02, London*, 2002.
- [23] H. Shan, F. Blagojevic, S. J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. A programming model performance study using the nas parallel benchmarks. In *Scientific Programming- Exploring Languages for Expressing Medium to Massive On-Chip Parallelism, Vol. 18, Issue 3-4*, August 2010.
- [24] STREAM: Sustainable memory bandwidth in high performance computers. <http://www.cs.virginia.edu/stream>.
- [25] K. D. Underwood, M. J. Levenhagen, and R. Brightwell. Evaluating NIC hardware requirements to achieve high message rate PGAS support on multi-core processors. In *SC07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, New York, NY, USA, 2007*.
- [26] Challenges for the message passing interface in the petaflops era. [www.cs.uiuc.edu/homes/wgropp/bib/talks/tdata/2007/mpifuture-uiuc.pdf](http://www.cs.uiuc.edu/homes/wgropp/bib/talks/tdata/2007/mpifuture-uiuc.pdf).