Mixed Mode Programming on Clustered SMP Systems

Ioannis Makris

MSc in High Performance Computing
The University Of Edinburgh
August 2005
Authorship declaration

I, Ioannis Makris, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;

2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

3. I have acknowledged all main sources of help;

4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

5. I have read and understand the penalties associated with plagiarism.

Signed:

Date: 26/08/2004

Matriculation no: 0460102
Abstract

The Mixed Mode programming model, also known as Hybrid OpenMP/MPI programming model, can potentially offer superior performance in clustered SMP systems compared to pure MPI. Due to the increased complexity and non-trivial implementations that this model involves however, it is very common for implementations to fail using this model efficiently, with inferior performance as a result. This project will focus on comparing the different kinds of Mixed Mode programming that can be employed in a typical numerical analysis problem. For the purposes of the test, an iterative three dimensional algorithm is used, several versions of which have been implemented. Benchmarks run on the HPCx service, an IBM p690+ cluster, resulted in all kinds of Mixed Mode codes performing slightly better than pure MPI, although one of them performed in a highly unpredictable way.
# Table of Contents

1 Introduction 1

2 Algorithm and implementations 3
   2.1 The Algorithm 3
   2.2 Serial Implementation 4
   2.3 Code design 5
      2.3.1 Abstracting Data 5
      2.3.2 Function naming conventions 8
   2.4 Decomposing the problem 8
      2.4.1 Decomposition in shared memory 8
      2.4.2 Load balancing methods 9
      2.4.3 Block distribution algorithms 10
   2.5 Compiler considerations 12
      2.5.1 Static arrays and pointer aliasing 12
      2.5.2 Loop extent known in compile-time 12
   2.5 Parallel Implementations 13
      2.5.1 Pure codes 14
         2.5.1.1 Pure OpenMP 14
         2.5.1.2 Pure MPI 15
      2.5.2 Mixed Mode Codes 17
         2.5.2.1 “Master Only” code 18
         2.5.2.2 “Funneled” code 19
      2.5.2 “Multiple” code 20

3 Testing Methodology 21
   3.1 The HPCx Service 21
      3.1.1 Hardware 21
      3.1.2 Software 22
   3.2 Timing the code 23

4 Analysis of the results 26
   4.1 Benchmarking pure codes 26
      4.1.1 Runs in 1 LPAR 26
      4.1.2 Runs in 8 LPARs 30
   4.2 Mixed Mode Benchmarks 31
      4.2.1 Runs in 1 LPAR 31
      4.2.2 Runs in 8 LPARs 34
      4.2.2.1 Master Only and Funneled benchmarks 34
      4.2.2.2 “Multiple” Mixed Mode code benchmarks 39

5 Conclusion 42

Appendix A: Timings 44

Bibliography 50
Acknowledgements

First and foremost, I would like to thank Dr Mark Bull and Dr Stephen Booth for their time and effort supervising this project, for their concise advising which proved very significant and for their patience as well. I am also grateful to Dr Lorna Smith for the help she offered when my supervisors were absent. Many thanks to all the nice people at EPCC who were very kind and helpful and finally, to my classmates who were always willing to share their knowledge and experiences.
Chapter 1: Introduction

Clustered SMP systems are dominant in HPC market since a few years, which is a result of their cost-effective design. There are clustered SMP systems in several scales, ranging from small Beowulf clusters to massive supercomputers. Due to the hierarchical level of shared and distributed memory, the programming model used on an SMP cluster is always a consideration.

The two commonly used models for parallelization are called “flat” and “hybrid”. The first model employs distributed parallelism irregardless to the shared memory hierarchy. Handling the memory hierarchy in an effective way is left to the libraries. The flat programming model can be expressed with MPI [6] and HPF [9] code. The hybrid model on the other hand requires the programmer to be aware of the memory hierarchy. The parallelization within the SMP node is typically done with threads, either using POSIX threads or OpenMP [7], which is the leading standard in thread programming. The Mixed Mode programming model [8] uses MPI for the distributed parallelization and OpenMP for thread programming within the SMP node.

The purpose of this project is to compare and analyse the performance of Mixed Mode codes in an implementation of a numerical analysis problem to the equivalent pure MPI code which is most commonly used way to perform parallelization.

The rest of the project report has the following structure:

Chapter 2 introduces the algorithm we use in the benchmark and discusses a few issues associated with real-world problem considerations. A more detailed description of the implementation of the algorithm follows, starting with the serial implementation. The details of the accompanying library we use in all the parallel implementations is discussed next followed by a short review of some candidate decomposition strategies. There is special mention in the strategies that were finally employed, discussing how they correlate to the library and their potential advantages and disadvantages. A short discussion on compiler issues that might affect performance is given as well and finally the parallel implementations are discussed, also focusing on issues that may impact performance.

Chapter 3 presents the hardware architecture of HPCx, the system on which the benchmarks are run. Details about the software configuration of the system that are related to our code’s performance are given as well. The strategy we used for timing the code is discussed next, along with a short description of the design of the timing code itself.
Chapter 4 has an analysis of the results we obtained, starting with the pure benchmarks followed by the mixed mode benchmarks. A comparison in the performance characteristics of the pure MPI code to the mixed mode codes is also attempted. For both the pure and mixed mode codes, results obtained from single and multiple node runs are examined separately. In many cases it is not absolutely clear what the cause of specific characteristics we observed in performance is and assumptions are made. Wherever possible, we back our assumptions with results obtained from additional tests.

Finally, Chapter 5 presents the conclusions of this project and some general considerations on Mixed Mode programming. Timing results are given in the Appendix.
Chapter 2: Algorithm and implementations

2.1 The Algorithm

The algorithm employed for the benchmarks is a Jacobi relaxation method that computes an approximation to Laplace’s second derivative equation. It is a very typical algorithm widely used in benchmarks of this kind as it involves computation, nearest neighbour communications and global reductions. Assuming that we have a rectangular three dimensional floating point data set “edge” produced by applying the Laplacian equation (1.1) over the data set “vol”,

$$\text{edge}_{i,j,k} = \text{vol}_{i+1,j,k} + \text{vol}_{i-1,j,k} + \text{vol}_{i,j+1,k} + \text{vol}_{i,j-1,k} + \text{vol}_{i,j,k+1} + \text{vol}_{i,j,k-1}$$  \hspace{1cm} (1.1)

the algorithm will incrementally approximate the original data set with iterations of the form (1.2).

$$\text{new}_{i,j,k} = \frac{1}{6}(\text{old}_{i+1,j,k} + \text{old}_{i-1,j,k} + \text{old}_{i,j+1,k} + \text{old}_{i,j-1,k} + \text{old}_{i,j,k+1} + \text{old}_{i,j,k-1} - \text{edge}_{i,j,k})$$  \hspace{1cm} (1.2)

In real world problems, care must be taken for the value of the boundary conditions. However, in order to keep the benchmark as simple as possible, there is no special treatment and boundaries are simply set to zero. The artificial data used with the benchmark should indeed have zero boundaries as they would remain unchanged after applying the Laplacian operator.

In order to estimate how close to the actual solution the new approximation gets, the residual is computed, as shown in equation (1.3). As the algorithm converges, the value of delta approaches zero, hence a desired approximation can be achieved for a given value of delta.

$$\Delta = \sqrt{\frac{1}{MN\Xi} \sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{k=1}^{\Xi} (\text{new}_{i,j,k} - \text{old}_{i,j,k})^2}$$  \hspace{1cm} (1.3)

Computing the residual is relatively expensive and therefore an efficient solution would only perform this task once every few iterations, but for our purposes it is preferable to have it always computed, in order for the effect it
has in performance to be more evident. Moreover, delta computation is not put into any effective use for terminating the algorithm but is only being computed redundantly.

Convergence of the Jacobi algorithm compared to other methods is very slow; however the speed of the algorithm itself should not be an issue because the purpose of the benchmark is to contrast the achieved performance among different implementations, rather than achieving a fast solution.

2.2 Serial Implementation

A very simple serial implementation of the algorithm would use 3 buffers, the first to hold the input data, the second one for the new array which is produced in every iteration and the third to hold the data that were generated in the previous iteration. We will refer to those buffers as edge, new and old respectively. After each iteration, it is essential to perform a copy operation from the new buffer to the old. This is how the code would be structured using this approach:

- iterate
  - read edge and old, write to new, accumulate squares to acc
  - compute Δ from acc
  - copy new to old

In the pseudocode, acc is a floating point value that accumulates the squares of the elements of the new array, as soon as they are computed.

It is easy to figure out that the last step can be avoided if the old and new buffers are operated in turn. It is desirable however that the algorithm has always the results (the new array) placed in the same buffer after execution. Thus it is important to assign the initial order of the two buffers depending on the number of iterations that will be performed. This is the pseudocode for the new algorithm:

- assign the order of buffers
  - iterate:
    - read edge and old, write to new, accumulate squares to acc
    - compute Δ from acc
    - swap buffers
2.3 Code design

Before we examine the characteristics of each implementation in particular, we will first present the design and features of the accompanying library modules used in common by all implementations. The code is written in C language, but despite that, there has been an effort to be as modular as possible, also following some very basic principles of object-oriented design. Porting to any object-oriented language like C++ or Java should be a relatively straightforward procedure, but we believe that unless additional functionality was to be added, there is no real need for using OOP in this code other than for making it look neater and cleaner. Using OOP constructs within a procedural language entails some pitfalls in using the program’s elements correctly and care has to be taken in order to avoid bugs that are sometimes difficult to trace.

2.3.1 Abstracting Data

Starting with the primitive Abstract Data Types (ADTs) used throughout the code, the most commonly used is the Triplet, which is a simple group of 3 integers. There is no significant functionality implemented for this ADT other than initialization and addition, but its use is vital for the clarity of the code. A second ADT that groups 3 floating point numbers is called Vector3 and is meant to be used with distribution algorithms. If there was any need for heavy use of vector mathematics, it would be a better idea to substitute this data type with an equivalent provided by a vector math library.

Due to the fact that all of the operations in the code are performed in regard to 3D data sets, it is desirable to have an ADT that is associated with a relevant implementation, and also encapsulates the information required to process the data set. This way the code becomes clear and concise and it is much easier to write or modify. The ADT that serves for that purpose is the Volume3D type. Although the code is written in C, it would be more appropriate to describe the Volume3D as a class, rather than a simple ADT; however this is not directly recognizable.

