An Investigation into High Performance Fortran vs. Message Passing Interface Performance

Sarah Farrier

August, 2008

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2008
Abstract

Previous studies investigating the performance of High Performance Fortran (HPF) as an HPC language have concluded that it fares less well than its alternatives. However, these studies are relatively out of date and do not necessarily indicate a modern view of the topic when the rapid change of hardware and compiler technology is considered. Two codes, which represent the typical behaviour of many scientific computations, are chosen to perform experiments in this area. They are tested on two separate systems: a Sun Fire X4600 and a Cray XT4, the latter of which was setup to run HPF code for the first time anywhere in the world on its category of system. The HPF codes are compared with one of the most popular HPC languages, Message Passing Interface (MPI). The report finds that, overall, HPF’s performance is indeed still lacking on small numbers of processors, and that any technology change within recent years has not contributed to a performance gain over its rivals. However, there is also an indication that HPF may scale acceptably on higher processor counts, which are more likely to be used in scientific computations.
Acknowledgements

I would like to show my appreciation to all those kind people who generously offered their help during the creation of this dissertation.

Academically, my supervisor Dr. Alan Simpson has been an invaluable source of project support; based alongside him at EPCC whilst working for Cray Inc., Jason Beech-Brandt provided excellent technical aid. I would like to extend my sincere thanks to them both. Code and further technical advice was also given by Dr. Lorna Smith, Jim Enright and Erland Davidson.

Personally, I would also like to give mention to the support of friends and family; in particular, to my husband James Duffield, for his encouragement and understanding throughout.
## Contents

Chapter 1 Introduction and Background ............................................................. 1  
  1.1 Project Basis .............................................................................................. 1  
  1.2 Literature Review ...................................................................................... 1  
    1.2.1 HPF Background ................................................................................ 2  
    1.2.2 MPI Background ................................................................................. 2  
    1.2.3 HPF vs. MPI Performance ................................................................. 3  
    1.2.4 Summary ............................................................................................ 4  
  1.3 Resources ................................................................................................. 4  
  1.4 Data Collection Methodology .................................................................... 5  
  1.5 Statement of Dissertation .......................................................................... 5  

Chapter 2 Overview of Codes ............................................................................. 7  
  2.1 Game of Life Code .................................................................................... 7  
    2.1.1 Design and Implementation .............................................................. 8  
    2.1.2 Results Data ....................................................................................... 9  
    2.1.3 Analysis of Results ........................................................................... 11  
    2.1.4 Conclusions ...................................................................................... 14  
  2.2 Image Processing Code .......................................................................... 14  
    2.2.1 Design and Implementation .............................................................. 16  

Chapter 3 Performance Comparisons .............................................................. 19  
  3.1 Introduction .............................................................................................. 19  
  3.2 Game of Life ............................................................................................ 19  
  3.3 Results Data ............................................................................................ 20
Chapter 1

Introduction and Background

1.1 Project Basis

The purpose of this project is to consider the performance differences between modern Message Passing Interface (MPI) and High Performance Fortran (HPF) implementations. The codes chosen include a typical cellular automaton program and a lattice-based application which involves image processing. Versions of the former code were already available in both languages prior to the start of the project, whilst a basic implementation of the latter is only available in MPI. It is within the project scope to develop an equivalent HPF version of the processing code, and to amend the MPI version as necessary.

It is important to note that previous work in the area of performance comparisons involving HPF and MPI have indicated that the software engineering process merits of HPF go some way towards offsetting any perceived performance advantages that codes written in MPI may achieve (see 1.2.3). However, the significance of this project lies in updating current knowledge. Work in this domain has been performed on relatively outdated machines and compilers, with little or no recent investigation to adequately assess the merit of each technology on similar applications. The interest of this project is to move someway into remedying that situation in the field of modern High Performance Computing (HPC).

An update on the consensus that HPF codes perform less well compared to MPI codes is sought. This expectation is based upon the comparatively low take-up of HPF technology within HPC environments in the time since previous work was published (reflected in its investment and development). See 1.2.1 for a more detailed discussion on this topic.

1.2 Literature Review

The first topics for discussion are brief introductions into the historical background and a technical overview of the technologies being compared. This section will
then give a summary of any performance comparisons which have been published.

1.2.1 HPF Background

HPF is a specification which allows the Fortran programming language to be extended with processor directives so that codes may be readily run in parallel. Its origins lay with a mixture of academics, government and industry-based experts and vendors forming the High Performance Fortran Forum (HPFF). Its popularity ensures its influence can be felt within most supercomputing facilities on many platforms (Chamberlain et al., 2000).

HPF includes many typical features such as data mapping support and specialised data parallel constructs. However, an important aspect of the language definitions is that it operates at a relatively high level of abstraction in comparison to other parallel technologies such as Message Passing Interface (MPI).

The standards were created in order to address the need to obtain the most optimal performance from parallel codes where the distribution of data was of particular significance (Rice University, 1993). HPF was based on a core language which enjoyed a period of time as the unchallenged choice for scientific and engineering applications (Aho, 2004) and Fortran still remains as one of the most preferred programming languages available to the community. Nevertheless, it is recognised that its popularity has declined somewhat in recent years, with C++ and C stepping in as viable alternatives (Bailey et al., 2005). In terms of modern HPC, its influence may be seen in the ongoing developmental efforts of Co-Array Fortran, which uses a related model for attempting to produce a scalable global address space by extending the Fortran language for Single-Program-Multiple-Data (SPMD) parallel processing. Co-Array Fortran similarly seeks to shift the burden of communication optimisation away from the programmer and onto the compiler. Nevertheless, as with HPF, these factors may be Co-Array Fortran’s saviour or its downfall; its success remains to be firmly established despite having been built as a competitor to MPI from the onset (Coarfa et al., 2003) and some indication that its compilers can produce code “roughly equal to that of hand-optimised MPI programs” (Dotsenko et al., 2004). The reasons for its lack of uptake are attributed to the immaturity of the language and the fact that a fledgling compiler is only available for Cray systems (Coarfa et al., 2003).

1.2.2 MPI Background

The Message Passing Interface Forum (MPIF) group developed MPI; the latest update was published in 1997. Since its creation, MPI has become a much more widely accepted standard than HPF in HPC environments. Chamberlain et al. state that it is now considered the de facto standard in the field of HPC. MPI operates at a much lower level than HPF and, in practical terms, is actually just a collection of library routines which may be called from a particular language’s
existing structure, using standard compilers. Pacheco (1996) argues that it is one of the most “powerful” options for parallel computation for this reason.

In MPI, explicit messages are passed between processors, with much more programmer effort focused on ensuring deadlocks are avoided during communication but with an overall greater control of data transmission between processing nodes. Aoyama and Nakano (1999) argue for MPI’s superiority for parallel processing because of this increased flexibility.

1.2.3 HPF vs. MPI Performance

In terms of software engineering practice, whether a particular application code achieves faster development and easier code maintenance using a particular language may be seen as a borderline-subjective matter, despite assertions that HPF requires up to ten times as little programming effort (Ding, 1998; Berthou and Fayolle, 2001). However, if the most important criterion for a code is its execution time, little recent research has revealed the extent to which the general assertion that MPI is faster is still true.

Ding (1998) suggests that for irregular domain decomposition problems, HPF is notably slower than MPI codes of similar nature, quantifying it with a 2- to 4-fold decrease in performance; the work also asserts that HPF codes scale less optimally in general. Even more significantly, Ding reveals that the HPF codes tested were noted to have highly inconsistent performances, with particular reference to seemingly minor code changes resulting in vastly under-performing or over-performing code. Chamberlain et al. (2000) concludes that HPF is amongst the worst of all HPC languages in terms of performance.

![Figure 1 - A Venn diagram to represent some of HPF’s language attributes compared to other HPC languages. CAF is Co-Array Fortran (discussed 1.2.1), F90+MPI represents what this project categorises as MPI, SAC is Single-Assignment C, and ZPL is the acronym of Z-level Programming Language. Reproduced from Chamberlain et al. (2000)](image)

Ding concludes, however, that for certain common scientific computations, such as LU factorisation and calculations involving vectors and matrices, performance was roughly equal to that of the MPI alternatives. Similarly, Adve et al. (1998) also suggest that with better compiler technology, HPF codes can achieve within
15% of the performance of MPI codes. Nevertheless, Li and Zamel (1997) offer conflicting conclusions from testing HPF and MPI versions of their reservoir simulator and parallel linear equation solver code to that of Ding. For both matrix-vector computations and calculating solutions to linear equations, the HPF code performed worse, approximately twice and five times slower respectively.

Ding (1999) later published a more recent account of how HPF vs. MPI performance measured up. This later report stated that, whilst improvements had been made to the HPF compilers available, and that reasonable performance may be noted in some codes tested, particular attention was required to achieve optimal performance in HPF codes. The work also confirmed the previous study’s suggestion that HPF code does not scale well to a large numbers of processors. Berthou and Fayolle’s 2001 investigation comparing MPI, HPF as well as OpenMP codes also supports the above finding (although Berthou and Fayolle’s testing was performed on a number of architectures, both sets of results relate to Cray’s T3E architecture, so it is reasonable to assume both studies would share similar conclusions, being only two years apart).

Finally, brief comment upon non-performance-related aspects of the two technologies must be taken into particular consideration.

Despite the acknowledgement that HPF codes are harder to optimise, Ding’s conclusions also argued in favour of HPF’s superiority regarding programmer effort. It is interesting to note that Adve et al. affirm the same, stating that HPF requires the programmer to undertake “substantial restructuring and hand-optimisation” to achieve decent performance; the authors cite the NAS benchmark complete re-write (after which HPF was declared to scale well) as evidence of this. Essentially: whilst it is much easier to develop working HPF code, it requires more effort than MPI to fine tune that code to reach the highest performance levels.

