Performance of Parallel Sparse Matrix Solvers on HPCx

Jianrong Chen

Edinburgh Parallel Computing Centre,
University of Edinburgh
I, Jianrong Chen, confirm that this dissertation and the work presented in it are my own achievement.

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

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;

I have acknowledged all main sources of help;

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;

I have read and understand the penalties associated with plagiarism.

Signed:

Date:

Matriculation no: 0340740
Abstract

IterSol is a program which was developed by D. Konstantinous in 2002, and optimized on the UK’s national supercomputer HPCx by Xing Chen in 2003, as their MSc projects. The HPCx system has been significantly upgraded to phase 2 in 2004 which introduces modifications to the hardware. It is interesting to investigate the performance of the IterSol on the new system and optimize it for the purpose of developing a reliable and robust solution for linear system of equations.

The comparison work is important throughout the project. A few factors produce the different performance of the same sparse matrix on the HPCx phase 2 against phase 1. The matrices Xing Chen used last year have been re-addressed to compare the different performance of the HPCx phase 2 and phase 1.

The refinement of the IterSol’s implementation and customization is necessary on the new high-end computing system. Bugs are inevitable in complex program development especially for parallel programs on high-end computers. Some bugs of the original IterSol program have been investigated and rectified in this project. However, some bugs are hard to solve in a short project period and the reasons for these bugs have been analysed.

A number of useful areas of future work are discussed at the end of this thesis. Although these efforts have not been completed, the project this year provides a clear guide for the future work for the IterSol to make it more competitive in the linear system equation solution.
List of Tables

Table 1: Phase 2 vs Phase 1 Hardware Comparison .................................................. 10
Table 2: Colony against HPS performance .............................................................. 15
Table 3: Matrices Inter-node communication vs Intra-node communication .............. 18
Table 4: Cage13 matrix timing with 200, 2000, 5000 and 10000 iterations .................. 20
Table 5: Solver module communication and computation timing summaries ............... 23
Table 6: Cage13.rua matrix with 200 iterations timing with optimised timing code ........ 23
Table 7: Verification of the doubling up code ......................................................... 33
Table 8: Matrices and the number of maximum processors for MPI_Buffer_attach() on HPCx system ................................................................. 35
Table 9: Matrix vector multiplication ---- The kernel of IterSol ................................. 42
Table 10: Performance of Matrix-Vector Multiplication ........................................... 42
Lists of Figures

Figure 1: The scalability of Cage12.rua matrix on phase 2 against phase 1 .......................... 12
Figure 2: The scalability of Cage13.rua matrix on phase 2 against phase 1 .......................... 13
Figure 3: The matrix-vector multiplication of cage12.rua and cage13.rua matrices on Phase 2 .... 14
Figure 4: The Ratio of performance of BiCG algorithm on IterSol on phase 2 against phase 1 .... 15
Figure 5: Solver module timing on cage13.rua matrix with different iterations ....................... 23
Figure 6: Solver module timing on cage13.rua matrix with new timing method ....................... 24
Figure 7: The speedup of cage12.rua and cage13.rua matrices of the optimized timing code .... 25
Figure 8: The first 100 iteration elapsed time of matrices .................................................... 27
Figure 9: The summary of the elapsed time of the first 100 iteration of cage24.rua, cage13.rua and cage26.rua matrices ................................................................. 29
Figure 10: The elapsed time against the total elapsed time of the first 100 iterations of cage24.rua, cage13.rua and cage26.rua matrices ............................................................. 29
Figure 11: The profile structure of IterSol ................................................................. 31
Figure 12: A simple Chaco output map file ............................................................. 33
Figure 13: The doubling up map file ........................................................................ 35
Figure 14: The original and optimized buffer size for MPI_Buffer_attach() ....................... 38
Figure 15: Scalability of cage24 and cage26 matrices on the HPCx phase 2 ....................... 38
Figure 16: Scalability of cage12 and cage13 compare with scalability of cage24 and cage26 matrices on the HPCx phase 2 ................................................................. 39
Acknowledgements

First of all, I would like to deeply thank my supervisors Dr David S. Henty and Dr Alan D. Simpson for their guidance and support through my project as well as my whole year study in the University of Edinburgh. They have always been available and kind for discussions and offered insightful ideas so that I could solve the difficulties in my project and write up the thesis here. Without their patient and kind help, I could not finish writing up my thesis for the MSc degree.

I deeply thank my course director Dr Judy Hardy for her kind help when I was ill near the deadline of this thesis. In addition, I need to appreciate Miss Jane Mitchell’s warm help from the beginning of the application of my study in Edinburgh University to the end of my project.

I am also like to thank Mr Xibei Jia for his guidance when I was debugging my programs and nearly stuck. His multi-language programming experience is significantly helpful during the development of my code.

Finally, I acknowledge support from Edinburgh University Alumni of Hong Kong in China. The entire coverage of the tuition fee scholarship which is initiative funded by alumni in Hong Kong and supported by the Edinburgh University Association (Hong Kong) for overseas student makes it possible for me to complete my study in Edinburgh.
Chapter 1

Introduction

1.1 Motivation

Linear algebra system solvers are one of the conventional computing applications in the scientific and engineering realm. It is publicly accepted that linear system, derived from the mathematic research dating back to the emerging date of the computer when two 20th century giants, Alan Turing and John von Neumann developed the stored-program machine and renewed the interests in matrices, especially on the numerical analysis of matrices [3], is the focus of contemporary mathematical and computational research. Moreover, as an important component of linear systems, sparse matrix operations dominate the performance in a wide range of scientific and engineering applications. IterSol, developed by Dimitris Konstantinous as his MSc dissertation in 2002, is a program for the solution of large sparse matrices computations. It is also a competent candidate in the scientific and engineering application.

