Comparing One-Sided Communication With MPI, UPC and SHMEM

C.M. Maynard
EPCC, School of Physics and Astronomy, University of Edinburgh,
JCMB, Kings Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK
Email: c.maynard@ed.ac.uk

***Abstract*—Two-sided communication, with its linked send and receive message construction has been the dominant communication pattern of the MPI era. With the rise of multi-core processors and the consequent dramatic increase in the number of computing cores in a supercomputer this dominance may be at an end. One-sided communication is often cited as part of the programming paradigm which would alleviate the punitive synchronisation costs of two-sided communication for an exascale machine. This paper compares the performance of one-sided communication in the form of put and get operations for MPI, UPC and Cray SHMEM on a Cray XE6, using the Cray C compiler. This machine has support for Remote Memory Access (RMA) in hardware, and the Cray C compiler supports UPC, as well as environment support for SHMEM. A distributed hash table application is used to test the performance of the different approaches, as this requires one-sided communication.**

I. Introduction

MPI has been the standard communication library used in High Performance Computing (HPC) for the past fifteen or so years. Two-sided, point-to-point communication, where the sending MPI task and receiving MPI task both call the MPI library, is widely used in applications. More recently, an alternative point-to-point communication model, namely one-sided communication has been implemented in Partitioned Global Address Space (PGAS) languages and libraries. In one-sided communication, one process accesses the remote memory of another process directly without interrupting the progress of the second process. Using this programming model could reduce the punitive synchronisation costs of a machine with 2^{30} processes or threads, compared to two-sided communication. There are now several PGAS languages or libraries which implement one-sided communications. Indeed, since the MPI 2 standard was released, MPI implementations also have one-sided communication in the form of put and get library calls. In this work the performance of one-sided MPI is compared with one-sided UPC and Cray SHMEM. MPI, UPC and SHMEM have differences in the way the memory accessible by RMA operations is arranged and the locking mechanisms used to prevent race conditions on particular data entries. These differences have performance implications and they are explored in detail.

Rather than a synthetic benchmark, an application benchmark is used to measure the performance. This application arose from the HPCGAP project[1]. The aim of this project is to produce a parallel version of the symbolic algebra software system, Groups, Algebra and Programming (GAP) [1]. The GAP software system is an interpreted software for symbolic algebra computations. It is written in C, which restricts the choice of PGAS models which could be deployed, ruling out for example, Co-Array FORTRAN. The GAP software system comprises of a kernel, which runs a cycle, creates and destroys objects and manages the memory. The HPCGAP project is implementing a parallel version of GAP for tightly coupled HPC machines. A complimentary project, called SymGridPar2 [2], is also implementing parallel symbolic algebra calculations by linking parallel Haskell [3] with many individual instances of GAP. In contrast to HPCGAP, this software targets loosely coupled architectures of grids or clouds.

As implied by its name, Group theory calculations are one of the targets for HPCGAP. A particular problem, called the orbit problem [4], is of interest to the symbolic algebra community. The problem is the calculation of the orbit of a set m acting on a group g. This calculation corresponds to a graph traversal problem, where at each vertex of the graph some computation of properties of the vertex is required, including the determination of which other vertices the vertex is connected. Keeping track of whether a vertex has been visited before is done by employing a hash table. Current orbit calculations are limited to certain size of group by the amount of memory available in serial. A parallel implementation of the algorithm would allow much larger orbits, corresponding to graphs with billions of vertices, to be computed. The characteristics of the distributed hash table, such as what fraction total memory does the table consume and how often are entries accessed would depend on both the group and the set acting upon it. Current serial implementations are unable to perform such calculations. A parallel implementation would enable a completely new class of problems to be solved, rather than improving the performance on current problems. A distributed implementation exists for a loosely coupled cluster, using multiple instances of different programs over UNIX sockets and exploiting task rather than data parallelism and is reported in [5]. However, for a HPC implementation on supercomputers a different

[1]see http://www-circa.mcs.st-and.ac.uk/hpcgap.php

approach is merited.

