



Profiling H2MOL on an IBM p690+ Cluster

Lorna Smith, Mark Bull, Arthur Trew, The University of Edinburgh
and Andrew Sunderland, Daresbury Laboratory

October 7, 2004

1 Synopsis

In this document we investigate the performance of the H2MOL code on HPCx. We investigate the speed up of the code on multiple processors, and the performance difference between the recent upgrade to HPCx and the previous system (Phase 1 and Phase 2). We consider the variation in performance with increasing problem size and the reproducibility of results. We profile the code to determine the computationally intense components, and investigate and optimise the communication overhead.

2 The System

HPCx is the UK's largest National High Performance Computing resource. It is operated by the HPCx Consortium, a consortium of the University of Edinburgh [1], through Edinburgh Parallel Computing Centre (EPCC) [2], the Council for the Central Laboratory for the Research Councils (CCLRC) Daresbury Laboratory [3] and IBM [4]. The project is funded by the Engineering and Physical Sciences Research Council (EPSRC) [5].

The HPCx system has recently undergone a technology refresh, from the Phase 1 system to the Phase 2 system. In this report we provide performance results for both systems.

2.1 Phase 1 System

The Phase 1 system of HPCx consisted of 40 IBM pSeries 690 Regatta nodes, each containing 32 1.3 GHz POWER4 processors (a total of 1280 processors). Each POWER 4 chip contained two processors, together with the Level 1 (L1) and Level 2 (L2) cache. Each processor had its own L1 instruction cache of 64

kB and L1 data cache of 32 kB integrated onto the chip. Also on-board the chip was the L2 cache (instructions and data) of 1.44 MB, which was shared between the two processors. Four chips (8 processors) were integrated into a multi-chip module (MCM). Four MCMs (32 processors) comprised one frame. Each MCM was configured with 128 MB of Level 3 (L3) cache and 8 GB of main memory, shared between the eight processors of the MCM.

The nodes in the Phase 1 system were connected via IBM's SP Switch2 (Colony) network. In order to make optimal use of the connections from the frames to the switch, each frame was logically partitioned into four 8-way nodes or LPARs (Logical PARTition). Each LPAR ran its own copy of the AIX operating system. Even when two LPARs in the same frame communicated, they used the Switch network, just like any other pair of LPARs.

2.2 Phase 2 System

The current Phase 2 system of HPCx consists of 50 IBM pSeries 690+ Regatta nodes, each containing 32 1.7 GHz POWER4 processors (a total of 1600 processors). The peak computational power of the HPCx system is 10.8 TeraFlop/s, or up to at least 6 TeraFlop/s sustained.

As with Phase 1, each a chip contains two processors, together with the Level 1 (L1) and Level 2 (L2) cache. On this system, each processor has its own L1 instruction cache of 64 kB and L1 data cache of 32 kB integrated onto one chip. The size of the L2 cache on board the chip (instructions and data) is 1.5 MByte, which is shared between the two processors. Again four chips (8 processors) are integrated into a multi-chip module (MCM) and four MCMs (32 processors) comprise one frame. Each MCM is configured with 128 MB of L3 cache and 8 GB of main memory. The total L3 cache of 512 MB per frame and the total main memory of 32 GB per frame is shared between the 32 processors of the frame.

The frames in the HPCx system are connected via IBM's High Performance Switch (HPS). Unlike Phase 1, each frame has only one 32-way logical partition. Each LPAR runs its own copy of the AIX operating system. HPCx is running AIX 5.2.

3 The Procedure

The code was compiled using XL Fortran for AIX, Version 8.1.1.4. It was compiled using the ESSL [6] and LAPACK [7] libraries and with the compiler options `-q64 -O3 -qarch=pwr4 -qtune=auto -qhot`.

A range of environment variables were specified within the batch script. These variables are listed in the Appendix.

In order to investigate the performance of H2MOL we have made extensive use of three tools: Xprofiler [8], MPITrace [9] and Vampir [10].

Xprofiler is a simple profiler that profiles both sequential and parallel code at the source level. With a GUI type interface, Xprofiler provides a profile of

CPU utilisation and execution time. Xprofiler does not report I/O or communication time.

MPITrace is an IBM tool that provides details of the amount of time spent in communication routines, including information on the number of times a particular routine is called, and the average message size for that routine.