The next year after Dimitris’ work, Xing Chen validated the correctness of the IterSol’s output and re-investigated the performance of the iterative solver on EPCC’s Sun Fire15K system, which is a Symmetric Multiprocessing (SMP) machine running Solaris on 52 Ultra-SparcIII processors. Xing also ported IterSol to the UK’s national supercomputer HPCx (clustered SMP IBM p690), looking at optimizing the parallel decomposition algorithms, investigating parallel scalability for larger matrices, general performance optimizations as well as mixed-mode MPI/OpenMP implementations.

The work performed by both Dimitris and Xing are the basis of this year’s project. The corresponding project in 2004 will focus on the decomposition of the input matrices and the preliminary optimization of IterSol’s scaling on the upgraded HPCx system. The recent upgraded HPCx system makes the optimization of the original IterSol in this project particularly timely. The upgraded HPCx architecture, named HPCx Phase 2, has been deployed since June 2004. Not only did the available number of processors increase, but also faster processors and faster interconnect among SMP nodes have been deployed. Furthermore, it is worth noting that phase 2 comprises 32-way rather than 8-way nodes on phase 1, which is the catalyst of the interest to investigate the most time-consuming decomposer module of the IterSol. Obviously, the comparison of the performance changes in terms of the modifications of the HPCx phase 2 becomes the beginning of the project this year.

Based on the original design by Dimitris and Xing’s optimization, it is hoped to do more work on the IterSol to make it a useful and robust sparse-matrix solver on supercomputers as a scientific and engineering application iterative solver.
1.2 Related work

Linear equation solvers, particularly sparse linear equation solvers, are major academic as well as industrial research aims. A wide range of libraries and packages have been developed to solve or help solve computational engineering applications.

As stated by Xing in his last year’s dissertation [2], Aztec is the most similar to IterSol because its stand alone decomposition package Chaco is used by the kernel of IterSol’s decomposition module in the last two years’ projects. However, as more and more processors and computational resources are available during the time since the original IterSol was developed, it is inevitably expected that advanced decomposition method of the input matrix can be implemented as the basic decomposition method in the time-consuming decomposition module. *ParMetis* is the newest parallel decomposition library which is suitable for the k-way partition of unstructured graphs. The advantages of utilization of parallel decomposition algorithms in IterSol on HPCx system are obvious, but the stability and reliability cannot be guaranteed while the *Chaco* library is mature for many years. The project performed in this year just focuses on the previous decomposition algorithms and continues utilizing the *Chaco* package as the decomposer kernel. Since bugs have been found while using *Chaco* to decompose the input sparse matrices, *ParMetis* will likely be the better choice of decomposition module for the next version of IterSol.

In addition to the decomposition and matrix-vector multiplication modules, the matrix libraries which are the input data for the solver cannot be ignored in this project. Xing Chen examined large matrices (the largest is cage14.rua, 1,505,785 rows and 1,505,7785 columns, 27,130,349 nonzero entries) on Lomond which is a SMP machine with 52 processors. The reason why Xing decomposed the biggest matrix on *Lomond* rather than on the HPCx is unclear. However, bugs have been found on *Chaco* when huge size matrices were feed as inputs to the IterSol. The problem cannot be solved unless the source code of Chaco is modified to be able to run in 64-bit mode.

At present, Matrix Market provides a wide range of sparse matrices from just 100 rows * 100 columns to 5 million rows * 5 million columns. We will investigate on HPCx phase 2 very large matrices. The original IterSol only can read the Hawell-Beoing format (H/B) and Raw format matrices (RAW). However, Matrix Market as well as a wide range of matrix generating toolkits, provides Matrix Market format matrices which motivates the work in this project to read Matrix Market format matrices. In addition, synthetic matrices are stored as CSR format directly for the purpose of being accessed by the code itself easily.

1.3 About this Document

The dissertation will be organized in the order as the work has been performed.

Chapter 2 will focus on the comparison of the HPCx phase 1 and phase 2. First, an overview of the UK’s national supercomputer HPCx is described. Second, the recent upgraded HPCx phase 2 is compared with the HPCx phase 1 from the architecture point of view. Finally and the most
important in this chapter, the same experiments which Xing did in 2003 on the HPCx phase 1 will be re-investigated on phase 2. Therefore, the upgraded performance of phase 2 can be seen at the end of this chapter and it is also the basis of the optimization and analysis work in the following chapters.

Chapter 3 addresses the decomposition module of IterSol and intends to find out the most optimized decomposition algorithms which are available in the *Chaco* library and look deeply for the reason why they give better performance.

A project is not always performed successfully and some bugs of the original *IterSol* code have been found out in the development and experiment processes. The original timing function cannot calculate the elapsed time of BiCGSTAB method properly which means the Xing’s results of the elapsed time per iteration of the BiCGSTAB method is challenged. This issue will be addressed in chapter 4.

Next, in chapter 5, addressing the large matrices on HPCx phase 2 will become the focus. However, Chaco cannot decompose large matrices properly on HPCx phase 2 because of the memory limitation. Rather than exchanging the *Chaco* library, the approach which generates simulated outputs of the *Chaco* library is highlighted.

Chapter 6 intends to address very large matrices. The scaling of the HPCx phase 2 will be shown on matrices which have various sizes and sparsity structures. In this chapter, very large matrices and large number of processors of HPCx system will be utilized.

All works will be concluded in Chapter 7 and future work and the expectation of the work which has not been done will be discussed in the final Chapter.
Chapter 2

Basic Comparison of HPCx Phase 1 and Phase 2

2.1 Overview of the HPCx Phase 2 vs Phase 1

HPCx is the UK’s national supercomputer as well as one of the biggest academic computers in Europe. It was initially set up as a 6-year project at the end of 2002 as a co-operation between Edinburgh Parallel Computing Centre and Daresbury Laboratories as well as IBM corporation. Nevertheless, the upgraded HPCx system has been deployed recently. Work hereby begins at the comparison of the architecture differences between HPCx Phase 1 and HPCx Phase 2.