A distributed hash table application is used to test the performance of the different approaches. This application requires the use of RMA, as a node determining the hash of a particular data object will know the position in the table to enter object, and thus be able to determine which remote node holds that entry. This would be problematic for a send and receive model as the sender would know the identity of the receiver, but not *vice versa*. Moreover, the receiver would not know when it should be expecting to receive the data. This problem does not arise with RMA operations of the put and get type.

Comparing the performance of one-sided MPI and UPC on a Cray XE6 was reported in [6]. This machine, called HECToR[2], is the UK national High Performance Computing Service. The compute nodes have subsequently been upgraded from two AMD 12-core processors (Magny-Cours) to two AMD 16-core processors (Interlagos). The benchmarks have been repeated and extended for one-sided MPI and UPC and these new results are compared to those for SHMEM.

II. Implementation of a distributed hash table

A simple hash table C code was used as a demonstrator. This C code creates integer typed objects from some random element, computes the hash of the object using a suitable hash function, and inserts a pointer to the object into the hash table. If a pointer to a different object is already resident, the collision counter is incremented and the pointer inserted into the next unoccupied table entry. The code repeats this process so that the populated hash table can be probed, thus the performance of both creation and access of the hash table can be measured. There are three points to note, firstly, the amount of computation is small compared to the memory access. The code is completely memory bound and therefore the performance of a parallel version would be communication bound. Secondly, to reduce the amount of memory transactions in the serial code, the *pointer* to the object is inserted into the hash table, not the object itself, thus reducing the amount of memory access required. In a parallel version, some of the pointers would point to remote objects, which would be undefined locally. Thus in a parallel version, the *object* itself must be stored in the hash table. This has the consequence of further increasing the communication costs. For the UPC version, a model where a *shared* pointer to a *shared* object can be envisioned. Indeed, this could even have less remote memory access (RMA) operations if the pointer is smaller than the object and thus be faster. However, this would consume more of the shared memory, as both the object and the pointer live in the shared space. From the perspective of comparing UPC with MPI and SHMEM, they would no longer be doing the same thing, and this model was not considered for this paper. Thirdly, the cost of RMAs is greater than local memory accesses, so the parallel code will be *slower* than the serial one. However, the motivation is not to execute existing programs faster, but allow larger hash tables to be created in parallel than can be done in serial. Moreover, in a real orbit calculation a significant amount of calculation is required for each element, and by performing the calculation in parallel it would be possible to reduce the overall execution time.

[2]http://www.hector.ac.uk/

A. One-sided MPI

The usual send and receive communication pattern used in MPI codes is not well suited to a distributed hash table. The hash of the object is used to identify the rank of the MPI task which owns that entry in the hash table. However, this node has no way of identifying the rank of an MPI node it is about to receive data from. Indeed, none, one or more tasks may be trying to send a message to a given node at any given time. MPI tasks could be set up to poll for incoming messages, but this would have extremely poor scaling behaviour. In the MPI 2 standard one-sided communication was introduced, which can be employed to negate this requirement. The one-sided MPI communications perform RMA operations. These RMA operations can only access memory which has been specifically set aside in *window*. The memory has to be allocated first, and then the window containing the memory is created with a call to the MPI function, `MPI_Win_create`. Shown in figure 1 is the memory pattern created. In the figure, the size of the arrays on each rank contained in the window is shown to be the same, but this doesn't have to be the case.

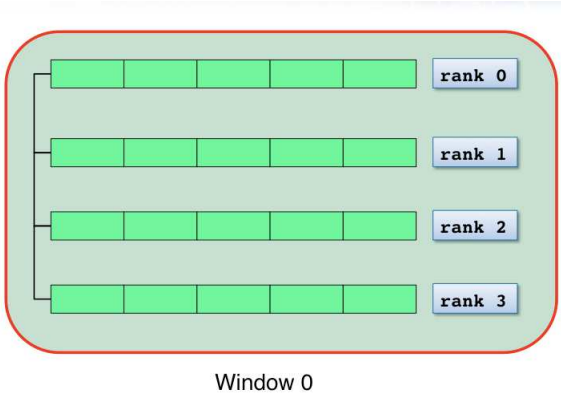

Figure 1. A schematic diagram of the memory pattern for one-sided MPI.

The memory contained in the window can now be accessed by *put* and *get* functions `MPI_put()` and `MPI_get()`. Both the rank and position of the memory location can be specified so that individual words can be accessed.

