Comparing UPC and one-sided MPI: A distributed hash table for GAP

C.M. Maynard

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

Abstract

The GAP (Groups, Algebra and Programming) software is an interpreted programming language for symbolic algebra computation. It also provides a library of mathematical functionality. A key computational pattern for the GAP community is the orbit problem, that of a group acting upon a set. Computationally this maps onto the graph discovery problem. The enumeration of very large orbits corresponds to the traversal of a graph with billions of vertices. A hash table is used to check whether a vertex has been visited before during the computation. The large memory requirements of such a computation necessitates using a distributed memory machine.

Building a parallel version of GAP is the goal of the HPC-GAP project. Message passing (MPI) and PGAS (UPC) are considered as the models for parallelisation. UPC has some advantages over MPI as some of the data structures anticipated in a parallel implementation of GAP can be simply constructed as shared objects in a PGAS model. Moreover, some of the communication patterns are not suited to the synchronous send and receive model of message passing. For example, in a parallel implementation of a hash table, the task or thread which computes the hash of an object, then knows the table entry and thus whether hash table access is remote or local. For MPI, the usual send - receive mechanism is compromised because the receiving rank cannot determine when, and from whom a message is to be passed. One-side MPI communications can be used to circumvent the problem. Windows of remote access memory are created, and guarded by locks. In UPC, the natural, shared arrays are used, again guarded by locks, However, the locking strategy for MPI and UPC is different. In this paper, the performance of a distributed hash table implemented in UPC and one-sided MPI in a C code using the Cray compiler on an XE6 machine is compared, and UPC found to have better performance. Thus confirming UPC as the choice of parallel model for HPCGAP.

PGAS 2011 15-18 Oct, Galveston Island, USA

1. Introduction

Symbolic algebra is not a traditional consumer of HPC resources. Indeed, most if not all of the computation processes integer data types rather than floating point data types, which are more commonly used in numerical computing. Floating point performance, as measured by flop/s and used to quantify the performance of a HPC system, is therefore not a relevant measure for a symbolic algebra calculation. However, symbolic algebra calculations do require large amounts of computer memory. For some large calculations the memory for even server sized systems can be exhausted, thus limiting the size of a calculation. A distributed memory parallel computer can offer much greater memory than a single machine and thus problems of much greater size can be tackled.

The GAP software system [1], is an interpreted software for symbolic algebra computations, written in C. It comprises of a kernel, which runs a cycle, creates and destroys objects and manages the memory. It has a large library with rich mathematical functionality, which operates on the objects. Packages with functionality not already contained in the library can be loaded. These packages may contain structural functionality, for example input and output, or a message passing package, or even mathematical functionality. The HPCGAP project[1] 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

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

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 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 approach is merited.

There are two parallel models of HPC which could be used to parallelise GAP. Message passing with MPI would be a standard approach, and a PGAS model employing UPC would be the alternative approach. The orbit problem is a critical use case for HPCGAP, so the performance of MPI and UPC for a distributed hash table is used as a design discriminant for the programming model. In this paper we report on the performance of a hash table coded in C, and then parallelised using UPC and MPI. This is used to decide which programming model will be used in HPCGAP.

2. Implementation of a distributed hash table

In order to measure and compare the performance of MPI and UPC 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, 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.

2.1 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.

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

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

memory assigned to the window belonging to the locked task is unaccessible to other task 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.

2.2 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 reside on 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 our implementation, the non-collective UPC function `upc_global_lock_alloc()` is 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). 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.

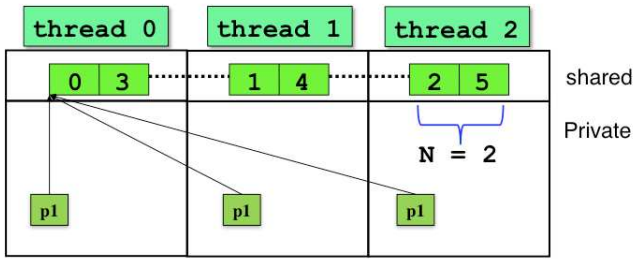

Figure 2. A schematic diagram of the memory pattern for UPC.

3. Performance Comparison

The code was compiled with the Cray Compiler Environment (CCE) version 7.3.1 on an XE6 called HECToR, the UK national supercomputing service[2]. 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 2.1 GHz 12-core processors and the communications network utilises Cray Gemini communication chips, one for every two compute nodes.