The newest HPCx system comprises a new architecture of IBM hardware. By the end of June 2004, the HPCx Phase 2 system comprises 50 Regatta nodes and each Regatta node contains 32 Power4+ 1.7 GHz processors, which mean that 1600 powerful processors make up the current HPCx system. Three major modifications of the HPCx system may take the performance of the IterSol to higher levels --- the higher clock rate of the processors, the faster interconnecting switches among different logical partitions (LPAR) and the changes of the size of the LPAR are relevant to the higher performance of the IterSol.

The processors of HPCx Phase 2 are based around the p690+ frames. It is now equipped with 1.7GHz Power4+ processors rather than the previous 1.3GHz Power4 processors. It is obviously expected that the serial performance will increase 1.7/1.3 = 1.308 times. This is the standard ratio criteria of the comparison in this project.

The other major difference between the phase 2 and phase 1 is that the SP Switch 2 has been replaced by the High Performance Switch (HPS) which is known as “Federation”. It is also expected that the large scale performance will benefit because of the reduction of the communication time among different LPARs.

The implementation of the HPS leads to the changes of the size of the configurable frames. The HPCx phase 1 was configured with 8-way LPARs logically because this configuration allowed 4 pairs of networking adaptors per frame to attach to the original Colony Switch. However, the deployment of the HPS changed the configuration because of the better performance and faster communication obtained in the HPS. Thereafter, the HPCx Phase 2 is operating as a 32-way clustered machine.

In addition to these three major modifications, the cache and size of the memory pages have been enlarged resulting in interesting cache effects in the experiments. It is reasonable to expect that the overall performance of the IterSol on the Phase 2 might be more 1.308 times than that of phase 1 because of the faster communication and cache factors. The hardware comparison between the
HPCx phase 2 and phase 1 are shown in table 1.

<table>
<thead>
<tr>
<th></th>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>40 p690 Frames</td>
<td>50 p690+ Frames</td>
</tr>
<tr>
<td></td>
<td>1.3 GHz Power4 Processors</td>
<td>1.7 GHz Power4+ Processors</td>
</tr>
<tr>
<td></td>
<td>4 x 8 Way LPARS per Frame</td>
<td>1 x 32 Way LPAR per Frame</td>
</tr>
<tr>
<td></td>
<td>1280 processors in total</td>
<td>1600 processors in total</td>
</tr>
<tr>
<td>Interconnect</td>
<td>SP Switch 2 with Colony PCI adaptors</td>
<td>HPS Switch</td>
</tr>
<tr>
<td>Disk</td>
<td>18 Tbytes GPFS</td>
<td>36 Tbytes GPFS</td>
</tr>
<tr>
<td>Rmax Linpack</td>
<td>3.41 Tflop/s</td>
<td>6.19 Tflop/s</td>
</tr>
</tbody>
</table>

Table 1: Phase 2 vs Phase 1 Hardware Comparison [13]

The next section will focus on the performance of matrices used on the phase 1 system and re-address them in the phase 2 system.

### 2.2 Performance comparison of HPCx Phase 2 vs Phase 1

This chapter concentrates on comparison of performance between the HPCx phase 1 and phase 2 on the cage12.rua and cage13.rua matrices. Since the HPCx system is impossible to go back to phase 1, all results in this chapter concerning phase 1 are based on the experiments completed by Xing Chen last year. It has been verified by Dimitris that the heuristic algorithm --- Recursive Spectral Bisection (RSB) [11] is the best decomposition method compared with Linear, Block-Striped and Random algorithms. (see Appendix D for more description of these decomposition algorithms). It is worthwhile noting that the timings in this chapter include the I/O elapsed time which Xing used 10,000 iterations in BiCGSTAB algorithm to overlap the effects of the I/O elapsed time. (more timing issue will be discussed in Chapter 4).

What makes the parallel computer more interesting than the serial computer is that the parallel computers, including Symmetric Multi-Processors (SMP) machines and distributed clusters of machines that include a combination of single processors and SMP machines, can get higher performance in both data-intensive and time-intensive cases. However, in the clustered setting such as the HPCx system, the communication overhead might impede the speedup of deploying more processors. The *IterSol*’s performance is basically limited by the explicit message passing as well as the implicit synchronization. As far as *MPI_Allreduce()* is concerned, *IterSol* executes global reduction on scalar variables which means that the size of the message passing is constant and the operation only depends on the number of communicating processors as well as the synchronization of themselves. Obviously, the decomposition strategy is important to minimize the number of communicating processors. It is also relevant to the data reuse. *Chaco* library has been a competitive serial decomposition package since 1995 as a convenient decomposition library for many scientific and engineering applications.

Recursive Spectral Bisection (RSB) decomposition algorithm has been proved by Dimitris to be the best one among Linear, Block-Striped and Random decomposition algorithms in *Chaco* library.
Hierarchical Recursive Spectral Bisection decomposition algorithm has also been proved by Xing to be better than pure RSB algorithm in IterSol package. Since the HPCx phase 2 comprises 32-way rather than the previous 8-way, the hierarchical RSB should be two categories: RSB8 stands for the same decomposition algorithm as Xing used last year, and RSB32 means the new hierarchical decomposition algorithm for the phase 2 settings. We used these contexts in the following part of the thesis.

Figure 1 shows the scalabilities of cage12.rua matrix on phase 2 and phase 1 with different RSB algorithms. The general performance of the cage12.rua on phase 2 is obviously better than that on phase 1 no matter which RSB algorithm is deployed. RSB, Hierarchical RSB8 and Hierarchical RSB32 are nearly the same while no more than 32 processors are used because the 32-way frame machine does not know the differences among the different decomposition algorithms. However, Hierarchical RSB32 is obvious the best decomposition algorithm when more than 32 processors are available. Since cage12.rua matrix is not a large matrix in the test cases, its scalability on phase 2 is not good. The inter-box communication makes the scalability drop dramatically and the performance of 128 processors on cage12.rua matrix is nearly the same as that of 10 processors. However, the figure shows that the performance of one box is good, which means that all communications happen in the frame rather than using the HPS. It is nearly two times faster than phase 1. Bearing in mind that the CPU clock rate is 1.308 times faster on phase 2 against phase 1, the better speedup might be the better switching modules and the cache effects.