1.2.4 Summary

As the above summary of previous work shows, very little has been done recently in this area. Ding’s two studies claimed a decreased difference could be seen comparing HPF and MPI on the test architecture in just a few short years, reinforcing the idea that compilers and technologies change rapidly. As such, it is felt that now is an ideal time to check the statements previously made during similar comparisons and ask how valid they are today.

1.3 Resources

In terms of hardware, the main project machine to be used will be HECToR (High End Computing Terascale Resources), a supercomputer which is based at the EPCC (Edinburgh Parallel Computing Centre) at the University of Edinburgh. The architecture of this machine is a Cray XT4 system which holds over five thousand 2.8Ghz dual-core AMD Opteron processors, with an overall peak
performance of 63 Tflops; each duel-core processor has 6GB of DIMM memory available, and each core has its own internal cache. It also boasts a high-speed interconnect as well as extremely good latency. It is a general-purpose machine built for multiple disciplinary computation, and is used in fields from epidemiology and meteorology to chemistry and biochemistry.

However, also available is the same facility’s smaller NESS machine, a Sun Fire X4600 server unit consisting of 34 2.6Ghz Opteron cores each with 2G of memory in a shared memory architecture. This GNU/Linux 2.6.18-running system enables jobs to communicate quickly using the shared memory, but does not allow jobs that span two nodes over the slow network connect, so the maximum size of any parallel job is therefore 16 processes (the number of cores in each NESS node).

The software resources involved in the project are mainly the two compilers. Fortunately due to the help received from Jason Beech-Brandt at Cray, support for HPF is included on the XT4; at the time of writing it is believed that this project is the first of its kind in the world to execute HPF code on HECToR’s category of system. For the HPF codes, pghpf (v7.0-7 for a 64-bit target on x86-64 Linux) is available on NESS, and pgf90 (v7.1-4 for a 64-bit target on x86-64 Linux) is available on HECToR. These widely-distributed and well-supported compilers are developed by the Portland Group. For the MPI code, the popular open-source MPICH2 (v1.0.5) implementation of the MPI library will be used via mpif90.

1.4 Data Collection Methodology

The dissertation includes many results tables. A short note on the methodology used to collect data is appropriate. It is most relevant to mention that all execution runs were performed three times; of the timings given in this document, the fastest of the three was chosen for inclusion here. The justification for this is that using the mean average – as at first might seem acceptable – is not a statistically rigorous number, being prone to distortion by extremes. On the other hand, using the median is less attractive than simply using the fastest time noted since there are many more situational things that can influence a code to run slower, and not many that can influence it to run faster. Nevertheless an attempt has been made to ensure that the results are reliable and reproducible; for example, on NESS a job submitted may request to reserve the maximum amount of processors yet only execute on less than that number. More details are given in A1.1.

1.5 Statement of Dissertation

The dissertation has been structured to give an overview of the codes’ design and implementation in Chapter 2. Initial performance comparisons have been given in Chapter 3; the format gives the first code’s data tables of results followed by an analysis, then the second code’s information in the same manner. Chapter 4 outlines further experiments performed – namely, the codes’ instrumentation, and
an investigation into intrinsic routine efficiency. Technical conclusions are given in Chapter 5 alongside suggestions for future work, followed by a brief project review which identifies the project’s level of success.
Chapter 2

Overview of Codes

2.1 Game of Life Code

The first application to be evaluated for performance is John Horton Conway’s famous cellular automaton, the Game of Life. Devised in 1970, it may be described as a zero-player simulation game, in that the initial configuration of the theoretically-infinite grid of square cells evolves without further external influence. It is a code which is popular in the field of emergent computing, which studies complex behaviour arising from seemingly simple interactions.

Both an HPF and an MPI version of the code already existed, the former having been developed by the project author during the MSc teaching semester, and the latter obtained from Dr. Lorna Smith of EPCC.

The Game of Life is a good candidate code to investigate the performance differences of HPF and MPI. The reason for this is because it is a sample code that is similar to the typical two-dimensional grids that are commonly used in scientific computation. It is important to note that one of the main features of the Game of Life is that it is computationally simple for each processor to deal with locally – communication is a much more significant part of its overall runtime.

The problem size chosen to work on was a grid of $1024^2$ elements using 500 generations of evolution. At the start of the project, various configurations were run to indentify an acceptable problem size, and there are several reasons why this particular one was eventually chosen. Firstly, during these initial test runs this grid and generation count gave an acceptable amount of work to measure for computation. However, it was also small enough to ensure a small execution time, which served two purposes – to ensure quick turnaround of batch-submitted runs, and to not waste unnecessary computation time. As a power of two, the grid size also divides neatly amongst the processor counts to be used, which an important consideration for load balancing purposes.

In the Game of Life, the game grid’s individual cells are given initial binary states, after which, evolution occurs up to a pre-determined number of generations. In
simpler simulations, the starting positions of “live” cells vs. “dead” cells may be pre-set to a given pattern; in others, a random distribution of the cell states may be used.

At each timestep of the computation, neighbouring cells determine the subsequent state of a given cell, based on a few simple rules:

- any cell with exactly two or three alive neighbours has no change to state;
- any cell which is bounded by less than 2 or more than three live neighbours dies due to loneliness or overcrowding respectively;
- a dead cell with 3 neighbours is given the alive status.

An example of Game of Life output per timestep may be seen in Figure 2 below.

![Figure 2 - An example of the Game of Life. Partially reproduced from Neider et al. (2004)](image)

2.1.1 Design and Implementation

Since both languages’ codes are pre-developed programs, any discussion here must focus on the most relevant aspects to the investigation rather than a general background on their implementation. Specifically, the remainder of this subsection will focus on an experiment into HPF array elements’ neighbour-related operations.

An important area of the Game of Life in HPF is the kernel used in the code. Each element’s interaction with its neighbours is one of the most fundamental operations of the program, and there are several ways it may be implemented. Three methods were investigated and are given below. The motivation behind these tests were to ensure a good version of one of the code’s most central operations was chosen to assess the overall quality of HPF performance.
boardneighbours = 
&
( CSHIFT(board,SHIFT=1,DIM=1) + &
 CSHIFT(board,SHIFT=-1,DIM=1) + &
 CSHIFT(board,SHIFT=1,DIM=2) + &
 CSHIFT(board,SHIFT=-1,DIM=2) + &
 CSHIFT(CSHIFT(board, SHIFT= 1, DIM=2), SHIFT= 1, DIM=1) + &
 CSHIFT(CSHIFT(board, SHIFT= 1, DIM=2), SHIFT=-1, DIM=1) + &
 CSHIFT(CSHIFT(board, SHIFT=-1, DIM=2), SHIFT= 1, DIM=1) + &
 CSHIFT(CSHIFT(board, SHIFT=-1, DIM=2), SHIFT=-1, DIM=1) )

Code Example 1 - 12 step CSHIFT neighbour operation

tmp_compact = board+cshift(board,1,dim=2)+cshift(board,-1,dim=2)
board = tmp_compact+cshift(tmp,1,dim=1)+cshift(tmp,-1,dim=1)-board

Code Example 2 - Compacted CSHIFT neighbour operation

board(i,j) = f77_neighbours(i+1,j-1)+ f77_neighbours(i+1,j) +
f77_neighbours(i+1,j+1) + f77_neighbours(i,j+1) + f77_neighbours(i-1,j+1) + f77_neighbours(i-1,j)+ f77_neighbours(i-1,j-1)

Code Example 3 - Fortran 77-style neighbour operation

Code Example 1 displays a CSHIFT operation which uses a naïve approach to reference the current cell’s eight surrounding elements. It uses four straight shifts and four nested ones to obtain data from its vertical-horizontal and diagonal neighbours respectively.

Code Example 2 is much smaller, and uses fewer shifts; it has been structured so that the computation of row data from the two neighbours is performed in a single stage, followed by the vertical neighbour stage. It is the combination of operations when dealing with identically-dimensioned neighbour elements which makes Code Example 2 optimised into a more efficient CSHIFT than Code Example 1.

Code Example 3 is a traditional element-referencing statement which avoids the use of the CSHIFT operation using a nested DO loop for both axis. It relies on f77_neighbours to be two elements larger than board i.e. identical to board but with a set boundary of zeros at each edge.

2.1.2 Results Data

The data from testing these operations upon the hardware used in the project follow.
### Table 1 - 12-step CSHIFT performance data on NESS

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Operation</th>
<th>Time (s)</th>
<th>Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>extended 12-step cshift</td>
<td>9.063</td>
<td>1.000</td>
</tr>
<tr>
<td>2 Processors</td>
<td>extended 12-step cshift</td>
<td>3.866</td>
<td>2.344</td>
</tr>
<tr>
<td>4 Processors</td>
<td>extended 12-step cshift</td>
<td>2.377</td>
<td>3.813</td>
</tr>
<tr>
<td>8 Processors</td>
<td>extended 12-step cshift</td>
<td>1.412</td>
<td>6.419</td>
</tr>
<tr>
<td>16 Processors</td>
<td>extended 12-step cshift</td>
<td>0.871</td>
<td>10.405</td>
</tr>
</tbody>
</table>

### Table 2 - Compact CSHIFT performance data on NESS

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Operation</th>
<th>Time (s)</th>
<th>Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>compact cshift</td>
<td>3.211</td>
<td>1.000</td>
</tr>
<tr>
<td>2 Processors</td>
<td>compact cshift</td>
<td>1.714</td>
<td>1.873</td>
</tr>
<tr>
<td>4 Processors</td>
<td>compact cshift</td>
<td>0.991</td>
<td>3.240</td>
</tr>
<tr>
<td>8 Processors</td>
<td>compact cshift</td>
<td>0.540</td>
<td>5.946</td>
</tr>
<tr>
<td>16 Processors</td>
<td>compact cshift</td>
<td>0.271</td>
<td>11.849</td>
</tr>
</tbody>
</table>

