Performance of Fortran Coarrays on the Cray XE6

David Henty
EPCC, The University of Edinburgh
The King's Buildings, Mayfield Road
Edinburgh, UK
d.henty@epcc.ed.ac.uk

***Abstract*—Coarrays are a feature of the Fortran 2008 standard that enable parallelism using a small number of additional language elements. The execution model is that of a Partitioned Global Address Space (PGAS) language. The Cray XE architecture is particularly interesting for studying PGAS languages: it scales to very large numbers of processors; the underlying GEMINI interconnect is ideally suited to the PGAS model of direct remote memory access; the Cray compilers support PGAS natively. In this paper we present a detailed analysis of the performance of key coarray operations on XE systems including the UK national supercomputer HECToR, a 90,000-core Cray XE6 operated by EPCC at the University of Edinburgh. The results include a wide range of communications patterns and synchronisation methods relevant to real applications. Where appropriate, these are compared to the equivalent operation implemented using MPI.**

***Keywords*-Fortran; parallel programming; benchmark.**

I. Introduction

Partitioned Global Address Space (PGAS) languages such as Unified Parallel C [1] have been the subject of much attention in recent years, in particular due to the exascale challenge. There is a widespread belief that existing message-passing approaches such as MPI will not scale to this level due to issues such as memory consumption and synchronisation overheads. PGAS approaches offer a potential solution as they provide direct access to remote memory. This reduces the need for temporary memory buffers, and may allow for reduced synchronisation and hence improved message latencies. Some modern distributed memory architectures allow for Remote Memory Access (RMA) directly over the interconnect, meaning the PGAS model maps directly onto the underlying hardware. PGAS features have been introduced into the Fortran 2008 standard with coarrays [2]. Programming using coarrays has many potential advantages compared to MPI. Amongst these are simplicity, compiler checking and scope for automatic optimisation of communications by the compiler. Coarrays can also be introduced incrementally to existing MPI codes to improve performance-critical kernels.

Coarrays have a long history on Cray systems, with implementations dating back to 1998 on the Cray T3E. The original coarray extensions proposed by Bob Numrich and John Reid [3] were implemented as an option in Cray Fortran 90 release 3.1. The direct RMA capabilities of the T3E torus interconnect enabled this PGAS model to be implemented easily and efficiently. The first generation of more modern Cray systems, the XT architecture, used the Seastar interconnect which had an underlying message-passing (i.e. two-sided) data-transfer model. As a result, PGAS languages had to be implemented on top of some software layer such as GASNET [4] to emulate true RMA capabilities. However, the most recent Cray XE systems use the GEMINI interconnect which once again offers native RMA capabilities. Coupled with the increasing maturity of coarray support in Cray's Fortran compiler, this makes it an interesting time to evaluate the performance of coarrays on machines such as the Cray XE6.

II. Fortran Coarrays

The fundamental extension is a new declaration syntax enabling variables (scalars or arrays) to be replicated across multiple *images* executing in an SPMD fashion, e.g.

```
real, dimension(10), codimension[*] :: x
```

declares that a one-dimensional array x with 10 elements exists on every parallel image. The distribution across images is achieved by having an additional *codimension*, indicated by square brackets, which spans the images; its size is set automatically at runtime. Remote data is accessed simply by indexing within the codimension, e.g.

```
x(2) = x(3)[7] ! get value from image 7
x(6)[4] = x(1) ! set value on image 4
```

If the codimension is omitted then access refers to the local copy, meaning there need be no overhead for local operations on coarrays. Multiple codimensions are also supported.

A number of image synchronisation mechanisms are available, e.g. to enable the programmer to ensure that remote writes have completed at the required point of execution. These include global synchronisation, mutual synchronisation with one or more remote images, critical sections and locks. In general, Fortran does not perform any automatic synchronisation: although this maximises performance, it very much places the onus for ensuring correctness on the programmer.

A. Fortran Coarray Benchmark

It is important that any benchmark measures characteristics of performance that are relevant to a wide range of applications. Given that Fortran coarrays are relatively new to the standard, it seems rather early to port full applications codes, although a number of performance studies have already been done on kernels such as the NAS Parallel Benchmarks [5]. We prefer to focus on a small number of low-level operations and measure their performance in isolation. In all cases the basic data type is a double precision floating-point value.