Synchronisation can be controlled by several mechanisms. For the implementation employed here, an MPI barrier

function, `MPI_Win_fence()`, is used during initialisation. Locks are deployed to control access to the RMA memory window and prevent race conditions. The lock is set using the `MPI_Win_lock()` function, specifying which RMA window and which task's memory are locked. All the memory assigned to the window belonging to the locked task is inaccessible to other tasks until the task is unlocked. The memory can now be accessed by the MPI task holding the lock, and is released with the `MPI_Win_unlock()` function.

B. UPC

The UPC implementation uses shared memory for the distributed hash table. In this example the memory is allocated dynamically using the collective call `upc_all_alloc()`. In contrast to the MPI version, the shared memory is addressable and doesn't require any special functions to access it. Moreover, memory allocated in this way is contiguous. The pointer declaration and memory allocation are shown below:

```
shared [B] int *hashtab;
int nobj;
nobj=2*N/THREADS+1;
hashtab = (shared  [B] int *) \
 upc_all_alloc(THREADS,nobj*sizeof(int));
```

where `N` is a run time variable. The blocking factor `B` however, is a literal, which has to be known at compile time. The environment variable `THREADS`, is evaluated at run time if the code is compiled dynamically. Setting the size and shape of shared arrays in UPC is critical to achieving good performance, so as to minimise the amount of remote memory access. A common procedure to match the compile time blocking factor with the run time number of threads is to compile different binaries for different numbers of threads. A feature of the hash table is that if the hash function is good enough, the access pattern to the table is essentially random. This implies that *any* array distribution is as good as any other, so the default round-robin distribution, with no blocking factor, is employed. Specifically, this means sequential memory addresses in the shared array have affinity with different threads. Shown in Figure 2 is a schematic diagram of the memory allocation and distribution with no blocking factor.

Synchronisation of the threads is achieved in a similar way to one-sided MPI. The function `upc_barrier` is used after the initialisation and locks are used to protect against race conditions. In UPC, an array of locks can be declared and in [6], the non-collective UPC function `upc_global_lock_alloc()` was used to allocate memory to the locks. The non-collective function was used for performance. When using the collective version, all the locks live in shared memory on thread zero. The size of the array of locks can in principle be any size (up to the total amount of shared space). However, this only scaled to a certain size. Profiling revealed that it was the allocation call itself which was performing badly. A better way to allocate shared memory for the locks [7] is as follows:

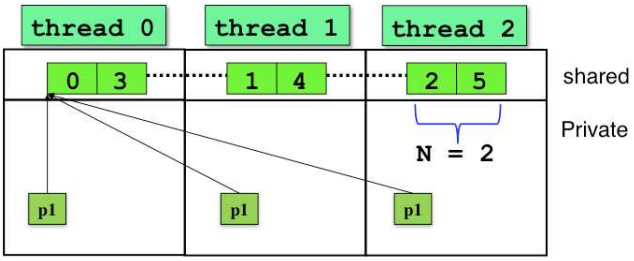

Figure 2. A schematic diagram of the memory pattern for UPC.

```
for  i=0;i<THREADS;++i){
 upc_lock_t* temp=upc_all_lock_alloc();
 if(upc_threadof(&lock[i ]==MYTHREAD){
  lock[i] = temp;
 }
}
```

This construction has much better scaling behaviour. In this implementation the size was set to the number of threads, thus an entire thread's worth of data is locked in one go. In principle, a much finer grained locking strategy could have been implemented, which may result in improved performance at the cost of consuming more of the shared memory allocation. To start with at least, the simplest implementation was deployed. Moreover, the pointer function `upc_threadof()` is used to determine which thread the hash table entry belongs to.

C. SHMEM

The SHMEM implementation uses put and get library calls to achieve RMA. Only variables that are *symmetric* across processing elements (pes) can be accessed in this way. Symmetric variables have the same size, type and relative address on all pes. Moreover, they must be non-stack variables and so must be either global or *local static*. Dynamic memory allocated using the SHMEM call `shmalloc()`. Shown in figure 3 is a schematic diagram of shared memory in SHMEM.