### Table 3 - Fortran-77 operation performance data on NESS

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Operation</th>
<th>Time (s)</th>
<th>Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>f77 statement</td>
<td>8.546</td>
<td>1.000</td>
</tr>
<tr>
<td>2 Processors</td>
<td>f77 statement</td>
<td>3.676</td>
<td>2.235</td>
</tr>
<tr>
<td>4 Processors</td>
<td>f77 statement</td>
<td>2.330</td>
<td>3.668</td>
</tr>
<tr>
<td>8 Processors</td>
<td>f77 statement</td>
<td>1.480</td>
<td>5.774</td>
</tr>
<tr>
<td>16 Processors</td>
<td>f77 statement</td>
<td>0.929</td>
<td>9.199</td>
</tr>
</tbody>
</table>

### Table 4 - 12-step CSHIFT performance data on HECToR

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Operation</th>
<th>Time (s)</th>
<th>Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>extended 12-step cshift</td>
<td>1.408</td>
<td>1.000</td>
</tr>
<tr>
<td>2 Processors</td>
<td>extended 12-step cshift</td>
<td>1.221</td>
<td>1.153</td>
</tr>
<tr>
<td>4 Processors</td>
<td>extended 12-step cshift</td>
<td>0.708</td>
<td>1.988</td>
</tr>
<tr>
<td>8 Processors</td>
<td>extended 12-step cshift</td>
<td>0.330</td>
<td>4.266</td>
</tr>
<tr>
<td>16 Processors</td>
<td>extended 12-step cshift</td>
<td>0.148</td>
<td>9.529</td>
</tr>
<tr>
<td>No. of Processors</td>
<td>Operation</td>
<td>Time (s)</td>
<td>Scaling</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>1 Processor</td>
<td>compact cshift</td>
<td>0.785</td>
<td>1.000</td>
</tr>
<tr>
<td>2 Processors</td>
<td>compact cshift</td>
<td>0.612</td>
<td>1.282</td>
</tr>
<tr>
<td>4 Processors</td>
<td>compact cshift</td>
<td>0.317</td>
<td>2.473</td>
</tr>
<tr>
<td>8 Processors</td>
<td>compact cshift</td>
<td>0.177</td>
<td>4.423</td>
</tr>
<tr>
<td>16 Processors</td>
<td>compact cshift</td>
<td>0.029</td>
<td>26.861</td>
</tr>
</tbody>
</table>

Table 5 - Compact CSHIFT performance data on HECToR

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Operation</th>
<th>Time (s)</th>
<th>Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>f77 statement</td>
<td>1.194</td>
<td>1.000</td>
</tr>
<tr>
<td>2 Processors</td>
<td>f77 statement</td>
<td>1.195</td>
<td>0.999</td>
</tr>
<tr>
<td>4 Processors</td>
<td>f77 statement</td>
<td>0.609</td>
<td>1.960</td>
</tr>
<tr>
<td>8 Processors</td>
<td>f77 statement</td>
<td>0.301</td>
<td>3.964</td>
</tr>
<tr>
<td>16 Processors</td>
<td>f77 statement</td>
<td>0.134</td>
<td>8.921</td>
</tr>
</tbody>
</table>

Table 6 - Fortran-77 operation performance data on HECToR

2.1.3 Analysis of Results

Interestingly, the Fortran-77 style operation has a similar time to the longer CSHIFT operation but scales slightly less favourably. Overall, however, the compacted CSHIFT operation on NESS is the best performer, up to a factor of two better than the others (this would make sense due to the superior data reuse of the operation and less communication).
Figure 3 - Graphs to show neighbour-related operation performance and scaling on NESS
Figure 4 - Graphs to show neighbour-related operation performance and scaling on HECToR

From the data above, it is also evident that there is a performance difference dependant upon the operation chosen for measurement on both the NESS and HECToR machines, yet the two systems still exhibit the same trends.

On NESS, the compact CSHIFT operation is significantly faster than the other two, which display similar runtimes. However, the scaling of the 12-step operation is the best at four and eight processors; the compact operation scales best at sixteen processors.

On HECToR, there are very similar findings to those reported for NESS. However, there is much less of a difference in the runtimes of the three operations on lower numbers of processors. For example, on NESS, the slowest operation (the 12-step CSHIFT) takes six seconds longer than the fastest (the compact CSHIFT) on a single processor; on HECToR this figure is less than half
a second. An interesting area of HECToR’s results is the jump from eight to sixteen processors, which speeds up drastically, however, this might be explained by the optimal distribution of data on sixteen processors as a square mesh or cache.

The overall difference in the two machines’ execution times would most likely be attributed to the particularly low latency levels on HECToR, which would result in the favourable timings shown compared to those of NESS.

2.1.4 Conclusions

As a result of the above discussion, the compacted CSHIFT was chosen for final implementation in the code since its runtime and scaling showed the most favourable figures from the candidate code fragments compared. It illustrates the complexity of HPF code’s performance depending on seemingly trivial implementation decisions.

2.2 Image Processing Code

The second code involved in the project is a lattice-based message-passing image processing application.

As with the Game of Life, the methods used in the image processing code share features with standard HPC applications. An example of this may be seen in the solutions to partial differential equations given by Jacobi method iterations which execute until convergence. However, unlike the Game of Life code, complex computation is a particular feature of the image processing code. As such, it is particularly interesting for parallel scaling and timing investigation due to the relatively large computational effort required to produce an image from the given edges, as well as involving neighbourly communication.

An MPI version of the code already exists from the EPCC teaching semester courtesy of two fellow students. Section 2.2.1 Design and Implementation describes aspects of the HPF version implementation which was undertaken during the project period.

The code’s input is the result of an edge-detection algorithm which has been applied to a non-colour MxN-sized image. The problem size chosen was the largest example image to hand, a 600x840 image of the Edinburgh skyline, to ensure an acceptably long execution time for measurement without being wasteful of computational resources. The pixels’ values are determined using the formula given in Equation 1. It illustrates that if a specific pixel in the input image has no edge (i.e. it has the same value as its four neighbours) then the same pixel location on the output image will be set to zero. Conversely, an edge is found by identifying that the pixel differs greatly from its neighbours and the formula’s output will be large in magnitude. Pixels considered to be out-of-bounds are considered to be set to zero. See below for an example of this.
\[ edge_{i,j} = image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - 4 \cdot image_{i,j} \]

Equation 1 - Edge-detection formula, reproduced from Henty (2007)

Figure 5 - Edge detection example, reproduced from Henty (2007)

The image processing application at hand involves an attempt to reverse this process in an iterative manner using the equation given in Equation 2 using vertical and horizontal neighbours. See Figure 6.

\[ new_{i,j} = \frac{1}{4} (old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j}) \]

Equation 2 - Edge-detection formula reversal process, reproduced from Henty (2007)

Figure 6 - Edge-detection formula reversal process

During each iteration, the value of the pixels in the reconstructed image is considered, to serve as stopping criteria (as opposed to performing a fixed number of iterations with no regard for desired accuracy). More specifically, the delta is calculated. This delta (\( \Delta \)) is the difference between the last iteration’s image and the current one’s image. See Equation 3. As such, calculation effort is terminated when the output image is sufficiently accurate to the original i.e. the difference between compared images, and thus the delta, is small. The accuracy used in the code for this project is a precision of \( \Delta 0.1 \).
\[
\Delta^2 = \frac{1}{MN} \sum_{i=1; j=1}^{i=M; j=N} (new_{i,j} - old_{i,j})^2
\]

Equation 3 - Stopping criteria for the computation, reproduced from Henty (2007)

The MPI code uses a 2-D domain decomposition and non-blocking communication for halo-swapping of data between nearest-neighbour boundary points, and uses collective communications for the sharing of data amongst processors. In the MPI version, which required slight adjustments to work correctly (see 5.3), this involved the use of operations such as Broadcast and Reduce. See Figure 7.

![Figure 7 - 2D domain decomposition in MPI](image)

2.2.1 Design and Implementation

The following section outlines the approach which was used to construct an HPF version of the image processing code for the project.

Firstly, the domain decomposition of the HPF code must be constructed in a similar manner to the one used in MPI. However, HPF has some key differences in the way this feature is implemented – the main one being that it is far more straightforward for the programmer to specify the decomposition in HPF, to the point where any future decomposition change is a trivial task. In MPI, there is a need to read the code’s input data file into a temporary buffer array and distribute appropriate sections amongst the available processors for computation i.e. the portion of the data array a process gets must be manually decided using its Cartesian coordinates and several Broadcast and Reduction data operations are needed between processors. To contrast with HPF, no explicit communication of the assignments takes place by the programmer. Instead, the DISTRIBUTE directive partitions the data space to the given parameters.

!HPFS$ DISTRIBUTE imageInputArray(BLOCK,BLOCK)

Code Example 4 – DISTRIBUTE directive
Another important feature of the HPF version of the image processing code is its use of the EOSHIFT function for array manipulation to fill in the boundary values at the image edge. This transformational function is similar to the more common CSHIFT function in that it takes a given input array and shifts it along a given dimension in a circular manner to a specified amount.

\[ \text{EOSHIFT}(\text{imageInputArray}, \text{SHIFT}, \text{BOUNDARY}, \text{DIM}) \]

**Code Example 5 - EOSHIFT operation**

The first dimension shifts column-wise and the second dimension row-wise; at the same time, a positive shift value bumps the array elements to the right and a positive one moves them to the left. An important feature of EOSHIFT is that any value moved off the edge of the given array has its value filled by the boundary value supplied in the argument list; if one isn’t supplied, then the default value of zero is used (or false if the array is a logical array, or null if a character array). An example of this can be seen in below, where the array A is transformed using an EOSHIFT operation to produce array B.

\[ B = \text{EOSHIFT}(A, -1, \text{DIM}=2) \]

**Code Example 6 - EOSHIFT example**

Another feature which the HPF code utilises is extrinsic procedures; these allow a developer to specify a region of a program as being something other than standard HPF code. Importantly, they offer the ability to perform operations on a local subset of data (using F90_LOCAL) or on only a subset of the processors involved in a wider computation (using F90_SERIAL).
Extrinsic routines basically move the program into a local execution model wherein each processor works on its allocation of data only, but each processor remains active i.e. a SIMD situation wherein great care must be taken to allocate local portions of arrays to each processor. They also allow message passing procedures to be called from within HPF. Essentially, the code segment is compiled using HPF but will only ever execute on a single processor within the computation i.e. the compiler treats the routine as if it were completely serial. In essence, extrinsic procedures allow HPF to be extremely flexible.