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 four times (once for creation, three times thereafter), with a varying numbers of elements in the hash table. In the first instance, we report on weak scaling, that is, where the number of elements per processing element is fixed. Shown in Table 1 are the times in seconds taken to execute the MPI distributed hash table. The table is divided into two, the data in the top section were measured from runs on a smaller test and development XE6, with only 32 nodes, 768 cores. The reported data is the mean of 10 runs, with the error estimated from the standard deviation. The data in the bottom section

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

was taken from runs on HECToR. The data from the top section is plotted in Figure 3.

# MPI tasks	Time (s) / LV (thousand) 10	25	50	100
12	0.98(8)	1.45(8)	2.4(1)	4.39(7)
24	1.02(5)	1.66(3)	2.9(1)	5.39(4)
48	3.6(2)	6.8(3)	11.9(2)	22.5(3)
96	4.7(2)	9.1(3)	17.0(2)	34.1(4)
192	5.6(3)	12.2(2)	24.9(3)	52.4(8)
384	7.07(5)	16.2(3)	35.2(7)	82.(1)
768	10.0(1)	27.4(5)	64.2(8)	161.(2)
768	9.45	24.0	54.5	132.1
1526	17.4	49.0	124.6	354.2
3072	33.1	104.5	240.4	594.1

Table 1. 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. Upper section: data from runs on development machine, lower section: the main HECToR service machine.

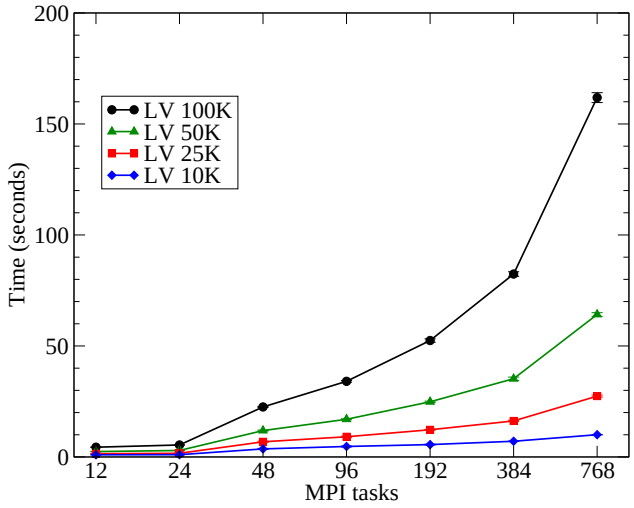

Figure 3. Weak scaling for MPI. Time taken versus number of cores, for different numbers of elements in the hash table. "LV" denotes local volume and refers to the number of elements local to each MPI task.

The most obvious pattern to note is that the code does not scale, in fact the time taken increases quickly as the number of processing elements 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. The difference between intra- and inter-node communication cost can be clearly seen in Figure 3 and Table 1. It takes approximately a factor of 4 times longer for 48 cores than for 24. Thereafter, the increase in time as the number of cores increases is less severe, but it increases none the less. The other obvious trend from the data is that a larger local volume takes longer to run. As the hash function produces a random distribution across the hash table, there is no data locality, so a larger local volume means more communication.

Shown in the bottom section of Table 1 is run time from the main HECToR service machine. There is no difference in hardware specification between the machines. As the variance data in the top half of the table is small, only one run is reported for large numbers of processors. A run with 768 processors is repeated for reference and is slightly faster on the main machine, possibly revealing an "edge effect" on the smaller machine. Overall the trends remain the same. This is shown in Figure 4.

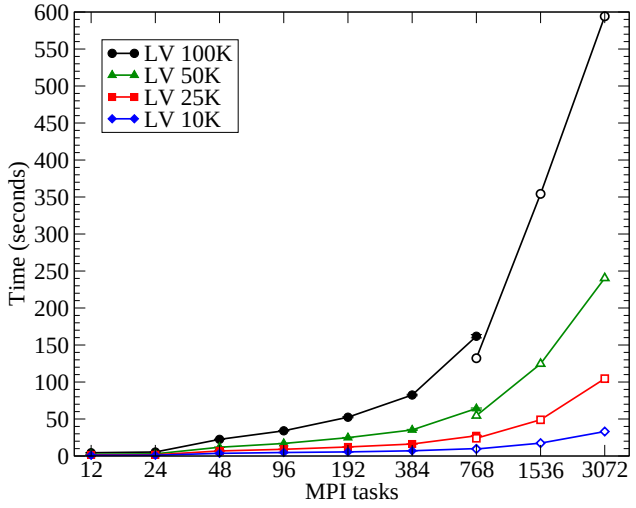

Figure 4. Weak scaling for MPI. Time taken versus numbers of cores, for different sizes of hash table. The open symbols denote data from run times on the main HECToR service machine. "LV" denotes local volume and refers to number of elements in the local hash table.