Vampir (Visualisation and Analysis of MPI Resources) is a post-mortem trace visualisation tool from Intel GmbH, Software and Solutions Group. It uses the profiling extensions to MPI and permits analysis of the message events where data is transmitted between processors during execution of a parallel program.

4 The Code

H2MOL is a code from the Multiphoton and Electron Collision Consortium (CCP2) written by Ken Taylor & Daniel Dundas, Queens University Belfast. It solves the time-dependent Schroedinger equation to produce estimates of energy distributions for laser-driven dissociative ionization of the H_2 molecule. A cylindrical computational grid is defined with ϕ , ρ and Z coordinates, with the Z domain distributed amongst an array of processors arranged logically in a triangular grid (to take advantage of symmetry). A feature of the way this code has been written is that it specifies as constant the number of grid points per processor in the Z -direction. Thus with increasing numbers of processors it is working with an increasingly refined mesh i.e., for this benchmark, perfect scaling would be represented by a flat timing profile across the different processor counts.

5 Reproducibility

All the results presented in this report have been run on a system with other user codes running. Each job has exclusive access to the nodes on which it is running, and does not share processors or memory with other applications. Each node does however run a copy of the operating system, which may influence the performance of the code.

Each node has 4 switch adaptors, which are connected to two switch nodes. While the adaptors are exclusive to a node, the two switches are shared between 8 nodes. Hence the performance of the code can be influenced by other user applications located on different nodes.

Lastly, files are written and read to two separate I/O nodes, hence switch adaptors, processors and memory are all shared between user applications on these nodes. This can also influence the performance of the code.

All the results presented here have been run multiple times, and the lowest timings reported.

6 H2MOL Performance

Figure 1 shows a graph of execution time against processor number. Results are shown for the Phase 1 system and the Phase 2 system (with and without large pages and with the new switch upgrade).

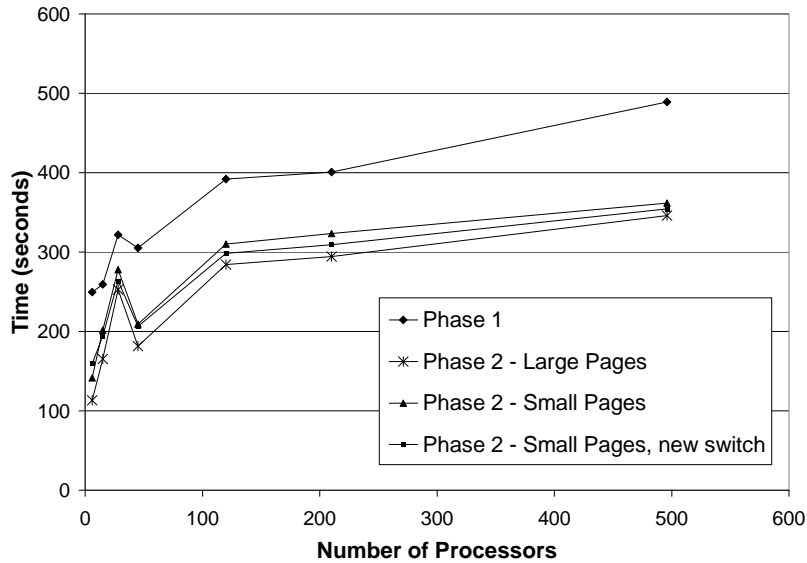


Figure 1: Execution time versus processor number for H2MOL.

As mentioned earlier, perfect scaling would be represented by a flat timing profile across the different processor counts. The unusual profile observed here is due to the nature of the code. H2MOL runs on a specific set of processor numbers. In this example the code has been run on 6, 15, 28, 45, 120, 210 and 496 processors. See Table 1 for the distribution of processes per frame.

H2MOL reads and writes large amounts of data to main memory. On HPCx, each MCM has its own L3 cache and main memory. Although on Phase 2 a processor on one MCM may access the L3 cache and main memory on another MCM (if, for example, they require the extra memory) a processor usually writes to the L3 cache and main memory on its own MCM.