Note that there are several restrictions which extrinsic subprograms must adhere to which relate to the logic of how they are used; an example of this is that no reference to any subroutine which realigns or redistributes data must be apparent since these directives are meaningless in a serial environment. Another obvious example is that any reference to a subroutine that has an extrinsic kind of HPF is also illegal.

Overall, this category of extrinsics is very relevant to areas of code which are inherently serial and which do not influence the overall runtime of the application as a whole (since there is an associated cost for any call to an extrinsic routine). Any I/O functionality such as the writing of the final calculated image in the project code does not need to be performed on multiple processors, and extrinsic procedures offer an acceptable solution.

As such, the HPF code developed makes use of F90_SERIAL as one of the more straightforward and natively supported options to deal with file transactions. An indication of the way in which this is done is given in Code Example 7 below; it shows that there is the ability to call Fortran 90 Serial routines within the HPF code by specifying as such in the INTERFACE block.

```fortran
INTERFACE
EXTRINSIC (F90_SERIAL) SUBROUTINE doSomethingToImageInSerial()
   INTEGER*4, INTENT(IN) :: imageInputArray
   // do something locally in serial using the interface between
   // HPF and a serial/local execution model section
END SUBROUTINE doSomethingToImageInSerial
END INTERFACE

Code Example 7 - Extrinsic procedure in HPF
Chapter 3

Performance Comparisons

3.1 Introduction

This section of work describes the performance comparisons of the Game of Life and Image processing codes on both machines in HPF and MPI. It is a black box performance investigation, in the sense that an in-depth knowledge of the code is not considered; instead, the codes are viewed by their external, measurable attributes.

Note that speedup (S) has been calculated by the serial execution time on a single processor over the parallel execution time, see Equation 4.

\[ S(N, P) = \frac{T(N, 1)}{T(N, P)} \]

Equation 4 – Speedup.

3.2 Game of Life

Verification of results for the Game of Life was performed by ensuring that both language codes gave the same number of live cells initially, and for increments throughout the computation as well as at the final generation. This number was also confirmed as consistent across multiple numbers of processors on both machines.

The methodology used to collect data has been given in 1.4.
3.3 Results Data

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>40.840</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>22.904</td>
<td>1.783</td>
</tr>
<tr>
<td>4 Processors</td>
<td>11.086</td>
<td>3.684</td>
</tr>
<tr>
<td>8 Processors</td>
<td>9.600</td>
<td>4.254</td>
</tr>
<tr>
<td>16 Processors</td>
<td>3.399</td>
<td>12.015</td>
</tr>
</tbody>
</table>

Table 7 - HPF Game of Life on NESS (1024*1024 gridsize, 500 generations)

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>5.540</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>4.461</td>
<td>1.242</td>
</tr>
<tr>
<td>4 Processors</td>
<td>3.001</td>
<td>1.846</td>
</tr>
<tr>
<td>8 Processors</td>
<td>1.394</td>
<td>3.975</td>
</tr>
<tr>
<td>16 Processors</td>
<td>1.750</td>
<td>3.167</td>
</tr>
</tbody>
</table>

Table 8 - MPI Game of Life on NESS (1024*1024 gridsize, 500 generations)

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>10.123</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>8.497</td>
<td>1.191</td>
</tr>
<tr>
<td>4 Processors</td>
<td>4.708</td>
<td>2.150</td>
</tr>
<tr>
<td>8 Processors</td>
<td>2.426</td>
<td>4.173</td>
</tr>
<tr>
<td>16 Processors</td>
<td>1.191</td>
<td>8.498</td>
</tr>
</tbody>
</table>

Table 9 - HPF Game of Life on HECToR (1024*1024 gridsize, 500 generations)

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>4.619</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>3.501</td>
<td>1.319</td>
</tr>
<tr>
<td>4 Processors</td>
<td>1.893</td>
<td>2.440</td>
</tr>
<tr>
<td>8 Processors</td>
<td>0.929</td>
<td>4.972</td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.570</td>
<td>8.104</td>
</tr>
</tbody>
</table>

Table 10 - MPI Game of Life on HECToR (1024*1024 gridsize, 500 generations)
3.4 Analysis of Results

3.4.1 Language Performance Comparison

Viewing the Game of Life results, it can be seen that on a single processor on NESS, the MPI version executes in just over five seconds, whilst the HPF code takes five times as long when given an identical problem size and generation count. This large initial difference between the two languages’ speed is reduced as processor number increases; however, the HPF code remains slower than MPI throughout testing, as shown in Figure 10.

![Figure 10 - Graph to show Game of Life in HPF vs. MPI on NESS](image)

On the HECToR machine, there is much less of a difference between the performance of the MPI and HPF Game of Life. This is shown below in Figure 11.
However, it remains true that HPF is slower at any number of processors; basically, HECToR follows the same pattern of performance that NESS indicated during tests. For example, whilst HECToR's HPF executes the code for approximately ten seconds, MPI performs the same problem size and generation count solution in under five. Another important finding is that HECToR's results show a much more consist execution time ratio between the two languages, as Figure 12 below illustrates. The exact reasons behind this trend are not obvious, and serve as one of the motivating factors behind section 4.1.
MPI’s better overall performance may be explained somewhat by the explicit processor-to-processor communication which the developer must specify, the details of which are rather more hidden in the HPF alternative. This topic has been discussed in past chapters relating to previously published work; see 1.2.3 HPF vs. MPI Performance. Since HPF’s communication is purposely abstracted to a higher level so the programmer does not get involved in explicit communication operations, there may be some efficiency here which contributes to HPF’s seeming lack of speed.

Despite this overall conclusion, it is important to note that when the code is executed on sixteen processors, the runtimes of the languages are much more similar on NESS, despite MPI following its previous trend of being faster. This is true of both machines used for testing. Interestingly, in terms of speedup on NESS, HPF has a figure of over twelve when using the highest number of processors, whilst the speedup offered by the MPI code remains relatively stable (between one and four) on any number of processors.

The impressive speedup of the HPF code on the higher-latency machine should be highlighted in conjunction with the poor runtime of the code on fewer processors. Essentially, these results might indicate that on certain types of architecture HPF performance is extremely poor using small numbers of processors, compared to MPI; but that given larger numbers of processors, HPF’s superior speedup may result in better performance. Since computational science mainly relies on its codes’ execution on large numbers of processors, the earlier assertion that HPF is slower than MPI using one or two processors might not be particularly relevant to the HPC discipline – but that the finding of better scaling of HPF on the NESS machine is.

Unfortunately, sixteen processors is the limit of the hardware available for language comparison on the two machines (in reality this is limited by the processor availability on NESS); nevertheless, it remains an interesting question as to whether there might be similar runtimes between MPI and HPF given more processors than are currently presented, or if HPF may perform better under some circumstances. This is discussed in section 5.2.

### 3.4.2 Machine Performance Comparison

To summarise in terms of performance for the same language code on the two test machines, it should be pointed out that MPI's timings on both machines does not appear to be very different.

However, whilst the speedup of the HPF Game of Life code has previously been discussed in this chapter (i.e. having a noteworthy speedup despite poor performance on low numbers of processors), it should be documented that HECToR's HPF speedup values are much less impressive. The HPF version of the Game of Life fares significantly better on HECToR than on NESS, with the greatest difference being using a single processor wherein the two machines have a 30 second difference. There is a much narrower range of speedup
amongst different numbers of processors, albeit still scaling to eight on sixteen processors. On HECToR, both language versions’ timings suggest similar scaling of code, but especially at higher numbers of processors. For example, at sixteen processors, the speedup of both is approximately eight. As such, on HECToR the difference between speedup in the two code versions is fairly minor.

The reason behind HECToR’s superior performance may be answered through several reasons. NESS has a shared memory node architecture which uses a shared bus; this might suggest that, at some point, the contention for resources amongst the available processors could make the code memory-bound. The overheads involved in the code are threefold – basic overhead of the software (which includes factors like the extra initialisation work involved in parallelising the code compared to serial i.e. extra loop indexing effort), load balancing where each node attempts to perform at a similar level to its peers, and communication overhead. It is suggested that on NESS, the runtime becomes dominated by access to data rather than computation due to the shared memory architecture, which could explain the generally poorer performance. At the same time, HECToR boasts a non-shared memory architecture as well as a high-speed interconnect, which again may go someway towards explaining the given results (please also refer 1.3 for more detail on this subject). This is investigated further in section 4.1.

### 3.4.3 Problem Size Experiments

In order to test some of the ideas discussed above, see the following data in Table 11 which should be compared to Table 7 and Table 8 in the following discussion.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>MPI</th>
<th>HPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>128.05</td>
<td>Speedup</td>
</tr>
<tr>
<td>2 Processors</td>
<td>77.89</td>
<td>1.64</td>
</tr>
<tr>
<td>4 Processors</td>
<td>58.10</td>
<td>2.20</td>
</tr>
<tr>
<td>8 Processors</td>
<td>35.31</td>
<td>3.63</td>
</tr>
<tr>
<td>16 Processors</td>
<td>31.90</td>
<td>4.01</td>
</tr>
</tbody>
</table>

Table 11 - Execution times for Game of Life HPF/MPI code versions on an 5000x5000 grid for 500 generations on NESS

The above table shows NESS’ performance on a problem size which is approximately twenty-five times that of the one used throughout testing, but the decomposition dictates that boundary length is only around five times bigger. Comparing to previous results, the runtimes show similar behaviour. The MPI version of the Game of Life is roughly five times faster than HPF (on the smaller grid the MPI code was roughly eight times faster). However, the 5000^2 grid in
HPF fails to close the performance gap with MPI, and remains approximately three times as slow throughout; this is contrary to the trend on the smaller problem size wherein HPF eventually displays similar runtime to MPI. See Figure 13 for a graphical representation.

![Larger Problem Size Grid for Game of Life on NESS](image)

Figure 13 - Execution times for Game of Life HPF/MPI code versions on an 5000x5000 grid for 500 generations on NESS