Synchronisation is achieved by calling the library function `shmem_barrier_all()` for global synchronisation and by locks to protect shared memory regions from race conditions. Locks must be symmetric and of type `long int`. Once again an array of locks, of size the number of pes is declared. The array of locks is declared on *each* pe. The library calls which clear, set and test the locks are collective, so that if, for example, pe 1 sets lock element A, no other pe can set that lock element until pe 1 has cleared it. This allows good performance at the cost of consuming an amount of

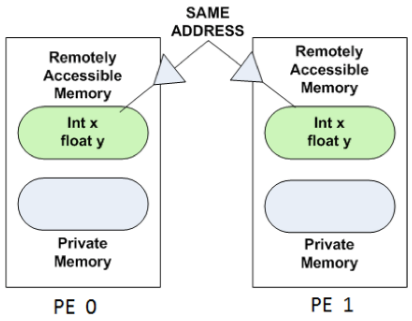

Figure 3. A schematic diagram of the memory pattern for SHMEM. Reproduced from [8]

symmetric memory of size which depends on the number of pes.

III. Performance Comparison

The code was compiled with the Cray Compiler Environment (CCE) version 8.0.1 on an XE6 called HECToR, the UK national supercomputing service. CCE supports UPC and directly targets the Cray XE6's hardware support for Remote Direct Memory Access (RDMA) operations. Each compute node contains two AMD 16-core processors (Interlagos) and the communications network utilises Cray Gemini communication chips, one for every two compute nodes. The SHMEM library is Cray SHMEM (xt-shmem/5.4.2).

For each run, the size of the integer object from which the hash table is made is set at eight bytes, the hash table visited two times (once for creation, once thereafter), with a varying numbers of elements in the hash table. In the first instance, weak scaling, that is, where the number of elements per processing element is fixed, is reported. Shown in Table I are the times in seconds taken to execute the MPI distributed hash table. The same data is plotted in figure 4.

# MPI tasks	Time (s) / LV (thousand) 10	20	40	80	160
32	1.12	1.19	1.83	2.73	4.56
64	4.34	5.93	5.21	7.72	11.5
128	4.26	5.46	6.56	11.0	18.2
256	5.34	6.87	10.3	17.9	32.6
512	7.32	8.50	13.3	24.7	60.4
1024	9.12	12.9	22.8	54.1	109.
2048	15.9	23.16	49.0	104.	258.
4096	26.8	49.3	88.3	200.	408.
8192	45.6	100.	236.	545.	709.
16384	85.2	178.	407.	606.	915.

Table I
TIME TAKEN FOR MPI CODE IN SECONDS, FOR DIFFERENT NUMBERS OF PROCESSORS, FOR DIFFERENT NUMBERS OF ELEMENTS IN THE HASH TABLE. "LV" DENOTES THE LOCAL VOLUME, OR NUMBER OF ELEMENTS IN THE LOCAL HASH TABLE.

The scaling performance of the one-sided MPI distributed hash table is extremely poor. The time taken increases

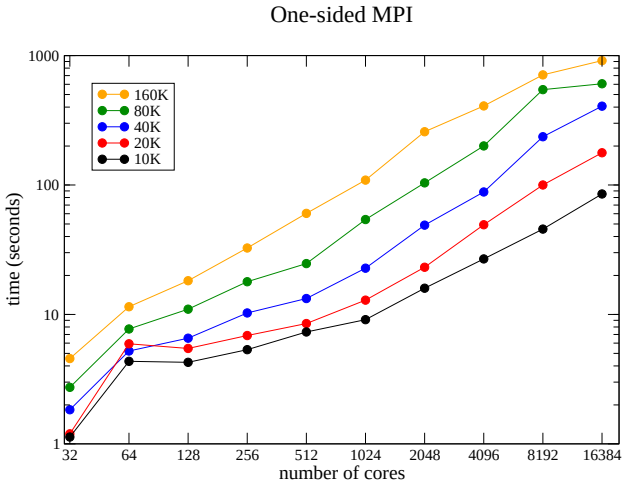

Figure 4. Weak scaling for MPI. Time taken versus number of cores, for different numbers of elements in the hash table. "LV" denotes local volume and refers to the number of elements local to each MPI task.

