Evaluation of Productivity and Performance Characteristics of CCE CAF and UPC Compilers

Sadaf Alam, William Sawyer, Tim Stitt,
Neil Stringfellow and Adrian Tineo,
Swiss National Supercomputing Center (CSCS)

ABSTRACT: *Coarray Fortran (CAF) and Unified Parallel C (UPC), representatives of the Partitioned Global Address Space (PGAS) paradigm, have been introduced over a decade ago. However, a widespread adoption by code developer communities has yet to materialize due to a lack of general availability of supported systems and software. Emerging Cray machines that include PGAS code development tools for x86 platforms and a network infrastructure that is optimized for both message-passing MPI and PGAS communication patterns, offer opportunities for code and performance portability across Cray systems and other similar cluster platforms. Therefore, we explore if and how scientific applications that are developed for a variety of platforms could target and benefit from the PGAS paradigm on systems with commodity and custom hardware and software components. Two Cray platforms, a Cray XT5 machine and a Cray X2 vector system, have been targeted for performance and productivity evaluation of PGAS compilers, which reveal that critical missing pieces of information such as remote memory mapping restrict generation of efficient code, particularly on the XT5 machine, which is composed of AMD Opteron multicore processors and a custom communication network. Strategies adopted to reduce remote memory accesses for the targeted micro-benchmarks and a representative kernel for stencil-based filter operations reduce runtimes on the XT platforms but are found to be less effective on the X2 platform. We conclude that a multi-platform CCE compiler would be essential for development and tuning of PGAS applications for upcoming systems with commodity and custom components.*

KEYWORDS: PGAS, CAF, UPC, Cray XT and X2 systems.

1. Introduction

The Partitioned Global Address Space (PGAS) programming model offers programming flexibility of a globally-shared memory programming model while introducing the concept of data locality that is similar to a distributed-memory model. In other words, globally shared data structures and functions (collectives) that access them, as well as data structures representing the locality and distribution of data across multiple processing elements are two important concepts of the programming paradigm. Part of the global shared data structure is available locally and a compiler or interpreter could make use of this locality of reference to provide improved performance. As a result, a user or code developer typically does not need to manually optimize and fine-tune the application to reduce accesses to remote data structures, while the language constructs could aid in the automatic optimization. Unlike message-passing models, these remote accesses are performed without explicit messaging i.e. they exploit one-sided communication primitives. Unlike globally shared programming model instances such as OpenMP, the PGAS model and its instances allow code developers to articulate locality of shared data types and offer mechanisms to control data mapping and distribution. Likewise, these restrictions permit the underlying compiler and runtime systems to distribute and schedule remote memory accesses in an optimal manner without maintaining a global, uniform access view of memory on a distributed memory system. Two instances of the PGAS model; Coarray Fortran (CAF) and Unified

Parallel C (UPC), have been available to the user communities for over a decade. However, there has been a lack of adoption in user communities, which is largely attributed to the availability of portable code development environments for these languages and scalable systems that showcase productivity and performance benefits for these languages over message-passing and shared-memory programming paradigms.

The current Cray Compiler Environment (CCE) provides embedded compilers for CAF and UPC languages alongside its MPI and OpenMP tools on the Cray XT and vector series systems [1][2][3]. Coarray Fortran (CAF), which is now integrated into the FORTRAN 2008 standard proposal, extends the Fortran language syntax with a construct called *coarrays*, which are essentially data structures that are shared between different *images* of a program [14]. Accesses to these co-arrays result in remote memory accesses that are similar to remote memory *put* and *get* operations. Similarly, UPC is an extension to the C language offering benefits of the PGAS model to programs written primarily in C [9]. In UPC, program instances are called *threads* and data is divided up between shared and private spaces. In addition, language qualifiers are provided which describe whether data is shared and how arrays should be distributed among available threads. The number of threads can be specified at both compile and runtime.

Although the CAF and UPC specifications do not address the issue of interoperability with other programming paradigms, such as MPI, some compiler instances, for example CCE, do not prevent combination of the two models. While it is efficient for an evolutionary code development approach, some issues remain, which are identified in this paper.

The primary motivation for evaluating the productivity of the PGAS compilers is to assess their benefits in terms of usability, execution and potential for significantly higher performance on the next-generation Cray interconnect. It is for this reason, even though we do not evaluate performance on the SeaStar network interface on the XT platforms and the network API called Portals [10], we attempt to gain an insight into compiler transformations that are likely to impact performance on both current and also future generation scalable systems that are expected to have Cray XT design characteristics, i.e. a processing node based on commodity microprocessor technology and a custom network interface. Therefore, we attempt to evaluate and analyse performance of a subset of test cases on the Cray X2 vector platform where both the processor and interconnect technology are proprietary and identify the factors that influence efficient code generation on XT and X2 platforms[1]. On the next-generation Cray systems an optimized communication API for PGAS languages is expected to be available, which could potentially be exploited in an optimal manner by the CCE compilers.

Our experiments with a set of micro-benchmarks and a stencil-filter operation that has been implemented in MPI, OpenMP, CAF and UPC reveal that there is a lack of information flow between different levels of the compiler and runtime system on the XT platform that prevents loop-level transformations for PGAS applications. On the X2 platform however, processor-level optimization such as vectorization are retained for serial and PGAS versions. To gain further insight into code generation and most importantly to explore code portability of PGAS code development using CCE compilers with other multi-platform compilers, we compare and contrast performance on multi-platform PGAS compilers such as the GCC Intrepid [5] and an alpha release of the Rice CAF compiler [7]. For small test cases with simple PGAS constructs, there is portability between CCE and other compilers while for complex structures, such as the team level synchronization operation in CAF and certain memory allocation operations, the users have to modify the code for different compilers. Hence, we argue for the availability of CCE PGAS compilers for other x86-based cluster platforms that would facilitate code development for applications by a community of developers and targeted for a wide range of x86 systems.