The Volume3D type consists primarily of a Triplet that holds the dimensions of the data set and a pointer which is meant to be pointing to contiguous data. Note that although Volume3D implements three-dimensional data, there are no multi-dimensional arrays employed. Other components exist as well that serve for internal purposes. They hold information about the state of the object or simply are variables which would be frequently used. Hence in an object-oriented language these variables would have been declared as “private”.
There is no restriction on whether a Volume3D object points to dynamic or static data, there are constructors for both cases. The dynamic constructor allocates data according to a given extent (received as a Triplet argument) and initialises accordingly. The second constructor, along with the extents, also receives the pointer to the allocated data. It is up to the user of the class that this pointer points to an adequately sized array. This is a violation to the rules of object oriented design principles, but it is not particularly important for these codes, simplicity is probably more important.

The second ADT which is used very frequently within the code is the SubVolume3D. This can be thought as a reference to a sub-region of a Volume3D object, without either providing implementation for the data or pointing to a specific location in memory. It is therefore possible for a SubVolume3D to be used in reference to several Volume3D objects, which is a particularly useful property for these codes. It consists of two Triplets, the first one being the offset of the Volume3D sub-region it refers to, and the second is the extent of this sub-region. We should note that an instance of a SubVolume3D is not meant to be used with volumes of specific size, as long as the regions it points to are within the limits of the volume. The actual relation between the Volume3D object and the SubVolume3D in terms of code design is presented in the class diagram of Figure 2.1, while a more practical view of what they actually stand for is shown in Figure 2.2. Note that the aggregation relation between the Volume3D and the SubVolume3D classes presented in Figure 3.1 is conceptual and does not have a direct correspondence to the code.

Although there is nothing complicated about the design so far, it may already seem to be over-engineered; however due to the complicated nature of mixed-mode programming, it will become obvious that it is essential for the code to have a well defined design.

Figure 2.1: Class diagram with the basic classes.
As we already mentioned, it would be more sensible to describe the program’s ADTs as classes rather than simple ADTs, because this is what they were designed for. There are some issues with this approach however that don’t actually have a straightforward way to be implemented in a procedural language. One of these issues is the initialization, creation and destruction of objects within the code which has to be made explicitly. This doesn’t hurt the design directly, but care must be taken that every object is initialized and destroyed within the scope that is used. Ignoring this requirement is likely to cause the program to crash and this unsafety makes evident that our implementation is more suitable for an object oriented language. Other than that, it adds up a few lines for the extra function calls which make the code less pleasant to read. The example given in Figure 2.3 illustrates the transformation required in an object oriented C++ code in order to fit our C implementation.

```cpp
//C++ implementation
//...
{   ClassName A(1, 2, 3);
    //use instance A
}
```

```
//C implementation
//...
{   ClassName A;
    ClassName_Init();
    ClassName_Construct(1, 2, 3);
    //use instance A
    ClassName_Destruct();
}
```

**Figure 2.2: A practical view of the relation between Volume3D and SubVolume3D types.**

**Figure 2.3: Initialization of objects in C.**
2.3.2 Function naming conventions

The naming convention used for function names was employed to make easier for member functions to be recognized. In principle, a member function would consist of a class name followed by the name of the function, as shown in Figure 3.3. Furthermore, the first argument to this function should always be a reference (pointer) to the object it operates for. There is much hassle about what naming conventions to follow in such cases, but we selected this particular naming convention because it also resembles the explicit C++ notation when calling member functions.

2.4 Decomposing the problem

The code was designed to be modular, in order to allow a great deal of flexibility to potential distribution algorithms, without any concern about the underlying mechanism that will put this to work. In this section we will analyze how the aforementioned design is correlated to the actual implementation and ways that certain decomposition strategies can be implemented. We will first discuss problem decomposition in the shared memory model followed by decomposition using message passing. Finally, we will focus on combining strategies from the two paradigms to come up with potential ways to decompose the problem using mixed mode programming.

2.4.1 Decomposition in shared memory

There are several ways to decompose the problem in shared memory programming, some being more obvious than others. The most obvious way is to use a single data set which is shared among threads and assign each thread to process a part of it. Due to the fact that each thread also references neighbouring elements that have been assigned to other threads, it is important that all threads are synchronised between iterations. Another way which resembles message passing decomposition is to use intermediate buffers to store the boundaries (halos) of each region. This method indicates that as soon as a thread processes the part it is assigned, it has to store the boundaries in a specified buffer and replace them with the buffer that corresponds to the neighbouring thread. We refer to memory reads-writes of this kind as pseudo-messages. This method avoids the overhead of MPI library calls for swaps inside the node, but it is not a particularly good idea to use it for this kind of problems. It is more appropriate for algorithms that need to exchange small messages frequently without needing to synchronise (stable matching algorithms for example can potentially benefit from this model in a better way).
Finally, it is also possible to use both shared data set and pseudo-messages depending on how the problem is decomposed. For example, pseudo-messages could be used only to prevent false sharing that occurs in a certain direction while decomposition along other axes is done with simple sharing. This kind of implementation is significantly more complicated though and as we will also see later, performance indicates that there is no reason for this additional complexity. We have decided to use the first kind of decomposition only, assigning parts of shared data sets to threads, which is also the simplest way to do it.

2.4.2 Load balancing methods

It would be helpful to provide a definition for each kind of decomposition we refer to, since it happens that these terms do not always come into agreement in scientific literature. We therefore define block decomposition to be the kind of decomposition where each worker is assigned a single series of consecutive elements. In this sense, “consecutive” does not imply that elements are also contiguous in memory, unless the problem we refer to is single dimensional. Figure 2.4 illustrates an example of a 2D block distribution of a two dimensional rectilinear data set, where each worker is assigned a series of consecutive elements, that are not contiguous in memory.

![Figure 2.4: Elements assigned to a worker using block distribution (left) and cyclic distribution (right).](image)

In contrary to block distribution, cyclic distribution is the kind of distribution where each worker is assigned either multiple elements (simple cyclic) or series of consecutive elements (block cyclic) in a round-robin fashion. This is particularly useful in cases where the problem can be load imbalanced. Both
the block distribution and the cyclic distribution methods are doing static decomposition, which means that the working domain of each worker is decided before the data set is actually processed and does not change while it is being processed.

Other decomposition methods also supported by OpenMP assign the working domain dynamically, as the computation is being made. This is what the *guided* and *dynamic* scheduling clauses serve for in OpenMP. In the case of the Jacobi algorithm, it is possible that some load imbalances appear when block distribution is employed, as a result of uneven distribution, but it is not a severe problem that needs to be addressed with a load balancing mechanism. This is because these mechanisms add up significant overheads and they are also prone to cache thrashing, so they should be used with care. The algorithm we are working on is inherently load balanced and heavily reliant to the cache and for this reason alone it is obvious that the most appropriate distribution method is the block distribution. It does not take many changes in the code for the SubVolume3D class to refer to strided data, which would at least make block-cyclic distribution possible. However it is a better idea to address these minor load-imbalances by employing a better algorithm for static distribution.

### 2.4.3 Block distribution algorithms

Distributing data using block distribution is not necessarily trivial. It can be really difficult if the optimal distribution is sought. Block distribution algorithms can be categorised in those that attempt to decompose the problem evenly, and those that allow uneven distribution, possibly in order to load balance computation in relation to communication. There is nothing particularly complicated with algorithms that fall in the first category, except that they should be able to handle the situation where the dimensions of the dataset are not perfectly divisible with the selected decomposition. The simplest way to handle this is to use the quotient of the division incremented by 1 for the dimensions of all but the last block and place the remainder to the last worker. For example, if a dataset with 101 elements was to be decomposed among 10 workers, then the first 9 workers would get 11 elements each and the final worker would only get 2. This is what actually happens in HPF block decomposition. A better approach is to assign each worker the quotient of the division and distribute the remainder among the workers. Using this method with the previous example, 9 workers would get 10 elements each and one gets 11, which performs slightly better than the first. Our implementation uses this algorithm.

Uneven distribution can be sought to make communication overlap with computation. For the Jacobi algorithm, we can notice that there is a load imbalance in communication for the tasks that touch the borders of the lattice.
They have to perform less communication and it is more likely that they have to wait for the other tasks to finish. It can thus be a good idea to assign them more elements to work on in order not to wait, which means that computation has to be unbalanced, hence distribution has to be uneven. Our implementations were designed having this in mind so they can work together with an algorithm that distributes unevenly. Although distribution can be uneven, the arrays used in the computation are always even and statically allocated. What happens in this case is that they are padded and extra elements are used accordingly. The factor that defines the amount of padding which is proportional to the problem size is \( \Lambda \) (lambda). Thus, if \( \Lambda \) is set to 1.5, then the dimensions of the arrays are increased by 50% and when it is set to 1, it is the same as if the distribution was meant to be even from the beginning. This is illustrated in Figure 2.5. Since the distribution algorithm we are using does not make uneven distribution, \( \Lambda \) was set to 1.

![Figure 2.5: Uneven distribution of data using padded arrays](image)
2.5 Compiler considerations

The compiler is likely one of the most important factors that contribute to the final performance we can obtain from our benchmarks. There are too many issues related to using the compiler efficiently in mixed mode programming applications, in our case particularly, this is mostly focused in cache optimizations and assumptions on pointer aliasing. It is not our intention to make an analysis on compiler optimization issues related to mixed-mode programming and therefore the code has been intentionally written in a way that can hinder the compiler to optimize some specific loops, whenever these optimizations cannot be applied in every parallel implementation. This way we try to have a more fair comparison that is purely focused on performance of different parallel codes rather than how the compiler can optimize in each case. It should be noted however that mixed mode programming is less likely to benefit from compiler optimizations because of its complicated nature. We will discuss some of these issues in the following paragraphs.

2.5.1 Static arrays and pointer aliasing

There are many reasons that can cause statically allocated arrays to be accessed faster than dynamically allocated which are only accessed through pointers, the most important being the compiler’s assumption about pointer aliasing. This means that the compiler can assume that a number of pointers may overlap (i.e. may point to the same data) and is therefore not able to perform optimizations like reordering instructions for better pipelining because data dependencies would cause the program to be incorrect. Some ways to work this issue around is using the C99 restrict keyword or explicitly ordering the compiler to make the assumption that pointers do not overlap. The latter entails the very dangerous situation that although the code might work when the flag is specified, a change in the code does not come into agreement with this assumption causing the program to behave erroneously. Such kinds of problems are sometimes difficult to trace. The compiler is also more likely to align static arrays better than the dynamically allocated, which might have an effect in several architectures. The only restriction the C standard sets for dynamically allocated arrays is to be aligned in a boundary specified by the largest language type.