Since this code specifies as constant the number of grid points per processor, then as the number of processors within an MCM increases, the amount of data being accessed per MCM increases and the performance decreases. Hence the fewer processors used per node, the better the overall execution time. This explains the significant fluctuation in performance between 15, 28 and 45 pro-

Total Number of Processors	No of Processors Per Frame	No of Frames
6	6	1
15	15	1
28	28	1
45	15	3
120	30	4
210	30	7
496	31	16

Table 1: Total number of processors, number of processors used per frame and number of frames

cessors. For larger processor numbers, the code scales well.

As expected, the Phase 2 system performs better than the Phase 1 system, primarily due to an increase in the clock speed of the processors. The Phase 2 system was originally configured using large pages, but was changed to small pages. Many applications reported little performance difference, however H2MOL shows some reduction in overall performance.

Lastly, the Phase 2 system has recently undergone an upgrade in the switch, which has increased the bandwidth and reduced the latency. This has resulted in a slight improvement in the overall performance of H2MOL.

To understand these curves further, we have investigated the ratio of performance between Phase 1 and Phase 2 on various processor numbers and problem sizes. We have also investigated the amount of time spent in the computationally intense routines of the code.

7 Phase 1 versus Phase 2 Ratios

The ratio of performance between the Phase 1 and Phase 2 system (using small pages and the new switch) is shown in Figure 2. This ratio fluctuates between a high of 1.6 and a low of 1.2. It is interesting to note that this ratio increases gradually with higher processor number.

The fluctuation at lower numbers of processors can be explained by differences in the Phase 1 and Phase 2 systems. On the Phase 1 system, processes were allocated to fill MCMs, while on Phase 2 processes are allocated to MCMs in a round robin fashion. Hence on the Phase 2 system the code is running with fewer processes per MCM, giving better performance (as discussed above). This effect is more pronounced for the lower processor counts, hence the larger variation in performance for 6 processors over 28 processors.

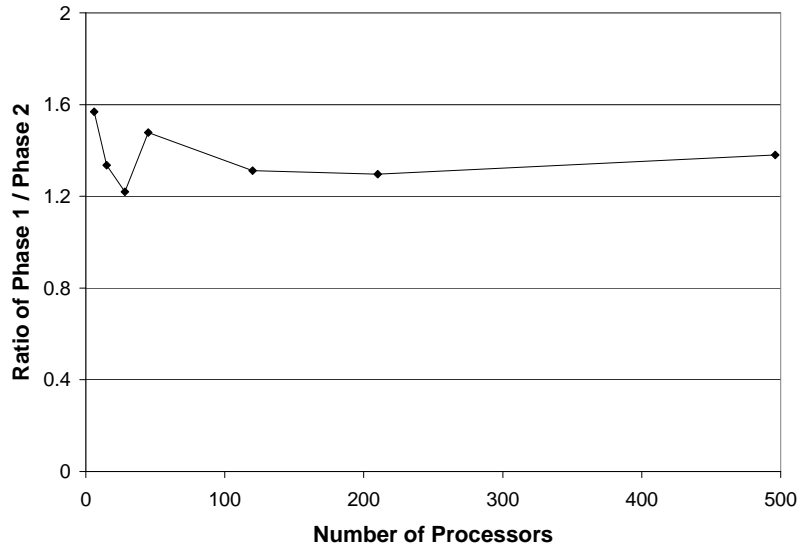


Figure 2: Ratio of Phase 1 execution time over Phase 2 execution time (with the switch upgrade) for a range of processor numbers.

8 Profiling

To understand where the computationally intense components of the H2MOL code are, we profiled the code on 28 processors using Xprofiler and simple timer calls. Table 2 contains details of the amount of time spent in the computationally intense routines of the code.

The majority of the time is spent in the routine Apply Hamiltonian, the BLAS routine CAXPBY and the routine inner product. The routine inner product consists of multiple global communication calls. Further investigation of the Apply Hamiltonian routine shows that the majority of the time in this routine is spent in the BLAS routine ZGEMM and in carrying out point to point communications.

As the majority of the computation is carried out using optimised BLAS routines, the computational part of the code is relatively well optimised. We have therefore focussed our attention on the communication routines - both the collective and point to point operations.