The benchmark design is described in [6], which also presents initial results from a range of Cray systems (X2, XT and XE). That paper also has some comparisons with results on an Intel Infiniband cluster, obtained using the coarray support recently available in the Intel ifort compiler.

III. Cray XE6 Results

Although some XE6 performance results were presented in [6], they were mainly used to illustrate differences in performance between different compilers (Cray and Intel) and between architectures with and without native hardware support (e.g XT and XE). Not surprisingly, performance on the XE systems was much better than for the XT, particularly in terms of reduced latency and synchronisation overheads. Here we present new data, obtained on more recent XE systems with Interlagos processors, and a much more detailed analysis of performance. It is also interesting to investigate if and how the Cray compiler has improved in terms of its coarray support.

Results were obtained on two machines. The first is HECToR, the 90,112-core UK National Supercomputer operated by EPCC at the University of Edinburgh on behalf of EPSRC. HECToR comprises 2816 compute notes, each containing two 16-core AMD Opteron Interlagos 2.3 GHz processors, connected by the GEMINI interconnect. We also had access to an internal Cray development system which is the same basic architecture but with 2.1 GHz CPUs. The difference in clock speed had no noticeable effect on the coarray communications performance. We used version 8.0 of the Cray Fortran compiler; the internal Cray systems were running pre-release software.

A. Point-to-point transfer

The simplest data transfer is the equivalent of an MPI ping-pong, i.e. data is repeatedly transferred between a pair of images. In MPI this is achieved using a pair of send/recv calls. Using coarrays an image can either write data remotely to the other image (using a *put*), or it can read data directly from the other image (using a *get*). Equally important, however, is the choice of synchronisation mechanism. Using coarrays, only one image actively participates in the data transfer (the sender for put; the receiver for get) meaning that synchronisation calls must be inserted explicitly by the user to ensure correctness. For a ping-pong with remote writes this means mutual synchronisation after every put so that the target image knows when the data has arrived and can then proceed to return the data with its own put call. Similarly, for remote reads mutual synchronisation is also required after each get so that the source image knows when its data has been read, meaning it can then proceed to read the data back from the target with its own get call. If these synchronisation calls are omitted then the pattern will not be a ping-pong: the images will not wait for each other so will be performing a simultaneous ping-ping. More importantly, we will have an incorrect code and will almost certainly be transferring the wrong data.

Figure 1 shows the time taken for small messages using coarrays, and using `MPI_Send` for comparison. Synchronisation is either global (`sync all`) or point-to-point (`sync images`). The data is for put, although the timings for get are almost identical. All MPI timings presented here come from the standard Intel MPI Benchmark [7], although we have checked that simply inserting MPI calls directly into our coarray benchmark framework gives equivalent results.

To ensure that the GEMINI network is used rather than shared-memory copies, the tests are done using two images (or MPI processes) placed on different compute nodes of the XE6. For coarrays, this is not necessarily the same test as having fully populated nodes where all but one of the images on each node simply does not participate. For global synchronisation (using `sync all`) the non-participating nodes will still have to make the collective synchronisation call do avoid deadlock, increasing the overall time taken. It should be noted that the results in Figure 1 are a best-case scenario: not only are the global synchronisation overheads minimised by having a single process per node, that process also runs on core 0 which has direct access to the GEMINI network without having to communicate over any additional internal hyper-transport links.

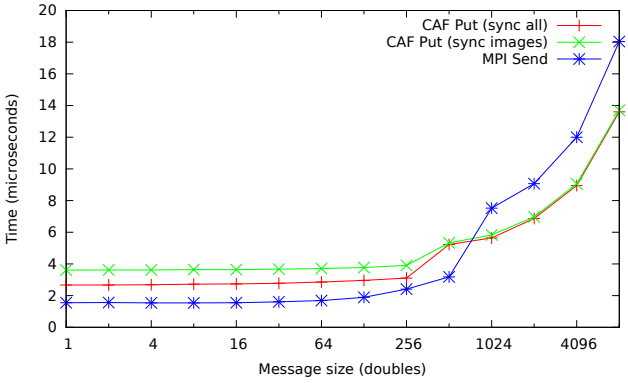

Figure 1. Latency of point-to-point data transfer using coarrays (with `sync all` and `sync images`) and `MPI_Send`

The results show that the MPI ping-pong is faster than

using coarrays for fewer than 1024 doubles. For these small messages, the coarray time is dominated by the synchronisation call, with point-to-point synchronisation about a microsecond more expensive than global synchronisation. Above 1024 doubles the superior bandwidth of coarray transfers leads to coarrays outperforming MPI. Interestingly, the point-to-point and global synchronisation overheads also appear to be the same in this regime. To fully understand the performance of coarrays it would be necessary to measure the synchronisation overheads in isolation, which is attempted in Section III-B.