The layout for this paper is as follows: in section 2, we introduce our target systems and test cases. Results and performance analysis of the test cases are presented in section 3. In section 4 we discuss productivity aspects of the code development environment and related tools. Summary of our findings, conclusions and future plans are outlined in section 5.

2. Test Cases and Target Systems

In addition to micro-benchmarks such as the memory-bandwidth STREAM benchmark [13] and a two-dimensional matrix-multiply code that are included in CAF and UPC test suites [7][8], we implemented a two-dimensional stencil-filter benchmark in CAF, UPC, MPI and OpenMP. Many applications using the Swiss National Supercomputing Centre computing facilities operate on regular grids and perform a local filtering operation, in which a given field value is updated with a weighted average of neighbourhood values. This is commonly referred to as a *stencil* operation. In this benchmark, a

[1] On XT series platforms, the GASNet API, which is available for wide ranging systems, provides the PGAS communication interface for the CCE compilers [11]

point-centred square stencil with configurable size (with radius n=1, 2, 3 etc.) is applied to all points in a two-dimensional field i.e. every field value is updated by a weighted average of 9, 25, 49 etc. point values.

In the parallel implementation, the two-dimensional field is decomposed into a two-dimensional grid of rectangular blocks (see Figure 1). In the typical implementation the local "computational" domains are extended by a *halo region*, a padding of points whose values must be updated from domains of neighboring images in a *halo update*. For the filter update, the local domain can be broken down into three distinct subsets[2]:

1. The *exclusive* region "E" which is more than *N* layers from the computational region boundary. The update for this region requires only point values local to the image, and can thus be performed without communication.
2. The *computational* region "C" which encompasses the points which are computed on the grid by a given image. The region C-E (the 'non-exclusive' part of the computational domain) can only be filtered after the halo has been updated; it can be further broken up into eight sub-regions, conditional on the dependences of the eight neighboring images.
3. The *total* domain "T" consisting of all points allocated on the image. This will generally include a halo consisting of *N* layers beyond the computational domain boundary. The values at these points are kept consistent with the corresponding points in neighboring domains with the halo update. They are not filtered locally.

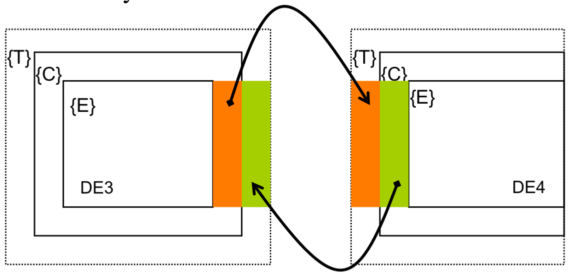


Figure 1: Domains for the stencil filter benchmarks

The fundamental filter algorithm is the following:

i. Initiate the non-blocking update of the halo regions
ii. Perform the filter on the exclusive domain on each image
iii. Complete the update of the halo regions
iv. Perform the filter on the remainder of the computational domain.

The explicit separation of the algorithm into a communication-only (halo exchange) and communication-free (filter) part was motivated by the experiences of [6], which illustrated that current CAF compilers have difficulty in aggregating communication for coarrays within loops, even for very simple constructs. On the other hand [15] illustrated that excellent CAF performance, in comparison to MPI, can be obtained for halo exchanges, if the underlying communication network offers a hardware-supported global address space, such as on the Cray X1 or X2. We consider two different synchronization concepts involving steps iii) and iv):

a. Wait until all images synchronize, indicating that all halo regions have been successfully updated, and then proceed to the filtering of C-E. This variant is well suited to architectures with an efficient global synchronization.
b. On a given image, synchronize with individual neighbors, and update the corresponding periphery sub-region as soon as the halo has consistent information. There could be an advantage to such local synchronization if global synchronization operations are expensive.

With these concepts in mind, we have implemented three MPI code variants:

1. MPI "Trivial": The total region corresponds to the computational region of the image's domain, but a halo-ed temporary array is used for the actual filter operation, using the four steps mentioned previously. Eight non-blocking sends and eight receives are posted in step (i), the exclusive region is filtered, and step (iii) consists of an MPI_WAITALL operation. After the C-E region is filtered in step (iv), the computational region of the temporary array is copied back to the image domain.
2. MPI "SendRecv": Again the image's total region corresponds to the computational region of the image's domain but in this case send-receive communication pairs are scheduled (e.g., the north-west / south-east images exchange halo information) and are executed with MPI_SendRecv. After the exclusive region is filtered, the MPI_SendRecv pairs are performed, and after each pair the corresponding portion of exclusive region is filtered. In this case the

[2] Terminology taken from the Earth System Modeling Framework [4]

communication is synchronous, so step (i) is not performed.

3. MPI "Halo": In this variant the local domain already contains the halo region (T-C), but is otherwise much like the "Trivial" version. The disadvantage of this version is that the halo region is visible in the filter interface, even if the caller does not have to perform the halo exchange in advance.