2.5.2 Loop extent known in compile-time

A compiler is often able to perform optimizations in a loop whose extent is known during compile time. Depending on the architecture of the system the code is aimed for, this could possibly enable the compiler to unroll the loop,
vectorize or even simplify. In order to take advantage of this effect, it is essential to perform static decomposition in compile time, probably using the preprocessor.

Of the implementations we have discussed, it is possible for the pure MPI version to take advantage of this effect in a rather straightforward way. This is because in the MPI model each process holds a separate set of arrays that can be addressed explicitly in the program in the same way for all processes, as long as the decomposition has been decided statically. This is also possible in the case that the problem size is not divisible with the number of processes, but requires a second loop to be written for this purpose alone (it takes some trickery). A pseudo-message implementation with threads may also take advantage of static arrays the same way as the MPI. In general, using this feature effectively requires the programmer to make significant tradeoffs in how general the code is, which sometimes ends up being messy as well. In all implementations of the Jacobi algorithm examined, there is no way for the compiler to know the extent of the loops. This is firstly because the extent is always set by a distribution algorithm and secondly because the loops where the computation takes place are inside a function in a different file. If static distribution was employed, the compiler could possibly know about the loop extent in the MPI code, if cross-file optimizations were enabled.

2.5 Parallel Implementations

There are five parallel implementations of the algorithm we will be comparing; a pure OpenMP implementation, an MPI implementation and three Mixed-Mode implementations referred to as “MasterOnly”, “Funnelled” and “Multiple”. Some variations of these implementations are examined as well. Before we proceed with the implementations though, we should present the convention we are using for memory contiguity. The letters x, y, z are used respectively to i, j and k and this is shown schematically in Figure 2.6.

![Figure 2.6: Relation to memory contiguity and the variable names we use for axes](image-url)
2.5.1 Pure codes

Although the two pure codes follow a different programming paradigm, they share many common characteristics. These codes were intentionally built to follow the same design principles in order to be able to bind easily and form the mixed mode codes. Hence, in this section we only describe the characteristics that are particular to each code, rather than the full implementation.

2.5.1.1 Pure OpenMP

The pure OpenMP code uses the aforementioned block distribution algorithm to decompose the arrays across a team of threads, so the OpenMP for work sharing construct is not used. The shape of the decomposition is set statically inside the config.h file, but the extents of the region that each thread will be working on are set dynamically. The problem can be decomposed in several dimensions. Among threads there are shared Volume3D objects called vdata, vbuf1 and vbuf2 that correspond to the edge, new and old arrays accordingly. Each thread has a private instance of a SubVolume3D struct named sv_thr_fprc which is set by the distribution algorithm and is used to reference a region in all three arrays. After the distribution stage, each thread runs the initialization algorithm that fills this region with artificial data. Initialization is performed for all three arrays.

**Figure 2.6: Schematic representation of the Pure OpenMP code.**
The computation work is done within a single parallel region that starts before calling the Jacobi function. The Jacobi function is fed with references to the three volumes and the local subvolume, as well as two pointers to functions that will be called before and after each iteration. These functions are placed inside the main.c file and serve for computing $\Delta$ and placing barriers. This functionality is actually implemented inside the “after” function while the “before” function is empty. The “before” function is placed in order for the code to be symmetric, since the Pure OpenMP code serves as a basis for the “Funneled” and “Multiple” Mixed Mode codes. There are two barriers in the “after” function that are vital for the program to be correct. One barrier is required for preventing a thread to start computing after an iteration, while another thread with adjacent data is still computing on the previous iteration. Either of the barriers can serve for this purpose. In order for $\Delta$ to be computed, there is an atomic operation with all threads summing up their local square sums to a shared value. It is essential that one thread sets this value to zero before the others start summing up. A second barrier is needed after they finish summing so that all threads compute the square root for the complete sum.

2.5.1.2 Pure MPI

The MPI implementation is structured in the same way as the OpenMP implementation, in order to be able to bind with it and form the mixed-mode codes. When the MPI application starts, the Cartesian topology is first set up, according to what has been defined in the config.h file. The distribution algorithm is then run, and the MPI datatypes are set for the halo swaps. The types of the halos are not defined in the same way for all dimensions and this also affects performance. The types for the back-forth and upper-lower swaps are defined as vectors with a given count, stride and blocklength. In this case, blocklength is set to the length of the row, while count is set to the length of the opposite dimension’s array. The value of stride must also be set for the gaps which are present because of the halos and possibly because of some padding as well. This way, back-forth and upper-lower swaps can be done using “1” in the count number of the MPI sends and receives. Types for the left-right halo swaps are defined firstly with an MPI vector whose count and stride are used in the same way as the other halo types, but with blocklength being equal to 1. This only defines a single element with the relevant displacement. Next, this vector is used to define an MPI struct that consists of the aforementioned vector and the MPI_UB type which sets the upper bound. The upper bound’s displacement is equal to one slice (i.e. width X height of the array). The type of this struct defines a single column in the array with the relevant to the left-right swap displacement. It is therefore
required that the swaps are performed using the value of the depth in the count argument of the MPI sends and receives.

**Figure 2.7: Schematic representation of the Pure MPI code.**

**Point to point communication**

Before the computation stage, halo swap is performed. This is done using non-blocking sends and receives in the following way:

- post all receives
- post all sends
- wait (either for all, or just for the receives)

The “wait” in the end is actually an MPI_Waitall which can be set to wait either for all the communications to complete, or just for the receives, in either case the program will work, however it does not make any difference in performance. The only thing that changes in this case is that point to point communication takes less time for some processes, but they will also have to wait longer in the collective communication stage for the $\Delta$ computation.

Although a program using the first approach will work, it is not correct not to wait for the sends at all, because there is some memory allocated from the MPI library for each non-blocking send which is not released until the sends are cleared with a wait or a test statement. In case a program can benefit from not waiting for the sends, it is a good idea to do this sometime accumulatively. Our pure MPI implementation does not wait for the sends, so the average waiting
time reported in the benchmarks for the collective operation (which functions as a barrier) is larger than those in the Mixed Mode runs.

An alternative to this approach is to use the combined send-receive function that the MPI library provides to avoid deadlocks. Using this approach the halo swaps should be performed in the following way:

- send-receive forth-back halos
- send-receive upper-lower halos
- send-receive left-right halos

This approach was also used in [1] but it sets too many restrictions in the way halo swaps are performed. The problem here is that send-receives may cause too many processes to wait inactively for three consecutive times until they receive the data. In this code the effect in performance should be more intense than [1] because the problem is decomposed in three dimensions. Non-blocking sends and receives are expected to perform better in any system with a decent MPI implementation, as the library should be able to schedule messages in a more efficient way. It is discussed in [2] that non-blocking sends and receives have indeed superior performance on HPCx.

Collective communication

In order to compute \( \Delta \), the variable that was used for each process to accumulate squares within the procedure has to be reduced globally. It is also important that all processes hold the value of the global sum and for this reason a call to MPI_Allreduce is made. This functions pretty much like an MPI_Barrier in this case, because the processes that were able to finish swapping earlier, would have to wait for the others to finish the computation stage that they entered later, it is therefore denoted as a process barrier in Figure 2.7.

2.5.2 Mixed Mode Codes

The structure of the Mixed Mode codes is the same as the pure codes and they are all constructed by combining the pure OpenMP and pure MPI codes, each one with some small modifications. There are a few common features among the mixed mode codes that are worth mention. Firstly, they start with the MPI_Init_threads rather than the MPI_Init used typically in single threaded MPI programs. This MPI-2 function is used here to assure that the MPI implementation provides the thread safety level required by the code. Another common characteristic is the two phase decomposition, first for the processes
and then for the threads, using the data obtained from the process decomposition.

### 2.5.2.1 “Master Only” code

This implementation uses one parallel region for the computation stage, but the rest of the code is run by one thread only, which is responsible for performing all the MPI communication. The thread safety required by the MPI library is `MPI_THREAD_SINGLE`. The reduction of the square sums is implemented with an OpenMP reduction clause. Due to the fact that some of the variables that are used locally by each thread are also needed in the serial region, these variables are stored in static arrays whose size is proportional to the number of threads used. In order to avoid false sharing which could be a result of threads writing in adjacent places of such an array, it is meaningful to have a lot of empty spaces between the elements. In the config.h file there is a `CACHE_LINE` preprocessor defined constant which should be set to a value equal to or greater to the cache line of the lowest hierarchy cache.

![Figure 2.8: Schematic representation of the “Master Only” code.](image)

In Figure 2.8 there is only one barrier in this computation, the implicit barrier that OpenMP places on the closure of the parallel region, but the opening of the parallel region can also be thought to act like a barrier here. Each time the parallel region opens or closes threads, there are no threads spawned or killed. Threads other than the master thread do some busy waiting when a serial
region is executed and resume when they enter a parallel region again. But they cannot enter a parallel region unless the master thread reaches this point; it therefore acts like a barrier, despite it is not presented in the graph.

2.5.2.2 “Funneled” code

The Funneled Mixed Mode implementation of the algorithm is a combination of the pure MPI and the pure OpenMP codes with very few modifications. Like the pure OpenMP code, it also implements the whole algorithm within a single parallel region. MPI communication is performed within this parallel region by the master thread and therefore the thread safety level required is MPI_THREAD_FUNNELED.

![Figure 2.9: Schematic representation of the “Funneled” code.](image_url)

The reduction of delta sums is performed with an atomic operation in the after function, and not a reduction clause as in the “Master Only” code and as in the OpenMP code, two explicit barriers are needed for this purpose. This code was expected to perform slightly better than the “Master Only” code due to reduced OpenMP overheads.
2.5.2 “Multiple” code

This code has a number of differences to the other two codes and is also more complicated. While “Master Only” and “Funneled” codes employ the obvious nested parallelisation strategy of OpenMP nested within the MPI, the “Multiple” implementation operates them in the same level. Each thread communicates directly with the neighbouring thread from the neighbouring task in order to swap the part of the halo(s) it is adjacent to. This involves an extra initialization stage to map the threads that are opposite in the decomposition scheme. When a thread reaches the halo-swapping section, it only executes those sends and receives that correspond to the halos of the opposite process, using tags accordingly in order for messages not to mismatch.

There is also a greater number of variables that have to be made private, while in the other two codes were shared and this includes individual pointers to the halos, the MPI types and all the ADTs that refer to these structures. Additionally, the MPI library will also have its data kept for every thread. It is interesting to note here that not all halos of one side are the same. This means that each thread may define a different type than the adjacent thread, but this is not an issue because opposite threads on different tasks always define the same types for the halos they need to exchange.