Figure 1: The scalability of Cage12.rua (130,228 X 130,228) matrix on phase 2 against phase 1
The timing is based on 10,000 iterations and it includes I/O timing.
Figure 2 shows the scalability of cage13.rua matrix on phase 2. Since the cage13.rua matrix is roughly 16 times larger than the cage12.rua matrix, the scalability of cage13.rua matrix is much better. RSB32 is the best decomposition algorithm, especially while 64 processors are available. However, the speedups drop quickly after 64 processors and when 128 processors are deployed, the performances drop again to nearly the same as 10 processors.

The kernel of the IterSol, sparse matrix-vector multiplication portion, does not have any communication among processors, therefore it is necessary to investigate the performance of the matrix-vector multiplication with single processor as well as running the code with various number of processors and comparing the timing of MATVEC which is the elapsed time of matrix-vector multiplication. The HPCx phase 2 comprises two floating-point units each of which could deliver one result per clock cycle [10], the nominal peak performance of each processor should be 6.8 Gflop/s in terms of the computational architecture theory.

In figure 3, the kernel matrix-vector multiplication performances of cage12.rua and cage13.rua matrices on phase 2 are shown. It is the sparse matrix-vector multiplication performance of the input matrices, which means that there is no communication among processors. The matrix-vector multiplication is also the kernel of the IterSol [1]. Unlike the method performances which include plenty of communications among processors, there is no significant difference between cage12.rua matrix and cage13.rua matrix. The matrix-vector multiplication makes the pure RSB algorithm linear speedup and RSB32 the worst algorithm. Since the elapsed time of the matrix-vector multiplication of cage13.rua matrix is longer than that of cage12 matrix, the kernel performance of hierarchical RSB algorithm on cage13.rua matrix is closer than that of cage12.rua matrix. However, the kernel of the IterSol is a small time-consuming portion of the whole solver module which has been shown in Xing’s dissertation and will be investigated deeply in the next Chapter.
Figure 3: The matrix-vector multiplication performance of cage12 (Top) and cage13 (Below) matrix
The timing is based on 10,000 iterations and it includes I/O timing.

Figure 4 shows the ratio of the BiCGSTAB algorithm performance on phase 2 with cage12.rua and cage13.rua matrices against those on phase 1 which are shown in Xing’s dissertation last year. Performances of both cage12.rua and cage13.rua matrices benefit at least 1.5 times faster on phase 2 compared Xing’s results on phase 1. The worst algorithm pure RSB on cage12.rua matrix even obtains over 2.5 times faster benefit on phase 2 against phase 1. All the algorithms and matrices we use in the tests benefit more than 1.308 which is the ratio of the clock rate of processors between the HPCx phase 2 and phase 1. It is no doubt that cage12.rua matrix can get more benefits from the upgraded system than cage13.rua because the communication overhead affects the smaller matrix cage12.rua matrix more than the larger matrix cage13.rua in the certain condition.
To summarise up this chapter, we compare the hardware and configuration modifications of the HPCx phase 2 against phase 1. Results have been shown that the general performance of phase 2 is better than phase 1. In addition to the processor clock rate ratio of phase 2 against phase 1 which comprises 1.308 times faster (overall above 1.5), the general performance of phase 2 is more than this number because of the replacement from Colony to High Performance Switch (HPS) as well as the cache effects. In addition, we introduce a new hierarchical RSB algorithm comprises the upgraded HPCx 32-way settings. It is obvious that the hierarchical RSB32 decomposition algorithm is the best one among pure RSB and the Xing’s hierarchical RSB8 algorithm on the HPCx phase 2. In the next chapter, the reason of the better performance of phase 2 shown in this chapter will be addressed deeply.
Chapter 3

Comparison of the Decomposition Performance between the HPCx Phase 1 and Phase 2

It is crucial to choose the best decomposition algorithm which is available in Chaco library for the IterSol to load balancing the work done by processors and minimise the communication among processors especially the most expensive inter-box communications. Dimitris and Xing also described the importance of the decomposition algorithms and optimized the decomposition module in their dissertation. Since the switching hardware has been changed on the HPCx phase 2, it is worthwhile re-addressing the decomposition and work out the best decomposition algorithm of the IterSol for the future experiments.

3.1 The communication of the clustered SMP system

3.1.1 The cost of the inter node communication of HPCx phase 2

The processor should be 1.308 times faster than the previous phase 1, so higher performance contributed to the speedup should be basically IBM’s High Performance Switch which is called Federation. It might make sense if we compare the graph of cage13.rua matrix on phase 1 and phase 2. How much is Federation better than Colony? It is expected that the latter one should be much slower than the first one because the inter-node communication is much slower than the communication happen inside one node.

The difference between the colony and HPS can be referred in HPC Challenge [14] which is updated every day of benchmark results of a lot of supercomputer systems such as IBM p690 and Cray T3E. Table 2 shows the performance of colony switch and HPS running the HPL library which is a solver for a randomly generated dense linear system of equations. The HPS obtains 1.67 times faster than the colony switch in terms of the data shown in the table. Although this case is not the same as the HPCx system and the performance is relevant to the specific system, it is obvious that the HPS is faster than the colony switch.
In addition, Xing Chen stated that the intra-node communication is roughly six times faster than the inter-node communication [2]. We assume that the cache effect modification is small enough that it can be ignored in the test. Since the inter-node communication is getting faster because of the better performance of HPS, the inter-node communication is less six times slower than the intra-node communication.

