

Mixed Mode Applications on HPCx

Jake Duthie, Mark Bull, Lorna Smith, Arthur Trew
The University of Edinburgh, Mayfield Road, Edinburgh, EH9 3JZ

1 Abstract

Five different OpenMP implementations of a simple Jacobi algorithm have been developed and their performance compared and contrasted. Comparing the best of these codes with an equivalent pure MPI implementation shows that the OpenMP code is considerably faster, due mainly to the use of direct read and writes to memory.

However an equivalent mixed OpenMP / MPI version of the code shows poorer performance than the pure MPI code. While the collective operations are faster, partly due to the less processes being involved in the MPI call, all of the application sections are slower. This is due to the inclusion of the threads shared memory communications and also due to cache problems. In addition, the mixed point-to-point communications are slower than the pure MPI, due to threads having to re-cache data after MPI calls and from communication traffic between nodes becoming more dominant.

2 Introduction

Shared memory architectures have become more prominent in the HPC market, as advances in technology have allowed larger numbers of CPUs to have access to a single memory space. In addition, manufacturers have clustering these SMP systems together to go beyond the limits of a single system. As clustered SMPs such as HPCx have become available, it has become more important for applications to be portable and efficient on these systems.

Message passing codes written in MPI are obviously portable and should transfer easily to clustered SMP systems. Whilst message passing is required to communicate between nodes, it is not immediately clear that this is the most efficient parallelisation technique within an SMP node. In theory a shared memory model such as OpenMP should offer a more efficient parallelisation strategy within an SMP node. Hence a combination of shared memory and message

passing parallelisation paradigms within the same application (mixed mode programming) may provide a more efficient parallelisation strategy than pure MPI.

Whilst mixed mode codes may involve other programming languages such as High Performance Fortran (HPF) and POSIX threads, the most widespread mixed mode codes focus on mixing MPI and OpenMP. In this report we investigate the potential benefits of mixed mode programming for HPCx. We use a simple Jacobi algorithm and develop a range of pure MPI, OpenMP and mixed MPI/OpenMP implementations and compare and contrast their performance.

3 Background

There are many different approaches to mixed mode programming, as noted in Rabenseifner[10]. For example, the two models can operate at the same level in the code, with each model controlling different sections of the code. The main focus of this report however, is to use MPI for coarse-grain parallelism (*i.e.* principal data decomposition), and OpenMP for fine-grain parallelism on each MPI process.

There are several theoretical reasons why mixed mode programs should be faster on Clustered SMP systems than pure MPI. For example:

- Intra-node communication is replaced by theoretically faster direct reads/writes to memory, thereby eliminating the overhead of calling the MPI library.
- Pressure on the interconnect is reduced as messages need only be sent via the interconnect for inter-node communication. In addition, the use of OpenMP can result in less aggregate data being sent across the interconnect, which would certainly improve performance.

These advantages are discussed in Chow & Hysom[5] amongst others. In addition, Smith[1] and Henty[14] point to the following additional advantages:

- Mixed mode codes can reduce the performance hit from an MPI implementation that is unoptimised for intra-node communication.
- Replicated data MPI codes are limited by the memory of a single process, a mixed version may increase this limit to the memory of an SMP.
- MPI codes that have a restriction on the number of processes that they can be used, can be generalised by adding OpenMP threads.

Despite all these proposed benefits to mixed mode programming, works in the literature are mixed in their reports of its performance. Cappello & Etiemble[7], Smith[1], Henty[14], and Chow & Hysom[5] all demonstrate that the pure MPI codes outperform their mixed counterparts irrespective of the underlying architecture. In some cases, this difference can be quite marked [14]. However, He & Ding[8] and Giraud[6] both report improved performance with

mixed codes; the former gaining a factor of 4 increase, but does itself go on to note that this is at variance with much of the rest of the literature.

Rabenseifner[10] points to several problems inherent in mixed mode code that may explain these mixed results. For example:

- The inter-node communication bandwidth may be better utilised by the pure MPI code when compared to its mixed counterpart. For example, the pure MPI code's may overlap its communication traffic on the inter-connect.
- To ensure the code is portable to platforms without a thread safe MPI library, it is necessary to make MPI calls from 1 threads. This often introduces additional synchronisation.

Some of these problems can be avoided by investing more time in the development of the mixed code, typically by using more complex decomposition at the OpenMP level.

While performance studies have been mixed, there are some real industrial codes being developed using mixed mode programming[4].

4 The algorithm

The algorithm used in this study is an inverse of a simple edge-detection algorithm used for image processing; such a kernel is typical of regular domain decomposition codes with nearest-neighbour communication. Given an input 2D $M \times N$ greyscale image, the "edge" pixel values (located at point (i, j)) can be built up from the individual image pixels using the equation:

$$edge_{i,j} = image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - 4 \times image_{i,j} \quad (1)$$

For the opposite approach, taking an edge-type input file and generating the original image, the algorithm then takes the form:

$$new_{i,j} = \frac{1}{4} \times (old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j}) \quad (2)$$

where *edge* is the edge input generated using equation 1 above; *old* is the image value at the start of the current iteration; and *new* is the image value at the end of that iteration.

The algorithm operates over regular 2D input data sets (the edge-type input files), which lends itself naturally to a domain decomposition parallelisation strategy.

Convergence of the *new* values is monitored via a residual value. The iterative process is then terminated when this residual falls below a specified value, indicating the required accuracy has been achieved. The following equation is used:

$$\Delta^2 = \frac{1}{MN} \times \sum_{i=1; j=1}^{i=M; j=N} (new_{i,j} - old_{i,j})^2 \quad (3)$$

and then $\Delta = \sqrt{\Delta^2}$ is calculated to get the residual value.

5 The serial code

The main iteration loop for the image processing algorithm is shown in Figure 1. Algorithm performs the Jacobi algorithm operation, delta the residual calculation and update the update to the current image values in preparation for the next iteration. This loop is run for a fixed number of iterations, or when the residual falls below a specified value, whichever occurs first.

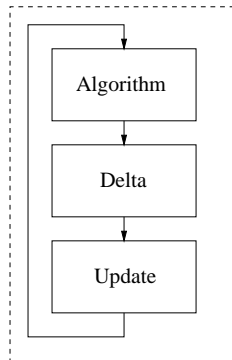


Figure 1: Schematic kernel design for the Jacobi code

6 The parallel code

Several different parallel versions of the code have been developed using pure MPI; pure OpenMP and mixed MPI / OpenMP.

6.1 Pure MPI

The pure MPI code uses a regular 2D data decomposition strategy, with the number of processes in each dimension chosen at compile time. This 2D decomposition is well suited to this problem, as the initial data itself is very regular. Hence the code is well load balanced, and more complex forms of data decomposition are not required.

Point-to-point communications (the `point-to-point` section) are carried out using `MPI_Sendrecv` and the collective operation `MPI_Allreduce` is used

to calculate the residual value (the `collective` section). Timing calls are carried out using `MPI_Wtime`.

6.2 Pure OpenMP

Five different versions of the pure OpenMP code were developed, based on three different methods of decomposing the work over the threads, and two different approaches to designing the parallel construct. The three different work decomposition strategies are shown in Figure 2 and the five versions described below.

6.2.1 Version 1.

The simplest version of the OpenMP code uses 3 `parallel for` directives, one per loop in the kernel, placed on the outermost (i) index.