Similarly, there are three CAF code variants:

1. CAF "Trivial": Much like MPI "Trivial", the computation region is put into a coarray and immediately thereafter all images are synchronized, thus coalescing steps (i) and (iii).

```
V(1:m,1:n) = dom(1:m,1:n)  !internal region
V(1-halo:0,1:n)[EE,myQ] = dom(m-halo+1:m,1:n)
!to East
V(m+1:m+halo, 1:n)[WW,myQ] = dom(1:halo,1:n)
!to West
V(1:m,1-halo:0)[myP,NN] = dom(1:m,n-halo+1:n) !
to North
V(1:m,n+1:n+halo)[myP,SS] = dom(1:m,1:halo)  !
to South
V(1-halo:0,1-halo:0)[EE,NN] = dom(m-halo+1:m,n-
halo+1:n) !  to North-East
V(m+1:m+halo,1-halo:0)[WW,NN] = dom(1:halo,n-
halo+1:n) !  to North-West
V(1-halo:0,n+1:n+halo)[EE,SS] = dom(m-
halo+1:m,1:halo) !  to South-East
V(m+1:m+halo,n+1:n+halo)[WW,SS] =
dom(1:halo,1:halo) ! to South-West
sync all
```

Thereafter the entire computational region is filtered, thus coalescing steps (ii) and (iv):

2. CAF "Get": emulates a one-way communication "get" operation to perform the halo exchange, in essence the reverse of the communication in (1) above:

```
dom(1-halo:0, 1:n) = Z(m-halo+1:m,1:n)[WW,myQ]
!  from West
dom(m+1:m+halo, 1:n) = Z(1:halo,1:n)[EE,myQ]
!  from East
dom(1:m,1-halo:0) = Z(1:m,nhalo+1:n)[myP,SS]
!  from South
dom(1:m,n+1:n+halo) = Z(1:m,1:halo)[myP,NN]
!  from North
dom(1-halo:0,1-halo:0) = Z(m-halo+1:m,n-
halo+1:n)[WW,SS] !  from South-West
dom(m+1:m+halo,1-halo:0) = Z(1:halo,n-
halo+1:n)[EE,SS] !  from South-East
dom(1-halo:0,n+1:n+halo) = Z(m-
halo+1:m,1:halo)[WW,NN] !  from North-West
dom(m+1:m+halo,n+1:n+halo)  =
Z(1:halo,1:halo)[EE,NN] !  from North-East
```

The array *dom* is used subsequently on the right-hand side of the filter, in which the coarray *Z* is updated.

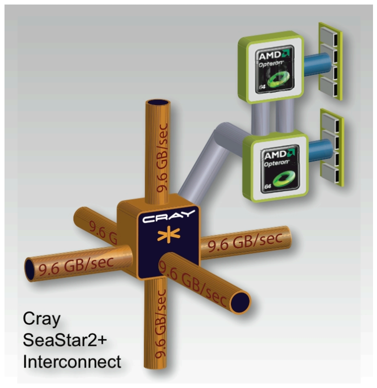
3. CAF "Put": pushes the data to the remote image's coarray, as done in 'Trivial', but synchronizes with an immediate neighbor before updating the portion of C-E which requires that neighbor's halo data. We thereby gain experience with local synchronization points, as opposed the global synchronization imposed by the *sync all* operation.

```
sync images( (myQ-1)*p+WW )    ! West
do j=1+halo,n-halo
  do i=1,halo
   sum = 0.
   do l=-halo,halo
      do k=-halo,halo
        sum = sum+stencil(k,l)*V(i+k,j+l)
      enddo
    enddo
   dom(i,j) = sum
  enddo
enddo
```

UPC and OpenMP variants of the stencil benchmark only implement the trivial filter. MPI, CAF and UPC experiments are run on both target systems, a Cray XT5 and a Cray X2 platform. OpenMP results are considered as outside the scope of this study since OpenMP cannot scale beyond a single node and therefore are not discussed here.

Our target Cray XT5 machine is located at CSCS while the Cray X2 access was kindly provided by the Edinburgh Parallel Computing Center (EPCC). Cray XT series and X2 systems can coexist as a single system but have distinctive processing, networking and programming characteristics (compute nodes are shown in Figure 2).

For instance, the Cray XT5 system is composed of 2.4 GHz dual-socket, hex-core AMD Opteron processors while the X2 processing node is a 4-way SMP node sharing a global address space (the CPU is a 1.4 GHz vector processors with 8 vector pipes). The memory bandwidth of the X2 system is considerably higher. On the XT5 system, the aggregate memory bandwidth is 25.6 GB/s while on the X2 system there are four 28.5 GB/s channels with two levels of cache. The XT5 processor has three cache levels and access to memory is not uniform unlike the X2 node. The network interface is also different on the two systems. The Cray XT5 system has a Cray proprietary SeaStarII network interface card and the nodes are connected in a three-dimensional mesh topology. The X2 system on the other hand has Cray proprietary YARC router chip connecting nodes in a fat tree topology.



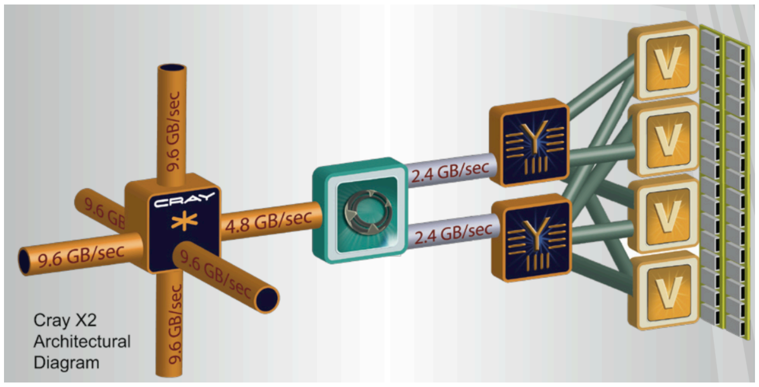


Figure 2: Cray XT5 node (left). Cray X2 node (right)

The programming environments also have distinct features, for example, the Cray XT5 system runs Cray Linux Environment (CLE), while the X2 system has a different variant of Linux. Although the Cray Compiler Environment (CCE) is used on both platforms, on the XT5 system version 7.2.x was available while on the X2 system version 6.0.0.x of C and Fortran compilers were available.