### 3.1.2 The Communication and Computation ratio

What makes the differences among different decomposition algorithms? Dimitris showed that the better the decomposition algorithm is, the less communication among different processors need while halo-swapping and distant data transmit in the solver module. He also proved that the RSB algorithm is one of the best decomposition approaches in the Chaco library. In 2003, Xing investigated the cage12.rua matrix (130,288 X 130,288) with BiCGSTAB algorithm and concluded that the computation and communication ratio is 0.203 which means that 83% of the total elapsed time was used for communications among processors. We will investigate cage12.rua matrix again in Chapter 4 for the computation and communication ratio of IterSol on the upgraded HPCx system. However, can we predict the highest ratio of the computation and communication of IterSol’s kernel?

The amount of computation is easy to be estimated. Let us suppose matrix A is N x N, then for each element of the product b of matrix A and vector v, we will perform n times multiplications, as well as n-1 times additions. There are n elements of b, so the total number of floating-point operation is

\[
N \times (N + (N-1)) = 2N^2 - N
\]

We also assume that addition and multiplication operations take the roughly same amount of elapsed time, which is called TC. Therefore, the estimated computation time is \((2N^2 - N) \times TC\).

On the other hand, we also estimate the communication costs. The number of floating-point numbers that have to be communicated is n (to send a row of A), +1 (to send back the response) for each element of b, for a grand total of

\[
N \times (N+1) = N^2 + N
\]
Again, we can assume the total communication time is roughly \((N^2 + N) \times TCO\), therefore the ratio of the communication and computation is \(((N^2 + N) / (2N^2 - N)) \times (TC / TCO)\)

Since the cost of a single floating-point operation is much less than the cost of communicating one floating-point number, we intend to make this number as small as possible. If the problem is large enough, which means the size of the matrix A is significantly large, it seems that the communication overhead could be reduced. However, the ratio of communication and computation is roughly independent of N. Based on the mathematical theory, the larger the size of the matrix A, the ratio just gets closer to 0.5 which means that there are always 50% communication cost in the \emph{solver} module against matrix-vector multiplication and this will always be a problem. However, it is not a significantly critical problem right now because the IterSol’s kernel is far away from 50% communication and computation ratio and it is believed that elegant implementation and better decomposition algorithm can help to reduce the communication demand. We will see better communication and computation ratio on the upgraded HPCx system in section 4.1.

Please note that the analysis above is based on the density matrix but it is still meaningful. On sparse matrix application context, it is unnecessary to send the zero portion of the given matrix. Therefore, the ratio of communication can be smaller which means that the \((TC / TCO)\) is much larger than 1. On the other hand, computation is nearly the same as our analysis. So the ratio of communication and computation could be larger than 50% based on the elegant implementation. Please also note that the analysis above is based on the number of the communication and computation rather than the time of the communication and computation comparison.

### 3.2 Importance of the decomposition algorithm

The decomposition algorithms for partitioning graphs generated from the sparse matrix file are paramount in the IterSol because it is expected to take a plenty of time to decompose a large sparse matrix evenly and the results of the output can affect the iterative solver to compute the final solution. Dimitris analysed how crucial the decomposition algorithm is in his dissertation, comparing four algorithms – RSB, Linear, Block and Random used in \emph{Chaco} library. He concluded that the RSB is the best decomposition algorithm \emph{Chaco} library can offer for the IterSol.

Furthermore, Xing extended the \emph{decomposer} module to multi-level decomposition. He indicated that re-decompose the sub-domain into sub-sub-domain can get higher performance in the \emph{solver} module. But why is the multi-level decomposition better than the simple decomposition in IterSol? The new HPCx system comprises 32-way that means each LPAR consists of 32 processors rather than original 8 processors. What is the communication change in the new settings?

#### 3.2.1 The inter communication and intra communication of Chaco’s algorithms

The HPCx system is clustered SMP machine which comprises 50 shared memory frames and each frame has 32 independent units. There are two different kinds of communications --- Inter
communication and Intra communication. The inter communication is much more expensive than
the intra communication because it would utilize the switching module between nodes which is
much slower than the shared access memory in the same frame. In 2002, Xing concluded that the
intra-node communication is roughly 6 times faster than the inter-node communication based on
the experiments he did on Colony switch hardware. It is reasonable to predict that the inter-node
communication is faster than the previous experiments because of the modification of switch
module.

Four decomposition algorithms have been implemented in the decomposer module of IterSol ---
Random, Linear, Block-striped and Recursive Spectral Bisection (RSB). Dimitris and Xing proved
that the RSB decomposition algorithm is the best one in the decomposer module. The
methodology they used in their dissertation is based on the curves of the speedup graphs. However,
it is necessary to address the communication volume of the decomposition algorithm in the Chaco
library for the solver module of the IterSol. The work performed in the next section will answer
this question.

3.2.2 The comparison of the RSB and hierarchical RSB algorithms

We addressed the performance of cage12.rua and cage13.rua matrices in chapter 2 and concluded
that the hierarchy RSB decomposition algorithms (RSB8 on phase 1 and RSB32 on phase 2) are
better than the simple RSB decomposition algorithm. The larger the matrix is, the better speedup it
can obtain. The question is why the hierarchical RSB algorithm is better? The idea of hierarchical
RSB decomposition algorithm is that we decompose the input matrix to sub-matrices and then
re-decompose the sub-matrices to sub-sub-matrices. It is obvious that choosing a good
decomposition algorithm is important because we need to load balance the work and decrease the
necessary communication among different processors. For the purpose of comparison of the
advantages of hierarchical RSB algorithm, it is necessary to compare the message exchange inside
the same node as well as outside the different nodes.