6.2.2 Version 2.

The second code attempts to minimise thread overhead by placing the entire iteration loop in one `parallel` region, and placing three `for` directives on the computation loops, again over the i index. This may result in some performance improvement due to a reduction in the software bookkeeping required.

6.2.3 Version 3.

In this code, the three `parallel for` directives have been replaced with `parallel` directives. Work decomposition within each parallel region is then carried out by placing `for nowait` directives over the *inner* loops in the double-loop nests. This forces work decomposition over the j index in contrast to the i decomposition employed in the first and second versions. The `nowait` clauses prevent synchronisation at the end of each iteration of i , which would otherwise incur a large amount of unnecessary overhead.

6.2.4 Version 4.

The fourth OpenMP code is designed to use a 2D decomposition strategy across the domain. A separate function has been written that allows the user to specify the number of threads required in each dimension; the function then determines the loop limits for each thread. This procedure is very similar to the data decomposition strategy employed in the pure MPI code, including giving domain-edge threads less work to perform. This version was implemented with three separate `parallel` regions.

6.2.5 Version 5.

Similar to version 4, this code carries out a 2D decomposition across the work domain. It is implemented using only one `parallel` region.

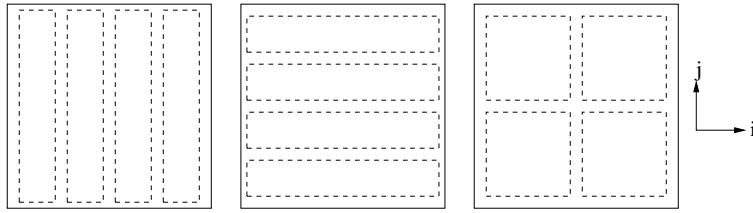


Figure 2: The three decompositions available with the OpenMP codes. The left hand panel describes the i -decomposition for versions 1 and 2. The centre panel shows the j -decomposition for version 3. The right hand panel shows one possible 2D decomposition for versions 4 and 5, although other methods including 1D decompositions are possible.

6.3 Mixed mode

Two different versions of the mixed mode code were developed. Both build on the pure MPI version and use `MPI_Wtime` for the timer calls. For both codes, MPI is used to perform the primary (coarse-grained) data decomposition, with OpenMP threads spawned on each process to carry out the secondary (fine-grained) decomposition. All MPI function calls are made on only one thread, to ensure the code is portable to platforms with non thread safe MPI implementations.

6.3.1 Version 1.

This code uses a 2D MPI decomposition strategy and a 2D OpenMP decomposition strategy (as in the pure OpenMP version 4). This strategy allows for the greatest amount of flexibility in thread/process combinations. By using three separate `parallel` regions all MPI calls are performed outside of an OpenMP parallel region, ensuring the code will run on non threads safe MPI platforms.

This code has three distinct OpenMP barriers, one per `parallel` region. The delta loop's reduction is performed via a local sum on each SMP node using OpenMP, followed by an inter-node sum with MPI.

6.3.2 Version 2.

This code uses a 2D MPI decomposition strategy and a 1D OpenMP decomposition strategy (as in the pure OpenMP version 2). All MPI function calls are carried out inside `master` directives, to ensure portability to non-thread safe MPI platforms.

An explicit OpenMP barrier has been added between the point-to-point and algorithm sections, in order to ensure that all halos have been updated before continuing. The reduction in the delta loop forces a second barrier, and the

update loop requires a third as the old array must be updated before the halo-swaps are made. Note that for both versions of the mixed code a considerably higher degree of synchronisation is required than for the pure MPI code.

7 The HPCx service

HPCx consists of a cluster of 40 IBM p690 SMP frames, each containing 32 IBM Power 4 1.3GHz processors for a total of 1280 processors across the machine, and delivering up to 3.2 Tflop/s sustained performance.

Per frame, the 32 processors are subdivided into 4 Multi-Chip Modules (MCM), each with 8 processors. Each MCM contains 4 chips, with 2 processors per chip. The cache hierarchy of the MCM units reflects this progressive division: each processor has its own Level 1 cache (separate instruction and data caches); each chip has its own Level 2 cache, shared between the 2 processors; and finally each MCM has its own Level 3 cache, shared between the 8 processors. The actual cache sizes are given in Table 1.

Level	Organisation	Capacity
L1	Two-way, 128-byte line	32 KB per processor
L2	Eight-way, 128-byte line	1440 KB per chip; 720 KB per processor
L3	Eight-way, 512-byte line	128 MB per MCM; 16 MB per processor

Table 1: HPCx cache design and hierarchy

Each frame has 32 GB of main memory, shared between 4 MCMs. Each MCM is connected to main memory, and to the other MCMs, via a 4-way bus interconnect. Communication between frames is handled via IBM's SP Colony switch.

HPCx is configured to operate every MCM as a distinct Logical PARTition (LPAR), each with its own copy of the operating system. The main memory on each frame has also been subdivided to match this partitioning, with 8 GB of dedicated memory per LPAR.

Each LPAR runs its own copy of the IBM Unix operating system, AIX 5.1D. Compilation was performed using version 6.0.0.2 of IBM's xlc compiler and the flags: `-q64 -qarch=pwr4 -qtune=pwr4 -O3 -qsmp=omp:noauto` were utilised.

8 Results and analysis

This section presents the results for a test fixed size problem and for a larger benchmark problem including in-depth studies of particular features of the code.

8.1 Fixed problem size

A fixed problem size of 1000×1000 was used, irrespective of the number of processors. This results in a total global problem size of approximately 12 MB. The code was set to run for 10000 iterations.

8.1.1 Pure OpenMP studies

In this section we compare the performance of the different OpenMP codes. Version 3 of the pure OpenMP codes (1D; decomposition across the J index) gave very poor performance when compared to the other versions. This is because data in C is stored contiguously over the innermost array index, hence decomposing across this dimension leads to more cache invalidations. This version is not considered further. The features of the remaining codes are summarised in Table 2.

Version	Code Design
1	3 parallel for directives
2	1 parallel region; 3 for directives
4	3 parallel regions; 2D decomposition
5	1 parallel region; 2D decomposition

Table 2: Tested OpenMP code versions and their features

Figure 3 and 4 show the results for each of these codes, on one LPAR. The numbers shown on the x-axis of each graph correspond to the $M \times N$ directions respectively. Execution times were measured on 7 and 8 processors to assess the potential benefits of leaving one processors free for the operating system (see [2]). It is interesting to note that for all these codes, the 7-process runs are slower than the 8, although the relative scalability is quite poor.

For version 4 and version 5, the best performance is observed for a 1D decomposition over the M direction (outer loop). Again, this is because data in C is stored contiguously over the innermost array index (corresponding to the N direction here).

Version 2 has the best performance, with the 8-thread run outperforming version 4's 8×1 decomposition by around 8%. It should be noted that an 8-thread run of version 2 gives the same work decomposition as an 8×1 use of version 4. Hence the performance improvement cannot entirely be explained by the apportioning of work. One possible solution is that we have reduced the thread overhead in version 2 by creating a parallel region outside the main iteration loop. In addition, it is possible that the OpenMP functions outlined with the `for` directives by the compiler are more suitable for optimisation (under `-O3`) than other functions.