Figure 2.10: Schematic representation of the “Multiple" code.
Chapter 3: Testing Methodology

3.1 The HPCx Service

3.1.1 Hardware

All the codes for this report were run on the HPCx service [5], an IBM p690+ SMP Cluster which is currently in its second phase, out of three phases of upgrades included in its roadmap. The system comprises 1600 IBM Power4+ CPUs clocked in 1.7GHz, organized in 50 frames. Every frame contains 4 Multi-chip modules (MCMs) with 4 Power4+ chips each and every chip holds 2 processors. Cache is organized in 3 levels. L1 cache is local to each processor, L2 is local to the Power4+ chip (shared among the two processors of the chip) and L3 cache is local to the MCM (shared among the 8 processors of the MCM). Characteristics of each cache are presented in table 3.1.

<table>
<thead>
<tr>
<th>line</th>
<th>type</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction</td>
<td>128 bytes</td>
<td>64K</td>
</tr>
<tr>
<td>L1 Data</td>
<td>128 bytes</td>
<td>32K</td>
</tr>
<tr>
<td>L2</td>
<td>128 bytes</td>
<td>1.5MB</td>
</tr>
<tr>
<td>L3</td>
<td>512 bytes</td>
<td>128MB</td>
</tr>
</tbody>
</table>

Each frame has 32GB of main memory installed which is shared among the processors. About 27GB of this memory is available to the user while the rest 5GB is used by the operating system and the switch network. Although the memory is shared, main memory modules (memory books) attached to an MCM are faster to access from the processors connected to this MCM. A schematic representation of memory hierarchies in a single node is presented in Figure 3.1.

Node communication is performed through IBM’s high performance switch also known as “Federation”. It is possible to achieve 1.45GB/sec when performing inter-node communication between two tasks, while the total bandwidth between two nodes is 8GB/sec. Messages sent between nodes have latency of about 6μs. Messages can be delivered in two ways. Short messages can be sent directly between two tasks, while longer messages require a rendezvous protocol to be used which has an increased overhead. The length of the message for rendezvous protocol is defined by the MP_EAGER_LIMIT system variable and it is suggested that this value is set to 65536, which is the highest possible.
Figure 3.1: Schematic representation of memory hierarchies in an HPCx node.

3.1.2 Software

Logical Partitioning in the second phase of HPCx is configured for the whole node, hence LPARs (Logical Partitions) are now synonymous to SMP Nodes, unlike in the previous phase where each node had 4 LPARs, each one assigned to a different MCM. Each LPAR is running its own copy of the operating system, which is AIX 5.2.
AIX is configured to run with memory affinity support enabled [4]. This means that the operating system can recognize the division of memory with MCMs and organize data structures accordingly. In order to enable allocation of memory from the local MCM, the environment variable MEMORY_AFFINITY must also be set to MCM. When a page fault occurs, the operating system allocates memory from the memory attached to the MCM. It is important to structure the code in a specific way in order to take better advantage of this feature.

We used IBM's VAC compiler for AIX version 6.0.0.10 and in all cases the thread safe version of the C runtime library was used. This is assured when the compiler name has a “_r” suffix and in order to compile the benchmarks we used “mpcc_r” which also links the MPI library. The compiler flags we used were the following:

- `-q64` to enable 64 bit addressing,
- `-O4` which was found to optimize better than `-O5` in many cases,
- `-qarch=pwr4` and `-qtune=pwr4` to optimize for the POWER4 architecture specifically,
- `-qsmp=omp:noauto` in order to avoid any implicit form of parallelization with threads,
- `-qpluscmt` to be able to use C++ style comments in the code and
- `-NDEBUG` to do a macro expansion in the “assert” statements used within the code in order to avoid any additional overheads.

### 3.1.3 Timing the code

It is important that the timing is as accurate as possible, especially due to the fact that most of the runs we performed were meant to last less than a second. The timer used in all codes is OpenMP’s high precision timer which is implemented in `omp_get_wtime()` function. This function returns a double precision floating point value that corresponds to the seconds elapsed from the operating system’s initial value of the real-time clock. The result of this function is not consistent among the threads in a team, which means that comparing results of two different calls to `omp_get_wtime()` on different threads in order to obtain a timing, might not produce accurate results. For most of our benchmarks, is typical for threads or processes to have different timings for certain parts of the code, a result of message scheduling and load imbalance, among others. For this reason it would probably not be a very good idea to assign only one thread or process in a team to do the timing work, because it is very likely that other threads would report different results.
Funneled, blocking
problem size: 600x600x600
process decomposition: 4x2x4
thread decomposition: 1x4x2
iterations: 20, computing delta every 1 iterations

Total
Average: 0.368829
Maximum: 0.368835, in process [2, 1, 2]
Minimum: 0.368823, in process [1, 0, 2]

Computation
Average: 0.250934
Maximum: 0.251672, in process [1, 0, 3]
Minimum: 0.250064, in process [0, 1, 3]

Barrier 1
Average: 0.001692
Maximum: 0.002570, in process [1, 0, 0]
Minimum: 0.001066, in process [1, 1, 1]

Swapping
Average: 0.098309
Maximum: 0.112647, in process [2, 1, 2]
Minimum: 0.082782, in process [0, 1, 0]

Local Delta Reduction
Average: 0.000150
Maximum: 0.000180, in process [2, 0, 3]
Minimum: 0.000138, in process [1, 0, 3]

Global Delta Reduction
Average: 0.017720
Maximum: 0.033638, in process [0, 1, 0]
Minimum: 0.003434, in process [2, 1, 1]

timing difference: 0.000024

Figure 3.2: Sample output produced with the Funneled code

It is a much better approach to have all threads and/or processes perform timing and collect the results as soon as the computation finishes. This way it is possible to extract average timings, that are more useful than individual timings in most cases, and other information about specific threads or processes as well. Figure 3.2 presents the results obtained from running a mixed mode code. Notice that the timings for swapping and global reduction vary greatly among processes, thus it would be likely to obtain inaccurate results if average timings were not computed.
Finally, except for timing each part of the code individually, the whole run is timed separately. When the timings for specific parts of the codes are collected, they are compared with this timing to verify how accurate the timing was, this is what the “timing difference” result in Figure 3.2 refers to. It is fine to have a very small positive timing difference because there is also some small overhead associated with function calls and with the timing itself. Whenever the timing difference is large, this can be an indication that there is something wrong with the timing, or something wrong in general.

In order to avoid having inaccurate timings in codes that would possibly produce significantly different results among different runs, all benchmarks have were run at least twice. In cases where there was an insignificant difference between those two runs, one of the two results was kept. Whenever timings differed significantly, the code under consideration was run multiple times and the best timing was kept. Codes that appeared to produce results difficult to interpret were also run multiple times.

3.1.4 The Timer ADT

There are many sections of code to be timed and it is possible to end up making a mess, if timing statements are placed carelessly. In order to keep the code tidy, a timer’s functionality was abstracted around a Timer ADT with a few basic functions implemented for this type, like Start, Stop and Reset. In this way it is possible to time a portion of code by simply wrapping a start-stop around it. For every part of the code that is timed, a different timer has to be used. Timing statements may be placed in any of the files, although it will be needed after the computation that they are accessed from this part of the code where the output timings are computed. It is therefore appropriate to declare all the timers in a header file (timers.h) and include it wherever timings need to be performed. It should also be included by the file where the output timings are computed.
Chapter 4: Analysis of the results

4.1 Benchmarking pure codes

The first series of benchmarks focused on the pure MPI and pure OpenMP codes only. It is used to investigate how the different ways of decomposing the problem affect performance. The results obtained from these tests will help us to interpret performance in the mixed mode codes.

4.1.1 Runs in 1 LPAR

The first series of tests is carried out in one Logical Partition. Initially, computation will be decomposed in one dimension only, in order to investigate how each direction of decomposition affects performance. Tests are performed for 2, 4, 8, 16 and 32 processors, using a problem size of 300x300x300 elements. Unfortunately it was not possible to run the benchmark on a single processor because memory requirements for this problem size were too high. A direct comparison of MPI and OpenMP code performance is also attempted. In the following bar diagrams, the computation time refers to the same portion of code for both MPI and OpenMP tests. For MPI, the halo swapping which involves point to point communication is displayed, followed by the collective communication for the reduction required to compute $\Delta$. For OpenMP codes, timings of the first barrier are displayed, as well as timings of the $\Delta$ computation. Note that in this timing both the atomic operation and the second barrier are included, in most graphs however, these timings were several orders of magnitude smaller than the computation timings and for this reason they are not visible.

Figure 4.1 shows performance obtained by decomposing the problem on two processors. The first thing to notice is the poor performance that the pure OpenMP code has when decomposition takes place along the x axis. This is a result of false sharing, a situation caused when the two processors are writing to regions in memory that are closer than the size of the cache line. As a result, they invalidate each other’s cache line and data have to be read from main memory or a remote cache, all the time. This situation may also occur rarely when decomposing along the y axis for a small problem size, and it is even less common in z axis decomposition.

Other than that, there is a small difference between timings in most OpenMP and MPI codes. In the two processor tests particularly, OpenMP computation is slightly faster than computation in the MPI tests. For more than two
processors it is the MPI programs that are faster. Therefore it is not the case that the compiler performs any sort of optimization in one program better than in the other. After all, the code associated with the Jacobi computation is in a separate file and it is identical in both programs, so this should not be the reason.

![Figure 4.1: Pure OpenMP and MPI graph for 2 processor decomposition](image)

Timings for MPI halo swaps and collective communications are rather negligible at this stage, but there is a first indication that x decomposition is also more expensive for MPI communication. Taking a look at table 4.1 in the Appendix reveals that swapping in the x axis decomposition takes about double the time than in the z decomposition and about nine times longer than in y decomposition.

As the number of processors doubles, the effects of false sharing become far worse. While in the two processor decomposition tests, the OpenMP test that decomposed the program along the x axis took only 36% more than the average of the other two OpenMP tests, in the four processor test illustrated in Figure 4.2, it took about 125% longer. This is because intervals between segments assigned to different CPUs become even smaller, so more invalidations occur. It is worth noting however that in the previous case where we had 2x1x1 decomposition, in the very unlikely case that the two segments were processed in a perfectly synchronous way (i.e. when CPU1 processes element \([x, \beta, \gamma]\), CPU2 always processes \([x+150, \beta, \gamma]\) there would be no false sharing in the L2 cache, because the L2 line is only 128 bytes long. The same applies for 4 processor tests as well, and in fact it would take 10 processors in order for a synchronously processed problem to have L2 invalidations. Practically this would never happen and there will always be some invalidation saved due to asynchronous processing. For this reason, it is expected that false sharing will keep getting worse as the number of processors increases. Performance in these tests may also have small variations among runs, possibly a result of different binding of threads to processors by the operating system. We should also take into account that the MP_TASK_AFFINITY was not set.
to MCM for OpenMP runs. In this case it is more logical that when two threads are placed in processors of the same chip, they would not suffer from false sharing as heavily, as would happen if they were on a different MCM.