The ping-pong results are summarised in Table I where we also include the coarary results for get and the asymptotic bandwidths for completeness. Although the latency for coarray transfers is substantially larger than for MPI, it should be noted that in real coarray application we would not normally synchronise after each single-word transfer. We would expect that multiple small transfers would be enclosed by a single pair of image synchronisation calls, thus reducing the effective latency per transfer. This is different from MPI where synchronisation occurs on a per-message rather than a per-image (or per-process) basis.

Mode	latency (μs)	bandwidth (GB/s)
MPI Send	1.6	6.1
Put (sync images)	3.6	6.8
Put (sync all)	2.7	6.8
Get (sync images)	3.5	6.8
Get (sync all)	2.6	6.8

Table I
PING-PONG STATISTICS FOR MPI AND COARRAYS

B. Basic synchronisation overheads

The benchmark measures a variety of patterns of synchronisation which are discussed in detail in Section III-E. However, to attempt to understand the ping-pong results in Section III-A the basic timings for global and pairwise synchronisation are required. We plot the time taken for a `sync all` in Figure 2, alongside `MPI_Barrier` for comparison. The results are encouraging as they show that the coarray synchronisation out-performs MPI, whereas we previously observed the opposite [6].

We also measure the global and point-to-point synchronisation overheads for two images, each on separate nodes, as 2.3 μs and 3.6 μs respectively. Although these numbers appear reasonable in isolation, they do not make sense when compared to Table I. If we subtract these independently measured synchronisation overheads from the overall ping-pong latencies it does not give consistent results: it would appear that the point-to-point synchronisation overhead of 3.6 μs is an overestimate.

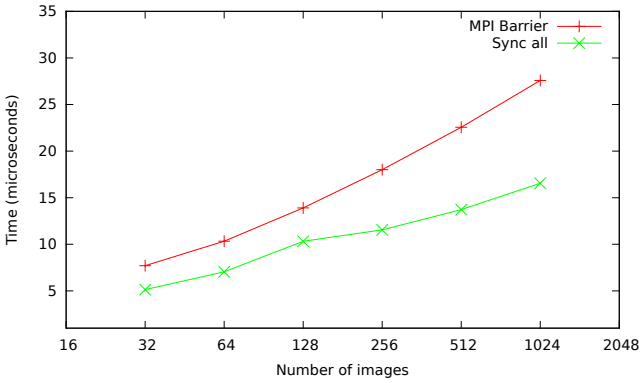

Figure 2. Time taken for global synchronisation

C. Aggregate Bandwidth

The ping-pong tests, although useful in understanding the fundamental characteristics of the interconnect, are not particularly relevant to the communications patterns used in real codes. Most importantly, real codes typically have all cores performing communications simultaneously (unless a hybrid model has been used with explicit threading on a node). The simplest way to investigate this with a benchmark is to perform multiple ping-pongs between pairs of images on different nodes. For example, with 64 cores on two fully populated nodes of HECToR this means pairing up core 0 with core 32, core 1 with core 33, etc.

The results for this "Multi-Put" benchmark are shown in Figure 3 for both synchronisation mechanisms. For comparison the curves for the single ping-pong benchmarks are shown for both coarrays and MPI, as well as those for multiple MPI Sends. The results for get are virtually identical.

