Mixed Mode Programming on Clustered SMP Systems.

Michal Piotrowski

August 25, 2006

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2006
Abstract

Clusters of shared memory nodes are most common and popular HPC solution. Parallel programming on this machines must combine the distributed memory parallelisation between the nodes with the shared memory parallelisation inside of each node. Most of these systems are programmed using the message passing paradigm. The Mixed Mode programming model could potentially exploit features of the SMP cluster architecture, thus resulting in a more efficient parallelisation strategy. It could possibly combine advantages of both OpenMP and MPI parallelisation paradigms.

Three different Mixed Mode versions were developed and compared to the pure MPI implementation. The results demonstrate superiority of the pure MPI over the Mixed Mode style in almost all parts of the benchmark. The only part were the Mixed Mode version shows better performance is collective communications.
Contents

1 Introduction ................................................. 1

2 Background ................................................. 3
  2.1 Parallel architectures .............................. 3
  2.2 Programming models ............................... 5

3 Benchmark codes ................................ .......... 10
  3.1 Jacobi Relaxation Algorithm ...................... 10
  3.2 Design and implementation details .............. 11
    3.2.1 Serial .............................................. 11
    3.2.2 Pure MPI ........................................... 13
    3.2.3 Master-Only ....................................... 16
    3.2.4 Funnelled ......................................... 17
    3.2.5 Multiple ........................................... 18

4 Results and analysis ..................................... 21
  4.1 Methodology .......................................... 21
    4.1.1 Hardware: HPCx Architecture .................. 21
    4.1.2 Software ........................................... 23
    4.1.3 Experimental procedure ....................... 24
  4.2 Pure MPI code .......................................... 25
  4.3 Pure MPI vs. Mixed-Mode ......................... 28
    4.3.1 Master-Only ....................................... 28
    4.3.2 Funnelled ......................................... 34
    4.3.3 Multiple .......................................... 36
  4.4 Simultaneous Multithreading ...................... 38

5 Conclusions ................................................. 41

A Experimental data ........................................ 43
List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Comparison of the HPCx Phase 2 and Phase 2a systems</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>Selected data gathered using the \texttt{hpmcount} tool</td>
<td>30</td>
</tr>
<tr>
<td>A.1</td>
<td>Different process topologies performance on 8 LPARs</td>
<td>43</td>
</tr>
<tr>
<td>A.2</td>
<td>Different process topologies performance on 16 LPARs</td>
<td>43</td>
</tr>
<tr>
<td>A.3</td>
<td>Master-Only version - 8 LPARs</td>
<td>44</td>
</tr>
<tr>
<td>A.4</td>
<td>Funnelled version - 8 LPARs</td>
<td>44</td>
</tr>
<tr>
<td>A.5</td>
<td>Multiple version - 8 LPARs</td>
<td>44</td>
</tr>
<tr>
<td>A.6</td>
<td>Master &amp; pure MPI versions - 8 &amp; 16 LPARs</td>
<td>44</td>
</tr>
<tr>
<td>A.7</td>
<td>Funnelled &amp; pure MPI versions - 8 &amp; 16 LPARs</td>
<td>44</td>
</tr>
<tr>
<td>A.8</td>
<td>Multiple &amp; pure MPI versions - 8 &amp; 16 LPARs</td>
<td>45</td>
</tr>
<tr>
<td>A.9</td>
<td>Simultaneous Multithreading - 8 LPARs</td>
<td>45</td>
</tr>
<tr>
<td>A.10</td>
<td>Simultaneous Multithreading - 16 LPARs</td>
<td>45</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Distributed memory architecture model ........................................ 3
2.2 Shared memory architecture model ............................................. 4
2.3 SMP cluster architecture model ................................................. 5
2.4 Mixed-mode programming model for a 2D array. [1] .......................... 7

3.1 Algorithm flow diagram - Serial version ..................................... 13
3.2 MPI Distribution ........................................................................ 14
3.3 Algorithm flow diagram - pure MPI version ................................. 15
3.4 3D array memory layout in C ..................................................... 15
3.5 MPI Communication ..................................................................... 16
3.6 OpenMP decomposition ............................................................... 17
3.7 Algorithm flow diagram - Funnelled version ............................... 18
3.8 Inter node communication - Multiple version .............................. 19
3.9 Algorithm flow diagram - Multiple version .................................. 20

4.1 Modifications to Power4 system structure. (P: processor, FBC: fabric bus controller, L2, L3: Level 2, Level 3 caches (reprinted from [11])) ........... 22
4.2 Pure MPI performance comparison for different process geometry. Timer data for runs performed on 8 LPARs (top) and 16 LPARs (bottom) ............. 26
4.3 Performance results of the Pure MPI and Master-Only runs on 8 LPARs ................................................................. 29
4.4 Performance results of the Master-Only and Pure MPI runs on 8 LPARs - with compiler optimisation disabled .................................................. 31
4.5 Performance results of the Pure MPI and Master-Only runs on 8 and 16 LPARs ................................................................. 34
4.6 Performance results of the Pure MPI and Funnelled runs on 8 LPARs ................................................................. 35
4.7 Performance results of the Pure MPI and Funnelled runs on 8 and 16 LPARs ................................................................. 36
4.8 Performance results of the Pure MPI and Multiple runs on 8 LPARs ................................................................. 37
4.9 Performance results of the Pure MPI and Multiple runs on 8 and 16 LPARs ................................................................. 38
4.10 SMT benchmark results for 8 LPARs (top) and 16 LPARs (bottom) ................................................................. 40
Acknowledgements

Most importantly I would like to thank my supervisor Mark Bull for his guidance, support, always cheerful attitude and infinite patience shown over the duration of the project.

I would also like to thank all fellow students for the great atmosphere they have created during this year.

Special thanks to Jackson Pauls without whom I probably would not manage to complete this studies.
Chapter 1

Introduction

Clusters of shared memory compute nodes, although not as efficient as Massively Parallel Processors (MPP), have became a very popular HPC solution in recent years. This is mainly due to their relatively low cost and availability, compared to the special-purpose MPP systems. The HPCx system, one of the biggest HPC systems in UK (listed at 59 place in top500 list [7] - August 2006), used by leading research groups, is an SMP cluster architecture. Furthermore, some of the large MPP manufacturers, are starting to replace single processors in their nodes with small SMP nodes (e.g. Blue Gene/P project).

With respect to the memory architecture, two different parallel programming strategies are used. For shared memory machines, where all processors are connected to the single shared memory space, a shared memory programming model is often used (e.g. OpenMP). Distributed memory architectures are typically programmed using message passing model (e.g. MPI). Both paradigms have their strong sides and weaknesses. For example the shared memory model is exceptionally good at handling load-imbalanced problems. SMP clusters are combination of shared memory (within SMP node) and distributed memory (cluster) architectures, thus both programming models could be applied and merged into a so called “mixed mode version” (also called “hybrid programming”). However, the vast majority of codes written for SMP clusters use pure MPI. Memory is treated as a fully distributed, ignoring the specific machine architecture.

The aim of this project is to investigate how a mixed mode programing performs on the HPCx system. Mixed mode code (we will focus on MPI/OpenMP) could potentially exploit features of the SMP cluster architecture, thus resultig in a more efficient parallelisation strategy. It could possibly combine advantages of both parallelisation paradigms in order to improve performance on a single SMP.

Three possible mixed mode programming styles will be compared, that is: Master Only, Funneled and Multiple. In master only style MPI communication is handled outside OpenMP parallel regions by the master thread. In Funneled model inter-node communication is still performed by a single thread, however MPI routines are called within parallel region. In the most sophisticated Multiple version, MPI communication routines are called simultaneous by more than one thread. Appropriate benchmark codes, based on a simple Jacobi algorithm
were developed. Computation and communication performance of the codes was compared to the corresponding results of the pure MPI version.

A new feature of the IBM Power5 processor - Simultaneous multi-threading (SMT) - will be investigated. SMT permits multiple independent threads to simultaneously access the resources of the processor, this might effect in better utilisation of available processor cycles. We have measured if and how this new technology improves both pure MPI and the mixed mode model.

Experience of past projects concerning mixed mode programming ([1], [2], [5], [6]) shows that producing well optimised and efficient hybrid code that will outperform pure MPI version is a challenging task. According to [6], the main difficulty may lie in the poor inter-thread communication efficiency; it is pointed that overlapping intra-node with inter-node communication could be particularly hard.

The report is structured as follows.

Chapter 2 defines the mixed mode programming model and explains the motivation behind this project in context of the SMP architecture and mainstream parallel programing paradigms applied in high performance computing industry.

Chapter 3 covers implementation details of the all developed benchmark codes. Briefly explains the Jacobi Relaxation Algorithm, which makes up the the core the benchmark implementations.

Chapter 4 describes the experimental procedure that was applied. Benchmark results presentation is followed by the in-depth analysis.

Chapter 5 ends the report with project conclusions and analyses.
Chapter 2

Background

2.1 Parallel architectures

The variety of systems developed for high performance computing makes parallel architectures classification difficult task. One of the major classification method is based on the system’s different memory architectures. This method is particularly useful when we try to understand underlying elements of different parallel programming paradigms.

Distributed memory

In distributed memory architecture (shown on Fig. 2.1) each processor has its own memory and each node (processor and memory) is connected to an interconnect network. Processors cannot directly access the memory of a remote node. In order to exchange data, processors have to use explicit message passing. The time needed for an access to the remote memory may vary depending on the node’s physical location; local memory is typically accessed much faster than remote memory.

Figure 2.1: Distributed memory architecture model
Figure 2.2 shows an example of a shared memory system often called Symmetric Multi-
processing, or SMP. In this multiprocessor system architecture multiple identical processors
are connected via the interconnect network to a single shared main memory. Therefore, aver-
age memory access time is identical for all processors, and is not connected to the processor’s
physical location. It is generally easier to program a shared memory machine, as programmer
doesn’t have to deal with explicit message passing. In SMP system any processor can work on
any task no matter where the data is located in memory. Tasks can be easily moved between
processors, hence the workload of the SMP system can be balanced efficiently. However,
shared memory systems do not scale very well, mainly due to limited memory bandwidth.
Shared memory programs may also suffer from bottlenecks, where many threads try to access
the same location in the memory.