The same run parameters were used for UPC, as were used for MPI. Shown in Table 2 are the data for the UPC benchmark runs. In the upper section of the table, the reported results are the mean of 10 runs, with the quoted error determined from the standard deviation of those ten runs. This data is also plotted in Figure 5.

In contrast to the MPI results, the UPC results show much better communication performance. Whilst it takes longer to execute the code with more processors, the increase is much slower. This can be seen in the figure, where the lines are much flatter. Moreover, the overall scale is much smaller, and so at large numbers of processors the performance is much better, *i.e.* the code executes much faster. The UPC benchmark code for the distributed hash table is much faster than the equivalent one-sided MPI code. However, as the number of UPC threads is increased to 768, the cost starts to increase more rapidly.

# UPC threads	Time (s) / LV (thousand) 10	25	50	100
12	3.83(5)	4.90(12)	6.53(5)	9.84(9)
24	7.14(10)	8.52(10)	11.0(1)	15.9(1)
48	7.59(16)	9.98(20)	13.7(2)	21.2(2)
96	8.34(20)	11.0(3)	15.5(3)	24.5(2)
192	9.0(4)	11.8(2)	16.8(2)	26.8(1)
384	10.1(1)	13.4(2)	18.9(1)	29.8(2)
768	14.5(6)	18.2(2)	24.7(1)	36.8(1)
786	19.2	23.3	29.1	41.7
1536	54.3	57.4	66.3	84.6

Table 2. Time take for the UPC code, in seconds for numbers of processing elements and number of elements in the hash table. The upper section shows data benchmark runs on the development machine, the lower section the main HECToR service machine. "LV" denotes local volume and refers to the number of elements in the local hash table.

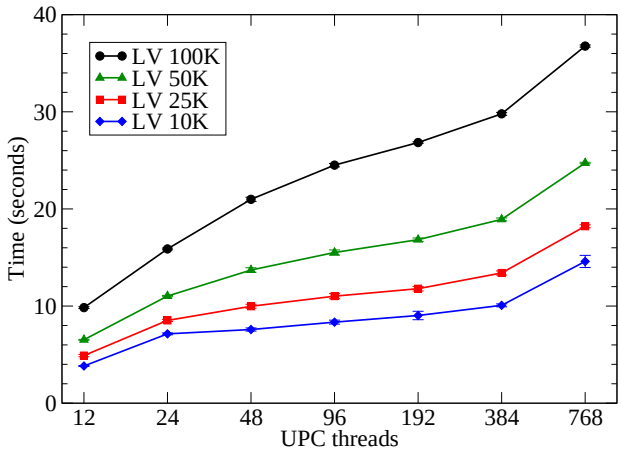

Figure 5. Weak scaling for UPC, time taken for different numbers of threads, for different sizes of hash table. "LV" denotes local volume and refers to the number of elements local to each thread, the "K" denotes thousand.

Shown in the lower section of table 2 is data taken from benchmark runs on the main HECToR service machine. As with the MPI data, the variance in the UPC data from the runs on smaller number of cores is small, so for the runs on larger machine partitions, the results are reported for only one run. These results are also shown in Figure 6. In contrast to the MPI data, there is no UPC data for 3072 cores. This is because a run-time environment variable cannot be set shared memory greater than 2 TByte [3]. The trend observed for large numbers of cores in Figure 5 can be clearly seen to continue in Figure 6 *viz.* the modest "scaling" seen for moderate numbers of cores breaks down beyond 768 cores. The execution time for 1536 cores takes approximately twice as long as 768 cores.

[3] Cray hope to extend this in the future.