9 Communication

In this section we consider the communications within the code (on the Phase 2 system). The amount of time spent in communications has been measured

Routine	Time (s)	Percentage
Total time	278.9	
Apply Hamiltonian	171.7	62
BLAS - CAXPBY	77.5	28
Inner product	22.5	8
Apply Hamiltonian: ZGEMM	75.8	27
Apply Hamiltonian: Point to Point Communication	47.9	17

Table 2: The computationally intense subroutines within H2MOL. The table provides execution times for each routine and their percentage of the total execution time

using MPITrace. MPITrace provides details of the amount of time spent overall in communications and in the most computationally intense MPI routines, on a per processor basis. Figure 3 shows the average amount of time (across processors) spent in communications. This demonstrates that the scaling of the code at high processor numbers is influenced by the communication routines, in particular by time spent in MPI.Recv and MPI.Barrier routines.

Table 3 shows the average, maximum and minimum amount of time spent across processors in each of the main communication routines. This clearly shows a wide variation in the amount of time spent in these routines on a per processor basis.

These results demonstrate that the majority of the communication time is spent in MPI.Recv and MPI.Barrier routines, and that this time varies significantly between processors. To understand this further, we have utilised the tool Vampir. Vampir gives a graphical representation of the communication pattern within the code. Figure 4 shows a graphical representation of the main communications going on within the code on 6 processors, for one iteration. This pattern is repeated for each iteration of the code.

Lines between processes represent MPI point to point and collective communications. Darker blocks represent time spent in communications, lighter blocks represent computation ('User_Code').

This clearly shows that process 3 and 4 appear to spend less time in user routines than processes 1, 2, 5 and 6. This pattern is repeated across the whole Vampir trace. The explanation for this is that we are using a full node on HPCx, but with only 6 processors, and these processors are allocated in a round robin fashion to different MCMs. As a result, two of the four MCMs will have fewer processors per MCM, resulting in less data going through the memory system. Hence these two processors (processors 3 and 4) will be faster than processors 1, 2, 5 and 6. By allocating all the processes to one MCM (using VSRAC [11]), we see this load imbalance has disappeared (See Figure 5). The overall execution time is, as we would expect, slower, as six processes are competing for the same memory bandwidth of an MCM. Note that this effect is observed for

Communication	Average Time (s)	Maximum Time (s)	Minimum Time (s)
Number of Processors = 6			
Total	13.74	19.08	11.00
MPI_BSend	0.66	1.25	0.25
MPI_Recv	8.70	13.02	5.20
MPI_Barrier	4.36	9.01	1.47
Number of Processors = 15			
Total	27.40	35.74	19.30
MPI_BSend	1.10	2.00	0.33
MPI_Recv	14.33	19.16	8.39
MPI_Barrier	11.94	19.36	6.59
Number of Processors = 28			
Total	51.98	73.93	30.21
MPI_BSend	2.23	3.74	0.53
MPI_Recv	27.21	46.78	12.42
MPI_Barrier	22.33	36.04	6.00
Number of Processors = 45			
Total	38.41	54.04	28.01
MPI_BSend	1.24	2.29	0.34
MPI_Recv	13.94	23.67	7.65
MPI_Barrier	23.16	36.34	10.54
Number of Processors = 120			
Total	61.89	92.71	39.16
MPI_BSend	2.50	4.70	0.66
MPI_Recv	25.97	48.54	14.89
MPI_Barrier	31.63	51.63	14.33
Number of Processors = 210			
Total	67.68	104.24	45.28
MPI_BSend	2.66	5.30	0.71
MPI_Recv	26.05	42.60	12.17
MPI_Barrier	36.61	61.76	20.89

Table 3: Timings for the main communication routines in H2MOL. Average, maximum and minimum times are given for different processor counts.