SMP Cluster

An SMP Cluster (shown on Fig. 2.3) is a combination of the previous two architectures. It is
a collection of shared memory systems connected via interconnect network; effectively it is
a distributed system, where a single processor node is replaced by SMP node.

This architecture could in theory combine advantages of both distributed and shared memory
systems. The problem of the limited scalability of the SMP system is solved. Now whenever
we would like to enlarge the system, we simply plug in another SMP node to the interconnect.
This is very convenient and flexible solution.

Explicit messages could be send only between nodes and data could be shared within the
node. That will reduce congestion of the interconnect and could improve the overall perfor-
mance. However, reduced number of messages is counterbalanced by increased frequency
of inter-thread communication, that is parallel reads and writes to the memory. That could
result in even greater overhead and lower program performance.
2.2 Programming models

The previous section briefly describes the most common hardware architectures applied in high performance computing. This section outlines the two main parallel programming paradigms and introduces the mixed mode programming model.

MPI

The Message Passing Interface (MPI) standard is a library specification for message-passing model. It was proposed as a standard by wide group of vendors, implementors and users. In the message passing model, data is distributed between all processors, which are executing separate versions of the code. This model match ideally to the distributed memory architecture, but in the same time it is very portable and can be used on most parallel architectures. Data distribution and communication are handled explicitly by the programmer. This often result in better performance, as the programmer has a greater control over the program, but on the other hand, it also means that programmer has full responsibility for the program efficiency.

In order to improve performance, MPI provides a number of optimised collective communication routines. In addition, many implementations used on SMP clusters are ’cluster aware’, this means that e.g. global reductions are first performed within the node to be then completed on inter-node level.

Synchronisation is done implicitly during communication, as processors must eventually agree that message has been received. If the communication is reduced to the minimum than synchronisation is minimised as well.

MPI is a very flexible model but in consequence developing using MPI is hard. Each processor can execute different part of the code at given time, this makes debugging very awkward and leads to many ’deadlock’ situations. Distributed data implicates load balancing problem which may result in poor resource utilisation. Although optimised, global communication can be still very expansive.
OpenMP

OpenMP is an industry standard [10] for shared memory programming. It is primarily based on compiler directives to invoke parallel computations on shared-memory multiprocessor computers.

Data resides in shared memory, hence load imbalance problems are easier to tackle than in distributed memory model. Communication is implicit, programmer only has to specify, using compiler directives, which parts of the code have to be perform in parallel. Hence, programming using OpenMP is easier and faster than using MPI. Generally, OpenMP is a better choice for shared memory machine and often makes better use of that architecture (e.g. collective operations, reduced communication overhead).

Mixed mode programming

Mixed mode programming is a combination of the above two programming models. (It could be a combination of any shared and distributed programming models, but we will focus on MPI and OpenMP). Mixed mode style naturally matches the SMP cluster architecture, where OpenMP could be used within the shared memory nodes, and messages could be exchanged between the nodes. Intuition says that this could potentially exploit features of the cluster and result in better program performance. Reading and writing to the memory should be faster than calls to the MPI library. To investigate this, we will perform series of benchmarks of different mixed mode styles and compare their achievements with pure MPI implementation.

There are several possible forms of the mixed mode programming. Rabenseifner [10] distinguishes different models depending on process / thread hierarchy, overlap of communication with computation and the number of threads calling communication routines. We will base our benchmarks on the hierarchical data distribution model, which is shown in Figure 2.4. Instead of 2D we will operate on 3D data set.

Figure 2.4 [1] shows an example of how 2D array is decomposed and distributed among MPI processes and OpenMP threads. First array is decomposed (2D decomposition) and distributed among MPI processes, which corresponds to dividing the array between SMP nodes. Sub-arrays are then distributed (1D decomposition) between OpenMP threads, which corresponds to the distribution within SMP node. We will follow that model in our benchmark codes.

Three different mixed mode programming models will be investigated, depending on the way the MPI communication is being handled we could distinguish [1]:

- **Master Only**, where all MPI communication takes place outside of OpenMP parallel regions.
- **Funnelled**, where communication may occur inside parallel regions, but is restricted to a single thread.
- **Multiple**, where more than one thread can call MPI communication routines.
In the Master Only model, inter-thread communication is performed by the master thread outside parallel region; it means that all other threads are idle during communication routine. It is a clear waste of processor cycles but it is also the most straightforward mixed mode technique.

The Funnelled model also employs only one thread for communication but MPI communication routines may be called within parallel regions. This means that while a selected thread exchanges data with neighbouring nodes, other threads can continue work on some other part of the program. This model allows programming techniques where computation is overlapped with communication.

The most sophisticated mixed mode model is the Multiple model, where communication is handled by more than one thread, thus communication may be overlapped with computation on the individual thread level.

**Why investigate mixed mode programming?**

SMP clusters are being successfully programmed using pure MPI. Why should we spend time investigating mixed mode and what could be potential benefits of using mixed mode programming model?

There are several promising scenarios where mixed mode could show its potential.

- **Code scales poorly with MPI**

  Mixed mode could help here, but only if the equivalent OpenMP version scales better. The problem with the pure MPI code scalability could be caused e.g. by load balance
problems that could be solved more easily using OpenMP. OpenMP gives also better performance on fine grain problems, where in MPI version communication overhead becomes dominant, thus reduces program scalability.

- Replicated data problems

Programs where data is replicated on all nodes are limited by the size of the memory available on the node. In MPI this is the size of the memory available on the processor. In mixed mode model data could be replicated just on the SMP nodes and shared between the threads, therefore data size would be limited to the memory of the whole SMP and it would be possible to study larger problems.

- Poorly optimised intra-node MPI

This problem is vendor specific, some MPI implementations are not optimised for shared memory architecture. Using OpenMP within the node could possibly improve overall performance of the program.

- Reduced communication

The number of intra-node messages, compared to the pure MPI version, will be reduced. However, the average message size will increase. Depending on the way that the communication is handled by the inter-connect, performance could rise or drop.

- Some shared memory algorithms may be more efficient

Collective communications are a good example of more efficient shared memory algorithm (e.g. OpenMP reduction). Utilising advantages of both programming models could improve efficiency of the program.

We have identified possible areas where mixed mode model could show its potential advantages. One could ask, if there are so many possible benefits that mixed mode programming could offer, why it isn’t the method of choice for SMP clusters in today HPC world?

There are several potential issues concerned with mixed mode programming identified by Smith and Bull [1]. First of all, mixed mode adds more complexity to the code development and maintainability. Secondly, it is not clear that we will obtain better performance, as OpenMP introduces additional overhead. The initialisation of parallel regions and the synchronisation at the end is expensive. Shared variables implicate use of additional synchronisation and sequential sections. OpenMP barriers are very expensive and might effectively decrease program efficiency. Finally, MPI is generally very portable, but mixed mode is less so.

Furthermore, Rabenseifner[2] experiments show that for several platforms, maximum internode communication bandwidth can be achieved only if communication is handled by more than one thread. This is due to specific interconnect architectures that support simultaneous transfer of many small messages rather than transfer of several big messages. This could potentially affect the master-only and funnelled models where communication is done by single master thread. However, more sophisticated Multiple style requires more complicated,
explicit distribution and communication handling which may result in additional overhead. It is still not clear if it will be more efficient technique than the others.

Additional complexity, not necessary resulting in better performance, is something that most project managers would not like to add to their ever growing list of risks. The aim of this project is to answer the question if and when developing mixed mode codes could be worthwhile, focusing particularly on the performance. We will compare three different styles of mixed mode programming. Research will be performed on state of art clustered SMP system - HPCx, described in detail in the next section.
Chapter 3

Benchmark codes

3.1 Jacobi Relaxation Algorithm

All benchmark codes are based on the Jacobi relaxation method, an iterative algorithm which finds discretised solutions to a set of differential equations. Algorithm that is used is a modified version of reverse edge-detection algorithm, which operates on 3D data set $x_{ijk}$, where the $\text{edge}$ is calculated as follows:

$$
\text{edge}_{ijk} = x_{i+1jk} + x_{i-1jk} + x_{ij+1k} + x_{ij-1k} + x_{ijk+1} + x_{ijk-1} - 6x_{ijk}
$$

(3.1)

With given $\text{edge}$ data, original data set can be determined iteratively using a Jacobi Algorithm:

$$
\text{new}_{ijk} = \frac{\text{old}_{i+1jk} + \text{old}_{i-1jk} + \text{old}_{ij+1k} + \text{old}_{ij-1k} + \text{old}_{ijk+1} + \text{old}_{ijk-1} - \text{edge}_{ijk}}{6}
$$

(3.2)

where $\text{old}$ is data calculated in previous iteration and $\text{edge}$ is the input data calculated in equation 3.1.

This algorithm meets all main requirements of the benchmark code that we are looking for. Calculating $\text{new}$ value is computationally expensive, algorithm works on the regular data set which can be easily distributed between processors and threads. Each step of the algorithm calculates the average of the values of its nearest neighbours, hence neighbouring processors have to exchange their halo areas each iteration. We could investigate how mixed mode model influences point-to-point communication, where compared to pure MPI, the number of exchanged messages will be reduced but their size will increase.

In order to study the performance of collective communications in a mixed-mode environment and monitor how much the data is changing we will compute the $\Delta$ value given by equation 3.3.
\[ \Delta^2 = \frac{1}{MNK} \sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{k=1}^{K} (new_{ijk} - old_{ijk})^2 \] (3.3)

In order to calculate \( \Delta \) value, each process/thread has to compute \( \Delta^2 \) locally, then perform global sum before a final square root. It will be interesting to compare how global MPI reduction performs comparing to mixed OpenMP/MPI reduction. In addition, we will be able to check if the algorithm converges and thus perform a correctness test.