3. Performance Evaluation and Analysis of Generated Code

The 2D stencil-filter CAF and UPC implementations were run on both the Cray XT5 and X2 using a 3x3 stencil and various domain sizes. On the Cray XT5, the code was compiled with "-O3 -h cpu=istanbul" flags. All algorithmic variants executed and passed consistency checks. On the X2, all MPI variants and the CAF "Trivial" variant ran successfully, but CAF variants "Put" and "Get" hang.

The results on 72 cores (6 nodes) of the XT5 are shown in Figure 3 and the X2 results are shown in Figure 4. The local domain on each core was a square with edge size varying from 64 to 8192. The MPI 'Trivial' version, in which only the intermediate buffers contain the halo, performs the poorest for larger problem sizes. The "SendRecv" variant, which implements pair-wise communication, is somewhat better than "trivial" for larger domain sizes. The best performing MPI variant is always the "halo" version, in which the domain contains the halo region from the onset. Up to a local edge size 2048, all MPI variants perform better than any of the CAF variants. However, at edge size 4096, the CAF "Trivial" version starts to outperform MPI "Trivial", after 5000, it outperforms also MPI "SendRecv", and at 7000 it seems to be approaching the MPI "Halo" performance. Unfortunately, larger problem sizes would not run on the XT5. The CAF "Get" and "Put" variants consistently performed less well than the "Trivial" version.

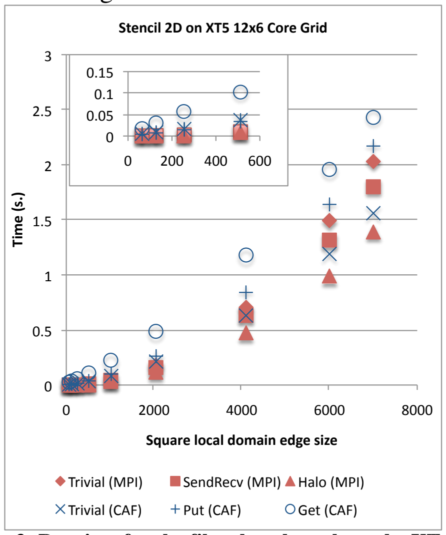


Figure 3: Runtime for the filter benchmark on the XT platform

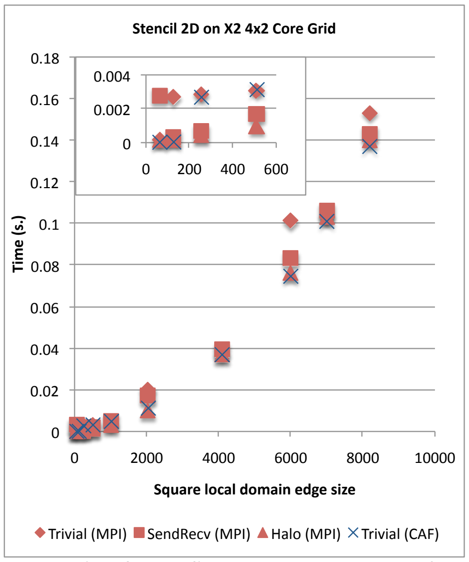


Figure 4: Runtime for the filter benchmark on the X2 platform

	Copy	**Scale**	**Add**	**Triad**
Local array (a, b, and c are regular Fortran arrays)	c(1:n)=a(1:n)	b(1:n) = scalar*c(1:n)	c(1:n) = a(1:n) + b(1:n)	a(1:n) = b(1:n) + scalar*c(1:n)
	Pattern-matched	Vectorized and unrolled x4	Vectorized and unrolled x4	Vectorized and unrolled x4
	8524.85 MB/s	8450.93 MB/s	8792.65 MB/s	8716.84 MB/s
Local coarray (a, b, and c are coarrays)	c(1:n)=a(1:n) b(1:n) = scalar*c(1:n)	b(1:n) = scalar*c(1:n)	c(1:n) = a(1:n) + b(1:n)	a(1:n) = b(1:n) + scalar*c(1:n)
	Pattern-matched	Unrolled x8	Unrolled x8	Unrolled x8
	8390.11 MB/s	5766.99 MB/s	6225.04 MB/s	6191.07 MB/s
Remote coarray, same node	c(1:n)=a(1:n)[2]	b(1:n) = scalar*c(1:n)[2]	c(1:n) = a(1:n)[2] + b(1:n)[2]	a(1:n) = b(1:n)[2] + scalar*c(1:n)[2]
	Pattern-matched	Unrolled x8	Unrolled x8	Unrolled x8
	3372.67 MB/s	1.42 MB/s	1.50 MB/s	1.50 MB/s
Remote coarray, different node	c(1:n)=a(1:n)[2]	b(1:n) = scalar*c(1:n)[2]	c(1:n) = a(1:n)[2] + b(1:n)[2]	a(1:n) = b(1:n)[2] + scalar*c(1:n)[2]
	Pattern-matched	Unrolled x8	Unrolled x8	Unrolled x8
	2673.65 MB/s	5.10 MB/s	4.03 MB/s	4.03 MB/s

Table 1: Language construct, optimization applied, and operation throughput for the 4 basic operations used in the STREAM benchmark, over 4 different access patterns

In view of the fact that GASNet calls are being invoked for the communication, the positive CAF results were a relative surprise. When the code is run on the X2, with hardware support for the global address space, the CAF "Trivial" version performs as well as the MPI "halo" version for larger problem sizes, and 5-20% better than the MPI "Trivial" version.