The performance of the MPI and UPC versions can be compared. For a small hash table at 1536 cores (local size = 10^4 elements, global size = 1.536×10^8) the MPI code is much faster, 17.4s for MPI compared to 54.3s for the UPC code. However, as the size of the hash table is increased, the MPI code slows dramatically. For the largest size of hash table examined, (local size = 10^5 elements, global size = 1.536×10^9) the MPI code takes approximately a factor *twenty* times longer at 354.2s to process the large hash table compared to the small one. In contrast the UPC code takes less than a factor of two longer at 84.6s to process a hash table a factor of ten larger in size. As the system size increases, so does the number of RMA operations. It is clear from the data that the UPC implementation is faster for large numbers of RMA operations, arising from large numbers of cores, or from large hash tables. However, at large numbers of cores, the performance of the UPC implementation starts to degrade, but for a large hash table, on such a system, the MPI performance is significantly worse.

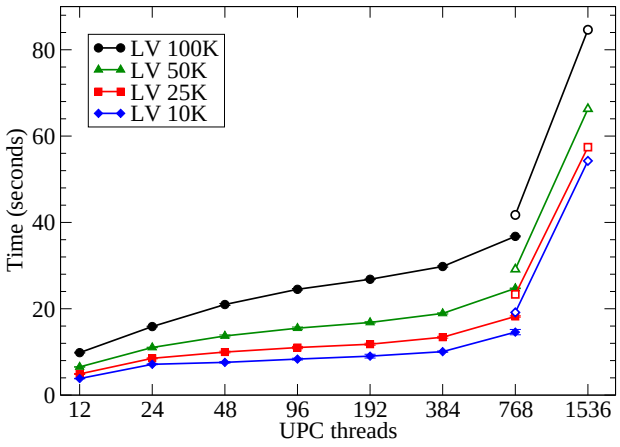

Figure 6. Weak scaling for UPC, time taken for different numbers of threads, for different sizes of hash table. The open symbols denote data from taken from benchmark runs on the main HECToR service machine. "LV" denotes local volume and refers to the number of elements local to each thread, the "K" denotes thousand.

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 Table 3 and Figure 7 are the results for a strong scaling analysis.

At each problem size, the benchmark was run for three different numbers of cores. As can be clearly seen, as the number of cores is increased for a fixed global hash table size, the speed of execution increases. The amount of communication *per core* decreases, so the amount of time taken is reduced. For the smallest hash table the curve is fairly flat and as the code is both memory and communication bound, increasing the number of cores further is unlikely to increase performance, thus the benchmark was only run for

	Time (s) [UPC]			Time (s) [MPI]		
	# elements ($\times 10^6$)					
# cores	4.8	9.6	19.2	4.8	9.6	19.2
48	13.7	–	–	22.5	–	–
96	11.0	24.5	–	17.0	34.1	–
192	9.0	16.8	26.8	12.2	24.8	52.4
384	–	13.4	18.9	–	16.2	35.2
768	–	–	18.2	–	–	27.4

Table 3. Time in seconds for both UPC and MPI, for different numbers cores, at constant size of the global hash table, for different sizes of global hash table.

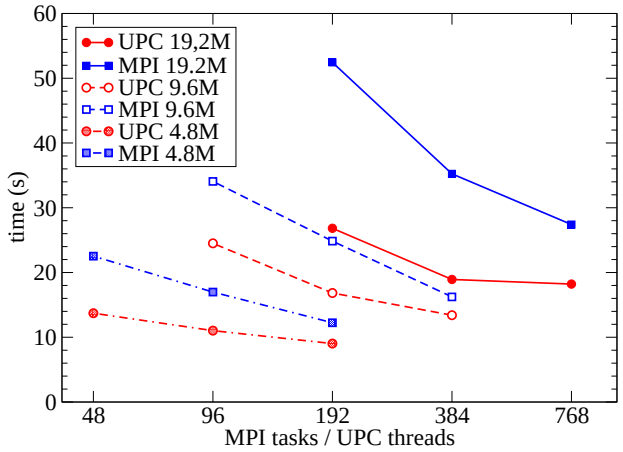

Figure 7. Strong scaling for both UPC and MPI, time in seconds versus number of cores. The lines show constant global size of hash table. The size of the global hash table is shown in the legend, "M" denotes 10^6.

a small number of cores. The largest hash table requires a large memory allocation and is only run on the larger core counts. Overall, the same pattern emerges, the UPC implementation is significantly faster than MPI version.