The given figures also show that when the problem size has been increased, the speedup of MPI improves but that of HPF worsens compared to smaller problem sizes. If the runtime of the code were limited by communication costs on NESS, it would be expected that large sizes show better scaling at higher processors (since processing now involves more local computation time and less competition for memory resources). Interestingly, the fact only appears to be true for MPI code, and HPF shows no superior scaling with larger problem sizes compared to smaller ones, which indicates that HECToR’s faster performance might not be explained only through NESS’ shared memory architecture (at least for HPF). It does remain true that HPF scales much better than MPI, however.

### 3.5 Image Processing

Verification of results for the Image Processing code was performed by ensuring that both language codes gave the same delta at initialisation, as well as for increments throughout the computation. A sum of the image array as well as a visual check on the output files produced was also performed after code execution. These numbers were also confirmed as consistent across multiple numbers of processors on both machines.

The methodology used to collect data has been given in 1.4.
3.5.1 Results Data

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>29.339</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>17.618</td>
<td>1.665</td>
</tr>
<tr>
<td>4 Processors</td>
<td>8.618</td>
<td>3.404</td>
</tr>
<tr>
<td>8 Processors</td>
<td>4.583</td>
<td>6.402</td>
</tr>
<tr>
<td>16 Processors</td>
<td>3.528</td>
<td>8.316</td>
</tr>
</tbody>
</table>

Table 12 - HPF Image Processing on NESS (600*840 input, 0.1 delta)

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>11.980</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>4.959</td>
<td>2.416</td>
</tr>
<tr>
<td>4 Processors</td>
<td>3.827</td>
<td>3.130</td>
</tr>
<tr>
<td>8 Processors</td>
<td>2.268</td>
<td>5.282</td>
</tr>
<tr>
<td>16 Processors</td>
<td>1.079</td>
<td>11.101</td>
</tr>
</tbody>
</table>

Table 13 - MPI Image Processing on NESS (600*840 input, 0.1 delta)

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>11.543</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>8.959</td>
<td>1.288</td>
</tr>
<tr>
<td>4 Processors</td>
<td>4.794</td>
<td>2.408</td>
</tr>
<tr>
<td>8 Processors</td>
<td>2.262</td>
<td>5.103</td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.423</td>
<td>27.288</td>
</tr>
</tbody>
</table>

Table 14 - HPF Image Processing on HECToR (600*840 input, 0.1 delta)

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Execution Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>8.489</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>3.372</td>
<td>2.517</td>
</tr>
<tr>
<td>4 Processors</td>
<td>1.505</td>
<td>5.640</td>
</tr>
<tr>
<td>8 Processors</td>
<td>0.540</td>
<td>15.728</td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.325</td>
<td>26.096</td>
</tr>
</tbody>
</table>

Table 15 - MPI Image Processing on HECToR (600*840 input, 0.1 delta)
3.5.2 Language Performance Comparisons

Firstly, just as with the Game of Life codes, HECToR follows roughly the same trends as seen on NESS in that the performance is slightly better on every processor count in MPI. This difference ranges from being by a large amount with fewer numbers of processors, to being a smaller difference at higher numbers of processors. See Figure 14 and Figure 15 for illustration of this behaviour.

Figure 14 – Graph to show Image Processing in HPF vs. MPI on NESS

HPF code is three times slower than MPI on a single processor on NESS (approximately 30 seconds and 10 seconds respectively); once more, this difference reduces slightly as processor count increases. At sixteen processors on NESS, HPF is once more slower MPI. HECToR’s data shows that it also follows the same pattern, albeit its HPF and MPI results on the highest number of processors are even closer than on NESS.

Overall, these results display the trend noted in the Game of Life in that the gap between HPF and MPI performance reduces with higher numbers of processors, further underpinning the ideas discussed in 3.4.1. An interesting avenue of experimentation would be to try and identify the regions of inefficiency in HPF in order to determine why on lower processor counts it performs so poorly, and if the prediction of acceptable HPF performance due to superior scaling is one of merit. Such efforts are described in Chapter 4.
3.5.3 Machine Performance Comparisons

As is to be expected due to knowledge of the system and the Game of Life results, HECToR’s runtimes outperform the NESS timings by a significant amount. For example, the HPF on HECToR executes in twelve seconds whilst the NESS machine takes three times as long on a single processor; the difference on two processors is that HECToR is twice as fast when running identical HPF code. On HECToR the overall language trend continues, albeit HPF and MPI have differences to a much lesser degree than on NESS. This type of behaviour might be the result of memory bandwidth congestion on NESS because it has a shared memory architecture, and may explain the results from the project code further in light of the discussion in 3.4.2 Machine Performance Comparison.

Both machines appears to suggest that HPF’s performance is impressive, although this speedup should be only be noted alongside its relatively poor performance with the lowest numbers of processors and admirable speedup, as with the Game Of Life codes.

3.5.4 Notes on Cache Effects

On several of the results tables, there is an interesting speedup between the execution times of different numbers of processors. Both Table 14 and Table 15 show results where this apparent hyperscaling may be seen. Cache effects may explain the seemingly very good runtime of the codes between eight and sixteen processors on HECToR using the Image Processing code. During the image transformation loops, the main array dealt with in memory is a 32-bit REAL datatype and is declared for the size of the input image (in this test case, 600x840
elements); its total memory footprint is 2016000 bytes used for that array. However, when the problem domain is split across multiple processors, the local problem is merely a portion of the total computational effort; the smaller the problem size, the more likely it is that the data will fit into the primary, quicker cache levels. This leads into a higher cache hit rate rather than repeated access to lower levels or main memory. For example, on both machines, the Level 1 cache is 64KB of memory, Level 2 is 1MB, etc. Main memory is an even larger and slower resource for data access.

As such, when the image processing code’s data set fits into the higher levels of cache, there is a marked performance improvement compared to the cases where it does not, which might explain the especially fast speedup at lower numbers of processors (despite increased communications time, which is worst of all on NESS with its shared memory bus, this cost is less than the cache miss delays which are apparent using a single processor). Moreover, these differences are become even more marked since the code makes repeated iterations over the same data region until the new image’s precision is to a desired accuracy.

3.6 Conclusions

The most prominent feature in the comparison of the programs on both machines is the marked difference between the runtimes. As a rule, MPI outperforms HPF without exception, and HECToR outperforms NESS – however, the degree to which these tendencies are true varies considerably, based on the machine and the number of processors involved in the computation.

The above results may be summarised in the statements that:

- MPI, without exception, outperforms HPF when given the same problem size, decomposition and generation count in the Game of Life in terms of execution time – and this is also true for the Image Processing code;

- the closer to sixteen processors the data point is, the narrower the HPF-to-MPI execution time gap is on NESS, with this not being so much more noticeable on HECToR;

- the execution time of codes run on HECToR is much better than on NESS, and indeed, HPF’s performance is much closer to MPI on HECToR;

- MPI and HPF appear to have similar scaling on HECToR yet on NESS HPF’s scaling is vastly superior to MPI regardless of problem size or code tested.
Chapter 4

Further Work Undertaken

This chapter will detail additional work which was undertaken to examine the code performance at a deeper level. This is a reasonable avenue of investigation based on the performance results noted in Chapter 3 which require further consideration. Firstly, instrumentation efforts are described for both codes; later, an experiment into the efficiency of HPF intrinsic is described. More detailed motivation discussion behind each of these activities is given in the introduction sections as appropriate.

4.1 Instrumentation

In order that more could be understood about the various timings achieved for the two codes, and in particular to find out why HPF appears to be so much more inefficient at lower numbers of processors than MPI, instrumentation has been performed. This provide insight on the ideas discussed in 3.4.2 regarding communication vs. computation effort and the potential memory bandwidth congestion on NESS.

4.1.1 Notes on the Effects of Instrumentation in HPF

Some warning on the intricacies of instrumenting code should be given here. To enable instrumentation to occur, the original HPF Game of Life and Image Processing codes had to be amended so that the split between communication and computation could take place. The following code extracts illustrate this issue further using the Image Processing code.

It can be seen in Code Example 8 that it is a two line computation for uninstrumented code (which in fact could be condensed into a single line albeit at a cost to readability); it has four EOSHIFT operations on a single array with an array-array subtraction with a final division.
newImageArray = (eoshift(oldImageArray,1,dim=2)+
eoshift(oldImageArray,-1,dim=2)+
eoshift(oldImageArray,1,dim=1)+
eoshift(oldImageArray,-1,dim=1));

newImageArray = (newImageArray-edge) / 4.0;

**Code Example 8 - Uninstrumented code example**

The code which is instrumented in Code Example 9 shows several more arrays used than before, which does slow the HPF code down and means more memory must be used than is absolutely required. It is also prudent to note that some of the classifications used for describing code activity are not necessarily as straightforward as it would first appear; for example, the EOSHIFT operations have been categorised as communication operations but this also involves a non-trivial proportion of local copying in reality.

There are also the calls to the clock which (as in MPI code) affects total runtime, albeit not significantly.

call system_clock(first_comm_1) ! FIRST COMMUNICATE
tmp1 = eoshift(oldImageArray,1,dim=2)
tmp2 = eoshift(oldImageArray,-1,dim=2)
tmp3 = eoshift(oldImageArray,1,dim=1)
tmp4 = eoshift(oldImageArray,-1,dim=1)
call system_clock(first_comm_2)
first_comm_total = first_comm_total + 
                   (first_comm_2 - first_comm_1);
call system_clock(third_compute_1) ! THIRD COMPUTE
newImageArray = ((tmp1+tmp2+tmp3+tmp4)-edge) / 4.0;

Code Example 9 - Instrumented code example

**4.1.2 Notes on Data Table Presentation**

The following sections’ data tables are given to show the split between communication and computation in the codes, given in the columns two and three. The fourth column is the sum of the previous two. The total time is the time for execution without input-output measured. The unaccounted time is the difference between the communication and computation sum and the total time given – essentially, it is calculated by subtracting column five from column four. The un-instrumented time is the result from the data tables in the previous chapter.
### 4.1.3 Results Data for the Instrumented Game of Life

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Communication</th>
<th>Computation</th>
<th>Communication + Computation</th>
<th>Total Time</th>
<th>Unaccounted Time</th>
<th>Un-instrumented Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>11.999</td>