To investigate the performance characteristics of the compiler-generated code and impact of remote communication operations, we select the CAF version of the STREAM benchmark to understand if and how remote memory accesses are scheduled, overlapped and synchronized by the compiler and the runtime systems. The STREAM benchmark gives the data throughput for a set of 4 basic operations on large coarrays (larger than cache size), on the same image and across different images. Table 1 offers some results collected for these operations against 4 different access patterns: local array (as a baseline measure of the operation cost), local coarray (as a baseline measure of the PGAS construct), remote coarray in the same node, and remote coarray in a different node. For each case, three pieces of information are displayed: (i) the language construct, (ii) the optimization applied by the compiler, and (iii) the operation throughput in MB/s. All tests were performed on the Cray XT5 available at CSCS.

In every case, the copy operation is pattern-matched, and replaced by an optimized library call, offering good performance overall. We could take the performance degradation that occurs when accessing another image (around 60% for a coarray in the same node and nearly 70% if it is in a different node) as the baseline cost of a remote versus a local access.

More interesting though is the disabling of vectorization for any construct that contains a coarray, even if it is only referenced locally (i.e. no brackets in the expression). This causes unnecessary performance degradation (~30%) for the local access with respect to the vectorized version. In the absence of vectorization, the compiler opts for loop unrolling for all operations that do not match a predefined pattern. Even so, the performance of scale, add or triad operation is about 1000 times slower when accessing a different image, which goes well beyond the baseline degradation exhibited by the library calls used for the copy operation.

We checked the code generated for these operations for potential problems. We were able to identify the networking API calls performed by the runtime, which are based on the GASNet conduit for Portals [10][12]. These include 2 ways of retrieving data: (i) *blocking gets*, and (ii) *non-blocking gets*, which are later synched. Even when unrolling the loop the 'gets' are issued either in a blocking fashion or synched early on. This means that the possibility of hiding remote access latency, by issuing non-blocking gets as early as possible and then synching to them as late as possible, is neglected. To get some insight into the effect of this statement reordering, we conducted a simple test. An alternate version of the benchmark was created, where the loop for the scaling operation was manually unrolled and the operations ordered so that all data needed in the unrolled body is fetched at the beginning, in a non-blocking fashion. Then, data is synchronized for completion of the data transfers.

upc_forall iteration statement using an integer as affinity expression	**upc_forall iteration statement using a shared pointer as affinity expression**

shared int arr[N];
[...]
upc_forall(i=0; i<N ;i++; i)
   arr[i] = get_value(i);
	shared int arr[N];
	[...]
	upc_forall(i=0; i<N ;i++; &arr[i])
	   arr[i] = get_value(i);

Table 2: Two equivalent alternatives to distribute the iterations for array arr between threads according to data locality

This offered a two-folds improvement over the automatically unrolled version, which shows the potential for hiding network latency by reordering the definitions and uses of shared variables, even for such a simple case. This test could not be performed for the other operations though, as the compiler would synch for the non-blocking gets even with the statements reordered.

Whatever the case, it is clear that the significant performance degradation observed for the operations that could not be replaced by an optimized library call cannot be solely attributed to the inability to hide network latency by reordering statements. It is unclear what the underlying reason is for such a gap in performance, be it runtime overheads, networking shortcomings, or some other issue.

On a secondary note, it can be observed that an improved throughput occurs for access to an image in a different node, than an image in the same node, between 2.5 and 3.5 times faster. At this stage, the reason for this counter-intuitive result is unclear. Since the compiler's inability to retain the microprocessor-level optimization, such as SSE vector, is a serious limitation, we ran similar set of experiments on the Cray X2 vector platform to find if these limitations exist on a vector system. Potentially, benefits of PGAS communication optimization could be offset by un-optimized code generated for the microprocessor. The X2 CAF compiler however retains the vector instruction optimization as shown below and the runtime data presented in Table 3 also confirms our findings.

```
On X2 system
791. 1  Vr------<      DO  j = 1,n
792. 1  Vr                b(j) = scalar*c(j)[2]
793. 1  Vr------>      end DO
```

	Single image	Two images
Copy	81.25	37.57
Scale	85.63	37.48
Add	57.54	34.95
Triad	60.37	34.95

Table 3: CAF STREAM results (GB/s) on X2

We therefore conclude that there is a limitation in the CCE x86 CAF compiler or some compiler dependency checks that fail on an x86 system preventing the compiler from vectorizing or retaining microprocessor-level optimization. If unresolved, this issue is likely to severely limit performance on a GEMINI based system where an x86 commodity multi-core processor will constitute a processing node.

For UPC, we considered a synthetic micro-benchmark to analyze the compiler's ability to apply optimizations automatically. For that we tested some variants of the *upc_forall* construct over statically defined shared arrays. A upc_forall loop is an iteration statement where the iteration space is subdivided between the threads running the program [9]. For this purpose, besides the typical initialization, condition, and update terms of a C for loop, there is a fourth term, known as the affinity expression, which defines which thread should execute which iteration. The affinity expression can be chosen so that each thread operates on the maximum amount of local shared data and the minimum amount of remote shared data, to deliver performance. This expression can be either an integer, which defines the loop indices to execute for each thread through a simple module operation, or it can be a shared pointer, in which case the iteration is performed by the thread whose pointed data is local. These two alternatives are shown for a simple array in Table 2.

Analyzing the generated code, we realize that, just as happened with the Fortran compiler, the loop cannot be vectorized even when accessing only local data. This causes the unoptimized version to perform ~60 times slower than a vectorized version operating on a private array of the same size. In addition, loop unrolling can only be performed for upc_forall loops with an integer affinity expression. In the case of a shared pointer affinity expression, the compiler is confused by a runtime function call to translate the pointer, which disables loop optimizations altogether, resulting in the unrolled version running twice as fast as the un-optimized version. Even when using different blocking factors, including the default round-robin shared array distribution, the compiler is still unable to recognize the contiguous chunk in a thread's shared space, to apply some sort of optimization.