Version 1 has a forced barrier at the end of its section, while version 2 does not. Hence, the performance gain for version 2 might be due to this lack of

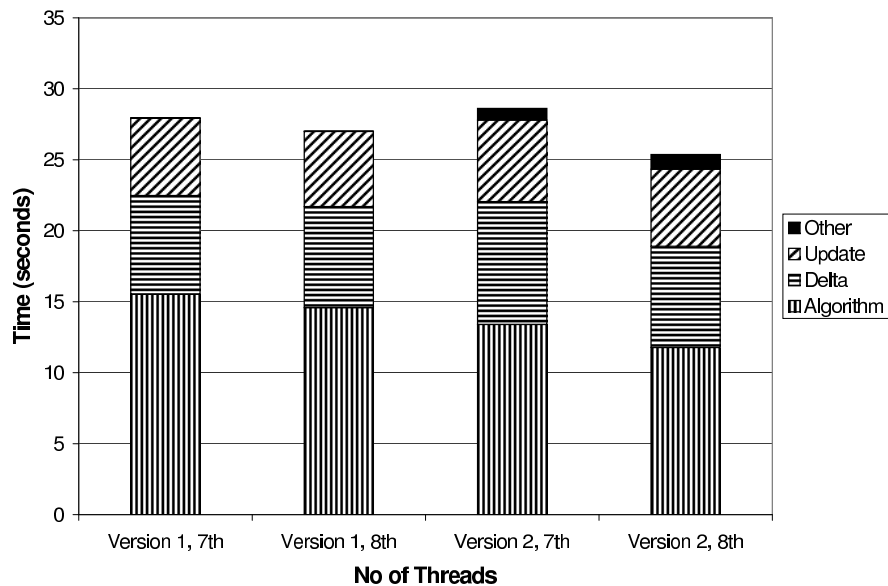


Figure 3: Timer data for pure OpenMP version 1 and 2 runs. Horizontal axis displays the number of threads used

barrier, however similar performance improvement is not observed for version 5, which also has no barrier at the end of the algorithm section.

8.1.2 The MPI code

Figure 5 shows the execution times for the pure MPI code on 1 LPAR. Again, it is interesting to note that the 7-process runs are slower than the 8-process runs, with relatively poor scalability.

The collective section of the code is relatively insensitive to the chosen process decomposition and the computation sections all show some performance improvement with a decomposition strategy over the the M direction, although this is less pronounced than for the OpenMP code. For the point-to-point section, the 8×1 decomposition strategy is 125% faster than the 1×8 . This is because a 1D topology that divides across the I direction send contiguous columns of data, whilst the inverse decomposition must first copy the correct elements into a 1D buffer before communication and unpack this after communication has completed.

Comparing this code to the best OpenMP delta section, shows that the OpenMP delta section is considerably faster than the corresponding MPI section. This is because this routine makes direct read and writes to memory, and avoids the overhead of calling the MPI library. This effect is also observed for the algorithm section of the OpenMP code, which is slightly faster than the

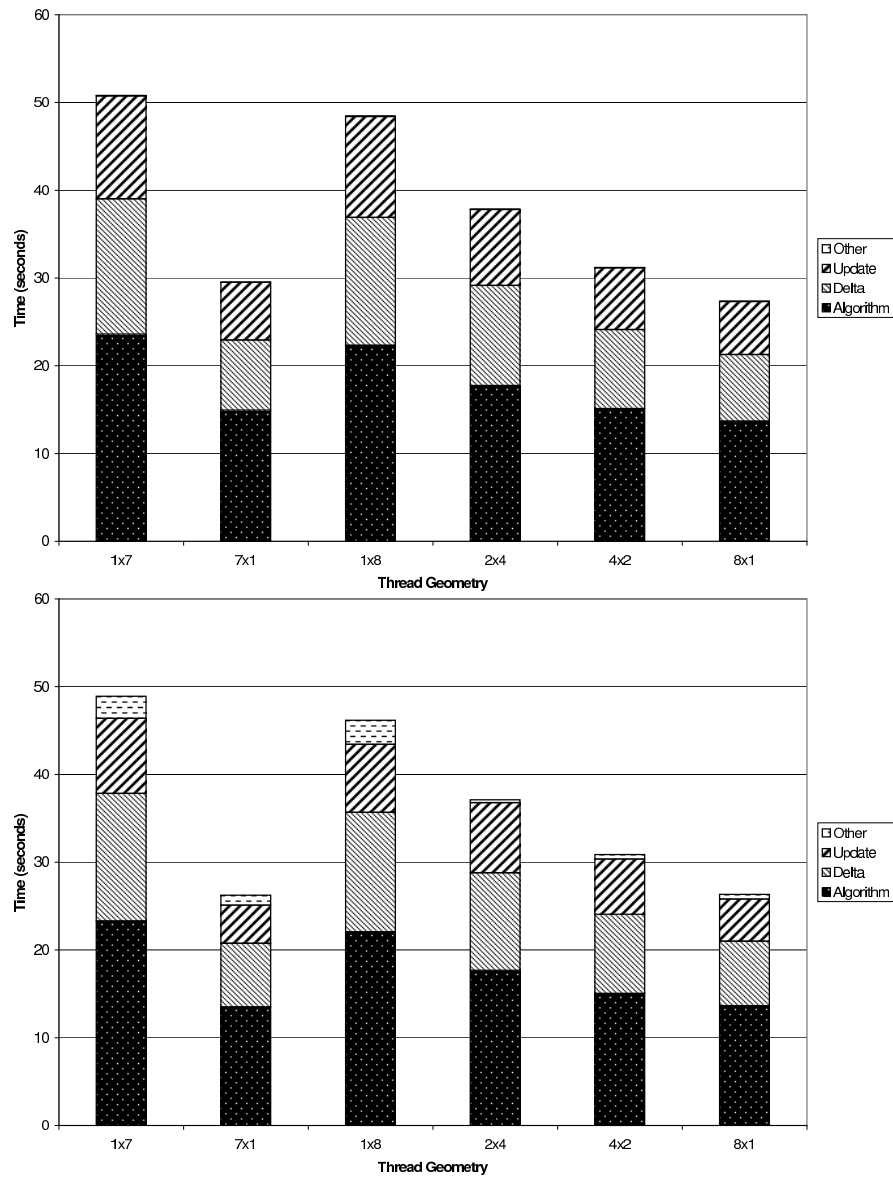


Figure 4: Timer data for pure OpenMP version 4 (top) and 5 (bottom) runs. Horizontal axis displays the given thread layout in 2D

corresponding section in the MPI code.

Hence, while OpenMP has the potential to give quite poor results for particular thread geometries, the best thread geometry outperforms the best pure MPI result by around 5%.