dramatically as the number of MPI tasks increases. As pointed out in the introduction, this example code does almost no computation. As the number of processing elements increases, the amount of communication increases, which in turn increases the amount of time the code takes to run. It is worth noting that figure 4 has a logarithmic scale for both axes, and the curve drawn through the data to illustrate the trend is approximately linear, showing how bad the scaling behaviour is. The data is not completely smooth, the reason for this could be dependent on the size and shape of the nodes allocated by the job scheduler, which is dependent on machine usage at the time [9]. In previous work [6], several runs for each datum where performed, with very little run time variation reported. However, this data was obtained from a small test and development machine which was not heavily used. It was thus unlikely to highlight this issue. Multiple runs to measure any run time variability were not included in this study as benchmarking large jobs with long run times multiple times is computationally expensive.

The same run parameters were used for UPC, as were used for MPI. Shown in Table II are the data for the UPC benchmark runs. This data is also plotted in figure 5.

In contrast to the MPI results, the UPC results show much better scaling performance. Indeed, the curves in figure 5 show only modest rises, up to quite large numbers of threads, around 4096. This performance data is better than that reported in [6]. As mentioned above, profiling the code with the Cray Performance Analysis tools revealed that allocating the shared memory for the array of locks with the function `upc_global_lock_alloc()` was taking considerable amounts of time. In particular, as the size of the array of locks was increased with the number of threads, the time taken increased dramatically, to the point where it was prohibitively expensive. Naively using the

# UPC threads	Time (s) / LV (thousand) 10	20	40	80	160
32	13.3	13.7	14.1	15.7	18.9
64	13.3	13.7	15.1	17.4	22.3
128	14.5	14.0	15.6	19.1	25.1
256	14.1	15.0	16.3	19.9	27.0
512	14.6	15.5	17.4	21.8	31.7
1024	15.5	16.7	20.3	26.8	39.8
2048	15.6	17.6	21.5	29.8	45.2
4096	17.0	20.1	26.2	38.2	58.4
8192	21.0	25.9	37.1	54.1	92.5
16384	34.8	43.3	64.0	98.6	167.4

Table II
TIME TAKE FOR THE UPC CODE IN SECONDS, FOR NUMBERS OF PROCESSING ELEMENTS AND NUMBER OF ELEMENTS IN THE HASH TABLE. "LV" DENOTES LOCAL VOLUME AND REFERS TO THE NUMBER OF ELEMENTS IN THE LOCAL HASH TABLE.

# SHMEM PEs	Time (s) / LV (thousand) 10	20	40	80	160
32	7.93	8.12	8.97	10.5	13.2
64	7.97	8.43	9.91	11.1	14.6
128	8.05	9.16	10.1	12.1	17.2
256	8.99	9.13	10.4	13.1	17.2
512	8.85	9.29	10.6	14.3	18.0
1024	9.12	9.81	11.2	16.3	20.1
2048	9.21	10.2	12.5	16.3	23.6
4096	9.86	11.9	15.1	21.1	27.7
8192	12.8	14.3	19.0	26.1	46.0
16384	20.7	26.7	31.3	46.0	68.6

Table III
TIME TAKE FOR THE SHMEM CODE IN SECONDS, FOR NUMBERS OF PROCESSING ELEMENTS AND NUMBER OF ELEMENTS IN THE HASH TABLE. "LV" DENOTES LOCAL VOLUME AND REFERS TO THE NUMBER OF ELEMENTS IN THE LOCAL HASH TABLE.

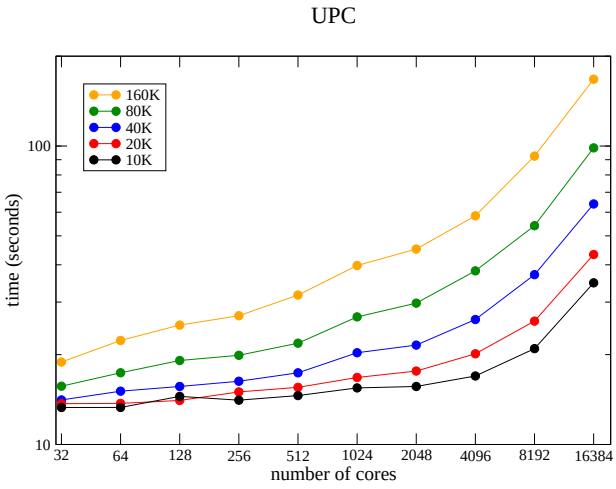

Figure 5. Weak scaling for UPC, time taken for different numbers of threads, for different sizes of hash table. "LV" denotes local volume and refers to the number of elements local to each thread, the "K" denotes thousand.