Table 3 (a) shows the message exchange of cage12.rua and cage13.rua matrices with RSB, RSB8
and RSB32. Bearing in mind that the inter-node communication is much more expensive than the
intra-node communication, we can see in Table 3 (a) not only the inter-node communication
message passing of pure RSB is larger than hierarchical RSB, but also the intra-node
communication message is almost the same. The 8-way RSB algorithm is also different from the
32-way RSB algorithm especially in the inter-node communication. Both cage12.rua matrix and
cage13.rua matrix comprise similar intra-node communication no matter whether 8-way or
32-way decomposition algorithm are deployed. It is known that the intra-node communication is
not expensive compared with the inter-node communication. We conclude that the advantage of
32-way RSB decomposition algorithm is basically based on the less inter-node communication. In
cage12.rua which is a 130,228 by 130,228 matrix, 32-way RSB algorithm indicates less 38.5%
inter-node communications than that of 8-way RSB algorithm. Again, in cage13.rua which is a
larger matrix, 32-way RSB algorithm alleviate roughly 36% communication volume between
different frames of the HPCx system.
<table>
<thead>
<tr>
<th></th>
<th>Cage12.rua</th>
<th></th>
<th>Cage13.rua</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inter</td>
<td>Intra</td>
<td>Inter</td>
<td>Intra</td>
</tr>
<tr>
<td>RSB64</td>
<td>65,319</td>
<td>181,524</td>
<td>192,891</td>
<td>608,948</td>
</tr>
<tr>
<td>RSB8+RSB8</td>
<td>54,516</td>
<td>197,374</td>
<td>159,856</td>
<td>579,647</td>
</tr>
<tr>
<td>RSB2+RSB32</td>
<td>39,537</td>
<td>201,188</td>
<td>101,980</td>
<td>539,168</td>
</tr>
</tbody>
</table>

Table 3 (a): Cage12 and Cage13 matrices Inter-node communication Vs Intra-node communication with RSB, Hierarchical RSB8 and Hierarchical RSB32 decomposition algorithm

<table>
<thead>
<tr>
<th></th>
<th>Cage12.rua</th>
<th></th>
<th>Cage13.rua</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inter</td>
<td>Intra</td>
<td>inter</td>
<td>Intra</td>
</tr>
<tr>
<td>RSB32</td>
<td>----------</td>
<td>202,570</td>
<td>----------</td>
<td>591,277</td>
</tr>
<tr>
<td>RSB2+RSB32</td>
<td>39,537</td>
<td>201,188</td>
<td>101,980</td>
<td>539,168</td>
</tr>
<tr>
<td>RSB4+RSB32</td>
<td>105,511</td>
<td>231,992</td>
<td>241,255</td>
<td>697,177</td>
</tr>
<tr>
<td>RSB8+RSB32</td>
<td>164,999</td>
<td>233,499</td>
<td>425,623</td>
<td>747,488</td>
</tr>
<tr>
<td>RSB16+RSB32</td>
<td>213,697</td>
<td>264,935</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3 (b): Cage12 and Cage13 matrices Inter-node communication Vs Intra-node communication with Hierarchical RSB decomposition algorithm

While utilizing more nodes to compute the cage12.rua and cage13.rua matrices with Hierarchical RSB decomposition algorithm RSB(n) + RSB32 ( n ∈ [0,16] ), the inter-node and intra-node communication are also interesting which is shown in Table 3 (b). We note that on cage12.rua matrix the inter-node communication increase dramatically from RSB2+RSB32 to RSB4+RSB32 comprising 166% increase. However, from RSB4+RSB32 to RSB8+RSB32, it increases 56% and again from RSB8+RSB32 to RSB16+RSB32, the communication just increases 29%. Based on these numbers, it is reasonable to predict that the maximum speedup lies between RSB2+RSB32 and RSB4+RSB32 which means 64 processors make the cage12.rua matrix get the highest performance and then tail off. Since the problem size of cage13.rua matrix is significantly larger than the size of cage12.rua matrix, the inter-node and intra-node communications are larger than the communication of cage12.rua matrix as well. In this chapter, 256 processors are the largest number of processors could be available because of the HPCx memory limitation itself. From RSB2+RSB32 to RSB8+RSB32 decomposition methods, the inter-node and intra-node communication of cage13.rua matrix increase linearly. The computation power for the sparse matrix-vector multiplication is increasing linearly while more processors are available. Therefore, the communication and the computation increase at the same time, it makes sense that after RSB2+RSB32 the speedup increases slowly.

To summarise this chapter, better decomposition algorithm could alleviate the communication overhead of the decomposer module. Since the inter-node communication is more expensive than that of intra-node communication, the inter-node communication overhead dominates the IterSol solver performance on a certain size of matrix. The less the communication happens in the solver module, the better performance we obtain.
Chapter 4

Correctness of timing function of the original IterSol

The timing function is critical to address the performance of parallel computing software. As far as IterSol is concerned, most figures and graphs are based on the measurements of the timing function. The timing function of the IterSol is developed by Dimitris based on the ANSI C `gettimeofday` timing subroutine which shall obtain the current time and express it as seconds and microseconds. However, it is trivial to develop a timing function, but it is tricky to put the timing function in proper places of the program as well as measure appropriate portions of the module of the program.

4.1. The problem of measuring the time per iteration in the solver module

Xing measured the time per iteration in the `solver` module with 10,000 iterations and took the average value. The experiments and figures shown in chapter 2 are also based on Xing’s measuring method. But why do we not execute the `solver` module 200 times and get the average value? It is obvious that the average value of 200 iterations shall be roughly equal to the average value of 10,000 iterations but the total charges of the budgets of the HPCx machine could be saved sharply.

It is not like what we expect when different numbers of iterations are executed in the `solver` module. Table 3 shows the timing of the cage13.rua matrix with various numbers of iterations from 200 to 10,000 times.