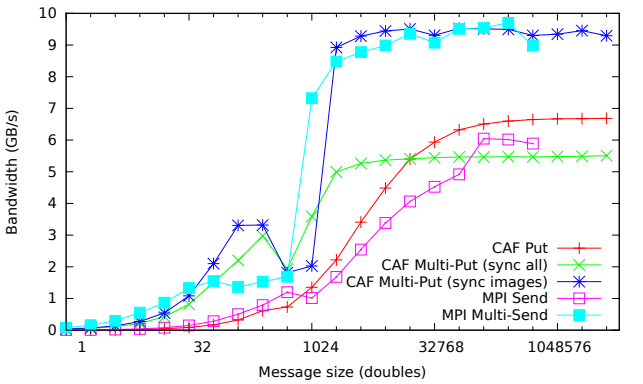

Figure 3. Bandwidth of single and multiple point-to-point data transfers using coarrays and `MPI_Send`

The multi-put results with global synchronisation are quite easy to interpret. Many data transfers are competing for the

same network connection, so the bandwidth is split almost equally between the 32 pairs of images. The aggregate bandwidth of 5.6 GB/s is therefore close to the 6.8 GB/s for a single put (assuming that the single put already saturates the bandwidth). This implies that there is some small overhead of around 20% from contention for the network.

The multi-put results for point-to-point synchronisation are almost twice this figure at 9.5 GB/s. This is simply because, in the absence of global synchronisation, the different ping-pongs naturally become de-synchronised with each other after sufficiently many iterations. In other words, when half of the images on a node are sending the other half are receiving. Since data is travelling in both directions we benefit from the bi-directional capabilities of the GEMINI network. The same is true for MPI where again the individual send/recv pairs are not mutually synchronised. This hypothesis has been confirmed by performing a test using global synchronisation but manually staggering the communications so that half the pairs send while the other half receive. Under these conditions, we achieve a bandwidth of 9.3 GB/s.

The only surprising feature of Figure 3 is the peak in the multi-put bandwidths between 64 and 512 doubles. Presumably the various thresholds for switching between different internal protocols have been optimised for MPI performance and should probably be changed for coarrays, although this was not investigated.

D. Strided data transfers

The standard ping-pong benchmark is not only unrealistic in assuming that just a single pair of processes are communicating: it also assumes a single block of contiguous data. In real applications the data is often non-contiguous, for example if it is a halo region which comprises a slice of some higher-dimensional array (see Section III-F). If we take the simple case of single words of data separated by a fixed stride then this can be expressed directly using Fortran array syntax or as an explicit DO loop, and we measure the performance of both constructs.

The results are presented in Table II where we show data for both the single and multiple versions of the ping-pong. As well as aggregate bandwidth, we report the time per double precision value as this enables a useful comparison with the previous ping-pong latencies. Since for these strided patterns we transfer many thousands of words between a single pair of synchronisation calls, the synchronisation overhead per word should be negligible. We find that performance is generally not sensitive to the stride between words, so here we use a stride of 2. Note that bandwidth figures are in megabytes per second, not gigabytes.

The "DO loop" results can be compared to the latencies in Table I. There is some agreement in that the time taken for put of a single word is marginally higher (around 0.1 μs) than a get. However, they also suggest that we have overestimated the synchronisation overheads in Section III-B.

Mode	time per double (μs)	total bandwidth (MB/s)
Single ping-pong		
Put (DO loop)	1.3	6.1
Put (array syntax)	0.3	25.3
Get (DO loop)	1.2	6.5
Get (array syntax)	1.2	6.6
Multi ping-pong		
Put (DO loop)	0.06	129
Put (array syntax)	0.02	364
Get (DO loop)	0.06	142
Get (array syntax)	0.06	141

Table II
RESULTS FOR STRIDED TRANSFERS

Unlike a contiguous ping-pong, strided transfers are a long way from saturating the interconnect bandwidth. However, this does mean they overlap very well with other strided transfers: when utilising all 32 cores in a node, we get a substantial increase in bandwidth.

What is also interesting is that these results give some insight into what optimisations the compiler is performing. The put results show that the compiler can take advantage of array syntax, presumably generating a single strided RMA call as opposed to multiple single-word calls. This gives an improvement of a factor of four, although we do not see the same benefit for get.

The compiler is similarly able to take advantage of vectorisation analysis to recognize that an explicit DO loop with stride-1 access (as opposed to the stride-2 access above) can be implemented as one large contiguous put or get, achieving 6.8 GB/s for these cases. However, it does currently have a blind spot where array syntax with an explicit stride 1 access pattern is not vectorised (although this is fixed in an upcoming compiler release).