### 3.2 Design and implementation details

All benchmark source codes are written in C, each of which is organised in three files. The entire algorithm flow with calls to the external communication and parallel logic routines is contained in main.c file. jacob3d.c includes definitions of functions that perform initialisation, communication and parallel decomposition. Functions declarations and initialisations of all global variables are contained in the jacob.h header file.

All versions of the program have to be run with specified parameters: number of processors, tolerance (which is the desired accuracy of the approximation), maximal number of iterations and number of processes in each of the three dimensions of the MPI decomposition.

```sh
./main proc_num tolerance maxiter xdim ydim zdim
```

For example to run the program on 256 processors, with 0.001 tolerance for at most 10 iterations, with 3D decomposition 8x8x4, one must enter:

```sh
./main 256 0.001 10 8 8 4
```

The following sections contain more detailed descriptions of program core functions, their modifications according to the different mixed mode versions, and analysis of different inter-node communication strategies.

#### 3.2.1 Serial

All versions of the algorithm use three static 3D arrays of doubles: `edge`, `new` and `old`. The main task of the benchmark code is to generate some computation load and communication traffic, we are not actually interested in reconstructing of the original data. Therefore, in order to simplify the code, no I/O operations are performed. All necessary data, that is `edge` array values, is generated at run time.
The algorithm flow in the serial version is shown in Figure 3.2. Iterative image reconstruction is performed until the result has a satisfactory precision level ($\delta < \text{tolerance}$) or the number of iterations reaches a predefined maximum.

Most of the time is spent in the `jacobistep()` function, which applies the Jacobi relaxation algorithm to the data set. It calculates the values of the new array as shown in Equation 3.2.

```c
void jacobistep(int Xmin, int Xmax, int Ymin, int Ymax,
                int Zmin, int Zmax, int Xstep, int Ystep, int Zstep) {
    int i, j, k;
    for (i=Xmin; i<Xmax; i+=Xstep) {
        for (j=Ymin; j<Ymax; j+=Ystep) {
            for (k=Zmin; k<Zmax; k+=Zstep) {
                new[i][j][k] = (old[i-1][j][k] + old[i+1][j][k] + old[i][j-1][k]
                                + old[i][j+1][k] + old[i][j][k-1]
                                + old[i][j][k+1] - edge[i][j][k]) / 6;
            }
        }
    }
}
```

Thanks to the input parameters that are passed to the `jacobistep` function, separate calculation of the halo region and inner volume can be performed by a call to the same function. This feature is used to overlap calculation and communication in the parallel implementations of the algorithm.

The delta calculation is carried with a frequency specified by the `interval` value. In most applications precision check is performed once per several iterations in order improve performance of the program. We use this value to reflect the relevant impact of this routine to the overall performance of the code.

```c
for (i=1; i<X+1; i++) {
    for (j=1; j<Y+1; j++) {
        for (k=1; k<Z+1; k++) {
            deltaSqr += (new[i][j][k] - old[i][j][k]) * 
                         (new[i][j][k] - old[i][j][k]);
        }
    }
}
delta = sqrt(deltaSqr/(X*Y));
```
At the end of each iteration the new approximated values are copied to the old array in the new2old() function.

```c
void new2old() {
    for (i=1; i<X+1; i++) {
        for (j=1; j<Y+1; j++) {
            for (k=1; k<Z+1; k++) {
                old[i][j][k]=new[i][j][k];
            }
        }
    }
}
```

If the calculated approximation is precise enough, or loop reaches the maximal number of iterations, program finishes its work.

### 3.2.2 Pure MPI

The pure MPI version was developed first, and all mixed-mode versions are modifications of this code. Parallel versions require the edge, new and old array to be distributed between all participating processors. Overall problem size is increased together with number of processors, as the local problem size is kept the same on each process.

![Algorithm flow diagram - Serial version](image-url)
At the beginning of the program the `MPI_Cart_create()` method is called. It returns a handle to a new communicator to which Cartesian topology information is attached. In this new communicator space, processes are arranged into a 3D cuboid (as shown on Figure 3.3) where the size of three dimensions are passed as parameters by the user. All communications between processors will be performed in the Cartesian communication space.

Processes arranged in this way allow for 3D data decomposition that matches the rank of the distributed edge array. This decomposition scenario is most popular solution in a wide range of scientific codes, e.g. fluid dynamics simulations.

Figure 3.4 shows a flow diagram of the pure MPI algorithm. Each iteration begins with a call to the `HaloSwap3D()` function that uses nonblocking communication routines to exchange halo regions between processors. Parameters that are passed to the function include the handle to the Cartesian communicator and `MPI_Vector` data types that are used to pack halo areas into one MPI message. The function returns a list of communication request handles.

The halo areas of the old array (which are being swapped) lie non-contiguously in memory. The relation of the array indexing to memory layout in C is shown in Figure 3.5. In order to send halo area located on the X plane we have to access disjoint addresses in memory. This would result in making consecutive MPI calls to send and receive each data element, which would be very inefficient and inelegant. In order to avoid this, a derived `MPI_Vector` data type is used, which allows us to concatenate blocks one data type separated by regular stride into a new data type, which is then exchanged between processes.

The halo swap is initiated with calls to nonblocking `MPI_Issend()` and `MPI_Irecv()` routines. Each process exchanges data with (at most) six neighbouring processes (as shown in the Figure 3.6). In order to determine the rank of the relevant processes taking part in the halo swap the `MPI_Cart_shift` routine is called. It takes a Cartesian communicator handle, displacement and coordinate dimension as the input parameters and returns rank of the shifted ‘source’ and ‘destination’ processes. In the case when process has no neighbour in given direction, the value `MPI_PROC_NULL` is returned, which prevents communication occurring when passed to the communication routines.
Figure 3.3: Algorithm flow diagram - pure MPI version

Figure 3.4: 3D array memory layout in C
After the halo swap is started, the program performs the Jacobi algorithm calculation on all non-boundary elements of the old array (see jacobiestep() function described in previous section). Thanks to the nonblocking communication routines used, we are able to overlap halo exchange with the Jacobi computation which may result in better utilisation of processor resources, and hence in better program performance. After calculation on the data located in the inner volume is finished, processes wait till all communication is done. Finally, computation of the halo region can be performed.

Each process updates its own part of the new array and if required, the residual value is calculated. In order to do this each process calculates deltasqrd value (which is squared value of local delta) which is then summed up from all processes and distributed back to all of them via a call to the global reduction MPI_Allreduce routine.

3.2.3 Master-Only

All three mixed mode versions implemented for the project purposes follow the hierarchical model as shown in Figure 3.7. Each cube represents a 3D sub-array that is part of the edge array distributed between processes; this sub-array has then been further divided between OpenMP threads using a 1D decomposition. Having distributed data in that way, each thread operates on contiguously address in memory (compare the thread distribution in Figure 3.7. with the memory layout in Figure 3.5). Experiments show that 1D thread decomposition is the best possible solution. Operating on large, contiguous chunks of data minimises probability of cache misses and prevents threads from false sharing.

Master-only is the most straightforward to implement of the mixed mode models. It differs from other models in the way the MPI communication is handled. All messages are sent outside the parallel regions by the master thread.

Modifications made to the pure MPI version are limited to adding several OpenMP directives. Thread parallelism is achieved by inserting omp parallel for directive whenever program operates on the edge, old and new arrays. The values of the iterator of the outermost loop is then distributed between threads, so each of them is working on its own ’slice’ of the 3D array.

The last modification made concerns residual calculation. Delta is first calculated locally,
summed up first by an OpenMP reduction clause and finally by MPI_Allreduce. Since shared memory global routines are usually faster, it will be interesting to compare this solution with the pure MPI one.

Since all MPI communication is done outside parallel regions, processing resources of threads that do not participate in data exchange are wasted. However, this style of programing is easy and fast to implement, especially when we have an existing MPI implementation. Adding several OpenMP directives does not require any further modifications to the code structure.

### 3.2.4 Funnelled

The Funnelled version also uses only one thread to perform MPI communication, however MPI routines may be called within OpenMP parallel region. The Parallel section of the algorithm is shown in the Figure 3.8.

At the beginning of each iteration one, of the threads start the non-blocking halo swap, performs the Jacobi calculation on its part of the non-boundary space, and waits for communication to complete. Unlike the master-only version, while one thread is performing inter-node communication, others are working on other parts of the code. All threads are then synchronised using OpenMP barrier.

Instead of using an OpenMP parallel for directive, which is not possible in this algorithm, loop bounds are specified for each thread and are passed as a parameter to the jacobistep() function. Loop boundaries are determined by the ThreadDistribute() function based on the thread ID value.
After the first barrier is reached by all threads, the program begins to perform calculations on the boundary areas. Each dimension is calculated separately by consecutive calls to `jacobistep()` function with relevant parameters. Each call is encapsulated in a separate `omp single nowait` block. This means that only one thread can enter the block while other continues to work on the following code. In other words, the first three threads that arrive to the single blocks will perform calculations on each part of boundary planes. Threads are synchronised once more before the delta calculation. Other parts of the code remain unchanged from the master-only version.

This style of mixed mode programming is more difficult to implement, as more modifications have to be made to the pure MPI version than for the master-only version. However, in return we are able to employ all threads when MPI communication is handled, which may result in better performance.

### 3.2.5 Multiple

The Multiple version of the algorithm uses yet a different way of handling MPI communication inside the parallel regions. Here, all threads participate in the exchange of the halo regions. Each thread sends and receives the part of the boundary data that it is working on independently of other threads. As shown in Figure 3.9, threads that are working on the end data slices have to communicate with five neighbour processes, while threads working on the inner data slices communicate only with four other processes.