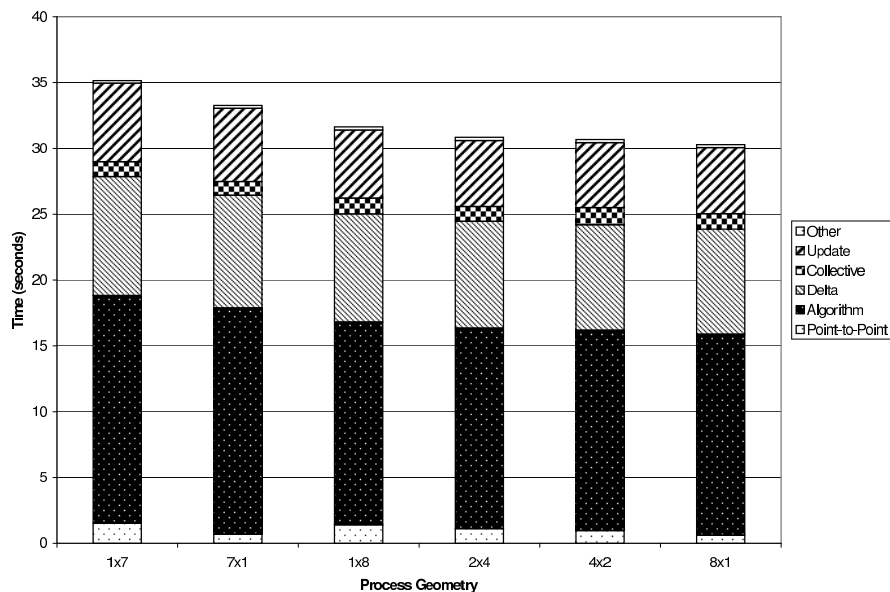


Figure 5: Timer data for the pure MPI code on 1 LPAR. Horizontal axis displays the given process layout in 2D

8.2 Mixed vs. MPI

In this section, the performance of the two mixed code versions (version 1: 2D MPI; 2D OpenMP; 3 parallel regions per iteration and version 2: 2D MPI; 1D OpenMP; 1 parallel region, three `for` directives) has been compared to the pure MPI code. The OpenMP decomposition strategy has been fixed at 8×1 for all runs.

Figure 6 and 7 show the results for 4 LPARs for the pure MPI code and mixed codes respectively, for various processors decompositions. These results demonstrate that version 2 of the mixed code is around 25% faster than version 1, and around 6% faster than the pure MPI code.

The point-to-point section for all the codes favours 1D decompositions over 2D, and favours decomposition over the M dimension. For the pure MPI code, the 2D arrangements are slower due to the underlying process-to-LPAR structure. For example, the 32×1 decomposition appears as repeated blocks of the structure shown in Figure 8. For this decomposition, only one inter-node communication is necessary between LPARs; all the remaining point-to-point communications occur within an LPAR. The situation for the 1×32 decomposition is similar, although now all communications are non-contiguous. In contrast, an 8×4 decomposition involves a different process topology, shown in Figure 9. Here it is now necessary for 4 inter-node communications take place per LPAR, reducing the point-to-point section performance.

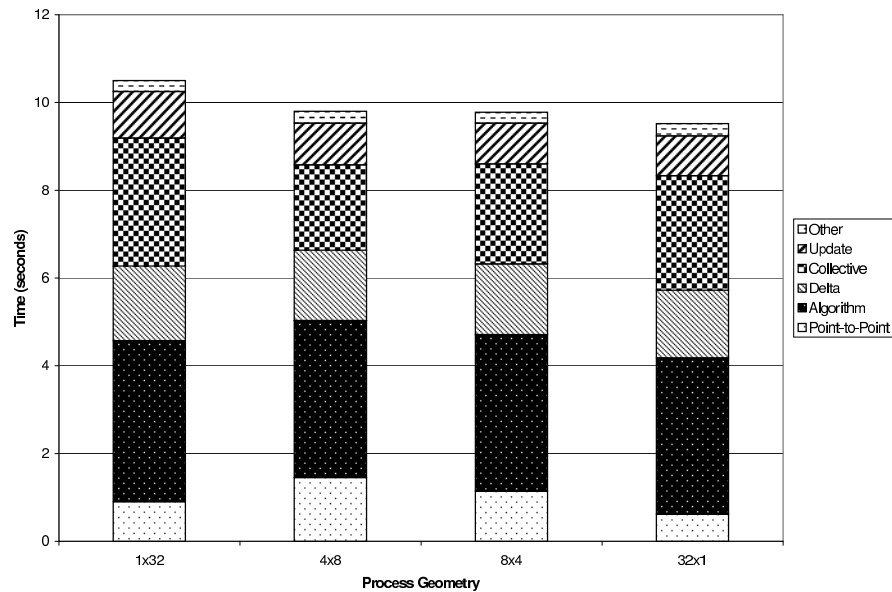


Figure 6: Timer data for the pure MPI code on 4 LPARs. Horizontal axis displays the given MPI process layout in 2D

The situation regarding the mixed codes' point-to-point sections is not as clear, and requires further investigation. As all communication traffic occurs over inter-node boundaries in the "1 process per LPAR" model, we would expect the fully non-contiguous send/receive layout to give the worst performance. However this is not the case.

All three computation loops are slower for the mixed code version 1, when compared to the pure MPI code. Comparing like with like, the best mixed loop (delta) is around 25% slower than its MPI counterpart, with the worst (update) being closer to 75% slower. These differences are too large to be due solely to the inclusion of additional memory accesses, but may be due to the overhead generated by the `parallel` regions. To test this supposition further, a mixed run on 4 LPARs was attempted using only 1 thread per process and allocated as many processes as processors: in effect a pure MPI run with the addition of the `parallel` overhead. The same process decompositions were chosen as for the pure MPI run, and the results are presented in Figure 10.

Comparing this graph to Figure 6, it is clear that the three computation sections are slower with this mixed version even with the same MPI process topology. In addition, the three computation sections show some improvement between the 8-thread and 1-thread mixed runs, demonstrating that replacing threads with processes improves performance. Since the code is essentially doing the same work, this suggests that the overhead from the three `parallel` regions is at least partly responsible for the poorer mixed performance.

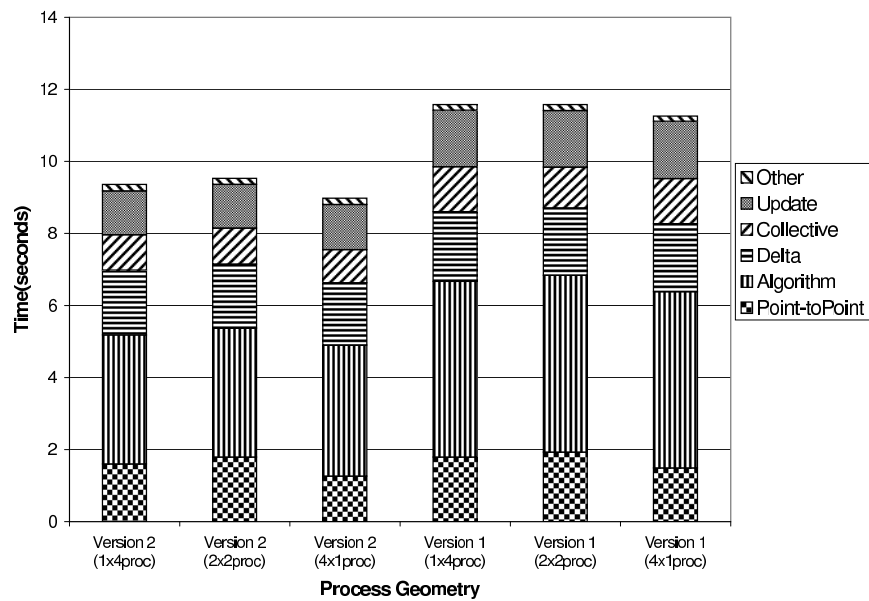


Figure 7: Timer data for version 2 mixed and version 1 mixed on 4 LPARs. Horizontal axis displays the given process layout in 2D.