<table>
<thead>
<tr>
<th>Time/Iterations</th>
<th>200</th>
<th>2,000</th>
<th>5,000</th>
<th>10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSB64</td>
<td>40,930</td>
<td>9,813</td>
<td>7,238</td>
<td>6,555</td>
</tr>
<tr>
<td>RSB8+RSB8</td>
<td>46,222</td>
<td>8,235</td>
<td>7,105</td>
<td>6,623</td>
</tr>
<tr>
<td>RSB2+RSB32</td>
<td>37,517</td>
<td>7,003</td>
<td>6,290</td>
<td>6,962</td>
</tr>
</tbody>
</table>

Table 4: Cage13 matrix timings with 200, 2000, 5000 and 10000 iterations. All experiments in this table is based on Dimitris’ and Xing’s code last two year. It tests the BiCGSTAB algorithm elapsed time per main iteration.

In terms of table 4, the timings of per iteration of BiCGSTAB loop are significantly different, especially while the number of iterations is not large enough. When the BiCGSTAB algorithm is executed 200 times in the `solver` module of IterSol, the elapsed time is roughly 39,000 micro-seconds depending on the decomposition algorithms. However, executing the loop 2,000 times makes the timing results drop dramatically to around 7,000 micro-seconds with rsb2+rsb32 decomposition algorithm. Again, the timing results drop slightly to 6,555 micro-seconds and 6,290 micro-seconds with 5,000 iterations and 10,000 iterations and the same decomposition algorithm respectively.
What makes this timing problem? The timing of per iteration of BiCGSTAB algorithm is calculated outside the main loop of the algorithm as a stand alone subroutine rather than in the BiCGSTAB main loop. Nevertheless, the elapsed time per iteration can be obtained in the main loop. Figure 5 shows the timing per iteration of cage13.rua matrix with 200 iterations and 10,000 iterations. The experiments are based on the timing of the original code developed by Xing with RSB2+RSB32 decomposition algorithm which means that 2 nodes take the communication and computation. The graph comprises 4 different columns for each iteration — *Send*, *Receive*, *Reduce* (*MPI_Allreduce*) and matrix-vector multiplication (*MATVEC*). What makes the different between 2 nodes is the reduction part of each iteration based on the Figure 5 (a). It is obvious that the timing of reduction of the first node (process 0 to process 31) is longer than that of the second node (process 32 to process 63). We also compare the details of the experiments on Table 5. What makes the surprised results is computation and communication ratio between 200 iterations and 10,000 iterations of BiCGSTAB algorithm on cage13.rua matrix. The computation and communication ratio is 26.7% while BiCGSTAB is executed 10,000 iterations. However, this value drops sharply to 4.15% while it is only carried out 200 iterations. We also address the intermediate number of iterations between 200 and 10,000. The computation and communication ratio comprises 23.0% and 25.1% on 2,000 iterations and 5,000 iterations respectively.

Table 5 shows that *send*, *receive* and *MatVec* operation of different iterations in the BiCGSTAB algorithm in IterSol are all stable increase while the number of iterations increase. However, the reduction operation is not the case. The reduction time of 200 iteration is much more than our prediction if the experiment with 10,000 iterations is correct. It is worth noting that the OTHERs which excludes the send, receive, reduce and matrix-vector multiplication timing are relatively stable in spite of how many iterations we choose in the BiCGSTAB algorithm main loop. It might be the startup overhead of the parallel programs. We note that in figure 5 (a) some reduction operations of specific iterations are much longer than the normal reductions, especially in the first 32 processes. However, it is different if we execute the code with 10,000 iterations. The startup overhead is nearly the same cost but the other operations take much longer elapsed time than those of 200 iterations. It is also worth noting that the reduction operations in 10,000 iterations case are much more stable than those of 200 iterations.

Table 6 and figure 6 show the optimized timing results of 200 iterations in the same experimental case. The *total* time in figure 6 we choose now is the total elapsed time of 200 iterations inside the BiCGSTAB main loop rather than including the I/O of the BiCGSTAB algorithm. There is still some startup overhead but it is more than 5 factors smaller than the previous test. The matrix-vector multiplication and the communication ratio is roughly 12.4% which is better than the previous 4.2% although it is not as good as the ratio of 10,000 iterations (28%).

- 23 -
Figure 5 (a): Solver module timing on cage13.rua matrix with RSB2+RSB32 and 200 iterations with original timing method.

Figure 5 (b): Solver module timing on cage13.rua matrix with RSB2+RSB32 and 10,000 iterations with original timing method.
Table 5: Solver module communication and computation timing summaries of cage13 matrix with 200, 2000, 5000 and 10000 iterations and BiCG algorithm. Max is the maximum elapsed time of 64 processes, Min is the minimum elapsed time of 64 processes and Sum is the total elapsed time of 64 processes.