In order to recognise which message is send by which thread and consecutively which message should be received by which thread, tagging system is used. A tag is an unique integer value which is based on the thread id and direction in which the message is being send.
A flow diagram of the parallel region is shown in Figure 3.10. Each thread starts a new iteration by initiating a non-blocking halo swap. Communication is then overlapped with computation performed on non-boundary data. Depending on the thread ID, each thread waits for a different number of Issends and Ireceivs to complete. This is represented in the code by different length arrays of MPI_Request handles passed into the MPI_Waitall() function.

Compared to the Funnelled version, no synchronisation is needed before threads start to work on their halo regions, as each thread waits individually for the relevant MPI communication to complete. Therefore, when a thread returns from the MPI_Waitall it knows that its boundary area is updated and it can carry on the calculation using that data. Threads with lowest and highest ID have to additionally perform calculations on the Y plane halo areas. Again, loop boundaries are determined explicitly for each thread, and passed as a parameter to the jacobistep() function.

This style of mixed mode programming is the most difficult to implement. We have to manage communication between each individual thread. However, this style might offer potentially the best performance, with reduced OpenMP overheads and synchronisation and better processor utilisation. A larger number of messages are sent, compared to Funnelled and Master-Only, however it is not obvious what impact this will have on performance.
Figure 3.9: Algorithm flow diagram - Multiple version
Chapter 4

Results and analysis

4.1 Methodology

4.1.1 Hardware: HPCx Architecture

All benchmark codes will be run on the HPCx Phase 2a machine, a state of the art SMP cluster system, currently ranked 59th on the top500 list [7] of fastest machines in the world, with a peak performance of 9216 GFlops.

Since HPCx system was recently upgraded from Phase 2 to Phase 2a system, it is very interesting to investigate how this changes the impact of mixed mode programing performance. According to the official HPCx specification ([3] and [8]) the HPCx Phase2 and Phase 2a systems feature:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Phase 2</th>
<th>Phase 2a</th>
</tr>
</thead>
<tbody>
<tr>
<td>System size</td>
<td>50 IBM p690+ nodes</td>
<td>96 IBM eServer 575 nodes</td>
</tr>
<tr>
<td>Node size</td>
<td>32 processors</td>
<td>16 processors</td>
</tr>
<tr>
<td>Proc arch.</td>
<td>IBM Power4</td>
<td>IBM Power5</td>
</tr>
<tr>
<td></td>
<td>1.7GHz 64-bit RISC</td>
<td>1.5GHz 64-bit RISC</td>
</tr>
<tr>
<td>Proc cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>64KB instruction, 32 KB data</td>
<td>64KB instruction, 32 KB data</td>
</tr>
<tr>
<td>L2</td>
<td>1.5MB shared between 2 proc</td>
<td>1.9MB shared between 2 proc</td>
</tr>
<tr>
<td>L3</td>
<td>128MB shared between 4 proc</td>
<td>36MB shared between 2 proc</td>
</tr>
<tr>
<td>Interconnect</td>
<td>IBM High Performance Switch (HPS)</td>
<td>IBM High Performance Switch (HPS)</td>
</tr>
</tbody>
</table>

One eServer LPAR (Logical Partition - used as synonym of the compute node), with its own
operation system, contains 8 Dual-Core Modules (16 processors) and has 32 Gbytes of main memory. More details about the IBM Power5 chip can be found in the next section. There HPCx service includes 96 IBM eServer LPARs that gives a total of 1536 processors available for users. IBM’s High Performance Switch (HPC) is used to provide inter node communication.

IBM Power5 processor

The main motivation of this project was to investigate the impact of new technology used in the new Phase 2a HPCx system to mixed mode programming. The most important upgrade concerns the processor chip, which was upgraded from Power4 to Power5 processor.

The 64-bit Power5 is new a generation, state of the art processor chip. According to [10], ‘it contains two microprocessor cores, chip and system pervasive functions and core interface logic. As listed above it is equipped with a 64KB level-1 data cache, a 32KB level-1 data cache, a 1.9MB level-2 cache and controls, a 36MB level-3 cache directory and controls and the fabric controller that controls the flow of information and control data between the L2 and L3, and between chips. Each core contains two fixed-point execution units (FXU - FX Exec Unit), two floating-point execution units (FPU - FP Exec Unit), two load/store execution units (LDU - LD Exec Unit), one branch execution unit (BRU - BR Exec Unit), and one execution unit to perform logical operations on the condition (CRU - CR Exec Unit)’ [10]. It also supports speculative, out-of-order execution.

As shown in Figure 4.1, in the Power5 system, the L3 cache has been moved closer to the processor, before fabric bus controller. Thus L2 cache misses are potentially less expensive and traffic on the inter-chip buses is reduced.

The completely new feature introduced in Power5 architecture is simultaneous multi-threading (SMT). The frequency of advanced processor has rapidly increased in recent years, however operations on memory are still the main bottleneck of the system. Even though caches, branch prediction and other sophisticated techniques are used, the processor has often to wait idle
for the process that is performing operations in other parts of the system (e.g. a memory load after a cache miss). In order to use the processor resources more efficiently, multi-threading has been introduced. Processor can now switch between threads to better utilise its processing power. Power5 has two microprocessor cores, each supporting 2 threads, giving 4 threads per chip.

### 4.1.2 Software

Each LPAR within the HPCx system runs its own copy of IBM’s AIX operating system. The latest AIX 5L version 5.3 is designed to provide support for all the features of the IBM eServer p5 system and 32- and 64-bit applications.

To compile both pure MPI and mixed mode codes we used `mpcc_r` the version of IBM compiler that automatically links MPI libraries. The `_r` suffix means that thread safe code is generated by the compiler, which is very important for OpenMP programming and is also recommended for MPI codes. Jobs are submitted on the HPCx system via LoadLeveller, a batch job scheduling application produced by IBM.

Following compilation options where used:

- **-q64**
  
  This option is used to enable 64-bit addressing, it also increases the amount of memory that a program can use and removes some of the restrictions on shared memory segments.

- **-O3**
  
  This option enables optimisation level three. Performs some memory and compile-time intensive optimisations. This level of optimisation have the potential to alter the semantics of a user’s program, e.g. reordering of floating point operations. However, no malfunction of the algorithm was detected when this level of optimisation was used.

- **-xarch=pwr5**
  
  This option specifies the general processor architecture, limits the code generated by the compiler to the instructions of the Power5 architecture.

- **-qtune=pwr5**
  
  This option specifies the Power5 architecture as the system for which the executable program is optimised.

- **-qsmp=omp:noauto**
  
  This option is used only when mixed mode versions are compiled. Enables OpenMP compliance, noauto option disables automatic parallelisation and optimisation.
In addition, some of the IBM MPI environment variables are used to specify how MPI running processes should behave. This allows to manipulate some of the tuning values which might improve the performance. The HPCx communication subsystem uses two methods to sent messages between processes. Messages whose size is smaller than a \texttt{MP_EAGER\_LIMIT} environment value (in bytes) are sent, immediately to the receiver, while larger messages are sent using a slower protocol. \texttt{MP_EAGER\_LIMIT} was set to its maximum value of 64K for all pure MPI and mixed-mode runs, which should result in better performance.

For pure MPI runs the \texttt{MEMORY\_AFFINITY} environmental variable was set to \texttt{MCM}. It instructs a process to use the memory that’s physical location is closest to the CPU. Furthermore, the IBM MPI implementation used on the HPCx system is cluster aware. This means that, by default MPI, uses shared memory for intra-node communication. It is, however, unclear if program performance benefits from that feature.

### 4.1.3 Experimental procedure

#### Problem size

A problem size of 192x192x192 per processor in fixed for experimental runs. This means that a total problem size is determined by a total number of processors used for computation and by their Cartesian topology in a communicator space. For example, for 256 processors arranged in 8x8x4 topology global problem size equals to 1536x1536x768. Since three arrays are used (\texttt{edge}, \texttt{old} and \texttt{new}), each of the same size, the maximum memory size used by the program is approximately 40.5 GB. It gives approximately 10.125 GB memory usage per computing node used.

With a problem size fixed per processor, we can study how the different number of employed computing nodes influences the performance. The amount of computation scales together with the system’s size, hence overhead connected with the inter-node communication and synchronisation is more visible.

#### Timing the code

Great care must be taken when it comes to timing the codes on HPCx. First of all HPCx is a shared resource, this means that resources (interconnect and processors) may be shared with system demons or other users. It is important to request exclusive access to all the nodes that will be used by our program. This is done by setting \texttt{node\_usage = not\_shared} in a job command file.

In order to get accurate timings the \texttt{MPI\_Wtime} function was used. It returns a double precision floating point number of seconds, representing elapsed wall-clock time since some time in the past. High resolution timings are obtained without adding significant overhead to the code.

Each version of the code was set to run for 11 iterations. However, only last the 10 iterations
were taken into consideration. The first run through the algorithm usually takes more time, as data has to be loaded into memory and program suffers from a greater number of cache misses. Because of the small number of iterations in total, the first iteration would distort the final result with greater effect. The convergence tolerance value was set to 0.0001 for each run. Although program never reached the required precision level, the converging delta value was used to determine program correctness.

It was observed that timing the same code several times produce times that fluctuated. These time differences were quite significant compared to the overall execution time. In order to collect more accurate results, each version of the benchmark code was run at least five times. The average value and the standard deviation was the calculated.

**Profiling tools**

As well as code timings, two profiling tools were used in order to make the performance analysis more complete. MPItrace, as the name suggest, is used to trace the inter-process MPI communication. It reports accurate time spent in MPI communication routines, and the number and size of the messages sent during the program’s run time. To use the MPItrace, program was liked with \texttt{-lmpitrace} during compilation.

The hpmcount utility, which is a Hardware Performance Monitor (HPM) Toolkit component, was used to provide detailed information on hardware performance counters, hardware metrics and resource utilisation - cache memory reuse in particular.

### 4.2 Pure MPI code

First runs of the pure MPI code were performed on 8 and 16 LPARs. The main target of this experiment was to determine the most efficient 3D process topology that would result in the best algorithm and scaling performance. The problem size of 192x192x192 is fixed per processor.