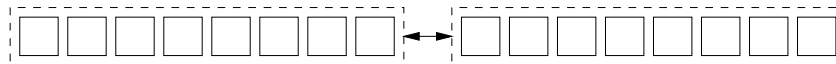


Figure 8: Representation of the process-topology for a 1D problem. Dashed blocks indicate LPAR boundaries; solid squares are individual processes.

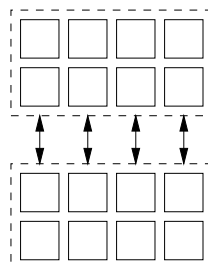


Figure 9: Representation of the process-topology for a 2D problem. Dashed blocks indicate LPAR boundaries; solid squares are individual processes.

This would imply that version 2 of the mixed codes should show better performance for this section, as there is only 1 parallel region. This is indeed

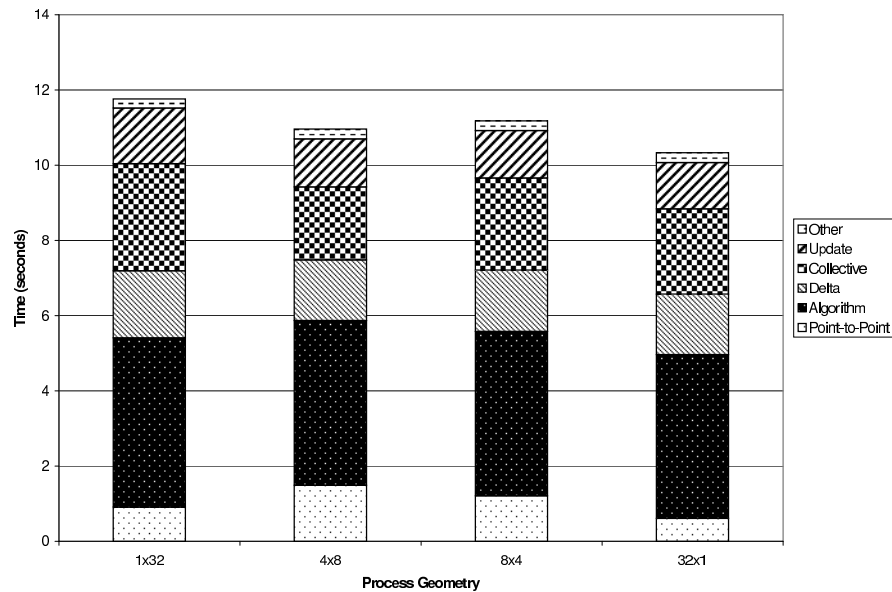


Figure 10: Timer data for mixed runs performed on 4 LPARs with 1 thread per process. Horizontal axis displays the given process layout in 2D

the case. To further test this hypothesis, a run on 4 LPARs with only 1 thread per process for version 2 was performed, in order to see how large the threaded sections overhead contributed to the runtime. This graph is shown in Figure 11.

Comparing this graph to Figures 6 and 10, we can now see that thread overhead has been eliminated from the algorithm section. The delta and update loops are still slower for the mixed code, but less so than before. The algorithm saving is the biggest performance benefit from version 2.

8.3 Scaling problem size

In this section we consider a problem with $M \times N$ equal to 450×450 per processor, a local problem size of approximately 2.43 MB. This exceeds the L2 cache per processor (720 KB), and fits comfortably into L3 (16 MB per processor). If this code scales ideally, all runs would take exactly the same amount of time to execute. The code was fixed to run for 5000 iterations, by setting the convergence tolerance to an extremely small value of 1×10^{-6} .

Note that in all cases, version 2 of the mixed code and version 2 of the pure OpenMP code were used for the analysis.

The results are presented first in order of increasing processor numbers, subdivided into **Small** (1 LPAR including pure OpenMP), **Medium** (2 and 4 LPARs), and **Large** (8 and 16 LPARs). A more in depth study of some specific

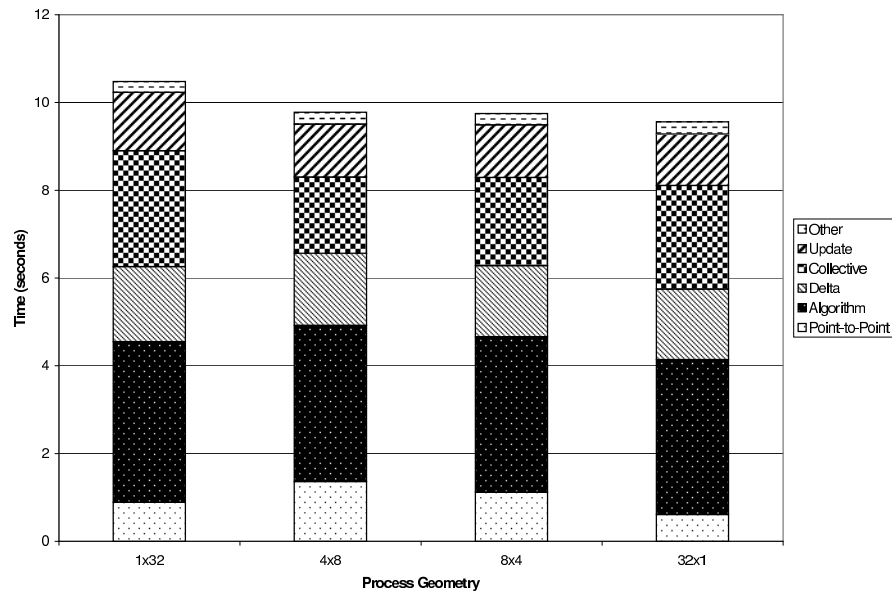


Figure 11: Timer data for a version 2 mixed run performed on 4 LPARs with 1 thread per process. Horizontal axis displays the given process layout in 2D.

features follows later.

8.3.1 Small

Graphs of the results for the OpenMP, mixed, and MPI codes on 1 LPAR are shown in Figure 12.

The pure OpenMP code gives the best performance, with an average runtime approximately 17% faster than either the mixed or MPI codes. The total runtime for the mixed and best (4×2) MPI code are very similar. As the (1 process; 8 thread) mixed code is performing no MPI communication on 1 LPAR, the increase in execution time from the OpenMP code is due solely to the computational sections.

The computation sections of the mixed and pure OpenMP codes are implemented in exactly the same way, however the algorithm and delta loops display performance drops of 25% and 23% respectively when moving to the mixed code. One possible explanation relates to compiler optimisations. The upper loop bounds in the OpenMP code are set by `#defines` M and N , but in the mixed they are governed by the `int` variables MP and NP . Hence the compiler may produce faster code when the upper loop bounds are known at compile time, as is the case with the OpenMP code.