While the computation time appears to take about the same in the rest of the tests, MPI communication takes more than 40% of the total time in the MPI tests. It is also very interesting that there is more time spent in the reduction, instead of the halo swapping which is a more expensive procedure, a result of a load imbalance in the halo swapping. As we have already discussed in the implementation, this is a false impression given by the graph, because in order to do the reduction, all processes have to send some data, hence the processors that reach this point earlier have to wait for the others to finish swapping. Placing a process barrier just before the reduction would cause most of the time currently associated with the collective operation to be spent in the barrier.

Increasing the number of processors to 8, makes more clear for MPI programs a situation that was also visible in the previous graphs. It seems that a z-axis decomposed problems take more time to complete than y-axis decomposed problems, due to the fact that they need more time for communication. This is unexpected, if we take into consideration that front-back halos associated with the z-axis decomposition are almost contiguous in memory (they only have very small gaps between succeeding chunks, in order to jump through the halos). Upper-lower halos associated with y decomposition, however, consist of chunks with very big gaps between them. Our initial expectations were that in both cases, prefetching should work equally well, if not better, for the z-decomposition case. Moreover, front-back halo swaps should also be more cache-friendly because the computation that preceded the swap should have up to 100% of the back halo in the cache. It is even stranger that the z-decomposed problem performs as badly as the x-decomposed one, which uses halos that are not contiguous at all.

Figure 4.2: Pure OpenMP and MPI graph for 4 processor decomposition
Decomposing the problem with 16 processors produces the expected performance in MPI runs. Problems decomposed along the z axis now take the same time to swap as those that employ y-axis decomposition. Although the fact that this strange behaviour stops when the number of processors exceeds 8, it is not really safe to say that this is an MCM related issue, because the allocation of processes is being made in a round-robin fashion once MP_TASK_AFFINITY=MCM has been specified. This means than in both cases processes would span across all 4 MCMs.

Finally, decomposing the problem on 32 processors follows the same pattern as with 16 processors. It is interesting that with the exception of false-sharing affected OpenMP runs, the rest were able to demonstrate linear speedup, despite the decomposition being one dimensional. In the last set of runs, the problem was also slightly load imbalanced, since 300/32 = 9.375, which means that some threads will take 9 slices, while others will take 10. We should therefore not expect significant differences in performance in the Mixed Mode codes depending on how threads are decomposed. This of course does not apply to the “Multiple” Mixed mode benchmarks, where threads are involved with communication individually and therefore thread geometry does matter.
4.1.2 Runs in 8 LPARs

The second series of tests involves MPI runs in various kinds of problem decompositions and is carried out on 8 LPARs. The problem size is 600x600x600 elements and it is proportional to the one employed for the previous tests, so direct comparisons may also apply in some cases. Like all MPI tests, these were also configured to run with MP_TASK_AFFINITY set to MCM. For the most part, the MPI test performed as predicted, with process geometries that minimise communication being favoured. Messages appear to be scheduled very nicely by the MPI library, something really beneficial in cases where there is decomposition along the x axis, which was found to be significantly more expensive in the previous tests.

![Figure 4.5: Pure OpenMP and MPI graph for 32 processor decomposition](image)

![Figure 4.6: Pure MPI graph for decomposition across 8 LPARs](image)
Although the best way to decompose the problem was found to be the 4x8x8 geometry, 8x4x8 did not perform badly at all. It is interesting to note that in the 4x8x8 scheme more time is spent in the reduction, rather than in swapping, which tells us that the cause of delay lies in a few processes taking longer to swap halos whilst in the 8x4x8 geometry most of the time is actually spent in the halo swapping. This is probably an indication that the MPI library in the second case can schedule messages in a more efficient way, which makes the effect of expensive left-right swaps less evident. Another thing to notice is that in all cases, decomposing along the z axis performs faster than decomposing along y, sometimes significantly faster, in contrary to the tests performed in one LPAR that showed z-decomposed problems communicating either equally fast or slower for a smaller number of processors. MPI handles inter-node communication in a different way than intra-node, where communication is not routed through the switch network. This is possibly a reason for this difference in performance, however this would not necessarily mean that the effect of z-axis decomposed problems performing worse than y-axis ones is not present anymore. It might still be present but since inter-node communication plays the key role in performance, it would not be visible.

The rest of the runs make clear that it is not a good idea to try avoiding left-right swaps because of their cache unfriendliness. Doing so greatly increases communication cost and in most of these cases, it takes more time to complete the communication rather than the actual computation.

### 4.2 Mixed Mode Benchmarks

In the following section we will analyse the timings we obtained by the mixed mode codes. At first we evaluate the results of the codes run in 1 node only and then we examine each code separately, for runs performed in 8 nodes. There are cases where the interpretation of the results becomes nontrivial.

#### 4.2.1 Runs in 1 LPAR

Although measuring mixed mode performance in one node is not the main goal of this dissertation, it will provide an interesting foundation for interpreting mixed mode codes run over multiple LPARs. It will also provide an indication of some more general characteristics of code performance.

Mixed mode codes were run with a problem size of 300x300x300 elements, using 4 processes with 8 threads each. They were configured to run with MP_TASK_AFFINITY set to TRUE and MEMORY_AFFINITY set to MCM,
in order to eliminate problems related to misplacement of tasks and remote memory references among MCMs. The decomposition used is 1x2x2 for the MPI tasks and 1x2x4 for the OpenMP threads. Results show the “Master Only” code to be performing significantly better than the others. This is not a really fair comparison because the code that performs the Jacobi iteration is slightly different and as a result a pointer aliasing issue which degrades performance in all the other codes is not present in this one. We will not focus on this issue as it is only related to the compiler.

![Mixed Mode code graph for decomposition in 1 LPAR.](image)

Other than that, Funneled and Master Only runs appear to be really close in the time spent for point to point and collective communication, which is very logical because both codes are doing this in the very same way. Performance in the “Multiple” test however is very poor compared to the other two. In this test the switch network is not involved and therefore the only rational explanation is a bad memory access pattern. Since communication is attempted at the same time, halos may also be accessed at the same time from the MPI library. With this kind of decomposition there are 5 threads per task that would be trying to communicate, one of them in 2 directions. It is ambiguous how the MPI library handles this situation and there are two distinct cases:

- communication is carried out for one thread at a time
- all threads communicate at the same time

In both cases, the memory access pattern can only be worse than those in Funneled and Master Only benchmarks where only one thread communicates. If communication is carried out at the same time for many threads, this might be a possible cause of cache thrashing which could provide an explanation for
this degradation in performance. If the MPI library performs communication for one thread at a time, then it could also be explained by the fact that parts of the halos are accessed for each thread rather than the full halo. This way swapping becomes complicated and it is more difficult for data to be prefetched effectively when swapping finishes for one thread and another takes over. Halo parts are also sparser in memory than full halos.

Different use of the MPI libraries does not affect benchmark results. The initialization in mixed mode codes is done with MPI_Init_threads in order to request the thread safety level that is required for the benchmark. The MPI implementation in HPCx ignores the value of the request parameter and returns a value according to the runtime library that was already set. Therefore, it can only assure that there is the required thread safety but does nothing more than that. Defining whether the thread safe version of the MPI library will be used is done by setting the variable MP_SINGLE_THREAD. If this is set to no or the variable is not defined, then the thread safe version is used. If set to yes, the unsafe version is used which may perform better due to the lack of thread locks associated with the thread safe version. Setting MP_SINGLE_THREAD to yes can be used with MPI programs requiring safety level that is less or equal to MPI_THREAD_FUNNELED, whereas programs that require the MPI_THREAD_MULTIPLE safety level should either have this variable set to no or not set at all. In our tests this variable was not set and therefore the thread safe library was used in all cases. Note that this has nothing to do with the “_r” suffix in the compiler’s name. This suffix implies that the thread safe compiler and C runtime library are used.

Finally, it appears that the barrier after the computation takes double the time of the same barrier in the Funneled run. This is something that would happen in case one or more threads were load imbalanced in computation, but this cannot be true because this would have the same effect in the Funneled run as well. In case the cache-thrashing scenario is actually accurate, another possible reason could be that cache thrashing affects the thread with the 2-way communication more than the others and as a result it takes more time to fill the cache back when it starts the computation loop, which means that other threads would have to wait longer. This should have a small hit in computation time as well though, but there is no clear evidence in the timings to support this scenario. Such timing differences are very closely approximating the timing accuracy so it would be better if they are not taken into account. We should also note that the “Multiple” mixed mode benchmark demonstrated a small variability in performance among different runs, which makes such kind of assumptions even less reliable.
4.2.2 Runs in 8 LPARs

It is a general observation that mixed mode codes performed exceptionally well, outperforming pure MPI code in most cases. Performance characteristics of “Master Only” and “Funneled” benchmarks were almost identical, but “Multiple” showed poor performance compared to the other two. The reasons are not perfectly clear on what is the cause of this poor performance.

4.2.2.1 Master Only and Funneled benchmarks

The most simple mixed mode implementation of the Jacobi algorithm, the “Master Only”, proved to be very effective for this problem. There was an initial concern that performance in this series of runs might be affected by the overhead associated with opening and closing the parallel region in every iteration, but the timings do not indicate that this has any noticeable effect. In a more general basis, OpenMP overheads do not seem to have any noticeable hit in performance because most of the time is spent in computation and swapping. For example, placing more barriers after the first barrier following the computation would also not affect performance. In this case, threads would be already synchronised so the only additional cost would be the extra synchronisation overhead. The timings we obtained for the extra barrier in a test written specifically for this purpose were very close to the timing precision. The reason for OpenMP overheads not having an effect is the problem size we are using which is quite large. In case a very small problem size was used, it would be very likely that the effect would be more obvious; however it is very unlikely to have a very small problem size with many iterations in a real world scenario.

The Funneled runs performed in a very similar way; in fact the performance ranking of attempted decompositions is almost identical to the Master Only benchmark. All the performance characteristics observed in the pure OpenMP code and the mixed mode benchmarks performed in 1 node are present in multiple node runs.