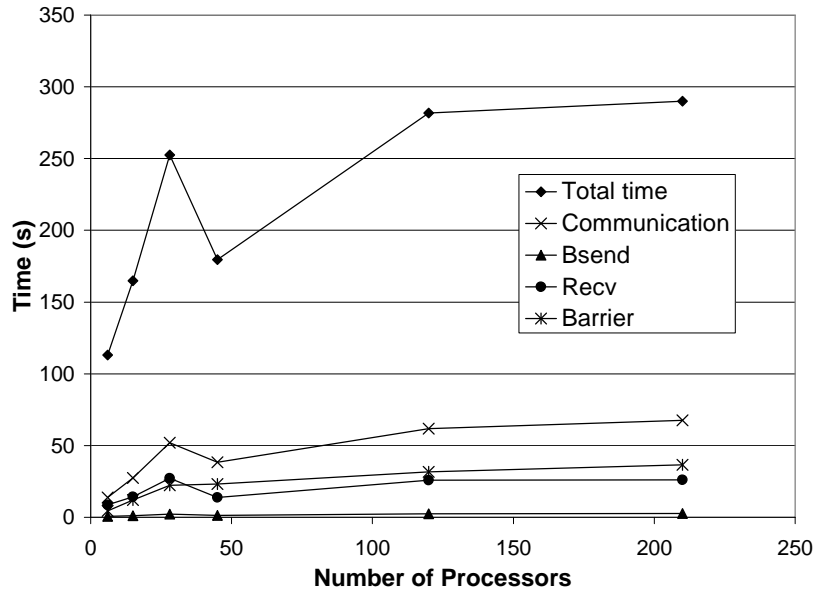


Figure 3: Communication time versus processor number for H2MOL.

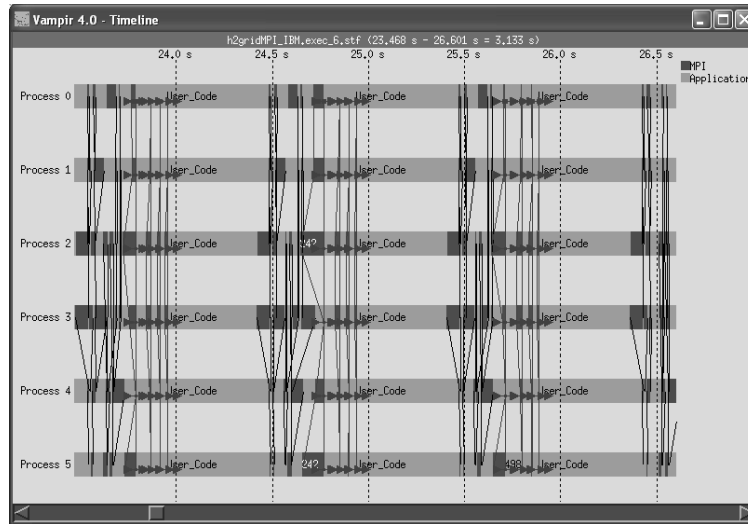


Figure 4: Graphical representation of the communication pattern in H2MOL for 6 processors (from Vampir).

other processor numbers.

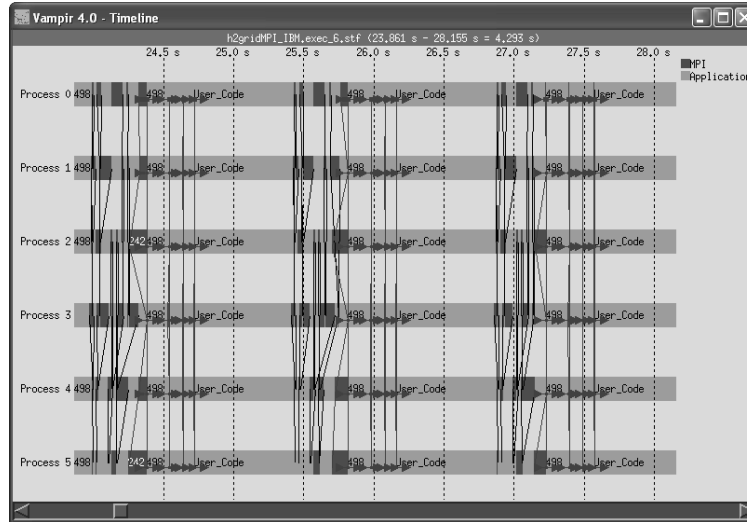


Figure 5: Graphical representation communication pattern in H2MOL for 6 processors located on the same MCM (from Vampir).

Figure 6 shows the point to point communications within the code. In this case they represent MPI_Bsend operations. It is clear from this diagram that the relatively large amount of time spent in MPI_Recv operations is due to processors waiting to receive data. This is partly due to variations in the amount of time spent doing computation ('user code') by each processor (explained above). Looking more closely however, we observe that certain processes seem to be spending time in MPI_Recv operations unnecessarily, as the matching send has already been posted.