<td>29.907</td>
<td>41.906</td>
<td>41.907</td>
<td>0.001</td>
<td>40.840</td>
</tr>
<tr>
<td>2 Processors</td>
<td>9.328</td>
<td>15.330</td>
<td>24.658</td>
<td>24.662</td>
<td>0.004</td>
<td>22.904</td>
</tr>
<tr>
<td>4 Processors</td>
<td>3.616</td>
<td>7.741</td>
<td>11.357</td>
<td>11.359</td>
<td>0.002</td>
<td>11.086</td>
</tr>
<tr>
<td>8 Processors</td>
<td>4.021</td>
<td>3.898</td>
<td>7.919</td>
<td>7.931</td>
<td>0.012</td>
<td>9.600</td>
</tr>
<tr>
<td>16 Processors</td>
<td>1.871</td>
<td>1.872</td>
<td>3.743</td>
<td>3.743</td>
<td>0.000</td>
<td>3.399</td>
</tr>
</tbody>
</table>

**Table 16 - Instrumentation of Game of Life in HPF on NESS**

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Communication</th>
<th>Computation</th>
<th>Communication + Computation</th>
<th>Total Time</th>
<th>Unaccounted Time</th>
<th>Un-instrumented Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>0.063</td>
<td>5.564</td>
<td>5.627</td>
<td>5.634</td>
<td>0.007</td>
<td>5.540</td>
</tr>
<tr>
<td>2 Processors</td>
<td>1.452</td>
<td>4.004</td>
<td>5.456</td>
<td>5.473</td>
<td>0.017</td>
<td>4.461</td>
</tr>
<tr>
<td>4 Processors</td>
<td>0.759</td>
<td>1.922</td>
<td>2.680</td>
<td>2.896</td>
<td>0.216</td>
<td>3.001</td>
</tr>
<tr>
<td>8 Processors</td>
<td>0.594</td>
<td>0.820</td>
<td>1.413</td>
<td>1.433</td>
<td>0.020</td>
<td>1.394</td>
</tr>
<tr>
<td>16 Processors</td>
<td>1.113</td>
<td>0.458</td>
<td>1.571</td>
<td>1.607</td>
<td>0.036</td>
<td>1.750</td>
</tr>
</tbody>
</table>

**Table 17 - Instrumentation of Game of Life in MPI on NESS.**

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Communication</th>
<th>Computation</th>
<th>Communication + Computation</th>
<th>Total Time</th>
<th>Unaccounted Time</th>
<th>Un-instrumented Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>5.749</td>
<td>4.697</td>
<td>10.446</td>
<td>10.447</td>
<td>0.001</td>
<td>10.123</td>
</tr>
<tr>
<td>2 Processors</td>
<td>5.516</td>
<td>3.306</td>
<td>8.822</td>
<td>8.824</td>
<td>0.002</td>
<td>8.497</td>
</tr>
<tr>
<td>4 Processors</td>
<td>3.174</td>
<td>1.517</td>
<td>4.690</td>
<td>4.693</td>
<td>0.002</td>
<td>4.708</td>
</tr>
<tr>
<td>8 Processors</td>
<td>1.743</td>
<td>0.681</td>
<td>2.423</td>
<td>2.425</td>
<td>0.002</td>
<td>2.426</td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.903</td>
<td>0.274</td>
<td>1.177</td>
<td>1.179</td>
<td>0.002</td>
<td>1.191</td>
</tr>
</tbody>
</table>

**Table 18 - Instrumentation of Game of Life in HPF on HECToR**
<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Communication</th>
<th>Computation</th>
<th>Communication + Computation</th>
<th>Total Time</th>
<th>Unaccounted Time</th>
<th>Un-instrumented Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>0.048</td>
<td>4.581</td>
<td>4.629</td>
<td>4.634</td>
<td>0.005</td>
<td>4.619</td>
</tr>
<tr>
<td>2 Processors</td>
<td>0.125</td>
<td>3.388</td>
<td>3.513</td>
<td>3.520</td>
<td>0.007</td>
<td>3.501</td>
</tr>
<tr>
<td>4 Processors</td>
<td>0.203</td>
<td>1.699</td>
<td>1.902</td>
<td>1.912</td>
<td>0.010</td>
<td>1.893</td>
</tr>
<tr>
<td>8 Processors</td>
<td>0.159</td>
<td>0.750</td>
<td>0.909</td>
<td>0.919</td>
<td>0.010</td>
<td>0.929</td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.208</td>
<td>0.341</td>
<td>0.548</td>
<td>0.558</td>
<td>0.010</td>
<td>0.570</td>
</tr>
</tbody>
</table>

Table 19 - Instrumentation of Game of Life in MPI on HECToR

4.1.4 Analysis of Results for the Instrumented Game of Life

Whilst it is reassuring that the graphs below show that as processor count increases the computation effort is halved at each data point, Figure 16 shows that HPF’s computation section takes up three quarters of its 40-second runtime on a single processor; to do the exact same computation, MPI takes less than six. This is an incredible difference, and indicates some extremely inefficient activity is going on in HPF in comparison to MPI. Similarly, what is considered to be communication takes up a large section of runtime on a single processor despite having no other cores to communicate with. This might be explained by the fact that a CSHIFT operation is actually just partially made up of communication between processors, and a good proportion is actually copy operations on local data.

At higher numbers of processors, HPF appears to scale better regarding its communication – the effort spent on communication and computation is approximately similar on sixteen processors, whilst in MPI the majority of time is spent on inter-node communication and not the calculation.
Figure 16 – Instrumentation of Game of Life in HPF on NESS

Figure 17 - Instrumentation of Game of Life in MPI on NESS

On HECToR, shown in Figure 18 and Figure 19, it is surprising to see that HPF’s computation time on a single processor is much less than on NESS. This is because both machines have the same AMD Opteron processors. Another relevant area for discussion is that at sixteen processors on HECToR, the MPI version is no longer dominated by communication cost. This is not true on NESS; it is reasonable to suggest that HECToR’s better performance is due to its high-speed interconnect structure.
Figure 18 - Instrumentation of Game of Life in HPF on HECToR

Figure 19 - Instrumentation of Game of Life in MPI on HECToR
### 4.1.5 Results Data for Instrumented Image Processing

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Communication</th>
<th>Computation</th>
<th>Communication + Computation</th>
<th>Total Time</th>
<th>Unaccounted Time</th>
<th>Uninstrumented Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>8.360</td>
<td>21.953</td>
<td>30.313</td>
<td>0.007</td>
<td>29.339</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>8.403</td>
<td>11.561</td>
<td>19.964</td>
<td>0.010</td>
<td>17.618</td>
<td></td>
</tr>
<tr>
<td>4 Processors</td>
<td>3.562</td>
<td>6.070</td>
<td>9.632</td>
<td>0.005</td>
<td>8.618</td>
<td></td>
</tr>
<tr>
<td>8 Processors</td>
<td>2.405</td>
<td>3.512</td>
<td>5.917</td>
<td>0.005</td>
<td>4.583</td>
<td></td>
</tr>
<tr>
<td>16 Processors</td>
<td>1.664</td>
<td>1.486</td>
<td>3.150</td>
<td>0.002</td>
<td>3.528</td>
<td></td>
</tr>
</tbody>
</table>

Table 20: Instrumentation of Image Processing in HPF on NESS

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Communication</th>
<th>Computation</th>
<th>Communication + Computation</th>
<th>Total Time</th>
<th>Unaccounted Time</th>
<th>Uninstrumented Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>0.018</td>
<td>12.098</td>
<td>12.116</td>
<td>0.003</td>
<td>11.980</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>0.382</td>
<td>4.413</td>
<td>4.795</td>
<td>0.004</td>
<td>4.959</td>
<td></td>
</tr>
<tr>
<td>4 Processors</td>
<td>0.316</td>
<td>3.353</td>
<td>3.669</td>
<td>0.004</td>
<td>3.827</td>
<td></td>
</tr>
<tr>
<td>8 Processors</td>
<td>0.675</td>
<td>1.179</td>
<td>1.854</td>
<td>0.004</td>
<td>2.268</td>
<td></td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.849</td>
<td>0.288</td>
<td>1.136</td>
<td>0.003</td>
<td>1.079</td>
<td></td>
</tr>
</tbody>
</table>

Table 21: Instrumentation of Image Processing in MPI on NESS

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Communication</th>
<th>Computation</th>
<th>Communication + Computation</th>
<th>Total Time</th>
<th>Unaccounted Time</th>
<th>Uninstrumented Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>5.396</td>
<td>6.100</td>
<td>11.496</td>
<td>0.003</td>
<td>11.543</td>
<td></td>
</tr>
<tr>
<td>2 Processors</td>
<td>3.956</td>
<td>4.750</td>
<td>8.706</td>
<td>0.004</td>
<td>8.959</td>
<td></td>
</tr>
<tr>
<td>4 Processors</td>
<td>2.284</td>
<td>2.498</td>
<td>4.782</td>
<td>0.003</td>
<td>4.794</td>
<td></td>
</tr>
<tr>
<td>8 Processors</td>
<td>0.776</td>
<td>1.427</td>
<td>2.203</td>
<td>0.002</td>
<td>2.262</td>
<td></td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.263</td>
<td>0.168</td>
<td>0.431</td>
<td>0.001</td>
<td>0.423</td>
<td></td>
</tr>
</tbody>
</table>

Table 22: Instrumentation of Image Processing in HPF on HECToR
<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Communication</th>
<th>Computation</th>
<th>Communication + Computation</th>
<th>Total Time</th>
<th>Unaccounted Time</th>
<th>Uninstrumented Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>0.010</td>
<td>8.451</td>
<td>8.461</td>
<td>8.462</td>
<td>0.001</td>
<td>8.489</td>
</tr>
<tr>
<td>2 Processors</td>
<td>0.105</td>
<td>3.239</td>
<td>3.344</td>
<td>3.345</td>
<td>0.001</td>
<td>3.372</td>
</tr>
<tr>
<td>4 Processors</td>
<td>0.097</td>
<td>1.367</td>
<td>1.464</td>
<td>1.465</td>
<td>0.001</td>
<td>1.505</td>
</tr>
<tr>
<td>8 Processors</td>
<td>0.088</td>
<td>0.441</td>
<td>0.530</td>
<td>0.531</td>
<td>0.001</td>
<td>0.540</td>
</tr>
<tr>
<td>16 Processors</td>
<td>0.092</td>
<td>0.216</td>
<td>0.308</td>
<td>0.309</td>
<td>0.001</td>
<td>0.325</td>
</tr>
</tbody>
</table>

Table 23 - Instrumentation of Image Processing in MPI on HECToR

4.1.6 Analysis of Results for Instrumented Image Processing

As with the Game of Life code, the Image Processing code’s instrumentation reveals significant communication levels at lower numbers of processors in HPF. This is true for both codes on both machines. There are also extremely high computation times in comparison to do the same functionality in MPI, and once more HECToR mirrors NESS’ trends, although with faster results.

Overall, the Image Processing code follows similar patterns to those noted in 4.1.4. As such, this data appears to support previous reasoning on the topic of why HPF shows poor performance on lower numbers of processors. Note, for example, the second columns of Table 22 and Table 23. The instrumentation of the Image Processing code has once more shown that this especially poor performance on small processor counts may be attributed to the area classified as communication in HPF. It is an area which involves the use of HPF’s intrinsic functions, as with the Game of Life.
Figure 20 - Instrumentation of Image Processing in HPF on NESS

Figure 21 - Instrumentation of Image Processing in MPI on NESS
4.1.7 Conclusions

Using the largest numbers of processors, HPF’s runtime is not significantly
dominated by communication costs on either machine, unlike MPI which appears
to suffer from network contention on the shared-memory NESS machine. However, at lower numbers of processors, there remains extremely slow
communication time in HPF. Moreover, on a single processor a lot of the communication time also appears unnecessary at first, unless previous comments regarding local data copying and section 4.1.6’s notes relating to inefficient intrinsic functions are taken into account. As such, the rest of this chapter will outline experiments designed to inform on this matter further.

4.1.8 An Experiment into CSHIFT Efficiency as an Intrinsic Operation

HPF’s performance appears to be less impressive than MPI in general. In an attempt to find out why this might be, one of the core operations of the Game of Life code is now compared to a similar operation. One reason this might be a factor is due to the particularly large communication time apparent when looking at the instrumentation data tables for HPF on lower numbers of processors.

Essentially, this is an attempt to identify the regions of HPF’s inefficiency, and to assess whether the prediction of acceptable HPF performance on higher numbers of processors due to superior scaling is one of merit. From previous experiments it is known that CSHIFT scales relatively acceptably compared to naïve code implementation performing similar operations, but an indication of whether an inefficiency when using lower numbers of processors might explain the poor performance is sought.

As such, this section shows CSHIFT’s performance on NESS compared to a similar operation call, namely SCOPY. SCOPY is a BLAS (Basic Linear Algebra Subprograms) library routine (Dongarra, 2002) which allows for the copying of a real or complex matrix into another. Its call is given in Code Example 10 below.