E. Synchronisation

In a real application, each image will normally have to communicate with a set of neighbours and then synchronise with them prior to entering any subsequent calculation phase. The pattern of communication (and hence synchronisation) will be different depending on the application. To investigate this, we benchmark a wide set of different synchronisation patterns. These include patterns with fixed numbers of neighbours: pairwise and with the six neighbours in a 3D grid. We also measure patterns where images synchronise with a variable number of neighbours n: the n nearest neighbours in a ring, or n randomly chosen neighbours.

Perhaps the most representative patterns for regular and irregular problems are the 3D and random cases respectively. In Figure 4 we plot the time taken for the regular 3D pattern, and random patterns with 4 and 10 neighbours (R4 and

R10). We also show the cost of global synchronisation for comparison.

The results show that for relatively small numbers of neighbours, point-to-point synchronisation is faster than global synchronisation for sufficiently many images. The crossover is at about 128 images for 4 random neighbours, or 1024 images for the 3D grid (six neighbours). However, it is clear that the global synchronisation is very highly optimised and for as few as 10 neighbours the crossover point will be above 10,000 images.

These results have changed quite significantly from those reported previously [6]. All the synchronisations are faster, but more importantly the cost is roughly constant with increasing numbers of images for all patterns. Previously, for example, the cost for the R10 pattern increased substantially with image number.

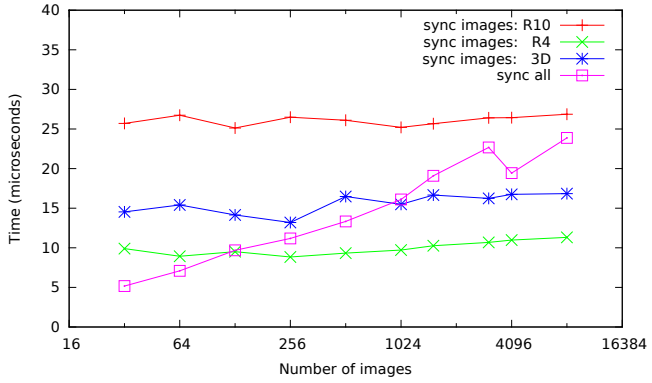

Figure 4. Time taken by various synchronisation patterns

F. Halo swaps

To simulate the communications and synchronisation in a simple regular domain application, we benchmark the performance of repeated halo swaps of the six external faces of a 3D array. We consider weak scaling with the array size per image held fixed, and implement put and get versions with local and global synchronisation. Halo swaps are coded using simple Fortran array syntax. The results are show in Figure 5 and Figure 6 for local array sizes of 10^3 and 100^3 respectively.

For the smaller array size, the fact that global synchronisation is very fast for small numbers of images means that this version initially outperforms point-to-point synchronisation. However, above 512 images the point-to-point version is always faster, in rough agreement with the results in Figure 4. In all cases, the get version outperforms put which is somewhat surprising given the results for strided transfers in Table II where it appeared that the compiler had much better pattern matching for strided puts than gets.

For the larger array size, the synchronisation overhead is much smaller in relative terms and so is much less important at small numbers of images. The get versions are faster than put for fewer than 2048 images regardless of the synchronisation method. Above this the synchronisation overheads become more significant and, for the largest number of images, the point-to-point version is fastest for both puts and gets.