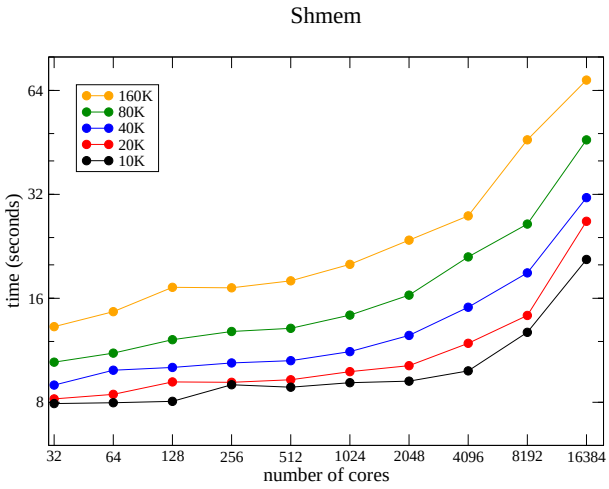

Figure 6. Weak scaling for shmem, time taken for different numbers of threads, for different sizes of hash table. "LV" denotes local volume and refers to the number of elements local to each thread, the "K" denotes thousand.

collective call `upc_all_alloc()` would allocated the data structure with affinity of the zeroth thread. This would have better performance for allocating memory for the data structure, but would result in poor performance for data access. However, the affinity of the locks can be distributed across all the threads using the loop construction described above [7].

Shown in Table III and plotted in figure 6 are the data for the SHMEM benchmark runs with the same parameters as used for MPI and UPC. The SHMEM results show good scaling with low overall clock times. In particular the weak scaling curves in figure 6 show only modest rises out to large (4096) numbers of cores.

Comparing the performance of all three implementations it is immediately obvious that one-sided MPI has greatly inferior scaling behavior to that of UPC and SHMEM, and that SHMEM has the best overall scaling performance. The performance of the same implementations can be compared in a strong scaling scenario, where the global system size is fixed, and the number of cores can be varied. Shown in figure 7 are the results for a strong scaling analysis, the data is the same as shown in the tables above. Here, clearly the MPI performance is much worse. However, one-sided MPI does perform better for smaller numbers of cores, particularly for small local hash table size, than UPC and to a lesser extent than SHMEM. Examining Tables II and III one plausible conclusion is that both UPC and SHMEM have a much larger set up time than one-sided MPI. If the performance for the on-node performance is considered, that is 32 cores, then this effect can be seen, *c.f.* 1.13s for MPI, 13.3s for UPC and 7.93s for SHMEM. It is also clear that both UPC and SHMEM scale better with size of local hash table. This suggests MPI performs worse than UPC and SHMEM for both number of RMA destinations and the number of RMA operations per core.

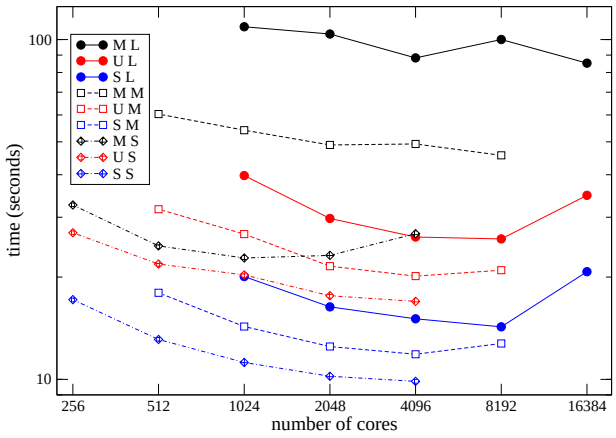

Figure 7. Strong scaling for MPI, UPC and SHMEM, time in seconds versus number of cores. The lines show constant global size of hash table. The legend is made from two symbols, the first refers to the implementation: "M" denotes MPI, "U" denotes UPC and "S" denotes SHMEM. The second symbol refers to the size of the global hash table: "L" denotes 163.84×10^6, "M" denotes 81.92×10^6 and "S" denotes 40.96×10^6.