Comparing the mixed code with the best MPI code, the delta and update sections are slower by 10% and 20% respectively. This drop in performance

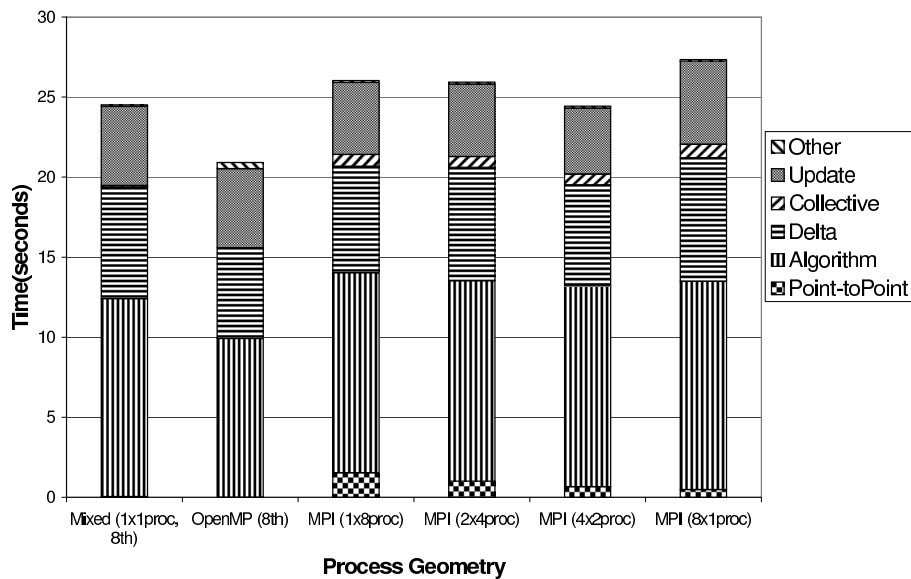


Figure 12: Timer data for OpenMP, MPI and mixed codes for the L3 scaling problem size on 1 LPAR. Horizontal axis displays the number of threads and processes used.

appears to be too large to be due solely to inter-thread communication.

Overall, the 1 LPAR results show that the performance improvement observed for the OpenMP code does not translate to a performance improvement for the mixed code.

8.3.2 Medium

Graphs of the mixed and MPI performance for 4 LPARs are shown in Figure 13.

The best mixed code decomposition is 4×1 , with the performance improvement due to the reduction in point-to-point time due to all-contiguous sends. The best MPI decompositions are 8×4 on 4 LPAR numbers.

Comparing the two codes, the best mixed results are slightly slower than the best pure MPI. In addition to the computation sections running slower for the mixed code, the point-to-point section is also slower. Only the collective communications section shows a performance improvement for the mixed code.

Finally, the scalability of the code is quite good, with the 1 LPAR to 4 LPAR runtimes only increasing by around 10%.

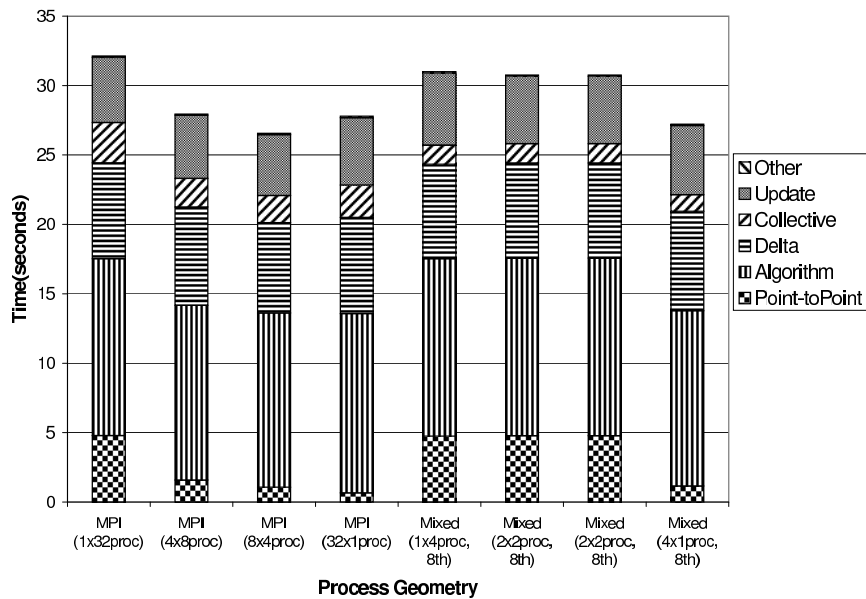


Figure 13: Timer data for mixed and MPI runs on 4 LPARs. Horizontal axis displays the given MPI process layout in 2D.

8.3.3 Large

Results for 16 LPARs are given in Figure 14. Note that these jobs were run for 2000 iterations, in order to reduce the runtime. The data has then been scaled up to be comparable with a 5000 iteration run.

The mixed and MPI times are very similar with the best mixed code run-times slightly faster than the best MPI runtimes (but not significantly).

The best mixed decomposition is 16×1 , due mainly to the faster point-to-point, as before. The mixed computation sections continue to show no real trend towards a favoured topology.

The point-to-point section shows the same behaviour as in all previous cases, with the mixed times slower than the MPI.

The mixed code only shows real performance gains in its Collective section with its computation sections and the properly-synchronised point-to-point all showing an increase in runtime compared to the pure MPI code. The next two sections are intended to explain these trends.

8.3.4 Point-to-point communication study

To investigate the point-to-point behaviour further, a 4 LPAR mixed run was performed, first with 2 processes per LPAR and 4 threads per process, and then with 4 processes per LPAR and 2 threads per process. For both cases, only the

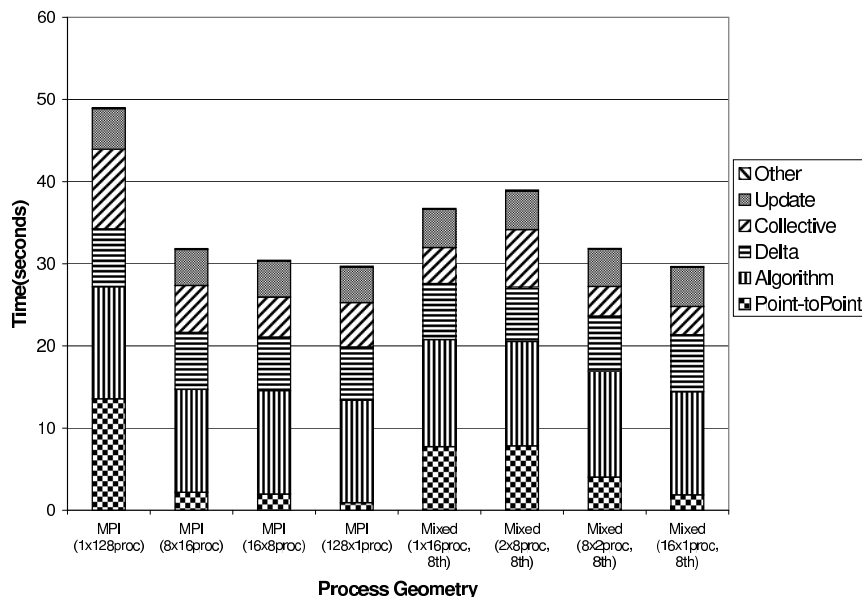


Figure 14: Timer data for mixed and MPI runs on 16 LPARs. Horizontal axis displays the given MPI process layout in 2D.

predicted “best” process topology was used (8×1 and 16×1 respectively) based on the previous data.