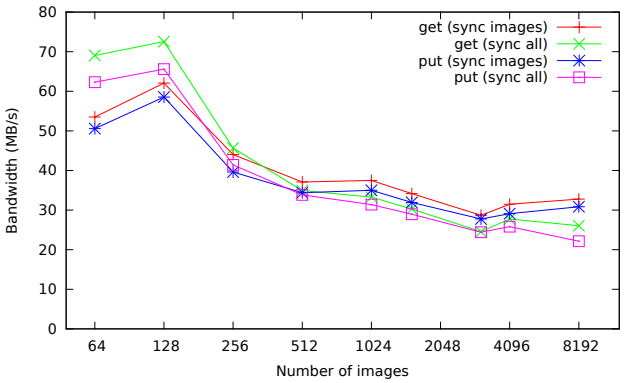

Figure 5. Performance of 3D halo swaps for 10x10x10 local arrays

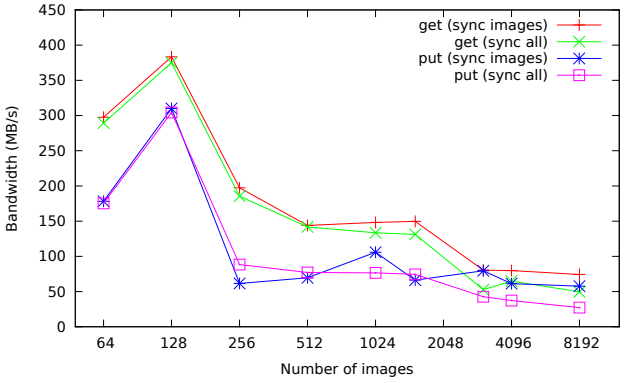

Figure 6. Performance of 3D halo swaps for 100x100x100 local arrays

IV. Conclusions and Further Work

The results show here are generally encouraging, showing significant improvements in performance since 2011 and with coarrays outperforming MPI in several cases. It is clear that point-to-point synchronisation can have significant performance benefits compared to global synchronisation for large numbers of images. However, this does require that the number of neighbouring images is less than around 10. The performance of strided transfers is substantially less than for contiguous ones. Although this could easily be improved by explicitly copying data to and from temporary buffers, this rather defeats one of the purposes of coarrays which is to enable users to write simpler source code. There are however some cases where the conclusions are quite clear, e.g. that

halo swaps should be implemented using gets rather than puts.

There are several cases where it is difficult to interpret the results quantitatively, e.g. the small message latencies do not appear to be the sum of the separately measured data transfer and synchronisation times. This is not particularly surprising as we are making simple assumptions about what runtime calls the compiler generates and how these are implemented. These assumptions may not be true, as illustrated by the substantial difference in performance between strided puts and gets. It would require a more detailed analysis of the binary executable to fully understand these issues.

Although the Cray compiler continues to improve in its support for coarrays, we have encountered a number of bugs (particularly at higher levels of optimisation). These have always been fixed very rapidly once reported, but it is important that all coarray codes have some form of internal verification. The benchmark used here can be run in a debug mode where the correctness of all data transfers is checked (although this is not enabled by default as it severely impacts performance). Note that this also picks up user bugs, often arising from race conditions due to incorrect synchronisation. As the benchmarks execute for many iterations, it is essential to use iteration-dependent data for all verification: a synchronisation bug may not manifest itself on the first iteration.

In the future we plan to extend the benchmark to measure the effect of overlapping communications and calculation, although this may require additional directives to inform the compiler that this is safe to do (e.g. `PGAS DEFER_SYNC` for the Cray compiler). We also plan to make the coarray benchmark suite publicly available.

ACKNOWLEDGEMENTS

I would like to thank Harvey Richardson of Cray Inc. for many useful discussions and for running the benchmark on internal Cray systems. This work has been supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

REFERENCES

[1] T. El-Ghazawi, W. Carlson, and J. Draper, *UPC Manual v1.2*, June, 2005, `www.gwu.edy/~upc/docs/upc_spec_1.2.pdf`

[2] J. Reid, *Coarrays in the next Fortran Standard*, April 21, 2010. `ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf`

[3] R. Numrich and J. Reid, *Co-Array Fortran for Parallel Programming*, ACM Fortran Forum, vol 17, no 2, pp 1-31.

[4] D. Bonachea1, P. Hargrove, M. Welcome and K. Yelick, *Porting GASNet to Portals: Partitioned Global Address Space (PGAS) Language Support for the Cray XT*, Proceedings of the 2009 Cray User Group (CUG2009), available at http://www.cug.org/.

[5] C. Coarfa, Y. Dotsenko and J. Mellor-Crummey, *Experiences with Sweep3D implementations in Co-array Fortran*, The Journal of Supercomputing, 36(2):101-121, May 2006. (Special Issue on Computer Science Research Supporting High-Performance Applications, Rod Oldehoeft, guest editor.)

[6] D. Henty, *A Parallel Benchmark Suite for Fortran Coarrays*, Proceedings of Parco 2011 (to appear).

[7] *Intel MPI Benchmark User Guide and Methodology Description*, Version 3.2, Intel Corporation.