Original matrix multiply	Alternative matrix multiply

`shared [N*P/THREADS] int a[N][P],c[N][M];`
`shared [M/THREADS] int b[P][M];`
`[…]`
`upc_forall (i=0; i<N; i++; &c[i][0]) {`
`    for (j=0; j<M; j++) {`
`        c[i][j]=0;`
`        for (l=0; l<P; l++)`
`            c[i][j]+=a[i][l]*b[l][j];`
`    }`
`}`
	`shared [N*P/THREADS] int a[N][P],c[N][M];`
	`shared [M/THREADS] int b[P][M];`
	`[…]`
	`for(j=0;j<M;j++){`
	`    for(l=0;l<P;l++){`
	`        b_val = b[l][j];`
	`        upc_forall(i=0;i<N;i++;&c[i][0])`
	`            c[i][j]+=a[i][l]*b_val;`
	`    }`
	`}`

Table 4: Two versions of a dense matrix multiply in UPC. The b_val variable must be manually hoisted to benefit from reduced communication, as the compiler is unable to perform this operation automatically

Since we observed different optimization patterns on the X2 platform as compared to the XT platform for the CAF code, we experimented using the canonical UPC matrix-multiply example on the X2 platforms with the UPC compiler. The compiler listings for the UPC code are shown below and correspond to our findings for the CCE CAF compiler. On the X2 platform, the compiler retains the vectorization and unrolling optimization while on the XT5 system only the loops were interchanged. Note that the serial version of the matrix-multiply code is vectorized by the Cray C compiler on the XT5 platform.

```
On X2
1------<   upc_forall (i=0; i<N; i++; &c[i][0])
{
1 V----<     for (j=0; j<M; j++) {
1 V            c[i][j]=0;
1 V r--<       for (l=0; l<P; l++)
1 V r-->         c[i][j]+=a[i][l]*b[l][j];
1 V---->     }

On XT5
1------<  upc_forall (i=0; i<N; i++; &c[i][0]) {
1 i----<    for (j=0; j<M; j++) {
1 i            c[i][j]=0;
1 i 3--<       for (l=0; l<P; l++)
1 i 3-->          c[i][j]+=a[i][l]*b[l][j];
1 i---->       }
1------>    }
```

To sum up, on the XT5 platform, the C compiler suffers the same limitations for UPC that were observed for the Fortran compiler with coarrays (including failure to adequately schedule network operations), plus new limitations related to unrecognized UPC runtime calls that disable loop level optimization completely.

To further test the performance with a more stringent communication pattern, we tested *MatrixMult*, a straightforward implementation of a typical dense matrix product, AxB=C. Matrixes are represented as statically defined shared arrays, and blocking factors are chosen so that each thread accesses only the local portion of matrix A and C, but all of the B matrix. The data decomposition is depicted in Figure 5.

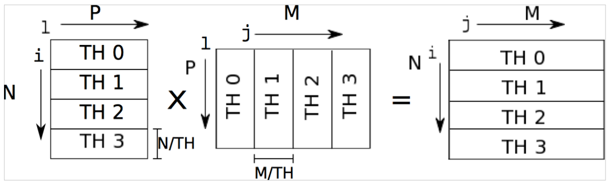


Figure 5: Data decomposition of the matrix multiplication in UPC

We have implemented 2 variations of this product, detailed in Table 4. The original version loops first on index *i* (upc_forall), then *j*, then *l*. In that case, each thread must access (M-M/TH)*P*N/TH remote elements, i.e., the whole of the B matrix that is not local, N/TH times. The alternative version loops first on the *j*, then on *l*, finally on *i* (upc_forall). This distributes the iterations of the inner loop rather than those from the outer loop, increasing the loop overhead but reducing the remote accesses per thread by N/TH, i.e., (M-M/TH)*P.

Figure 6 presents the speedup collected for both the original and the alternative version. The original version presents consistent slowdowns, whereas the alternative version scales to about half the linear speedup for a sufficiently large matrix (N=1000).

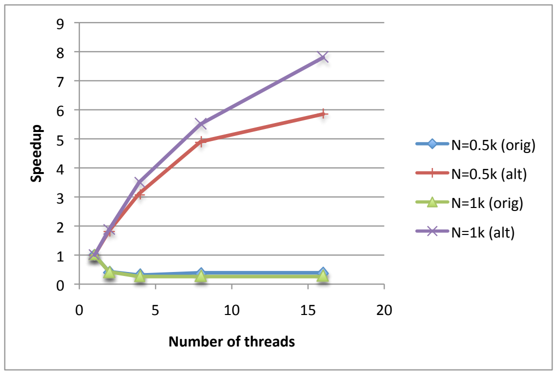


Figure 6: Speedups, with respect to the 1-threaded version, measured for the original and alternative matrix multiplication code in UPC, with square matrixes of 500x500 and 1000x1000

Two observations can be derived from this experiment. First, given the large increment in accessing remote data in another node, it is advisable to check the algorithm to communicate as less as possible, even incurring the extra runtime overheads. This situation can only be expected to dominate as we extend to many nodes and heterogeneous systems with vastly varying memory access profiles. Secondly, the performance of PGAS codes is still clearly hindered by issues that encompass all levels of the programming stack, from the compiler, to the runtime, to the network API (GASNet over Portals), and the SeaStar NIC. Improvements must be made along all these levels to make PGAS languages a competitive option for large systems. It should be noted that a matrix multiply example is as simple as it gets in the UPC world, and it is typically used as a motivating example for advertising the benefits of UPC as a productive language.