This can be explained by the way MPI handles buffered send operations. If the receiving process is not ready to receive when the MPI_Bsend operation occurs, the system will wait until the next time the MPI library is called before trying to send the message again. This results in some processes sitting idle unnecessarily, waiting for the MPI library to be called again. We can force the system to try and send this message more quickly using the environment variable MP_CSS_INTERRUPT. Figure 7 shows the point to point communications for H2MOL on 6 processors with MPI_CSS_INTERRUPT switched on.

As we can see we no longer observe the unnecessary waiting in MPI_Recv operations. It is interesting to note however that on a larger number of processors, say 120 processors, MP_CSS_INTERRUPT has little effect, and does not prevent these unnecessary waits in MPI_Recv (See Table 4).

Hence, while buffered send operations allow the point to point communications to be overlapped, and thus give better performance than standard synchronised sends, on larger numbers of processors the performance benefit

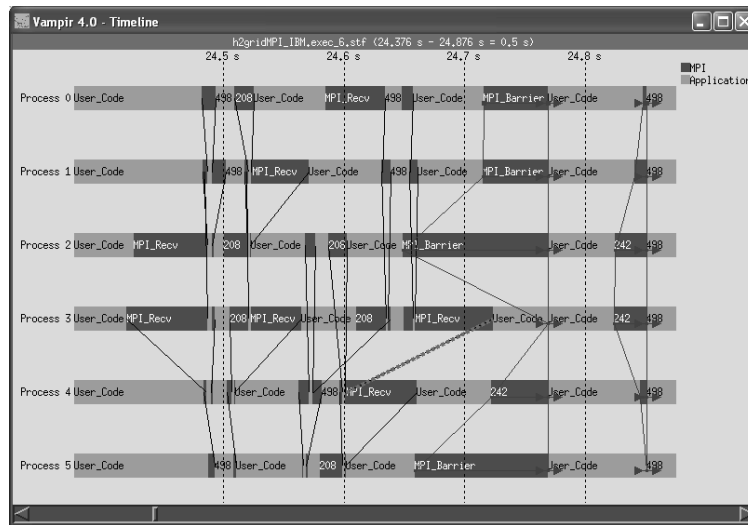


Figure 6: Graphical representation of the point to point communication pattern in H2MOL for 6 processors (from Vampir). The highlighted send operation between process 4 and 3 demonstrates that process 3 spends an unnecessary amount of time in the receive operation

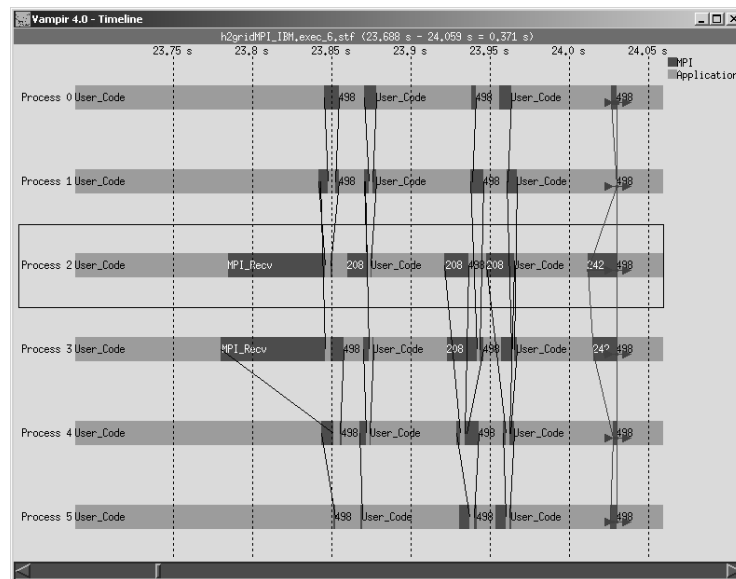


Figure 7: Graphical representation of the point to point communication pattern in H2MOL for 6 processors with MP_CSS_INTERRUPT (from Vampir).

is limited by unnecessary waits in MPI_Recv operations.

Overlapping of communications can also be achieved using non blocking communications. Hence the code has been modified to use MPI_SEND and MPI_RECV operations. The communication pattern for this operation on 6 processors can be seen in Figure 8. This provides an increase in the overall code performance of around 6 percent on 120 processors (See Table 4).