The Cray Performance Analysis Toolkit (Cray-PAT) was used to profile the MPI implementation. Between 60% and 80% of the execution time was spent in the `MPI_Win_unlock()` function. Thus demonstrating that *contention* between tasks, *i.e.* tasks trying to obtain the lock for an already locked task, is not adversely effecting performance. However, the MPI 2 standard does not define *when* RMA operations take place during a the lock/unlock phase or *epoch*. Merely that the lock and unlock operations define the start and end of that epoch. In this implementation of MPI, the lock, RMA operations, are in effect buffered until the unlock occurs, which synchronises the MPI tasks. As measured by Cray-PAT, the most of execution time is taken during the MPI unlock call, because this is when the one-sided MPI functions are executed across the window.

In the MPI implementation the lock/unlock pair of functions brackets each RMA operation. This potentially means each MPI task is holding the lock is doing so for the shortest possible time. However, the lock/unlock functions are therefore called several times during each transaction with the hash table. Altering this so that a single pair of lock/unlock functions are used to protect all the RMA operations for each hash table transaction had at most a very small impact on the timings. Contention between MPI tasks is not a performance limiting factor, nor is the lock/unlock calls themselves (despite the profiling results). Changing the locking strategy has little effect because the RMA operations dominate the time taken, even if they are actually processed during the unlock call.

	time (s)	
size (bytes)	UPC	MPI
4	30.1	51.3
8	29.8	52.4
16	30.4	69.3
32	31.4	116.0

Table 4. Time taken for benchmark runs with different sizes of integer objects UPC and MPI implementations. Number of elements in the hash table set at 1.92×10^7 on 192 cores.

One final set of benchmarks were run to check the performance of the two codes. The size of the integer data object was varied to see if the communication of smaller or larger chunks of data effected the performance. Shown in Table 4 are the benchmark data for different data sizes for a fixed number of cores, at a fixed size of hash table. Once again, the UPC implementation is clearly faster than the MPI version. Varying the size of the integer object has no effect on the time taken for the UPC code, whereas the performance of the MPI implementation degrades as the size of the object increases.

4. Conclusions

The UPC implementation of the distributed hash table is faster than the MPI implementation, except for the regime where there is a small number of elements local to each processor, and the regime where the global volume and the number of MPI tasks are small. Neither regime is likely to be accessed when processing large orbit problems. UPC scales better with the number of processing elements for both weak and strong scaling. Moreover, it shows better scaling both with the number of elements in the hash table, and the size of the objects in table. The one-sided MPI functions are not widely used in applications, at least the author is not aware of many, or indeed any applications using them. A possible explanation as to why the MPI version is slow compared to the UPC version is that whilst the implementation of MPI itself executes the RMA operations correctly, the MPI implementation has not been optimised due to the lack of applications to stress the implementation. This is of course, speculation, and may well not explain all the performance

gap, as it may be fundamental to the way MPI handles one-sided communication. Whilst uncovering the reasons for this would be very interesting, is not obvious to the author why this should be so.

The UPC implementation could potentially be improved. In UPC, the size of the array of locks could be defined the same size as the hash table. Thus, an individual data element could be locked rather than all the data local to a thread. This would reduce contention. However, this is likely to be a small gain, as the evidence from the albeit slower MPI version is that contention is not an issue. Moreover, the locks consume shared memory, which is needed for the hash table, so it is not clear that this strategy would be better.

The object of this exercise was to implement a realistic use case to decide which model of parallel programming would be best for HPCGAP. The results are clear, and UPC has been chosen as the model for parallelising HPCGAP.

5. Acknowledgements

The author would like to thank Max Neunhoffer 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. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*. The GAP Group, 2008. URL `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 Y. Shi, G. van Albada, J. Dongarra, and P. Sloot, editors, *Computational Science ICCS 2007*, volume 4488 of *Lecture Notes in Computer Science*, pages 617–624. Springer Berlin / Heidelberg, 2007. URL `http://dx.doi.org/10.1007/978-3-540-72586-2_90`. 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, Sept. 2010. ACM Press. URL `http://www.macs.hw.ac.uk/~dsg -/gph/papers/abstracts/new-strategies.html`. To appear.

[4] D. F. Holt, B. Eick, and E. A. O'Brien. *Handbook of computational group theory*. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2005. ISBN 1-58488-372-3. doi: 10.1201/9781420035216. URL `http://dx.doi.org/10.1201/9781420035216`.

[5] F. Lübeck and M. Neunhöffer. Enumerating large orbits and direct condensation. *Experiment. Math.*, 10(2):197–205, 2001. ISSN 1058-6458. URL `http://projecteuclid.org/getRecord?id=euclid.em/999188632`.