Global problem dimensions are changing together with changing process topology, thus each process have always exactly the same amount of work, access data in the same pattern and finally, the halo areas exchanged between processes have always the same shape. What is different is the process geometry and thus the main significant performance change that we should observe is the point-to-point communication performance.

Processes in different runs were arranged in the way that would give us more information about two matters that are not immediately obvious. First, we need to establish which dimension should dominate in the decomposition, that is in which dimension we should employ the greatest number of processes, which will be then replaced by the OpenMP threads in the mixed-mode version. Since we will use 1D thread decomposition and 16 threads per node, our choices are be limited. Secondly, we have to identify the topology that will efficiently scale form 8 to 16 LPARs. All mixed mode implementations will follow the most efficient and best scaling process decomposition.
Figure 4.2: Pure MPI performance comparison for different process geometry. Timer data for runs performed on 8 LPARs (top) and 16 LPARs (bottom)
Results of the experiment are shown in Figure 4.2. As expected the Jacobi calculation and update of the new array are not significantly affected by the different process geometry. We observe a slight fluctuation of the total time spent in these sections. However, it does not show any pattern that could suggest that it is connected to process layout. Although in the job submit file we have requested exclusive access to the whole LPARs, there still might be some operating system process sharing resources with our processes.

The pattern can be however observed in the point-to-point communication performance, which corresponds to the time spent in HaloSwap3D() and MPI_Waitall() routines. Although not that clear for 8 LPARs it becomes more apparent for 16 LPARs that the better performance is achieved when greater number of processes is used in the first dimension and smaller number on the third dimension. Runs where 8 and 16 processes where used to decompose the X dimension, compared to the cases where 2 processes were used, are up to 140% faster. Furthermore, when we swap the number of processes used in the last two dimensions in the 8x8x2, 8x8x4 16x8x2 and 16x4x2 decompositions, we observe up to 60% loss in the communication performance.

It is obvious that different process arrangements in the Cartesian communicator produces different communication pattern between tasks. It is important to understand how this pattern influence the program performance. In order to do this we have to recall the 3D array memory layout in C shown in Figure 3.4. One can see that the elements located along the Z dimension (which correspond to the outermost array’s k parameter) are lying contiguously in the memory, while elements lying along Y and X dimension form respectively stripes and single elements separated by a regular interval. Therefore, halo areas located on the X plane of the 3D array are formed from the elements lying linearly in the address space, where halo regions that are exchanged along Y and Z dimensions have to be constructed from the elements distributed in memory. Therefore, the process of assembling the Y plane and Z plane halos lasts longer, and reduced communication in these dimensions results in better code performance.

Not surprisingly, the best scaling performance was observed for the codes that arrange more processors along the X dimension in the communication space. This supports the previous observation concerning the longer time required for assembling the halo areas located on thr Y and Z planes.

Thr delta calculation performance, which is the part of the code containing collective communication, was not taken under consideration when topology choice was made. It can be observed that for all runs where more processes were placed along the Z dimension the performance was lower for about 20% than other cases, but lower performance was also noted for 16x4x2 decomposition. This difference is however only 1% of the overall program performance.

Based on these experimental results it was decided to use 16x4x2 topology for 4 LPARs and 32x4x2 topology for 8 LPARs, thus extending the decomposition along X dimension. All mixed mode implementations are based on that process topology; all mixed mode version results were compared to the pure MPI codes where processes were arranged in that fashion.
4.3 Pure MPI vs. Mixed-Mode

In this section, the performance of all implemented mixed mode benchmark codes is studied and compared to the most efficient pure MPI version. As the result of the experiments described in the previous section, all pure MPI runs where performed using 16x4x2 decomposition for 128 processors and 32x4x2 when 256 processors where employed.

Studies of the performance on the 8 LPARs include experiments with different numbers of threads used, in order to determine the overhead of the parallel regions and the influence of inter-thread communication on the overall performance. In order to investigate the scalability of the codes, experiments were carried on 16 LPARs with 16 threads per node.

The problem size is fixed per process/thread number which always give 192x192x192 per processor. For example in the case where 16 threads per node are spawned, problem size per process equals to 3072x192x192, which gives 192x192x192 per thread after 1D OpenMP decomposition.

4.3.1 Master-Only

All mixed mode versions follow the decomposition scheme used in pure MPI runs, where the first dimension is the most extended dimension in the process topology. However, in this experiment we gradually reduce the number of processes in the X dimension, spawning number the number of threads in their place. Finally, we end up with a model where initial 2D MPI decomposition among the SMP nodes, is followed by the 1D thread decomposition inside the SMP node. 1D decomposition is performed along the X dimension; this means that the values of the iterator of the outermost loop are distributed between threads. This is the best possible mixed mode distribution in this particular case, where we minimise MPI communication traffic along most expensive Y and Z dimensions and distribute contiguous chunks of memory between OpenMP threads. See Duthie, Bull, Smith & Trew [13] for experimental results on different mixed mode decompositions on the HPCx Phase 1 system.

The results of the experiment are shown in Figure 4.3. The first column shows the performance result of the pure MPI code and is followed by Master-Only results where different number of threads were used. The Master-Only version is significantly outperformed by pure MPI version, especially in the Update and Point-to-Point sections of the code. Mixed mode with 16 threads per node is approximately 35% slower than the pure MPI implementation. Performance is slightly better when only 4 threads are spawned, but still the mixed mode version is by 18% slower. We will now take a closer look at the performance of different parts of the code.

Jacobi Algorithm

The performance of the \texttt{jacobistep()} function is similar for pure MPI and mixed mode runs. Except the case with 8 threads per node, the execution time differences in this part of
the code do not exceed the standard deviation value for each series of measurements. Slightly worse (approx. 15%) performance is obtained for the 8-threads run; this might be a result of a temporary lower capability of the HPCx system. Execution time fluctuations, often observed during the test measurements, were especially visible between runs performed over longer time intervals.

The mixed mode version does not seem to suffer from any penalties connected to the additional parallel region overhead. The amount of computation required for Jacobi algorithm calculation overshadows the time required for initiating the parallel region. From the same reasons we can partly eliminate parallel region overhead as an explanation for the very poor Master-Only performance in the update section.

Scaling the number of threads used inside the node, does not affect the Jacobi calculation performance. It also shows that 1D thread decomposition along the X dimension does not cause additional cache misses or false sharing between threads.

**Update**

The Update section timings shows a dramatic loss of performance in the Master-Only version. Mixed mode is up to 120% slower in this part of the code, which translates to 45% of the overall performance loss for 16 threads and 80% for 4 threads.
Although it was expected that computational sections of the code might performed slightly worse than the relevant parts of the MPI version, where no inter-thread communication is present. It is unlikely that such a difference is a result of additional reads and writes to the memory. Runs where only one thread per process is spawned (giving one process per processor), show that inter-thread communication and synchronisation cannot be blamed. The performance of the \texttt{new2old()} function is almost identical, no matter if 1 or 16 threads are spawned. Suspicion that cost of the parallel region initiation could be a source of the problem was ruled out in the previous paragraph, also supported by the results of a single thread run. Finally, thread 1D decomposition follows the scheme used in the Jacobi algorithm calculation, where it does not seem to reduce cache memory reuse.

This problem was further investigated by the \texttt{hpmcount} utility. We were looking for any abnormal behaviour of the program, especially for main memory and cache memory access patterns. Results of some of the measurements (16 threads per node were spawned for mixed mode version) are presented in Table 4.2 below.

<table>
<thead>
<tr>
<th>Resource Usage Statistics</th>
<th>Pure MPI</th>
<th>Master-Only (ave. per thread)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 D cache store misses</td>
<td>176,881,254</td>
<td>165,081,611</td>
</tr>
<tr>
<td>Data loaded from L2</td>
<td>22,091,076</td>
<td>21,082,351</td>
</tr>
<tr>
<td>Data loaded from L3</td>
<td>946,947</td>
<td>885,757</td>
</tr>
<tr>
<td>Data loaded from local memory</td>
<td>1,753,162</td>
<td>256,149</td>
</tr>
<tr>
<td>Data loaded from remote memory</td>
<td>308,642</td>
<td>1,782,024</td>
</tr>
</tbody>
</table>

Table 4.2: Selected data gateherd using the \texttt{hpmcount} tool

Results show that cache memory usage is almost on the same level in both implementations. Slightly less data is loaded (approx. 5% for L2 and 8% for L3) from L2 and L3 cache in the mixed mode version. That indicates that thread distribution did not introduce false sharing and that threads make as good use of the cache memory as the processes in the pure MPI version.

What is worrying however, is that the number of local and remote memory loads is exactly opposite in the pure MPI and mixed mode versions, to the disadvantage of Master-Only. It might have a relevant impact on the performance as the memory structure inside the HPCx SMP node is somewhat similar to the memory structure in the NUMA machine. Processors are integrated into a multi-chip module (MCM), each frame is composed from two MCMs, which gives four chips (8 processors) per MCM. Each MCM is configured with 128 MB of L3 cache and 16 GB of main memory [7]. Briefly, it means that accessing main memory that is located in remote MCM takes longer time than accessing memory that is sharing the MCM with the processor. Data gathered using \texttt{hpmcount} tool suggest that mixed mode version accesses data located in remote memory approximately 7 times more frequent than in the local memory. This would be a result of a very poor data placement. In other words it means that threads are working on the parts of the \texttt{edge, new or old} array that is stored in.
memory located on a different MCM. It is however not obvious for the author if this data can be applied for the mixed mode code analysis. It is not clear how hpmcount, which is run on the process level, identifies local and remote memory of each thread.

Although, without complete trust in the hardware counter measurements, greater care was taken during the data generation to assure that threads are accessing memory that is located closer to the CPU. Unfortunately it did not improve the overall performance of the mixed mode code, which could be expected after analysing the performance of Master-Only version where no threads where spawned.