These figures were then compared to the best runtime for the original 1 process and 8 thread mixed model and the pure MPI code from the 4 LPAR studies made earlier. This data is given in Figure 15.

This graph demonstrates that as the number of MPI processes in the mixed code is increased, and hence the number of point-to-point communications is increased, the execution time for these communications decreases. This is not what we would expect.

To help explain this, the point-to-point sections of the codes have been instrumented using the `libhpm` version of HPM [20]. HPM was used to monitor the usage of the L2 and L3 caches and main memory, with data recorded for each MPI process. Given that the actual problem resides in L3, one would expect most of the memory traffic to occur in L3. Some L2 usage would also be expected, but since L3 is many times slower in terms of access speed than L2, it is expected that L3 use would completely dominate the time spent in this section.

Instead, these code sections are dominated by unexpected main memory traffic. This traffic appears on the MPI processes in a very distinct pattern as can be seen in Figure 16.

Time spent in `Sendrecv` calls is not solely due to communication between processors, but also includes time spent gathering the data to be sent, and plac-

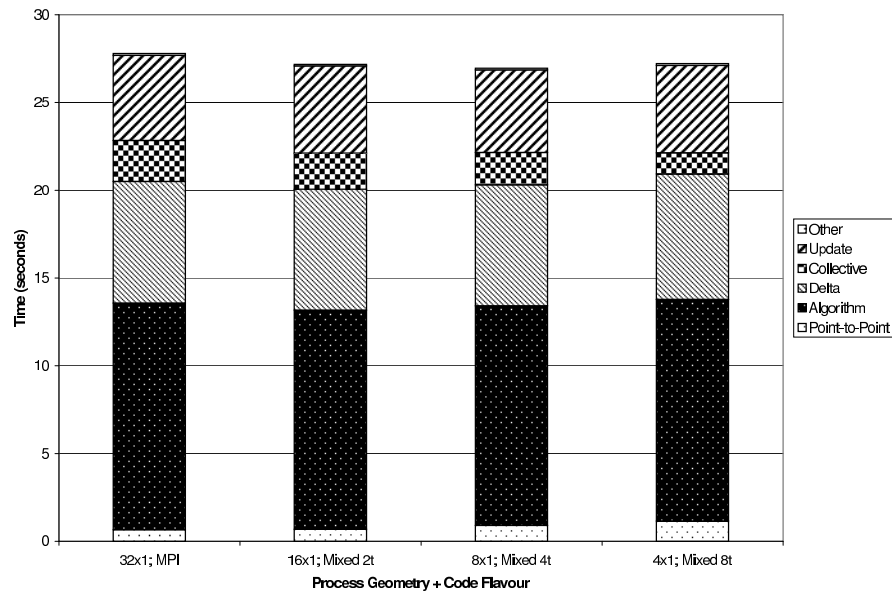


Figure 15: Timer data for differing process/thread combinations for the mixed code for 4 LPAR runs, and a corresponding pure MPI run. Horizontal axis displays the given MPI process layout in 2D, along with the number of threads per process used if referring to a mixed run.

ing this data in the correct halo after the receive. For the pure MPI run, the load usage is mostly flat, with spikes corresponding to processes that lie at the edge of an LPAR (recall that these are 1D process decompositions), possibly indicating the passage of data through the switch and then through the shared memory subsystem (and vice versa). On-LPAR communication is handled entirely in shared memory for point-to-point, and this may account for the base-level of memory loads on these processes.

For the mixed code, as the number of processes engaged in on-node communications decreases (and the number of threads increases), the “spikes” becoming progressively more dominant. This is due to the corresponding increase in main memory loads, which are extremely slow.

In addition, data locality also reduces the mixed code performance. As the mixed code only calls MPI functions on the master thread, the mixed code used has the data-to-communication pattern shown in Figure 17.

The master thread will first have to obtain the data to be sent from the cache of the processor running the edge-thread for the case of left/right sends, or from the caches of all the other threads in the case of up/down sends. This will take longer than a comparable process involved in a send, as it will already have the necessary data stored in cache.

Hence the mixed mode code’s point-to-point communication is slower for

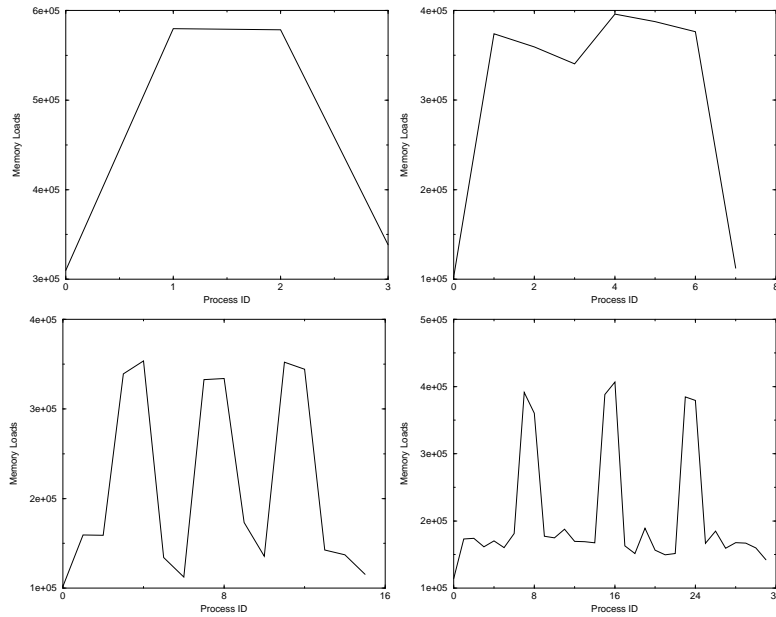


Figure 16: Line graphs showing the total number of main memory loads recorded on each MPI process for mixed (1p x 8t) (top left), mixed (2p x 4t) (top right), mixed (4p x 2t) (bottom left), and MPI (bottom right) on 4 LPARs.

two reasons. Firstly, there is a larger amount of memory traffic, and secondly a cache locality issue. Both problems could be avoided by redesigning the code to allow individual threads to communicate, as if they were processes, over LPAR boundaries by manually coding in methods for all thread IDs to be unique and all threads to be aware of their neighbours. This would bypass the second problem completely and reduce the first problem down to the pure MPI memory load behaviour.

8.3.5 Computation sections study

In this section we consider the performance characteristics of the computation sections in the mixed mode code. Each of the three sections had been instrumented with HPM instrumentation calls.

HPM reported that each computation section of the kernel spent some considerable time loading from main memory, despite the fact that the entire problem was designed to fit comfortably into L3 per processor. There were more loads taking place when threads were in use, and since memory loads are so expensive, this explains the poor performance observed for the mixed code on these sections.

However, the code should not be accessing main memory at all in these

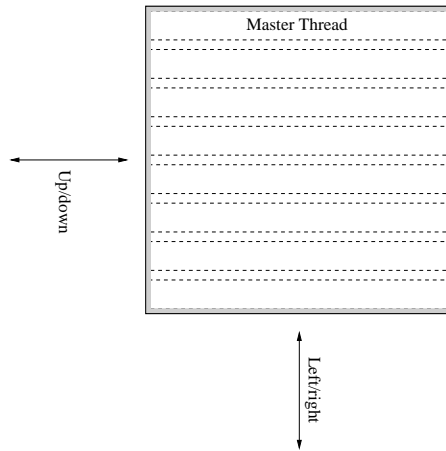


Figure 17: Representation of the relationship between thread data locality and the MPI communication pattern for a mixed code. The shaded area indicates the data halos on the process.

sections. For example, with the update section, all the memory traffic related to this operation should be taking place in L3. Instead, about 15% of the loads for the processes, and up to about 25% of the loads for the threads, are going to main memory. This equates to hundreds of loads from memory per thread/process *per iteration* of the kernel.