We can categorise the attempted decompositions in two major groups: those that have 1 task running per node (8 tasks in total) and those that have 4 tasks running per node (32 tasks in total) each one bound to an MCM. We also ran 3 tests with 2 tasks per node, hence 16 tasks in total which proved to perform remarkably well.

It is very difficult to define the optimal number of tasks required in order for the benchmark to perform well, because as we can see in the graphs, they are not clearly separated. There are a few factors that influence performance in each case that should be discussed separately.
Figure 4.8: “Master Only” performance graph for runs in 8 LPARs

Note that whenever two decompositions appear for a mixed mode code next to each other separated by comma (e.g. 1x2x4(8), 1x4x8(32)), the first one refers to the MPI decomposition and the second to the OpenMP. The numbers in parentheses are the total number of processes or threads.

Overlapping communication: The ability for the chosen MPI decomposition to overlap is vital for obtaining good performance. It is worth comparing two specific cases of decomposition to explain this in a better way. We will compare the 2x2x2 MPI decomposition with the 4x1x8.

In the first case, each process needs to send and receive 3 halos, each one being about 350 KB. There are 12 swaps to be performed in total and all of them are routed through the switch network, which adds 6μs latency for each swap. This should be about 0.00032 seconds latency for each task (18 iterations × 3 swaps per iteration × 6μs), a number which should not be a
consideration in performance. It is more expensive to swap left-right halos though which are associated with x decomposition. Although the amount of data exchanged in total is minimal comparing to other decompositions, the problem is that halos are big and there is not much to be done with scheduling. If the left-right halo swap takes longer to complete, there is nothing to overlap with and as a result the other processes will end up waiting.

Figure 4.9: “Funneled” performance graph for runs in 8 LPARs

In the second case, there are 52 swaps to be made per iteration with sizes of halos being significantly smaller (there are 44K and 22K halos). Some processes have to perform more halo swaps than others, however not all of these swaps are going through the switch network. This means that intra-node halo swaps can overlap with inter-node, which can greatly decrease swapping time, especially if the scheduler is smart enough to put inter-node swaps first.

**Exploiting the switch network:** In the previous example, there is another issue which possibly affects performance. In the 4x1x8 decomposition, there are many inter-node swaps that can take place in one node simultaneously, while with 2x2x2 decomposition only one process per node communicates. Since there is a lower speed limit for task-to-task communication, the
network is better exploited when multiple processes communicate, because the overall node-to-node communication speed is higher. But it is ambiguous whether this affects performance at all because the bandwidth of the switch network is really high, so it is not likely that the swaps can be bandwidth limited. If there was an impact in performance due to this particular cause, then all of the runs that use 1 task per node would not perform that well. Although the first places in the ranking are occupied by decompositions that used more than one processes per node, performance of some decompositions with only one process per node was also be very good.

**Number of threads:** It is presented in [1] that benchmarks run in phase 1 of HPCx showed that running fewer threads than the available number of processors proved to perform slightly better. For this reason we ran two of our tests using 30 threads per node, instead of 32. This did not prove to perform better though and as we can see in the graphs, in these cases computation time has also been increased. We should note that in phase 1, LPARs were configured per MCM, not per node as in phase 2.

**Halo size:** The size of the halos seemed to be the most significant factor in all MPI and mixed mode benchmarks performed so far, but this appears to have changed in the multiple node mixed mode runs. If we take a careful look at the results, we can observe that for those runs that do not perform intra-node communication at all, the size of the halos does not matter. It is more important not to decompose along the x axis. This is exactly the case if we compare the poorly performing 2x2x2 set of decompositions with the 1x2x4 which performed quite well, although it uses larger halos. When there are intra-node MPI communications (32 task runs), the situation is different. For example, 4x1x8 decomposition performed worse than 4x2x4 which has smaller halos, but not better than 2x2x8 which has somewhat larger halos but does not decompose in the x axis as much as the others. It appears that the final result derives from both inter-node and intra-node MPI decomposition.

**Comparing performance with pure MPI**

The Funneled code that employed the best performing decomposition compared to the best pure MPI run was found to be only 7% faster. This might not be a very big difference but we should also take into consideration that almost all Funneled and Master Only runs produced decent performance. The same was not true for many decomposition schemes in the pure MPI benchmark. This tells us that mixed mode codes are more tolerant in non-optimally selected decomposition schemes. It is possibly easier with our benchmarks to predict which decomposition is optimal, since they used cubic lattices. In real problems that do not use cubic lattices this would be less
obvious. It is worth reviewing a few timings taken from MPI and Funneled runs that used equivalent decomposition schemes:

Table 4.12: comparison of pure MPI and Funneled runs with equal geometry

<table>
<thead>
<tr>
<th>Decomposition</th>
<th>Pure MPI</th>
<th>Funneled</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI: 4x8x8, Funn.: (tasks)4x2x4, (threads)1x4x2</td>
<td>0.307238</td>
<td>0.368829</td>
</tr>
<tr>
<td>MPI: 2x8x16, Funn.: (tasks)2x4x4, (threads)1x2x4</td>
<td>0.326078</td>
<td>0.330333</td>
</tr>
<tr>
<td>MPI: 1x8x32, Funn.: (tasks)1x4x4, (threads)1x2x8</td>
<td>0.456736</td>
<td>0.302634</td>
</tr>
<tr>
<td>MPI: 2x16x8, Funn.: (tasks)2x4x4, (threads)1x4x2</td>
<td>0.486354</td>
<td>0.328346</td>
</tr>
<tr>
<td>MPI: 1x16x16, Funn.: (tasks)1x4x4, (threads)1x4x4</td>
<td>0.53487</td>
<td>0.302703</td>
</tr>
</tbody>
</table>

Figure 4.10: Comparison graph of selected pure MPI and Funneled runs.

We can see that except for the 4x8x8 MPI decomposition where the pure MPI version performed 20% faster, there is one more performing slightly better than the Funneled equivalent (1% faster). The other three decompositions resulted in 50% - 75% superior performance for the Funneled code. Since the pure MPI code is totally dependent on the decomposition, the user has to be really careful with it. The number of messages that need to be exchanged in total can be tens of times higher than in the mixed mode codes. In the mixed mode codes on the other hand, there is almost zero performance impact from the shared memory part as it can scale almost linearly. The MPI part is significantly lighter, so it cannot be devastating even if the decomposition selected is not the best. We should also note that in the pure MPI code the message scheduler has a vital role due to the large volume of messages that need to be exchanged. For this reason, if a good decomposition is used, it is possible to achieve very good performance because inter-node messages can
overlap efficiently with intra-node. This is actually why MPI performance is comparable to mixed mode for a small number of decomposition schemes.

4.2.2.2 “Multiple” Mixed Mode code benchmarks

The “Multiple” code is more complicated than the other two and also produced results that are more difficult to interpret. Although in many cases performance was better than the pure MPI code, there was not a single run that was superior to the Master Only or Funneled benchmarks using the same decomposition. It was also very strange that benchmarks did not produce the same results among different runs. Figure 4.11 presents the timings for some moderately performing runs of the “Multiple” code. Note that the ranking for the worst performing decomposition schemes is not accurate and should not be taken into consideration. It is possible that the same decomposition schemes perform much worse if they are re-run and sometimes total execution time exceeds 1 sec. We can observe that the ranking for the runs that used 32 tasks is not significantly different than those produced by the other two mixed mode codes. For these decomposition schemes, results were not very different among runs, but variability was present and it was not insignificant. The program starts behaving in a very strange way when less than 32 tasks are used. Variability increases when 16 tasks are used in total and it gets even worse for 8 tasks.

![Mixed Mode, Multiple](image)

*Figure 4.11: “Multiple” performance graph for runs in 8 LPARs*
It is very likely that the problem is related to MCM usage. According to [3], AIX spreads threads across the MCMs to balance memory accesses, but there is no guarantee on how this is done. In all of the tests where 4 tasks were placed per node (32 tasks in total), the TASK\_AFFINITY variable was also set to MCM and therefore all the threads of a task were placed in a single MCM. We will make an assumption on what possibly happens which may not be perfectly accurate. We will use as an example the (tasks)1x4x8 – (threads)1x4x2 decomposition scheme contrasted to the (tasks)1x2x4 – (threads)1x8x4 which is the equivalent for 32 threads per node. In the first case, there are 5 threads in each process that perform point-to-point communication. Of these threads, 4 are swapping one part of a halo, while there is one that has to perform communication for two parts that belong to two different halos. No matter how threads are spread, once TASK\_AFFINITY is set to MCM, each MCM will have exactly this configuration. The only thing that can change is the placement in different processors within the same MCM. In the second case however, due to the fact that this restriction is not present, it is possible for threads that have to perform point-to-point communication to be spread unevenly across MCMs. Thus, it can happen that an MCM might have very few threads that communicate, while another one has many. Figure 4.12 illustrates an extreme case, but it’s not unlikely for something similar to happen.

![Diagram of MCM and threads](image-url)

- **MCM**
- **Thread performing 1 comm**
- **Thread performing 2 comm**
- **Thread performing 0 comm**
Figure 4.12: The optimal binding of threads in a node (left) and an example of a really bad binding (right).

We assume that when many threads are trying to communicate simultaneously, cache thrashing occurs. If it happens in one node that the placement of threads is that bad, then the overall performance would be really bad. Proving that this assumption is correct would require that we find out the physical processors that each thread is bound to. Unfortunately, we were not able to find a way to do this. Although there are many ways to bind a thread to a specific processor, there is no straightforward way to allow querying the physical processor that a thread is already running on.

We performed another test though which seems to support our allegations. We ran the same test with the only difference that we put the MPI communication code in a critical section. This way, only one thread communicates at a time. Results of this test are presented in Figure 4.13 and we can clearly see that variations in performance did not cease to exist, but the difference in timings has significantly decreased.
Chapter 5: Conclusion

The results of the benchmarks demonstrated that the mixed mode codes were able to outperform the pure MPI codes in most cases. The selected decomposition scheme proved to be the most critical factor that affected performance. In contrast to other benchmarks performed on HPCx that employed the same algorithm but operated on 2-dimensional data, adding a third dimension makes more difficult to predict what is the decomposition that should be selected in order for the code to perform well. This is because of the different cost associated with decomposing the problem on specified axes and the fact that it is not always obvious how directional decompositions can interfere with each other. It is also sometimes beyond the understanding of the user how the MPI library will perform on a given decomposition scheme, which is determined by factors such as the message scheduling and the handling of threads for the mixed mode codes.