One could argue that writing PGAS applications in this way would be similar to writing MPI applications where a code developer is aware of the communication patterns and make every possible effort to reduce it. However, since we notice several differences using X2 compilers, especially for microprocessor optimization and a significantly higher performance for the PGAS operations, we run optimized versions of the matrix-multiply and stencil-filter benchmarks implemented in UPC on the two platforms. The results for the matrix multiply operation show slowdown for the second version of the matrix-multiply code since the internal upc_forall loop prevents vectorization of the outer loop. In the case of the stencil-filter benchmark, there is no significant slowdown as the pointer locality is improved on the X2 platform but no significant gains either. On the XT platform, performance of the alternate version increases linearly with problem size and number of threads. Hence, we conclude that in order to achieve and retain performance for PGAS applications it is imperative to have a compiler that retains microprocessor level optimization along with a supporting network hardware and runtime infrastructure.

4. Productivity Analysis

Our goal for this study is not to define a single metric for productivity and then attempt to measure it but to evaluate the useability of the PGAS development and execution environments and tools (debugging and performance) that enable code developers to port existing applications, to gradually introduce PGAS constructs in their existing parallel applications and to develop code from scratch where applicable. Unfortunately, at the Swiss National Supercomputing Center, there are no production level PGAS applications, therefore we relied on benchmarks available from PGAS compiler development teams and developed some of our test codes from scratch as described in the earlier sections. Likewise, we developed some synthetic test cases to analyse productivity of a code when an application team adopts an incremental approach to PGAS programming, i.e. to introduce PGAS constructs in the existing MPI programs. We define this as the hybrid or mix-mode programming.

The hybrid programming model of intermixing MPI for inter-node communication with a secondary programming model for intra-node computation (e.g. OpenMP) has provided a promising approach for exploiting ever-increasing system sizes with large distributed compute nodes containing many-core processors which share a global address space. It is pertinent to investigate the extent to which these new approaches can be made interoperable with the message-passing paradigm (MPI). We briefly investigate the ability of the CCE UPC/CAF compiler to support the intermixing of MPI with CAF/UPC. We describe some of the pitfalls and show the results of some initial experiments with this hybrid programming model.

The feasibility of interoperating MPI with UPC/CAF is determined by the compatibility of the respective communication layers required by each programming model. On Cray XT systems, UPC and CAF are implemented via the GASNet communication library, which in turn is implemented on top of the native Portals communication layer and not MPI (thereby improving performance and minimizing conflict with shared network resources and MPI data structures). Previous implementations of GASNet on Cray systems used the Portals-conduit for GET and PUT operations and the MPI-conduit for Active Messages. Under certain conditions (see Table 5) GASNet and MPI are designed to be compatible and therefore can be used together within the same network and the same application.

Condition	Remark
Initialization	The correct order of initialization for GASNet and MPI must be preserved
Interleaving	Interleaving of MPI and GASNet Blocking calls is prohibited as this can lead to deadlock

Table 5: Requirements for successful interoperation between MPI and GASNet

GASNet conduits may or may not initialize MPI internally, so a recommended strategy is to test for MPI initialization at the outset of the code, and therefore ensure correct startup behaviour when porting codes to different GASNet conduits[3].

.

[3] On Cray XT systems, GASNet automatically calls MPI_INIT() and manages the MPI initialization process.

```
integer,allocatable :: A(:)[:]

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,error)
call MPI_COMM_SIZE(MPI_COMM_WORLD,PEs,error)

if (rank.ne.0) then
  allocate(A(10)[*])
end if

call MPI_FINALIZE(error)
```

```
integer,allocatable :: A(:)[:]

call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,error)
call MPI_COMM_SIZE(MPI_COMM_WORLD,PEs,error)

if (rank.eq.0) then
  print *,"Deadlock"
else
  sync all
end if

call MPI_FINALIZE(error)
```

Table 6: Trivial Examples of Deadlock in Mixed CAF/MPI codes

The interleaving of MPI and CAF/UPC codes must be managed carefully to avoid deadlock situations. In particular, mixing of MPI and UPC/CAF blocking calls e.g. shared UPC accesses or CAF synchronisations, can create situations were the servicing of requests from remote nodes is prevented. In particular, the CCE CAF compiler has the option of creating blocking or non-blocking communication calls depending on the surrounding code within the execution segment (i.e. code delimited with a *SYNC MEMORY* instruction). When the SYNC MEMORY instruction is reached, all outstanding non-blocking transfers have to complete and under certain circumstances, where processes are in various states of MPI or UPC/CAF blocking, this is not achievable

Table 6 demonstrates two examples of mixed-mode CAF and MPI codes that can lead to trivial deadlock situations. Both SYNC ALL and ALLOCATE (as well as DEALLOCATE) of co-arrays are necessarily blocking among all CAF images. In both cases the master MPI process prevents the CAF statements from completing, as it does not partake in the synchronization. To ensure that deadlock is prevented, it is recommended that mixed codes utilize UPC or CAF in phases with MPI code i.e. all CAF (or UPC) statements are executed within a CAF (or UPC) phase, which is terminated with a CAF (or UPC) barrier. The following phase includes the MPI statements which are likewise terminated with a MPI_BARRIER()[4].