To investigate this cache behaviour, a code has been written which simply declares two 1D `float` arrays, fills them with random data, and adds each element together. The loop performing this addition was instrumented with HPM, and the code run on a single processor with varying total array sizes. It was compiled with the same options as the Jacobi code, for consistency. The results are given in Table 3.

Total Problem Size	L2 Loads	L3 Loads	Memory Loads
200 KB	15708880	10	0
1200 KB	91757819	1753632	538982
10 MB	701858669	68308725	17535457
100 MB	6917507218	528641734	432049931

Table 3: HPM cache/memory data obtained from the simple array-addition code, for varying total problem sizes.

The problem sizes were chosen to fit comfortably into L2 (200 KB), fill most of L2 (1200 KB), fit comfortably into L3 (10 MB) and fill L3 (100 MB) – recall that this code only uses one processor on an LPAR, but gets the entire LPAR to itself hence 1440 KB of L2 and all 128 MB of L3 are available.

This HPM data shows that the only the smallest problem size behaves as expected. When the data has expanded to fill L2, over a million loads from L3 and hundreds of thousands from main memory occur. As the problem increases into L3 the situation is worse, and when L3 is mostly full the number of loads going to L3 and main memory is roughly 50/50.

This “cache leak” is fundamentally a hardware problem; however the leakage is more apparent with OpenMP threads than with MPI processes, as was seen with the Jacobi code.

This cache leak problem can be partly explained: the L3 cache can choose not to retain new data if it is already highly utilised - it instead acts as a buffer for the main memory. However this does not explain why the effect should be so evident with problem sizes that have plenty of space in L3, nor why it appears in L2 use.

8.4 Summary

Five different OpenMP implementations of the Jacobi algorithm have been developed and compared and contrasted. Of these, the code using a single parallel region and a 1D decomposition strategy over the outer loop gave the best performance. This is due to a number of factors, such as reduces overhead from parallel regions, good cache use and improved compiler inlining of OpenMP functions.

Comparing this code with the pure MPI implementation, shows that the OpenMP code is considerably faster, due mainly to the use of direct read and writes to memory, hence avoiding the overhead of calling the MPI library.

This suggests that a mixed OpenMP / MPI version of the code might give better performance than the pure MPI code. However, the best mixed code shows poorer performance than the MPI code. While the collective operations are faster in the mixed code, partly due to the less processes being involved in the MPI call, all of the computation sections are slower. This is due to the inclusion of the thread’s shared memory communications and also due to a cache leak (described above). In addition, the mixed point-to-point communications are slower than the pure MPI, due to threads having to re-cache data after *masteronly* MPI calls and from cross-LPAR communication traffic becoming more dominant.

References

- [1] L.A. Smith; *Mixed Mode MPI/OpenMP Programming*; Technical Report Technology Watch 1, UKHEC; 2000.
<http://www.ukhec.ac.uk/publications/tw/mixed.pdf>
- [2] J. Hein and M. Bull; *Capability Computing: Achieving Scalability on over 1000 Processors*; HPCx Technical Report HPCxTR0301; 2003.

http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0301.pdf

- [3] D. Klepacki; *Mixed-Mode Programming*; IBM ACTC Workshop for Power3 presentation, SP SciComp '99; 1999.
<http://www.mhpcc.edu/doc/ACTC/MpiOpenMP.pdf>
- [4] D. Salmond; *ECMWF's IFS on an IBM p690 system with 960 Power4 processors*; ECMWF presentation at Third UKHEC Annual Seminar; 2002.
<http://www.ukhec.ac.uk/events/annual2002/salmond.pdf>
- [5] E Chow and D. Hysom; *Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters*; Lawrence Livermore National Laboratory technical report UCRL-JC-143957; 2001.
<http://www.llnl.gov/CASC/people/chow/pubs/hpaper.ps>
- [6] L. Giraud; *Combining Shared and Distributed Memory Programming Models on Clusters of Symmetric Multiprocessors: Some Basic Promising Experiments*; The International Journal of High Performance Computing Applications, Volume 16, No. 4, pp 425-430; 2002.
<http://www.paulchapmanpublishing.co.uk/journals/details/issue/sample/a031089.pdf>
- [7] F. Cappello and D. Etiemble; *MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks*; in the proceedings of SC2000, Dallas; 2000.
<http://www.sc2000.org/techpaper/papers/pap.pap214.pdf>
- [8] Y. He and C.H.Q. Ding; *An Evaluation of MPI and OpenMP Paradigms for Multi-Dimensional Data Remapping*; in the proceedings of WOMPAT 2003, LNCS 2716, pp 195-210; 2003.
- [9] S. Bova, C. Breshears, R. Eigenmann, H. Gabb, G. Gaertner, B. Kuhn, B. Margo, S. Salvini, and V. Vatsa; *Combining Message-passing and Directives in Parallel Applications*; SIAM News, Volume 32, Number 9; 1999.
<http://www.siam.org/siamnews/11-99/mpi.pdf>
- [10] R. Rabenseifner; *Hybrid Parallel Programming: Performance Problems and Chances*; in the proceedings of 45th CUG Conference, Columbus; 2003.
http://www.hlrs.de/people/rabenseifner/publ/cug2003_hybrid_1.pdf
- [11] R. Rabenseifner and G. Wellein; *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures*; The International Journal of High Performance Computing Applications, Volume 17, No. 1, pp 49-62; 2003.
http://www.hlrs.de/people/rabenseifner/publ/hybrid_HPCA_3.pdf

- [12] R. Rabenseifner; *Communication Bandwidth of Parallel Programming Models on Hybrid Architectures*; in the proceedings of WOMPEI 2002, part of ISHPC-IV, Kyoto; 2002.
http://www.hlrs.de/people/rabenseifner/publ/hybrid_wompei2002final.pdf
- [13] A. Jackson; *Mixed-Mode Parallelisation of a Discrete Element Model*; MSc in HPC Dissertation, University of Edinburgh, UK; 2002.
- [14] D.S. Henty; *Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling*; in the proceedings of SC2000, Dallas; 2000.
<http://www.sc2000.org/techpaper/papers/pap.pap154.pdf>
- [15] *HPCx Capability Computing*;
<http://www.hpcx.ac.uk>
- [16] *ASCI White*;
<http://www.llnl.gov/asci/platforms/white>
- [17] *ASCI Purple*;
<http://www.llnl.gov/asci/purple>
- [18] *MPI: A Message-Passing Interface Standard*; Message Passing Interface Forum, June 1995; <http://www.mpi-forum.org>
- [19] *OpenMP C and C++ Application Program Interface*; OpenMP ARB, March 2002; <http://www.openmp.org>
- [20] *Hardware Performance Monitor (HPM) Toolkit*;
<http://www.hpcx.ac.uk/support/documentation/IBMdocuments/HPM.html>