```
CALL SCOPY (N, X, INCX, Y, INCY)
```

**Code Example 10 - The BLAS SCOPY call**

As shown, the operation takes five parameters. They are:

- N is the number of elements in input matrix;
- X is a matrix with N elements;
- INCX describes the storage spacing between elements of SX (i.e. the offset or stride);
- Y is a matrix with N elements;
- INCY is the storage spacing between elements of SY.

The SCOPY operation is compared to CSHIFTs in both dimensions when a shift of zero (i.e. a straight copy) and a shift by one (involving minor data movement) is performed.
4.1.9 Results Data

Note that the achieved memory bandwidth has been calculated by the size of the data transferred (4*1024*1024 since it is a 32-bit REAL array of 1024*1024 elements) divided by the number of iterations (1000) which is then divided by the time taken for computation. 1000 was chosen as an acceptable number of iterations to give a large enough timing for accuracy but to not waste computational time on the machines.

<table>
<thead>
<tr>
<th>No. Processors</th>
<th>Operation</th>
<th>Time (s)</th>
<th>Achieved Memory Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>Shifted by one in X direction</td>
<td>17.562</td>
<td>238.828</td>
</tr>
<tr>
<td>1 Processor</td>
<td>Shifted by zero in X direction</td>
<td>15.703</td>
<td>267.102</td>
</tr>
<tr>
<td>1 Processor</td>
<td>Shifted by one in Y direction</td>
<td>17.781</td>
<td>235.887</td>
</tr>
<tr>
<td>1 Processor</td>
<td>Shifted by zero in Y direction</td>
<td>15.796</td>
<td>265.530</td>
</tr>
<tr>
<td>1 Processor</td>
<td>Standard SCOPY call</td>
<td>4.110</td>
<td>1020.512</td>
</tr>
</tbody>
</table>

Table 24 - For 1000 iterations on a single processor on a 1024*1024 grid on NESS

<table>
<thead>
<tr>
<th>No. Processors</th>
<th>Operation</th>
<th>Time (s)</th>
<th>Achieved Memory Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Processor</td>
<td>Shifted by one in X direction</td>
<td>7.733</td>
<td>542.375</td>
</tr>
<tr>
<td>1 Processor</td>
<td>Shifted by zero in X direction</td>
<td>5.812</td>
<td>721.646</td>
</tr>
<tr>
<td>1 Processor</td>
<td>Shifted by one in Y direction</td>
<td>7.926</td>
<td>529.210</td>
</tr>
<tr>
<td>1 Processor</td>
<td>Shifted by zero in Y direction</td>
<td>5.767</td>
<td>727.339</td>
</tr>
<tr>
<td>1 Processor</td>
<td>Standard SCOPY call</td>
<td>2.878</td>
<td>1457.187</td>
</tr>
</tbody>
</table>

Table 25 - For 1000 iterations on a single processor on a 1024*1024 grid on HECToR

4.1.10 Analysis of Results

Figure 24 to Figure 27 below show the results from NESS and HECToR; the following comments are relevant to both systems. Whilst HECToR offers faster runtimes, the general trend is the same as on NESS.

It is seen that SCOPY clearly outperforms CSHIFTs regardless of the amount of data to be shifted or the dimension of the shift. For example, on NESS SCOPY is three times as fast as a zero-shifted copy; on HECToR this difference is less noticeable, but only to as good as over two times faster.

At the same time, the peak memory bandwidth for CSHIFT is extremely disappointing. According to figures released by the manufacturer, listed in the references under AMD, the peak memory bandwidth of a machine such as NESS (a dual-core shared memory Sun Fire X4600 system with AMD Opteron processors which are 8000 series, AMD64e, 2.6Ghz) is 17k to 20k MB/Sec. This figure is given for an SCOPY operation on a quad-core system for the STREAMS
Benchmarks\textsuperscript{1}. The first generation of Opteron processors were released in 2003 and have slightly lower performance results (nearer 12k MB/sec). This illustrates that even when SCOPY achieves below the peak memory bandwidth of the system being tested on, CSHIFT performs even worse in comparison.

As an aside it is also interesting to note the clear difference between the two dimensions' results – the x-axis shifts show slightly faster times at every test on both machines. This possibly relates to the manner in which arrays are stored, and thus accessed, contiguously in memory. Fortran stores higher dimensional arrays as contiguous series of elements ordered by column (in the opposite manner to languages such as C, which stores by row), commonly referred to as column-major order. This refers to the manner in which the elements are stored linearly in memory, as opposed to physical address space.

![Execution Time on NESS on 1 Processor - Single Operation](image)

**Figure 24 - Execution time on NESS on 1 processor - Single Operation**

\textsuperscript{1} The STREAMs benchmarks are calculated by summing the amount of data the program reads and writes during execution; for a simplified example, if a code were to read 1Mb of data and then write 1Mb of data with a time of a second, the STREAM bandwidth figure would be published as 2Mb/sec. However, STREAMs benchmarks are designed to work with much larger datasets than those which can fit into the executing machine's cache. See McCalpin (1995).
Figure 25 – Achieved memory bandwidth on NESS on 1 processor - Single Operation

Figure 26 – Execution time on HECToR on 1 processor - Single Operation
4.1.11 Conclusions

HPF’s performance on lower numbers of processors may be explained by poorly performing intrinsic functions. The one tested here is CSHIFT, and despite previous experiments having shown that it actually scales relatively well using larger processor counts, it fails to achieve anywhere near the peak memory bandwidth figures the hardware manufacturer has published. More specifically, it performs noticeably slower than similar operations such as SCOPY by a factor of two on HECToR and by a factor of around three on NESS.
Chapter 5

Conclusions

5.1 Technical Conclusions

The principal area of investigation for the project was to assess the performance of HPF, using two codes which exhibit behaviour similar to standard scientific applications. This has been achieved through testing on two machines, one of which has run HPF code for the first time anywhere in the world. At the same time, a brief look into the available software kernels’ efficiency as well as an exploration into why HPF may be such an apparent underachiever in comparison to the more popular MPI has been undertaken.

The first conclusion from the project is that the kernel used in HPF codes affects runtime significantly. The structure of calling CSHIFT operations can be arranged to give much better performance results than an initial crude implementation may. This can be seen from the results of the operations termed a 12-step CSHIFT compared to those of a compacted one (see 2.1.2). At the same time, using a naïve Fortran-77 style call is slower and scales worse than the most optimised CSHIFT calls.

Another conclusion is the confirmation that modern HPF continues to be slower than MPI. This is particularly true when using lower numbers of processors. However, HPF’s lack of speed may in part be explained by inefficient intrinsic functions such as CSHIFT, which performs poorly compared to operations which might be expected to have similar performance (even when the latter achieves nowhere near the peak memory bandwidth of the system being tested on). As such, CSHIFT still remains a core weakness of HPF’s performance, as 4.1.8 illustrates.

HPF does, however, scale much more impressively; there are indications that if the work which is suggested in 5.2 were undertaken that it may be proven to be as fast as MPI on larger numbers of processors. As discussed in Chapter 3 Performance Comparisons, this may be much more relevant than the runtimes noted on less than sixteen processors, since standard supercomputers have tens
of thousands of nodes today and those numbers are rapidly increasing (Farber, 2008).

5.2 Further Work

Further areas of work that would be suggested for investigation are described below.

Primarily, the running of these codes on even larger numbers of processors would be an interesting avenue to pursue. Of course, this would only be available on the HECToR machine. Nevertheless, it would be enough to see further into the scaling behaviour of the codes, and to ascertain if there is any merit in earlier discussions’ reflection that HPF’s performance may be similar to MPI at larger processor counts. At project onset it was hoped that this could be achieved within the timeframe of work. The reason this was never achieved was due technical issues which meant that no job submission greater than sixteen processing cores was permitted for execution, despite working code which is ready for submission to larger numbers. These issues remain unsolved at the time of writing despite a Cray employee’s attempts to remedy the situation.