Of the three mixed mode codes that were tested, two of them, namely the “Master Only” and “Funneled” codes, proved to have the same performance characteristics. This was expected however because these two codes use the same kind of parallelism. The only difference they have is the usage of OpenMP and the overheads associated with it. However, we were unable to notice any penalty associated with OpenMP usage at all, because the problem size we used is quite large, hence most of the time is spent in computation. Thus, the pure OpenMP code was found to be able to scale linearly for up to 32 processors.

The third mixed mode code referred to as “Multiple” had significantly different performance characteristics than the other two. This was also expected because instead of the nested parallelization that the other two codes use, this code employs a flat parallelization strategy between MPI and OpenMP, with individual threads of neighbouring tasks being able to communicate with each other. Although the code performed in many cases better than the pure MPI code, it proved that this parallelization strategy is not good for HPCx, because the theoretical advantages that this solution has over the other two are undermined by the memory hierarchy and the thread-unfriendly MPI implementation. This is supported by the fact that serialising the communication greatly improved performance.
Besides their superiority in performance for optimally decomposed problems, the mixed mode codes were also found to have a decent performance in less optimal decompositions. The latter was not the case with the pure MPI code, which proved not to perform well for a number of decompositions.

Finally, there are many things that the programmer should consider when writing mixed mode programs and relate primarily to the design of the code. It is important that such codes are well designed because the increased complexity can lead to a code that is very difficult to maintain. This involves a reasonable amount of time in development. It is not very easy to answer the question whether it is really worth spending this additional time though. There is probably not a single answer to this question because the time needed for development can be dependent to various issues. For example, it could be more difficult to parallelise an already existing pure MPI code which was also written without having threads in mind, than to write it from scratch. But spending the extra time also involves the risk that the mixed mode code may fail to perform better than the pure MPI. There are a number of factors that determine that, such as the problem size and the algorithm that is implemented, so it is up to the developer to decide whether it is really worth it.
Appendix A: Timings

Note that whenever two decompositions appear for a mixed mode code next to each other separated by comma (e.g. 1x2x4(8), 1x4x8(32)), the first one refers to the MPI decomposition and the second to the OpenMP. The numbers in parentheses are the total number of processes or threads.

Table 4.1: Pure OpenMP and MPI timings for 2 processor decomposition

<table>
<thead>
<tr>
<th></th>
<th>1 x 1 x 2</th>
<th>1 x 2 x 1</th>
<th>2 x 1 x 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>4,186845</td>
<td>4,074716</td>
<td>4,146658</td>
</tr>
<tr>
<td>swap/barrier</td>
<td>0,058001</td>
<td>0,005561</td>
<td>0,014292</td>
</tr>
<tr>
<td>delta</td>
<td>0,055257</td>
<td>0,000186</td>
<td>0,000183</td>
</tr>
</tbody>
</table>

Table 4.2: Pure OpenMP and MPI timings for 4 processor decomposition

<table>
<thead>
<tr>
<th></th>
<th>1 x 1 x 4</th>
<th>1 x 4 x 1</th>
<th>4 x 1 x 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>1,941819</td>
<td>2,003651</td>
<td>1,919851</td>
</tr>
<tr>
<td>swap/barrier</td>
<td>0,277845</td>
<td>0,009788</td>
<td>0,218785</td>
</tr>
<tr>
<td>delta</td>
<td>0,678561</td>
<td>0,000242</td>
<td>0,52664</td>
</tr>
</tbody>
</table>

Table 4.3: Pure OpenMP and MPI timings for 8 processor decomposition

<table>
<thead>
<tr>
<th></th>
<th>1 x 1 x 8</th>
<th>1 x 8 x 1</th>
<th>8 x 1 x 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>0,986428</td>
<td>1,007124</td>
<td>0,961285</td>
</tr>
<tr>
<td>swap/barrier</td>
<td>0,223226</td>
<td>0,015761</td>
<td>0,166785</td>
</tr>
<tr>
<td>delta</td>
<td>0,595555</td>
<td>0,000341</td>
<td>0,448279</td>
</tr>
</tbody>
</table>

Table 4.4: Pure OpenMP and MPI timings for 16 processor decomposition

<table>
<thead>
<tr>
<th></th>
<th>1 x 1 x 16</th>
<th>1 x 16 x 1</th>
<th>16 x 1 x 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>0,486641</td>
<td>0,509045</td>
<td>0,506938</td>
</tr>
<tr>
<td>swap/barrier</td>
<td>0,13502</td>
<td>0,003812</td>
<td>0,265341</td>
</tr>
<tr>
<td>delta</td>
<td>0,278998</td>
<td>0,000493</td>
<td>0,391023</td>
</tr>
</tbody>
</table>
### Table 4.5: Pure OpenMP and MPI timings for 32 processor decomposition

<table>
<thead>
<tr>
<th></th>
<th>1 x 1 x 32</th>
<th>1 x 32 x 1</th>
<th>32 x 1 x 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPI</td>
<td>OMP</td>
<td>MPI</td>
</tr>
<tr>
<td>computation</td>
<td>0.24987</td>
<td>0.270879</td>
<td>0.250369</td>
</tr>
<tr>
<td>swap/barrier</td>
<td>0.100326</td>
<td>0.005890</td>
<td>0.092515</td>
</tr>
<tr>
<td>delta</td>
<td>0.210449</td>
<td>0.000802</td>
<td>0.220711</td>
</tr>
</tbody>
</table>

### Table 4.6: Pure MPI timings for decomposition in 8 LPARs

<table>
<thead>
<tr>
<th></th>
<th>4x8x8</th>
<th>8x4x8</th>
<th>2x8x16</th>
<th>8x8x4</th>
<th>1x8x32</th>
<th>2x16x8</th>
<th>1x16x16</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>0.246735</td>
<td>0.246662</td>
<td>0.247495</td>
<td>0.248428</td>
<td>0.252133</td>
<td>0.247734</td>
<td>0.246871</td>
</tr>
<tr>
<td>swap</td>
<td>0.027109</td>
<td>0.050696</td>
<td>0.023485</td>
<td>0.052544</td>
<td>0.036141</td>
<td>0.025031</td>
<td>0.04414</td>
</tr>
<tr>
<td>delta</td>
<td>0.033361</td>
<td>0.021384</td>
<td>0.055067</td>
<td>0.028256</td>
<td>0.168431</td>
<td>0.213556</td>
<td>0.243828</td>
</tr>
<tr>
<td></td>
<td>1x32x8</td>
<td>1x64x4</td>
<td>1x4x64</td>
<td>1x2x128</td>
<td>1x128x2</td>
<td>1x1x256</td>
<td>1x256x1</td>
</tr>
<tr>
<td>computation</td>
<td>0.246694</td>
<td>0.249145</td>
<td>0.253414</td>
<td>0.259</td>
<td>0.258913</td>
<td>0.268451</td>
<td>0.286868</td>
</tr>
<tr>
<td>swap</td>
<td>0.066799</td>
<td>0.083657</td>
<td>0.051558</td>
<td>0.089842</td>
<td>0.132314</td>
<td>0.22335</td>
<td>0.233332</td>
</tr>
<tr>
<td>delta</td>
<td>0.226478</td>
<td>0.253503</td>
<td>0.291587</td>
<td>0.296297</td>
<td>0.270707</td>
<td>0.425558</td>
<td>0.41115</td>
</tr>
</tbody>
</table>

### Table 4.7: Mixed Mode code timings for decomposition in 1 LPAR

<table>
<thead>
<tr>
<th></th>
<th>Master Only</th>
<th>Funnelled</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>0.210147</td>
<td>0.246065</td>
<td>0.246357</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0</td>
<td>0.005555</td>
<td>0.01307</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.037381</td>
<td>0.037605</td>
<td>0.063321</td>
</tr>
<tr>
<td>Local Delta Reduction</td>
<td>0</td>
<td>0.000401</td>
<td>0.00049</td>
</tr>
<tr>
<td>Global Delta Reduction</td>
<td>0.011356</td>
<td>0.012976</td>
<td>0.044239</td>
</tr>
</tbody>
</table>
Table 4.8: “Master Only” timings for decomposition in 8 LPARs

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Computation, Delta, Barrier</th>
<th>Swapping</th>
<th>Global Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x4x4(32), 1x4x2(8)</td>
<td>0.207977, 0.210147, 0.209283</td>
<td>0.02923, 0.037381, 0.037478</td>
<td>0.012253, 0.011356, 0.01659</td>
</tr>
<tr>
<td>1x4x4(16), 1x2x8(16)</td>
<td>0.209687, 0.210584, 0.205015</td>
<td>0.052405, 0.052732, 0.064082</td>
<td>0.006648, 0.00931, 0.010628</td>
</tr>
<tr>
<td>2x4x4(32), 1x4x2(8)</td>
<td>0.220489, 0.205523, 0.208635</td>
<td>0.052163, 0.063556, 0.065543</td>
<td>0.008059, 0.0143, 0.013795</td>
</tr>
<tr>
<td>2x4x4(32), 1x2x4(8)</td>
<td>0.204705, 0.202809, 0.206095</td>
<td>0.019528, 0.01845, 0.022419</td>
<td>0.019528, 0.01845, 0.022419</td>
</tr>
<tr>
<td>4x1x8(32), 1x2x4(8)</td>
<td>0.209042, 0.207058, 0.20825</td>
<td>0.208202, 0.208663, 0.217565</td>
<td>0.010692, 0.015156, 0.012388</td>
</tr>
<tr>
<td>2x2x4(16), 1x4x4(16)</td>
<td>0.20825, 0.208663, 0.217565</td>
<td>0.213941, 0.210728, 0.205303</td>
<td>0.025596, 0.020519, 0.021142</td>
</tr>
</tbody>
</table>
Table 4.9: “Funneled” timings for decomposition in 8 LPARs