We have ruled out several possibilities that could stand behind the very low performance of the new2old() function, we are left with a few options to investigate. Lets consider compiler optimisation. In the pure MPI version, the compiler has a full knowledge of the bounds of each loop iterator used while updating the new array. The problem may lay in the way the OpenMP directives are turned into the code by the compiler in the mixed mode version. Since loop parameters are distributed using a parallel for directive, compiler knowledge about them is limited and it cannot apply any advanced optimisations techniques.

In order to investigate this further, we carried out another experiment. Both pure MPI and Master-Only codes where compiled without -O3 option, that means that no optimisation techniques were applied by the compiler. However, -qsmp=omp:noauto option used to compile the Master-Only version forces second optimisation (-O2) level to be applied on the mixed mode version. Therefore, in order to determine which of the optimisation techniques does not apply to the mixed mode, we checked how the performance of the pure MPI version is changing together with different optimisation levels enabled.
Results are shown in Figure 4.3. The role of compiler optimisation is striking, difference in performance of the pure MPI code between optimisation level one and two equals approximately to 95%. Performance of the update section in the mixed mode with -O2 optimisation level is slightly better than in the pure MPI with -O1 optimisation level and significantly worse then in the mixed mode with -O2 optimisation level enabled. It would suggest that some of the -O2 optimisation techniques cannot be applied to the update procedure in Master-Only version. -O2 optimisation level include among others the loop-invariant optimisation technique. It is very possible that OpenMP directives make the optimisation of the update loop much harder task for the compiler, which would explain the weaker performance of the new2old() function.

**Delta**

The delta computation time does not differ much between both versions. Although reduction clause is used in the mixed mode version, performance of the delta loop is comparable with pure MPI. It proves the very high efficiency of OpenMP global reduction method, which did not add any significant overhead to the deltasqrd calculation time.

**Point-to-Point Communication**

Time required for Point-to-Point communication to complete is getting larger with the number of threads used in computation. In the run where 16 threads were spawned inter-node communication took about seven times longer to complete.

There might be several reasons behind the very poor performance of the Point-to-Point communication in mixed mode version of the code. First of all, number of inter-node messages sent in the mixed version is smaller than in pure MPI but their size is much larger. For example, when 16 threads are used, the problem size is 3072x192x192 per process, that gives an average size of the message send across Z dimension equal to 4608KB. It means that although number of messages is smaller, it is more likely that their size will be larger than the value specified by the MPI_EAGER_LIMIT global variable. This means that a slower interconnect communication protocol is used to exchange the messages.

The message assembling process might be another reason for the weaker Point-to-Point communication performance. Since in the pure MPI version each process exchanges the halo area that it is working on, it accesses the local memory in order to build the message. In case where the master thread is assembling a message that will be composed of halo regions of all the threads that are working inside the SMP node, it has to access remote memory, including remote MCMs. When we compare Point-to-Point communication time of the runs where 8 and 16 threads, where remote MCM is accessed, we will notice a dramatic performance loss of approximately 110%, whereas the difference between 4 and 8 threads equals to approx. 65%.

This situation is somewhat repeated when the master thread receives a message from the remote node. The halo region is stored int the master thread cache memory. When other
threads access their halo regions, they got to access remote memory where the master thread has stored the relevant data.

Furthermore, the master thread while assembling and disassembling MPI messages, fills its cache with the data that will not be reused in later parts of the computation. It is a serious handicap compared to other threads or processes in the pure MPI version, which are always working on the same part of the data. This may result in additional overheads in the update and delta calculation where the OpenMP barrier is used for synchronisation, as other threads have to wait for the slower master thread to complete. This is also reflected by the slightly worse cache re-usage (approx. 5% for L2 and 8% for L3) shown in Table 4.2.

Finally, while the master thread is performing the inter-node communication all other threads are remain idle. This means worse processor resource utilisation than in pure MPI implementation, where communication is performed more efficiently by all the threads.

**Collective**

The collective section in the mixed mode version was expected to perform better than its counterpart in the pure MPI. This is the result of the smaller number of MPI processes that take part in the global reduction. Part of the global reduction is performed during the \( \text{deltasqrd} \) calculation using OpenMP parallel clause. Better performance in this part of the code is however overshadowed by much worse performance of the update and Point-to-Point communication in Master-Only version.

**Scaling the problem size**

The following experiments were carried out to investigate scaling ability of the code. We have extended the number of the employed LPARs form 8 to 16. All mixed mode runs were performed using 16 threads per node. Problem size remains fixed per processor; that means that ideally both 8 LPARs and 16 LPARs execution time should be the same.

Results are shown in the Figure 4.5. Pure MPI again shows better performance than Master-Only version. Except the slightly slower Point-to-Point communication, all parts of the program scale exceptionally well. It proves that scaling the problem size along the X dimension gives the best performance results. The worse performance of the Point-to-Pint communication is the result of the increased number of processes taking part in the MPI communication. The performance of the \( \text{MPI_Allreduce()} \) is unaffected by the larger number of LPARs used. Surprisingly, worse performance of the Point-to-Point communication is counterbalanced by the faster delta calculation. This is not the result of greater number of processes involved, as the problem size remains the same. Only one series of timings was performed for the larger problem size, and this might be the result of execution time fluctuations, often observed on the HPCx system.

The Scalability of the Master-Only version is decreased mainly by the MPI global reduction, which was not expected as this part of the code was the only one that was significantly
faster than its pure MPI counterpart, when run on 8 LPARs. One of the expected advantages of the mixed mode version, that is better scaling of the point-to-point communication due to the smaller number of messages sent across the interconnect, is overshadowed by the exceptionally bad performance comparing to the pure MPI.

### 4.3.2 Funnelled

The main idea behind this implementation is to employ all the threads that do not participate in the inter-node communication. In Master-Only version, while the master thread is assembling the MPI messages and then exchanging them with its neighbours, all other threads are idle, thus available system resources are being waisted.

The Funnelled model introduces a more complicated approach where MPI communication routines are called within the parallel region. One of the threads, identified by its thread ID, performs all the MPI communication duties, meanwhile other threads can still work on the other parts of the code. However, data have to be decomposed among threads explicitly by the programmer. Additional explicit OpenMP synchronisation is also required.

Comparison of the Funnelled and pure MPI versions is shown in Figure 4.5. Unfortunately, the performance of the Funnelled version is much worse than pure MPI. It is approximately 42% slower for 16 threads and 21% slower when 2 threads per node are used. Furthermore, Funnelled version is about 15% slower than Master-Only version for 16 threads, while the performance for 2 threads is approximately the same. This results show that our attempt to
overlap MPI communication with computation among the threads was unsuccessful.

The Funnelled implementation suffers from the same disadvantages that were identified in the Master-Only version. We observe the same problem with the update section optimisation, which takes approximately the same amount time to execute. Point-to-Point communication timings are also very similar and become longer with increasing number of threads.

Worse performance than in the Master-Only version, especially for 8 and 16 threads, is observed in the computational section. The main source of overhead in this part is the explicit data distribution and use of the expensive `omp barrier`. As the thread responsible for the communication is busier than others, an attempt was made to distribute some of its work among the other threads. In addition, the Jacobi calculation in the halo regions is encapsulated in a `omp single nowait` section, which should result in a better computation load balance in the program.

Attempts to overlap the single thread MPI communication with computation was overshadowed by the greater OpenMP overhead. As observed in the Master-Only analysis, use of the `omp parallel` directives makes optimisation of the computational parts of the code more awkward for the compiler. Less effective explicit data distribution between threads and additional overhead connected with `omp single` and `omp barrier`, especially visible for 16 threads, result in even worse performance of the Funnelled version.

Figure 4.6 shows the comparison of the Funnelled version and pure MPI scaling performance.
Reasonable scaling result of the mixed mode is however meaningless when we compare the overall performance of both implementations. What is interesting is that the Funnelled version that is run on 16 LPARs does not show the bad performance of the `MPI_Allreduce` that can be observed on its Master-Only counterpart.

### 4.3.3 Multiple

The Multiple version is the most sophisticated implementation of the mixed mode programming model. Each thread maintain its own MPI communication policy independently from the other threads. Results of this approach are shown in Figure 4.7.

The Multiple version turns out to be the most efficient and stable model out of all that were implemented. It is however not enough to outperform the pure MPI version. Again the main performance loss is connected with the update section, which seems to be immune to any optimisation attempts by the compiler. The Jacobi calculation benefits from the reduced synchronisation, and performs even slightly better than in the pure MPI implantation. No inter-thread communication overhead is visible here. The delta calculation takes approximately the same amount of time in both versions of the code. MPI global reduction behaves very predictably and is much more efficient (by approximately 70%) than in pure MPI for 16 threads. This is, however, irrelevant to the overall code performance.

Except for the update section, the most disappointing is the low performance of the Point-to-Point communication. We can rule out the large size of the messages as the reason, as the size
Figure 4.8: Performance results of the Pure MPI and Multiple runs on 8 LPARs

of the messages does not exceed those sent in the pure MPI version. Furthermore, the number of messages sent is lower than in pure MPI, as messages exchanged inside the node along the Z dimension in the pure MPI version are replaced with much faster reads from the shared memory in the Multiple. This was expected to result in better performance of the Multiple implementation.

The issue connected with necessity of accessing the remote MCMs within the LPAR while assembling and disassembling the MPI messages is also gone. As each thread sends and receives halo areas that are lying on its ‘territory’, in the same way as is done in the pure MPI version. However, it does not improve the performance of the Point-to-Point communication.

Furthermore, MPI point-to-point synchronisation is relaxed, as the threads that are working on the inner volume of the data, communicate with at most 4 neighbouring nodes. This means that each thread is synchronised with only 4 other threads rather than 6 (processes) as in pure MPI version.

All these features of the Multiple implementation do not convert into a better program performance. It suffers mainly from same reasons in the same parts of the code as the other mixed mode implementations.