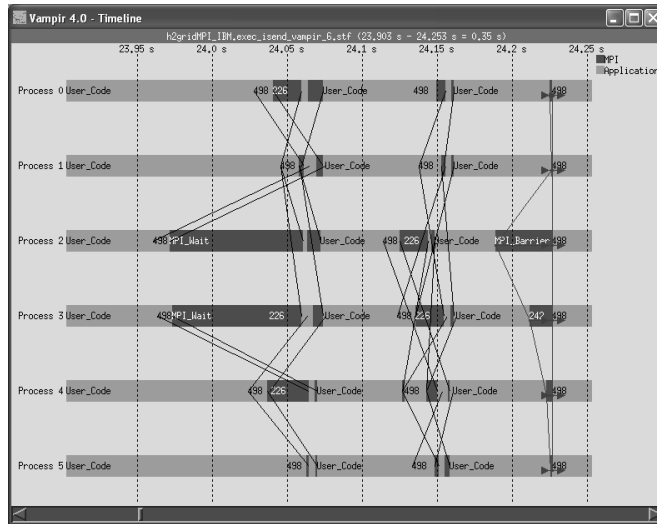


Figure 8: Graphical representation communication pattern in H2MOL for 6 processors for Isend operation (from Vampir).

10 Conclusions

We have investigated the performance of H2MOL on both the Phase 1 and Phase 2 systems of HPCx. This code scales well on both systems. It places heavy demands on the memory bandwidth, a fact that has highlighted the differences in the configuration of the two systems (for example, in the way the processes are allocated on a frame).

The time spent in collective communications is actually a result of time spent in barrier operations - this is a result of the variation in number of processes on each multi-chip module (MCM). Lastly, the point to point communications have been optimised by using non blocking send and receive operations, allowing these communications to be overlapped.

Operation	Time (s)
Number of Processors = 6	
MPI_BSend, MPI_Recv, no css	144.2
MPI_BSend, MPI_Recv, css	134.7
MPI_Send, MPI_Recv, no css	134.5
MPI_Send, MPI_Recv, css	134.2
MPI_Isend, MPI_Irecv, no css	131.1
MPI_Isend, MPI_Irecv, css	132.9
Number of Processors = 120	
MPI_BSend, MPI_Recv, no css	298.6
MPI_BSend, MPI_Recv, css	295.3
MPI_Send, MPI_Recv, no css	302.6
MPI_Send, MPI_Recv, css	302.6
MPI_Isend, MPI_Irecv, no css	279.4
MPI_Isend, MPI_Irecv, css	279.9

Table 4: Timings for H2MOL using different types of point to point communications, with and without MP_CSS_INTERRUPT.

11 Appendix

The following environment variables were setting within the batch script:

```
export MP_SHARED_MEMORY=yes
export MP_EAGER_LIMIT=65536
export MP_USE_BULK_XFER=yes
export MEMORY_AFFINITY=MCM
export MP_TASK_AFFINITY=MCM
export MP_SINGLE_THREAD=yes
export MP_SHM_CC=yes
```

References

- [1] *The University of Edinburgh, U.K.*
<http://www.ed.ac.uk>
- [2] *Edinburgh Parallel Computing Centre (EPCC), The University of Edinburgh, U.K.*
<http://www.epcc.ed.ac.uk>
- [3] *Council for the Central Laboratory for the Research Councils (CCLRC)*
<http://www.cclrc.ac.uk>
- [4] *IBM*
<http://www.ibm.com>

- [5] *The Engineering and Physical Sciences Research Council (EPSRC)*
<http://www.epsrc.ac.uk>
- [6] *ESSL: IBM Engineering and Scientific Software Library*
<http://www.hpcx.ac.uk/support/documentation/IBMdocuments/am501401.pdf>
- [7] *LAPACK – Linear Algebra PACKage*
<http://www.netlib.org/lapack/>
- [8] *IBM Parallel Environment for AIX.*
http://www-1.ibm.com/servers/eserver/pseries/library/sp_books/pe.html
- [9] *MPITrace documentation.*
<http://www.hpcx.ac.uk/support/documentation/IBMdocuments/mpitrace>
- [10] *Visualisation and Analysis of MPI Resources (Vampir), Intel GmbH, Software and Solutions Group.*
<http://http://www.pallas.com/e/products/vampir>
- [11] *Versatile System Resource Allocation and Control (VSRAC), P. Vezolle, J.A. Broyelle, F. Thomas, IBM, internal communication.*