IV. Conclusions

In this study the performance of one-sided MPI, UPC and SHMEM are compared using an application benchmark up to large numbers of cores. SHMEM has the best scaling behaviour. This is probably due the SHMEM library calls mapping in a straightforward way to the underlying protocol on the Gemini network of the XE6, called *dmapp*. The Cray SHMEM implementation is thus fairly close to what actually happens in hardware [9]. UPC performs well, although slower than SHMEM. It has good scaling behaviour. One-sided MPI in comparison performs poorly and in particular has poor scaling behaviour with both the number of cores and the size of the hash table. Whilst the reason for this is not apparent from this work, how the memory access is performed in the implementation is critical. Comparing the MPI Window construction to the shared memory of UPC and the symmetric memory of SHMEM one obvious difference stands out. The memory that can be accessed by SHMEM and UPC RMA operations is restricted and is special in some sense, whereas the MPI standard specifies that *any* memory can be included in a window. The MPI-3 working group on one-sided communication[3] notes that this is difficult to implement efficiently and proposes that an new type of window, with some restrictions on memory is included in the new standard for performance reasons. The performance of one-sided communication in MPI-2 is so poor, there are not many applications which use this feature of MPI. Indeed, the author isn't aware of any. It is thus an open question as to what practical purpose retaining the unrestricted memory window in the current proposals for MPI-3 serves.

[3]see http://meetings.mpi-forum.org/mpi3.0_rma.php

V. Acknowledgements

The author would like to thank Max Hungover at St Andrews University for the example hash table code and the HPCGAP project. This work was funded under EPSRC grant EP/G055742/1

References

[1] *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, The GAP Group, 2008. [Online]. Available: http://www.gap-system.org

[2] A. Zain, K. Hammond, P. Trinder, S. Linton, H.-W. Loidl, and M. Costanti, "Symgrid-Par: Designing a framework for executing computational algebra systems on computational grids," in *Computational Science ICCS 2007*, ser. Lecture Notes in Computer Science, Y. Shi, G. van Albada, J. Dongarra, and P. Sloot, Eds. Springer Berlin / Heidelberg, 2007, vol. 4488, pp. 617–624, 10.1007/978-3-540-72586-2_90. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72586-2_90

[3] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. W. Trinder, "Seq no more: Better strategies for parallel Haskell," in *Haskell Symposium 2010*. Baltimore, MD, USA: ACM Press, Sep. 2010, to appear. [Online]. Available: http://www.macs.hw.ac.uk/ dsg/gph/papers/abstracts/new-strategies.html

[4] D. F. Holt, B. Eick, and E. A. O'Brien, *Handbook of computational group theory*, ser. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2005. [Online]. Available: http://dx.doi.org/10.1201/9781420035216

[5] F. Lübeck and M. Neunhöffer, "Enumerating large orbits and direct condensation," *Experiment. Math.*, vol. 10, no. 2, pp. 197–205, 2001. [Online]. Available: http://projecteuclid.org/getRecord?id=euclid.em/999188632

[6] C. M. Maynard, "Comparing UPC and one-sided MPI: A distributed hash table for GAP," in *PGAS 2011*. Galveston Island, TX, USA: ACM Press, Oct. 2011, to appear. [Online]. Available: http://pgas11.rice.edu/papers/Maynard-Distributed-Hash-Table-PGAS11.pdf

[7] T. Johnson and S. Vormwald, Pers. Comm., Cray Inc., Feb 2012.

[8] T. Curtis and S. Pophale, "OpenSHMEM specification and usage," in *PGAS 2011*. Galveston Island, TX, USA: ACM Press, Oct. 2011, to appear. [Online]. Available: http://pgas11.rice.edu/etc/PophaleCurtis-OpenSHMEM-Tutorial-PGAS11.pdf

[9] T. Edwards, Pers. Comm., Cray Inc., Mar 2012.