A further consideration when contemplating mixing MPI and CAF/UPC codes on Cray XT systems, is that MPI processes, CAF images and UPC threads are all mapped to hardware in the same way i.e. each MPI rank, CAF image and UPC thread is represented as a process to the OS. MPI ranks and UPC threads are zero-based while CAF images are (rank+1) based. In this scheme it is currently impossible therefore to have different number of MPI processes and CAF images (e.g. request 1 MPI process for inter-node communication and *N* CAF images for on-node computation on a node with N cores). Therefore, two approaches can be undertaken to divide work amongst MPI and CAF/UPC processes:

1) With the same number of CAF images and MPI ranks (e.g. 12 each for a 12-core Istanbul compute node) 1/12th of the MPI ranks (one per node) can be employed to perform inter-node communication, while all CAF images can be used within a compute node for computation. It is envisaged though that this scheme will introduce unnecessary redundant MPI resources as such a strategy is scaled to large system sizes (> 100,00 cores)
2) Declare coarrays with 2D codimensions where the leading dimension is equal to the number of cores per node e.g. A[12,*] for a 12-core compute node. In this strategy the corresponding column in the coarray can be used to perform on-node computations and intra-node communication can be performed using CAF assignments. With the implementation of CAF *teams*, a team can be created for each node with high-speed SYNC TEAMS synchronizations between nodes. Currently, teams are not implemented within the CCE compiler but SYNC IMAGES can be used instead albeit with reduced performance.

Besides these potential issues, overall the code development experience with the CCE PGAS compiler is seamless (only an additional compiler flag needed). However porting code with other multi-platform compilers can lead to some code modifications. Particularly, our experience with the Rice CAF compiler (alpha version) required several modifications to the synchronization operations. Debugging tools such as TotalView can launch CAF and UPC jobs but do not provide any useful information about the language structures. Likewise the CrayPAT compiler could report some load balance information but programming paradigm specific information is lacking. In short, it is

[4] It should be noted that MPI_BARRIER and specific CAF or UPC barriers do not process the same types of outstanding communication requests. Therefore they cannot be used interchangeably.

entirely up to the code developer's ingenuity and effort to debug and tune these PGAS applications.

5. Conclusions and Future Directions

We have identified several shortcomings as well opportunities for targeting PGAS compilers for production level application development on the upcoming GEMINI based systems. First, we need to explore options for retaining microprocessor, loop-level optimization for the PGAS code particularly for compilers for commodity x86 platforms. This may involve investigation into both compiler and runtime systems and how the memory dependency information is exchanged between different layers of software stack. Second, the dynamic or runtime information, for example, mapping of CAF images or UPC threads, need to be fed back in such a way that a local memory copy operation could be performed instead of a remote one when applicable. Thirdly, for aiding code development for distributed application teams targeting multiple platforms (a large number of applications that are running on the Swiss National Supercomputing Center XT5 platform) the Cray compiler environment should be made available for x86 based cluster systems. It may only be a functional compiler but it will allow code developers to build codes on systems other than their target Cray platform. Finally, research and development programming paradigm-aware tools are lacking and could inhibit tuning and optimization efforts on the GEMINI based systems.

Acknowledgments

The authors would like to thank Dr Jason Beech-Brandt from the Cray Centre of Excellence for HECToR in the UK for providing access to the X2 nodes of the system. We also appreciate the feedback from Bill Long, Cray for advice on the CAF development of the stencil application.

References

[1] Cray Compiler Environment and Cray Performance Tools (CrayPAT) available at (http://docs.cray.com)
[2] Cray XT series computers (http://www.cray.com/Products/XT/Systems/)
[3] Cray XT5h System Overview (X2 processors) available at (http://docs.cray.com)
[4] Earth System Modeling Framework (ESMF), www.earthsystemmodeling.org
[5] Intrepid UPC compiler (http://www.intrepid.com/)
[6] PRACE Technical Report on the Evaluation of Promising Architectures for Future multi-Petaflop/s Systems, www.prace-project.eu/documents/d8-3-2.pdf
[7] Rice Co-Array Fortran 2.0 compiler, http://caf.rice.edu/
[8] UPC Benchmarking suite, George Washington University, http://upc.gwu.edu/download.html
[9] UPC Language Specifications, v1.2. Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005.
[10] R. Brightwell, B. Lawry, A. B. Maccabe, and R. Riesen. "Portals 3.0: Protocol building blocks for low overhead communication," IPDPS, 2002.
[11] D. Bonachea. GASNet Specification, v1.1 U.C. Berkeley Tech Report CSD-02-1207.
[12] D. Bonachea, P. H. Hargrove, M. Welcome, and K. Yelick, "Porting GASNet to Portals: Partitioned Global Address Space (PGAS) Language Support for the Cray XT," CUG 2009.
[13] J. McCalpin , "STREAM: Sustainable Memory Bandwidth in High Performance Computers," 2007, http://www.cs.virginia.edu/stream/.
[14] R.W. Numrich and J.K. Reid, "Co-Array Fortran for parallel programming," Fortran Forum, volume 17, no 2, 1998.
[15] P. H. Worley and J. Levesque, "The Performance Evolution of the Parallel Ocean Program on the Cray X1," CUG, 2004.

About the Authors

Sadaf Alam is a member of the Scientific Computing research group at the Swiss National Supercomputing Centre. She can be reached at Via Cantonale 2, 6928 Manno, Switzerland, Email: alam@cscs.ch.

William Sawyer is a member of the Scientific Computing research group at the Swiss National Supercomputing Centre. He can be reached at Via Cantonale 2, 6928 Manno, Switzerland, Email: wsawyer@cscs.ch.

Tim Stitt is a member of the National Supercomputing Services group at the Swiss National Supercomputing Centre. He can be reached at Via Cantonale 2, 6928 Manno, Switzerland, Email: stitt@cscs.ch.

Neil Stringfellow leads the National Supercomputing Services group at the Swiss National Supercomputing Centre. He can be reached at Via Cantonale 2, 6928 Manno, Switzerland, Email: nstring@cscs.ch.

Adrian Tineo received PhD in computer science from the University of Malaga, Spain. He is a member of the Scientific Computing Research group at CSCS. His research interests lie in parallel computing and optimizing compilers. He can be reached via email through atineo@cscs.ch.