<table>
<thead>
<tr>
<th></th>
<th>1x4x8(32), 1x4x2(8)</th>
<th>1x4x4(16), 1x2x8(16)</th>
<th>1x4x4(16), 1x4x4(16)</th>
<th>1x2x4(8), 1x4x8(32)</th>
<th>1x2x4(8), 1x8x4(32)</th>
<th>1x2x4(8), 1x5x6(30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>0.243637</td>
<td>0.246065</td>
<td>0.243697</td>
<td>0.245697</td>
<td>0.243472</td>
<td>0.261017</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.007706</td>
<td>0.005555</td>
<td>0.0087</td>
<td>0.006792</td>
<td>0.008662</td>
<td>0.003841</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.029034</td>
<td>0.037605</td>
<td>0.037033</td>
<td>0.051072</td>
<td>0.052577</td>
<td>0.051125</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.000148</td>
<td>0.000401</td>
<td>0.000407</td>
<td>0.000907</td>
<td>0.000922</td>
<td>0.000836</td>
</tr>
<tr>
<td>Global Delta</td>
<td>0.010481</td>
<td>0.012976</td>
<td>0.012833</td>
<td>0.009333</td>
<td>0.008516</td>
<td>0.009076</td>
</tr>
<tr>
<td>2x2x8(32), 1x4x2(8)</td>
<td>0.242857</td>
<td>0.25044</td>
<td>0.25079</td>
<td>0.245722</td>
<td>0.250934</td>
<td>0.251438</td>
</tr>
<tr>
<td>Computation</td>
<td>0.004988</td>
<td>0.004695</td>
<td>0.004633</td>
<td>0.00541</td>
<td>0.001692</td>
<td>0.004793</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.064382</td>
<td>0.063011</td>
<td>0.063345</td>
<td>0.065132</td>
<td>0.098309</td>
<td>0.09786</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.00015</td>
<td>0.000147</td>
<td>0.0015</td>
<td>0.000152</td>
<td>0.00015</td>
<td>0.000147</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.013742</td>
<td>0.010027</td>
<td>0.01139</td>
<td>0.016816</td>
<td>0.01772</td>
<td>0.017385</td>
</tr>
<tr>
<td>Global Delta</td>
<td>0.246851</td>
<td>0.243883</td>
<td>0.244454</td>
<td>0.244003</td>
<td>0.243546</td>
<td>0.259574</td>
</tr>
<tr>
<td>4x1x8(32), 1x2x4(8)</td>
<td>0.246851</td>
<td>0.243883</td>
<td>0.244454</td>
<td>0.244003</td>
<td>0.243546</td>
<td>0.259574</td>
</tr>
<tr>
<td>Computation</td>
<td>0.005286</td>
<td>0.005971</td>
<td>0.007459</td>
<td>0.006713</td>
<td>0.00689</td>
<td>0.003787</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.10147</td>
<td>0.121904</td>
<td>0.120689</td>
<td>0.207446</td>
<td>0.208425</td>
<td>0.20687</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.000149</td>
<td>0.000403</td>
<td>0.000409</td>
<td>0.000922</td>
<td>0.000917</td>
<td>0.000834</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.028141</td>
<td>0.019253</td>
<td>0.024333</td>
<td>0.013167</td>
<td>0.013139</td>
<td>0.01686</td>
</tr>
</tbody>
</table>
Table 4.10: “Multiple” timings for decomposition in 8 LPARs

<table>
<thead>
<tr>
<th></th>
<th>1x4x8(32), 1x4x2(8)</th>
<th>2x2x8(32), 1x2x4(8)</th>
<th>2x2x8(32), 1x4x2(8)</th>
<th>1x4x4(16), 1x4x4(16)</th>
<th>2x4x4(32), 1x2x4(8)</th>
<th>2x4x4(32), 1x4x2(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>0.243593</td>
<td>0.245822</td>
<td>0.243245</td>
<td>0.243732</td>
<td>0.250465</td>
<td>0.2502</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.008842</td>
<td>0.006657</td>
<td>0.00621</td>
<td>0.012299</td>
<td>0.005939</td>
<td>0.006088</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.035781</td>
<td>0.066121</td>
<td>0.067829</td>
<td>0.058283</td>
<td>0.065683</td>
<td>0.067604</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.000202</td>
<td>0.000201</td>
<td>0.000269</td>
<td>0.000498</td>
<td>0.000204</td>
<td>0.000206</td>
</tr>
<tr>
<td>Global Delta</td>
<td>0.014496</td>
<td>0.013555</td>
<td>0.016042</td>
<td>0.021075</td>
<td>0.013636</td>
<td>0.013078</td>
</tr>
<tr>
<td></td>
<td>1x4x4(16), 1x2x4(8)</td>
<td>4x2x4(32), 1x2x4(8)</td>
<td>4x2x4(32), 1x4x2(8)</td>
<td>4x1x8(32), 1x2x4(8)</td>
<td>2x2x4(16), 1x8x2(16)</td>
<td>1x2x4(8), 1x4x8(32)</td>
</tr>
<tr>
<td>Computation</td>
<td>0.246357</td>
<td>0.250886</td>
<td>0.250448</td>
<td>0.246797</td>
<td>0.246499</td>
<td>0.246127</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.01307</td>
<td>0.014748</td>
<td>0.01438</td>
<td>0.01508</td>
<td>0.010702</td>
<td>0.038669</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.063321</td>
<td>0.087354</td>
<td>0.091016</td>
<td>0.091304</td>
<td>0.13359</td>
<td>0.094487</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.00049</td>
<td>0.000205</td>
<td>0.000211</td>
<td>0.000203</td>
<td>0.001004</td>
<td>0.000689</td>
</tr>
<tr>
<td>Global Delta</td>
<td>0.044239</td>
<td>0.015349</td>
<td>0.015826</td>
<td>0.0192</td>
<td>0.02069</td>
<td>0.148784</td>
</tr>
<tr>
<td></td>
<td>1x2x4(8), 1x8x4(32)</td>
<td>1x2x4(8), 1x8x4(32)</td>
<td>2x2x4(16), 1x4x4(16)</td>
<td>2x2x2(8), 1x5x6(30)</td>
<td>2x2x2(8), 1x4x8(32)</td>
<td>2x2x2(8), 1x8x4(32)</td>
</tr>
<tr>
<td>Computation</td>
<td>0.243652</td>
<td>0.243899</td>
<td>0.244776</td>
<td>0.259719</td>
<td>0.244078</td>
<td>0.244401</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.036595</td>
<td>0.050078</td>
<td>0.021995</td>
<td>0.036356</td>
<td>0.036431</td>
<td>0.065183</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.096298</td>
<td>0.113155</td>
<td>0.127502</td>
<td>0.22887</td>
<td>0.22512</td>
<td>0.236434</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.005242</td>
<td>0.006624</td>
<td>0.00653</td>
<td>0.002835</td>
<td>0.005562</td>
<td>0.004216</td>
</tr>
<tr>
<td>Global Delta</td>
<td>0.160289</td>
<td>0.162881</td>
<td>0.240348</td>
<td>0.144329</td>
<td>0.158788</td>
<td>0.284013</td>
</tr>
</tbody>
</table>
Table 4.11: “Multiple” (critical swaps) timings for decomposition in 8 LPARs

<table>
<thead>
<tr>
<th></th>
<th>1x4x8(32), 1x4x2(8)</th>
<th>1x4x4(16), 1x4x4(16)</th>
<th>2x2x8(32), 1x2x4(8)</th>
<th>1x4x4(16), 1x2x8(16)</th>
<th>2x4x4(32), 1x2x4(8)</th>
<th>1x2x4(8), 1x8x4(32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>0.243389</td>
<td>0.243694</td>
<td>0.246524</td>
<td>0.247039</td>
<td>0.251379</td>
<td>0.243298</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.008088</td>
<td>0.01092</td>
<td>0.006303</td>
<td>0.01001</td>
<td>0.005793</td>
<td>0.013745</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.031587</td>
<td>0.048006</td>
<td>0.058638</td>
<td>0.051316</td>
<td>0.058653</td>
<td>0.073149</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.000201</td>
<td>0.000504</td>
<td>0.000206</td>
<td>0.00065</td>
<td>0.000195</td>
<td>0.001036</td>
</tr>
<tr>
<td>Global Delta</td>
<td>0.012862</td>
<td>0.016792</td>
<td>0.012029</td>
<td>0.01703</td>
<td>0.011615</td>
<td>0.00951</td>
</tr>
<tr>
<td></td>
<td>1x2x4(8), 1x4x8(32)</td>
<td>2x4x4(32), 1x4x2(8)</td>
<td>1x2x4(8), 1x5x6(30)</td>
<td>4x2x4(32), 1x2x4(8)</td>
<td>2x2x4(16), 1x4x4(16)</td>
<td>2x2x8(32), 1x4x2(8)</td>
</tr>
<tr>
<td>Computation</td>
<td>0.245917</td>
<td>0.251143</td>
<td>0.261251</td>
<td>0.251756</td>
<td>0.245541</td>
<td>0.24444</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.013321</td>
<td>0.005511</td>
<td>0.01087</td>
<td>0.010252</td>
<td>0.008754</td>
<td>0.007446</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.068332</td>
<td>0.062953</td>
<td>0.068546</td>
<td>0.082651</td>
<td>0.108627</td>
<td>0.061669</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.001035</td>
<td>0.000195</td>
<td>0.000958</td>
<td>0.000193</td>
<td>0.004076</td>
<td>0.000203</td>
</tr>
<tr>
<td>Global Delta</td>
<td>0.013994</td>
<td>0.02686</td>
<td>0.009954</td>
<td>0.016741</td>
<td>0.01769</td>
<td>0.070097</td>
</tr>
<tr>
<td></td>
<td>2x2x4(16), 1x8x2(16)</td>
<td>4x1x8(32), 1x2x4(8)</td>
<td>4x2x4(32), 1x4x2(8)</td>
<td>2x2x2(8), 1x4x8(32)</td>
<td>2x2x2(8), 1x8x4(32)</td>
<td>2x2x2(8), 1x5x6(30)</td>
</tr>
<tr>
<td>Computation</td>
<td>0.245579</td>
<td>0.247465</td>
<td>0.251297</td>
<td>0.244561</td>
<td>0.24494</td>
<td>0.260753</td>
</tr>
<tr>
<td>Barrier 1</td>
<td>0.008661</td>
<td>0.011436</td>
<td>0.008638</td>
<td>0.009231</td>
<td>0.008621</td>
<td>0.007784</td>
</tr>
<tr>
<td>Swapping</td>
<td>0.116805</td>
<td>0.088166</td>
<td>0.091899</td>
<td>0.188239</td>
<td>0.198955</td>
<td>0.190758</td>
</tr>
<tr>
<td>Local Delta</td>
<td>0.000493</td>
<td>0.000199</td>
<td>0.000194</td>
<td>0.001003</td>
<td>0.00104</td>
<td>0.000924</td>
</tr>
<tr>
<td>Global Delta</td>
<td>0.013831</td>
<td>0.044159</td>
<td>0.041011</td>
<td>0.007149</td>
<td>0.008484</td>
<td>0.007872</td>
</tr>
</tbody>
</table>
Bibliography