Figure 4.8 shows the code performance behaviour on 8 and 16 LPARs. It general shows good scaling, and keeps the the same characteristics independent of number of nodes used. Both pure MPI and Multiple runs where 16 LPARs where employed, show a slightly worse
performance in communication parts and slightly better performance in computational parts. The fact that runs for 8 and 16 nodes where undertaken in large time intervals might explain timing fluctuations.

4.4 Simultaneous Multithreading

In this experiment we will investigate the new feature of the IBM Power5 processor - Simultaneous Multithreading (SMT). By allowing multiple threads to simultaneously access the resources provided by a one physical processor, SMT tries to improve the efficiency of the utilisation the processor resources. Experimental procedure does not differ from the one we follow in the previous measurements. Again, we will scale the problem size along the X dimension and keep problem size fixed per processor. Applying SMT means that number of threads used per node will be doubled.

Results of the experiment are shown in Figure 4.10. All benchmark versions, where the SMT feature is enabled, prove to be more efficient then their ‘ordinary’ counterparts. Better performance in all computational parts of all the benchmark codes is observed, that is average improvement of approximately 20% in the delta (except the Master-Only version for 8 LPARs), update and the Jacobi algorithm calculation sections. It overshadows worse, and much worse particularly for the Multiple implementation, Point-to-Point communication and global reductions performance.

Better performance of the computational parts is the result of the more efficient processor
resource utilisation. While each processor handles two threads simultaneously, it can switch between one and another when, for example, one of them waits for a data load from main memory to complete. Furthermore, the Multiple implementation, where inter-node communication is more sparse than in pure MPI version, may benefit from even more sophisticated way of overlapping communication with computation.

Things become more complicated when it comes to analysing the program’s communication performance. In all versions we observe a loss of performance in the Point-to-Point communication sections. The performance drop is relative to the increased number of the messages, which have to be handled by a singular processor. Additional overhead might be also the result of context switching during the communication procedure. However, no performance drop was observed for the Multiple mixed mode version, which result in best performance gain due to the SMP for this implementation.

Global reduction performance also suffers form increased number of threads, messages and additional synchronisation that have to be handled be a processor. Again, performance of the collective section is slightly better for the Multiple implementation. Its value, however, lies in the measurement error interval and cannot be clearly interpreted.

Although performance of all the benchmark codes benefit from the SMT technology, the Multiple code in particular, pure MPI implementation is still the most efficient one. This experiment, however, proves the great capability that lay in the Simultaneous Multithreading technology and its benefits for the parallel computing.
Figure 4.10: SMT benchmark results for 8 LPARs (top) and 16 LPARs (bottom)
Chapter 5

Conclusions

The comparison of the pure MPI and mixed mode implementation shows the superiority of the pure MPI in terms of performance. All three mixed mode versions where outperformed in almost all sections of the code except the global communication. The most efficient Multiple mixed mode implementation turned to be 30% slower than the pure MPI.

Computational parts of the benchmark algorithm were differently affected by the mixed mode implementations. The Jacobi and delta calculation turned to perform similar in all codes except the Funnelled version, where problems where inflicted by poor load balance. Other than that mixed mode codes show to handle the inter-node communication in a highly efficient manner. Problem occurred with the update section performance which most probably lay in the way the OpenMP directives are turned into the code, which could make the compiler optimisation awkward. This was one of the major factors of the mixed mode implementations poor performance.

Through series of the experiments and previous work experience we have determined the most efficient process topology and thread decomposition. In the Master-Only and Funnelled style, all communication must be done by the master thread. It was expected that aggregated inter-node bandwidth we be achieved for the Multiple style, where more than one thread is used for the communication with other nodes. In addition, other then in the pure MPI, intra-node communication is substituted by direct access to the application data in the shared memory. Although slightly faster than in the Master-Only and Funnelled versions, Multiple Point-to-Point communication turned to be much slower than in pure MPI implementation. Reason to this wasn’t clearly identified and might be connected with the way that MPI library cooperates with multiple threads. For example if the message buffers have the same size in 16 thread runs as for single process run it would have a severe impact on the communication performance. Poor performance of the Point-to-Point communication was the second major factor of the overall worse performance of the mixed mode implementations.

The only section of the code that turned to be more efficient in the mixed mode versions was the collective communications. This was expected as less number of global messages have to be handled due to the fact that MPI_Allreduce routine was called by only one thread per SMP node instead of 16 precesses in pure MPI version. OpenMP global reduction
proved to be very efficient as well, as its use does not add any significant overhead to the
delta calculation. However, it turned to be not enough to outperform the pure MPI version.

The experiments with the SMT technology proved the vital influence of the multi-threading
to the parallel program performance. All versions of the benchmark code benefit from that feature. Although the Point-to-Point communication is slower when SMT option is enabled, due to larger number of messages that have to be handled by the processor, this was recompensed by the much better performance of the computational parts of the code. Greatest performance gain was observed in the Multiple version. However, it was not enough to outperform the pure MPI version, which was still approximately 25% faster with SMT enabled.

Mixed mode version, although outperformed on the HPCx system, should not be considered completely useless. The benchmarks performed during this project were mainly focused on the mixed mode inter-node communication and computation performance comparing to the pure MPI, but we were working on the regular mesh problem size. If we have to tackle more complicated problem, which characteristics wouldn’t be regular and balanced, mixed mode implementation might be favourite. More sophisticated and easier to implement load balance techniques could overshadow the worse inter-node communication performance. This is, however, a material for an another MSc project.
Appendix A

Experimental data

<table>
<thead>
<tr>
<th>Topology</th>
<th>Jacobi</th>
<th>Comms</th>
<th>Delta</th>
<th>Update</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8x2</td>
<td>1.6265</td>
<td>0.1115</td>
<td>0.1584</td>
<td>0.4666</td>
<td>0.01821</td>
</tr>
<tr>
<td>8x2x8</td>
<td>1.6005</td>
<td>0.1792</td>
<td>0.1606</td>
<td>0.4607</td>
<td>0.01912</td>
</tr>
<tr>
<td>2x8x8</td>
<td>1.6717</td>
<td>0.2322</td>
<td>0.1424</td>
<td>0.4970</td>
<td>0.01873</td>
</tr>
<tr>
<td>2x4x16</td>
<td>1.5964</td>
<td>0.2680</td>
<td>0.1810</td>
<td>0.4474</td>
<td>0.02093</td>
</tr>
<tr>
<td>2x16x4</td>
<td>1.7219</td>
<td>0.2902</td>
<td>0.1795</td>
<td>0.4522</td>
<td>0.01909</td>
</tr>
<tr>
<td>16x4x2</td>
<td>1.6725</td>
<td>0.1202</td>
<td>0.1634</td>
<td>0.4643</td>
<td>0.02171</td>
</tr>
<tr>
<td>16x2x4</td>
<td>1.7128</td>
<td>0.1949</td>
<td>0.1482</td>
<td>0.4696</td>
<td>0.01940</td>
</tr>
<tr>
<td>4x16x2</td>
<td>1.8770</td>
<td>0.2240</td>
<td>0.1539</td>
<td>0.5991</td>
<td>0.02021</td>
</tr>
<tr>
<td>4x2x16</td>
<td>1.7382</td>
<td>0.2773</td>
<td>0.1837</td>
<td>0.4771</td>
<td>0.02128</td>
</tr>
</tbody>
</table>

Table A.1: Different process topologies performance on 8 LPARs

<table>
<thead>
<tr>
<th>Topology</th>
<th>Jacobi</th>
<th>Comms</th>
<th>Delta</th>
<th>Update</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8x4</td>
<td>1.6082</td>
<td>0.1205</td>
<td>0.1634</td>
<td>0.4643</td>
<td>0.02171</td>
</tr>
<tr>
<td>8x4x8</td>
<td>1.6985</td>
<td>0.2041</td>
<td>0.1585</td>
<td>0.4359</td>
<td>0.02201</td>
</tr>
<tr>
<td>4x8x8</td>
<td>1.7779</td>
<td>0.2216</td>
<td>0.1500</td>
<td>0.4663</td>
<td>0.02724</td>
</tr>
<tr>
<td>2x8x16</td>
<td>1.8651</td>
<td>0.2258</td>
<td>0.1677</td>
<td>0.4616</td>
<td>0.03141</td>
</tr>
<tr>
<td>2x16x8</td>
<td>1.7897</td>
<td>0.2310</td>
<td>0.1610</td>
<td>0.4777</td>
<td>0.02754</td>
</tr>
<tr>
<td>16x8x2</td>
<td>1.6428</td>
<td>0.1302</td>
<td>0.1611</td>
<td>0.4514</td>
<td>0.02402</td>
</tr>
<tr>
<td>16x2x8</td>
<td>1.7665</td>
<td>0.1742</td>
<td>0.1709</td>
<td>0.4614</td>
<td>0.03012</td>
</tr>
<tr>
<td>8x16x2</td>
<td>1.7564</td>
<td>0.2496</td>
<td>0.1707</td>
<td>0.4661</td>
<td>0.02734</td>
</tr>
<tr>
<td>8x2x16</td>
<td>1.8195</td>
<td>0.2511</td>
<td>0.1582</td>
<td>0.4502</td>
<td>0.02178</td>
</tr>
</tbody>
</table>

Table A.2: Different process topologies performance on 16 LPARs
<table>
<thead>
<tr>
<th>Decomposition</th>
<th>Delta</th>
<th>Update</th>
<th>Jacobi</th>
<th>MPI_Allreduce</th>
<th>Point-to-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure_MPI</td>
<td>0.182</td>
<td>0.430</td>
<td>1.593</td>
<td>0.128</td>
<td>0.075</td>
</tr>
<tr>
<td>16-4-2/1t</td>
<td>0.183</td>
<td>0.895</td>
<td>1.620</td>
<td>0.107</td>
<td>0.093</td>
</tr>
<tr>
<td>8-4-2/2t</td>
<td>0.189</td>
<td>0.898</td>
<td>1.635</td>
<td>0.136</td>
<td>0.122</td>
</tr>
<tr>
<td>4-4-2/4t</td>
<td>0.160</td>
<td>0.881</td>
<td>1.574</td>
<td>0.057</td>
<td>0.165</td>
</tr>
<tr>
<td>2-4-2/8t</td>
<td>0.210</td>
<td>0.941</td>
<td>1.843</td>
<td>0.030</td>
<td>0.254</td>
</tr>
<tr>
<td>1-4-2/16t</td>
<td>0.165</td>
<td>0.956</td>
<td>1.704</td>
<td>0.033</td>
<td>0.538</td>
</tr>
</tbody>
</table>