A second area, which should be strongly considered, is the effect of strong vs. weak scaling on the languages, codes and systems tested. The project used strong scaling (static problem sizes without regard for the number of processors being executed on) throughout. However, weak scaling using fixed local problem sizes is an area of interest that was never able to be explored due to project time constraints. This might be more difficult with the image processing code due to the nature of the application’s input being pre-rendered images, but certainly not impossible.

Another area to be considered would be to profile the codes using commercially available tools, rather than relying on the manual instrumentation performed. This is due to the intricacies of manual instrumentation; profiling would enable a deeper, more hardware-centric view of the code behaviour. Appropriate tools for this work would include CrayPAT, which “provides access to a wide variety of performance experiments that measure how an executable program consumes resources while it is running” (Cray, 2002), available on HECToR. It would be especially useful when undertaken alongside the first suggestion for further work given above, using larger numbers of processors. For the purposes of this project, however, there were two reasons in favour of manual instrumentation. The first was an unfamiliarity with the profiling tools which would have necessitated additional learning in a relatively short project timeframe; the second was that the profiling tools offer a large, complex amount of information which, at the time of instrumentation, would have mostly been superfluous to requirements.

An additional area, which could be investigated to extend the general area of work, would be to compare and contrast the performance of Co-Array Fortran. This would be to see if it achieves where HPF is considered to have failed, and as
far as the actual results of this project confirms, continues to fail – which is to perform competitively against explicit communication languages such as MPI. This was never in the project scope originally but remains an interesting side interest to the main project.

5.3 Project Review

There are certain topics that any review of a project should discuss. Myllyaho et al. (2004) define these into four separate categories:

- positive aspects to the project;
- negative aspects to the project;
- areas noted for future improvement;
- and those aspects which remain confusing.

To summarise the positive aspects to this project, which do not need to be discussed in great detail, it is felt that the overall objective (to determine whether HPF performs better than or worse than MPI) was met satisfactorily. The project scope offered a secure environment in which the goals were realistically achievable – and throughout the project allowed for enough flexibility that areas of interest could be explored. It is these factors which underpinned an early personal confidence in the project which is crucial when working in an unknown or unfamiliar area. In this regard the project methodology used was much more similar to agile development software engineering methods than any strictly-staged waterfall model practice (Williams and Cockburn, 2003). For example, if it had been written explicitly, the incremental project development might have appeared as:

- obtaining the codes and getting them running, achieving initial results;
- production of missing HPF code;
- running batches of code submissions for reliable results;
- instrumentation and investigation behind the behaviour above.

The limited time available meant that a focus on what was practically achievable from the project start was more ideal than hoping for unrealistic targets, and an incremental approach to the project served this purpose well.

It has been also rewarding to prove that an XT4 can run HPF code successfully.

Another positive is that the approach towards supervision offered an excellent structure for the project to progress, namely the suggestion of post- and pre-meeting summaries being circulated. These allowed the people involved in the
project to ensure that no misunderstandings had occurred during discussions, and provided a useful source of reference between meetings.

A negative factor which should be noted during this review is that there was poor estimation of the amount of time required to achieve certain tasks, usually of a technical nature such as developing the missing HPF code quicker than expected or taking too long to perform CSHIFT experimentation. Along similar lines, a significant negative of the project was that during planning there was an assumption that the pre-developed codes worked as intended. Unfortunately for the project schedule, this turned out to be false; more time was spent trying to debug the image processing MPI code than developing an HPF one.

Improvements which are recommended involve the two negative aspects defined above. Firstly, the tendency towards poor estimation of time could mainly be attributed to lack of experience since it is a common weakness amongst student projects in particular. Planning for this by seeking advice from those with more knowledge on the topic, improving the awareness of one’s own abilities, and ensuring appropriate contingency time is available can all contribute towards more project success in this area. Secondly, there are two improvements to described as a result of the incorrect assumption that third-party codes work as intended. The first is that these codes should never be assumed to function well or to have been tested adequately. The second is an indication of MPI’s specific complexity when dealing with communication bugs which are so much harder to identify than in HPF code, due to the explicit nature of the language.

A noteworthy aspect to the project which remains confusing is the potential to run HPF codes on higher numbers of processors on HECToR. Since the further work section explains that this is an area of primary importance to continuing the achievements of this project, it is hoped a resolution presents itself shortly.
Appendix A

A1 Instructions for Code Execution

The following section will outline instructions for code execution for the codes and systems used throughout the project.

A1.1 NESS Execution

The following instructions are for compiling and submitting jobs on the NESS system which is based at ness.epcc.ed.ac.uk. The batch submission scripts are the *.sge files and the command to submit should be amended to be the required number of processors. Note that to do small test runs this approach is most acceptable but for reliable results 16 processors should always be requested (despite potentially using less for computation).

- For compiling and submitting the HPF Game of Life code, use:
  - pghpf -Mfree -o life life.hpf -lrt
  - qsub -cwd -pe mpi 4 life.sge

- For compiling and submitting the MPI Game of Life code:
  - mpif90 life.f -o life
  - qsub -cwd -pe mpi 4 life.sge

- For compiling and submitting the HPF Image Processing code, use:
  - pghpf -Mfree -o life process.f90
  - qsub -cwd -pe mpi 4 process.sge

- For compiling and submitting the MPI Image Processing code, use:
  - make
  - qsub -cwd -pe mpi 4 parallelImage2D.sge
For compiling and submitting the neighbour-execution experiments and the CSHIFT efficiency tests, use:\n\n- `pghpf -Mfree -o scopy scopy.hpf -lblas`
- `qsub -cwd -pe mpi 1 scopy.sge`

### A1.2 HECToR Execution

The following instructions are for compiling and submitting jobs on the HECToR system which is based at `login.hector.ac.uk`. The batch submission scripts are the `*.pbs` files; these should be amended to suggest the required number of processing cores. `Mppwidth` requests a given number of processing cores (at the time of writing the limit is 4096) whilst `mppnppn` sets the number of tasks to place on a dual core processor (valid choices are 1 and 2; when using a value of 1, the scheduler reserves twice as many nodes compared to when this value is 2).

- For compiling and submitting the HPF Game of Life code, use:
  - `./compile`
  - `qsub submit.pbs`

- For compiling and submitting the MPI Game of Life code:
  - `./compile`
  - `qsub submit.pbs`

- For compiling and submitting the HPF Image Processing code, use:
  - `./compile`
  - `qsub submit.pbs`

- For compiling and submitting the MPI Image Processing code, use:
  - `make`
  - `qsub submit.pbs`

- For compiling and submitting the neighbour-execution experiments and the CSHIFT efficiency tests, use:\n  - `./compile`
  - `qsub submit.pbs`

---

\[^2^ Note that the SCOPY routine should not be run on more than a single processor; amend the calls as necessary.

\[^3^ Ib id.\]
References