Table A.3: Master-Only version - 8 LPARs

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>Delta</th>
<th>Update</th>
<th>Jacobi</th>
<th>MPI_Allreduce</th>
<th>Point-to-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure_MPI</td>
<td>0.178</td>
<td>0.911</td>
<td>1.620</td>
<td>0.123</td>
<td>0.091</td>
</tr>
<tr>
<td>8-4-2/2</td>
<td>0.157</td>
<td>0.929</td>
<td>1.718</td>
<td>0.186</td>
<td>0.185</td>
</tr>
<tr>
<td>4-4-2/4</td>
<td>0.166</td>
<td>0.942</td>
<td>1.770</td>
<td>0.129</td>
<td>0.273</td>
</tr>
<tr>
<td>2-4-2/8</td>
<td>0.159</td>
<td>0.914</td>
<td>2.144</td>
<td>2.718</td>
<td>0.436</td>
</tr>
<tr>
<td>1-4-2/16</td>
<td>0.242</td>
<td>0.977</td>
<td>2.176</td>
<td>2.994</td>
<td>0.581</td>
</tr>
</tbody>
</table>

Table A.4: Funnelled version - 8 LPARs

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>Delta</th>
<th>Update</th>
<th>Jacobi</th>
<th>MPI_Allreduce</th>
<th>Point-to-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure_MPI</td>
<td>0.182</td>
<td>0.430</td>
<td>1.593</td>
<td>0.054</td>
<td>0.075</td>
</tr>
<tr>
<td>8-4-2/2</td>
<td>0.171</td>
<td>0.937</td>
<td>1.631</td>
<td>0.149</td>
<td>0.168</td>
</tr>
<tr>
<td>4-4-2/4</td>
<td>0.203</td>
<td>0.936</td>
<td>1.557</td>
<td>0.095</td>
<td>0.207</td>
</tr>
<tr>
<td>2-4-2/8</td>
<td>0.153</td>
<td>0.937</td>
<td>1.609</td>
<td>0.035</td>
<td>0.304</td>
</tr>
<tr>
<td>1-4-2/16</td>
<td>0.194</td>
<td>0.960</td>
<td>1.563</td>
<td>0.043</td>
<td>0.550</td>
</tr>
</tbody>
</table>

Table A.5: Multiple version - 8 LPARs

<table>
<thead>
<tr>
<th>Code Version</th>
<th>Delta</th>
<th>Update</th>
<th>Jacobi</th>
<th>MPI_Allreduce</th>
<th>Point-to-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure_MPI_8LPARs</td>
<td>0.182</td>
<td>0.430</td>
<td>1.593</td>
<td>0.054</td>
<td>0.075</td>
</tr>
<tr>
<td>Pure_MPI_16LPARs</td>
<td>0.141</td>
<td>0.451</td>
<td>1.589</td>
<td>0.057</td>
<td>0.134</td>
</tr>
<tr>
<td>Master_8LPARs</td>
<td>0.194</td>
<td>0.960</td>
<td>1.563</td>
<td>0.043</td>
<td>0.550</td>
</tr>
<tr>
<td>Master_16LPARs</td>
<td>0.227</td>
<td>0.957</td>
<td>1.71</td>
<td>0.178</td>
<td>0.539</td>
</tr>
</tbody>
</table>

Table A.6: Master & pure MPI versions - 8 & 16 LPARs

<table>
<thead>
<tr>
<th>Code Version</th>
<th>Delta</th>
<th>Update</th>
<th>Jacobi</th>
<th>MPI_Allreduce</th>
<th>Point-to-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure_MPI_8LPARs</td>
<td>0.182</td>
<td>0.430</td>
<td>1.593</td>
<td>0.054</td>
<td>0.075</td>
</tr>
<tr>
<td>Pure_MPI_16LPARs</td>
<td>0.141</td>
<td>0.451</td>
<td>1.589</td>
<td>0.057</td>
<td>0.134</td>
</tr>
<tr>
<td>Funneled_8LPARs</td>
<td>0.242</td>
<td>0.977</td>
<td>2.176</td>
<td>0.063</td>
<td>0.581</td>
</tr>
<tr>
<td>Funneled_16LPARs</td>
<td>0.179</td>
<td>0.974</td>
<td>2.32</td>
<td>0.067</td>
<td>0.645</td>
</tr>
</tbody>
</table>

Table A.7: Funneled & pure MPI versions - 8 & 16 LPARs

44
<table>
<thead>
<tr>
<th>Code Version</th>
<th>Delta</th>
<th>Update</th>
<th>Jacobi</th>
<th>MPI_Allreduce</th>
<th>Point-to-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure_MPI_8LPARs</td>
<td>0.182</td>
<td>0.430</td>
<td>1.593</td>
<td>0.054</td>
<td>0.075</td>
</tr>
<tr>
<td>Pure_MPI_16LPARs</td>
<td>0.141</td>
<td>0.451</td>
<td>1.589</td>
<td>0.057</td>
<td>0.134</td>
</tr>
<tr>
<td>Multiple_8LARPs</td>
<td>0.194</td>
<td>0.960</td>
<td>1.563</td>
<td>0.043</td>
<td>0.550</td>
</tr>
<tr>
<td>Multiple_16LARPs</td>
<td>0.222</td>
<td>0.960</td>
<td>1.48</td>
<td>0.079</td>
<td>0.586</td>
</tr>
</tbody>
</table>

Table A.8: Multiple & pure MPI versions - 8 & 16 LPARs

<table>
<thead>
<tr>
<th>Code Version</th>
<th>Delta</th>
<th>Update</th>
<th>Jacobi</th>
<th>MPI_Allreduce</th>
<th>Point-to-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure_SMT</td>
<td>0.182</td>
<td>0.430</td>
<td>1.593</td>
<td>0.054</td>
<td>0.075</td>
</tr>
<tr>
<td>Master</td>
<td>0.123</td>
<td>0.333</td>
<td>1.352</td>
<td>0.068</td>
<td>0.126</td>
</tr>
<tr>
<td>Master_SMT</td>
<td>0.165</td>
<td>0.956</td>
<td>1.704</td>
<td>0.033</td>
<td>0.538</td>
</tr>
<tr>
<td>Funnelled</td>
<td>0.186</td>
<td>0.859</td>
<td>1.381</td>
<td>0.121</td>
<td>0.667</td>
</tr>
<tr>
<td>Funnelled_SMT</td>
<td>0.242</td>
<td>0.977</td>
<td>2.176</td>
<td>0.063</td>
<td>0.581</td>
</tr>
<tr>
<td>Multiple</td>
<td>0.219</td>
<td>0.772</td>
<td>1.804</td>
<td>0.164</td>
<td>0.592</td>
</tr>
<tr>
<td>Multiple_SMT</td>
<td>0.194</td>
<td>0.960</td>
<td>1.563</td>
<td>0.043</td>
<td>0.550</td>
</tr>
</tbody>
</table>

Table A.9: Simultaneous Multithreading - 8 LPARs

<table>
<thead>
<tr>
<th>Code Version</th>
<th>Delta</th>
<th>Update</th>
<th>Jacobi</th>
<th>MPI_Allreduce</th>
<th>Point-to-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure</td>
<td>0.182</td>
<td>0.430</td>
<td>1.593</td>
<td>0.054</td>
<td>0.075</td>
</tr>
<tr>
<td>Pure_SMT</td>
<td>0.16</td>
<td>0.39</td>
<td>1.261</td>
<td>0.025</td>
<td>0.115</td>
</tr>
<tr>
<td>Master</td>
<td>0.165</td>
<td>0.956</td>
<td>1.704</td>
<td>0.033</td>
<td>0.538</td>
</tr>
<tr>
<td>Master_SMT</td>
<td>0.283</td>
<td>0.801</td>
<td>1.381</td>
<td>0.263</td>
<td>0.658</td>
</tr>
<tr>
<td>Funnelled</td>
<td>0.242</td>
<td>0.977</td>
<td>2.176</td>
<td>0.031</td>
<td>0.581</td>
</tr>
<tr>
<td>Funnelled_SMT</td>
<td>0.201</td>
<td>0.797</td>
<td>1.981</td>
<td>0.115</td>
<td>0.561</td>
</tr>
<tr>
<td>Multiple</td>
<td>0.194</td>
<td>0.960</td>
<td>1.563</td>
<td>0.043</td>
<td>0.550</td>
</tr>
<tr>
<td>Multiple_SMT</td>
<td>0.0996</td>
<td>0.791</td>
<td>1.151</td>
<td>0.033</td>
<td>0.518</td>
</tr>
</tbody>
</table>

Table A.10: Simultaneous Multithreading - 16 LPARs
Bibliography

http://www.epcc.ed.ac.uk/ markb/docs/sci_prog01.pdf

http://www.hlrs.de/people/rabenseifner/publ/cug2003_hybrid_1.pdf


7. Top 500 team. *Top 500 supercomputer sites*, 
http://top500.org, 2006

8. *HPCx Capability Computing*, 
http://www.hpcx.ac.uk, 2006

9. Power5 system microarchitecture, (IBM Journal of Research and Development, 49, 4-5) 


12. *OpenMP Application Programming Interface*,
    http://www.openmp.org, 2006

13. J. Duthie, M. Bull, L. Smith, and Arthur Trew *Mixed Mode Applications on HPCx*,
    (HPCxTR0403, 2004.)