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Operation</th>
<th>send</th>
<th>receive</th>
<th>Reduce</th>
<th>MatVec</th>
<th>Others</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 iter</td>
<td>Max</td>
<td>0.33</td>
<td>0.24</td>
<td>2.72</td>
<td>0.36</td>
<td>-------</td>
<td>10.14</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>0.11</td>
<td>0.06</td>
<td>0.43</td>
<td>0.30</td>
<td>-------</td>
<td>4.74</td>
</tr>
<tr>
<td></td>
<td>Sum</td>
<td>14.12</td>
<td>9.13</td>
<td>67.74</td>
<td>21.33</td>
<td>401.25</td>
<td>513.57</td>
</tr>
<tr>
<td>2,000 iter</td>
<td>Max</td>
<td>3.19</td>
<td>1.64</td>
<td>4.37</td>
<td>3.86</td>
<td>-------</td>
<td>16.07</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>1.12</td>
<td>0.64</td>
<td>1.17</td>
<td>2.87</td>
<td>-------</td>
<td>12.32</td>
</tr>
<tr>
<td></td>
<td>Sum</td>
<td>138.63</td>
<td>70.53</td>
<td>168.41</td>
<td>212.21</td>
<td>393.97</td>
<td>923.75</td>
</tr>
<tr>
<td>5,000 iter</td>
<td>Max</td>
<td>8.46</td>
<td>4.36</td>
<td>13.51</td>
<td>9.02</td>
<td>-------</td>
<td>33.37</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>2.78</td>
<td>1.75</td>
<td>5.51</td>
<td>7.26</td>
<td>-------</td>
<td>30.44</td>
</tr>
<tr>
<td></td>
<td>Sum</td>
<td>342.16</td>
<td>183.84</td>
<td>597.45</td>
<td>521.25</td>
<td>399.08</td>
<td>2043.78</td>
</tr>
<tr>
<td>10,000 iter</td>
<td>Max</td>
<td>22.11</td>
<td>9.53</td>
<td>30.28</td>
<td>21.73</td>
<td>-------</td>
<td>70.68</td>
</tr>
<tr>
<td></td>
<td>Min</td>
<td>7.02</td>
<td>3.96</td>
<td>8.40</td>
<td>16.44</td>
<td>-------</td>
<td>67.18</td>
</tr>
<tr>
<td></td>
<td>Sum</td>
<td>780.22</td>
<td>435.62</td>
<td>1386.83</td>
<td>1178.92</td>
<td>419.34</td>
<td>4200.93</td>
</tr>
</tbody>
</table>

Figure 6: Solver module timing on cage13.rua matrix with RSB2+RSB32 and 200 iterations with optimized timing method

<table>
<thead>
<tr>
<th>iterations</th>
<th>operation</th>
<th>send</th>
<th>receive</th>
<th>reduce</th>
<th>MatVec</th>
<th>Others</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 iter</td>
<td>Sum</td>
<td>15.52</td>
<td>8.025</td>
<td>59.5</td>
<td>22.70</td>
<td>77.55</td>
<td>183.3</td>
</tr>
</tbody>
</table>

Table 6: Cage13.rua matrix with 200 iterations timing with optimised timing code.

The Total time is the elapsed time inside the BiCGSTAB algorithm main loop
After optimizing the timing issue of the IterSol, we plot the scaling graphs of cage12.rua and cage13.rua matrices again. We have discussed the inter-node communication in section 3.3.2 and proved that how the inter-node communication increases in cage12.rua and cage13.rua matrices cases. Comparing the communication result in section 3.2.2, it makes sense that the figure 7 (a) (b) indicate the scaling characteristics of the cage12.rua and cage13.rua matrices on phase 2. Cage12.rua matrix reaches the highest speedup in 64 processors comprising the speedup approximate 14 and then tail off to 10 while using 128 processors. Bearing in mind that the inter-node communication is expensive and the intra-node communication of cage12.rua matrix from RSB2+RSB32 to RSB4+RSB32 increases 267% dramatically. Cage13.rua matrix can get better scalability because of the larger size of itself. It seems it can get higher speedup after 128 processors are available. In this chapter, 128 processors are the largest number of processors could
be available because of the HPCx memory limitation itself (see chapter 6 for details). From RSB2+RSB32 to RSB4+RSB32 decomposition methods, the inter-node and intra-node communication of cage13.rua matrix increase linearly. The computation power for the sparse matrix-vector multiplication also increases linearly while more processors are available. Therefore, the communication and the computation increase at the same time, it makes sense that after RSB32 the speedup increases slowly and keeps flat to RSB4+RSB32.

4.2 The necessary Barrier in the BiCGSTAB algorithm

The barrier is necessary to put in the BiCGSTAB algorithm in the solver module of the IterSol. There are two options to put the MPI barrier in the algorithm: inside the BiCGSTAB main loop and outside the BiCGSTAB main loop.

It seems reasonable to put the barrier outside the BiCGSTAB main loop because it guarantees all processes execute the BiCGSTAB main loop at roughly the same time to avoid the unpredictable timing result. However, the barrier inside the main loop can also guarantee the timing results, but it might affect the parallel performance. As describe above, we have to rectify the timing method from Dimitris timing subroutine to my timing subroutine which is basically standard ANSI C timing subroutine to guarantee the correctness of the timing result. This modification makes it necessary to put the barrier inside the BiCGSTAB algorithm to guarantee the timing result. Herein, all the experiments are based on the code which includes the barrier inside the BiCGSTAB algorithm in the solver module of the IterSol.

4.3 The unstable timing per iteration in BiCGSTAB algorithm in IterSol

The plotted graphs shown in figure 7 are based on the best timing of the first 100 iterations of the BiCGSTAB algorithm. Why we choose the best one? Are there significant differences among iterations? Graphs below help us to obtain the possible answers. For the purpose of comparison clearly, we introduce cage24.rua and cage26.rua matrices which are the doubled up cage12.rua and cage13.rua matrices respectively. More details for the double up matrices will be discussed in the next chapter.

Figure 8 (a) (b) (c) show the elapsed time of the first 100 iterations of cage13.rua, cage24.rua and cage26.rua matrices respectively. The graphs show that the elapsed time of the first iterations is basically not stable. The first iteration generally takes a lot of time on both cage13 and cage26 cases. However, during the middle of the first 100 iterations the elapsed timings of each iteration are relatively stable. Furthermore, there are two interesting issues which are worthy noting and investigating. First, the stable timings are periodic and different matrix has not only the different elapsed time but also the frequency. Figure 9 shows the comparison elapsed timings of the first 100 iterations with BiCGSTAB algorithm of the cage13.rua, cage24.rua and cage26.rua matrices. We note that the frequency of the elapsed time of the cage24.rua matrix is faster than those of cage13.rua and cage26.rua matrices. It seems the larger the matrix is, the smaller frequency it will be in the first 100 iterations of the BiCGSTAB algorithm. Secondly, we note that in figure 10, the elapsed time of the beginning iterations are significant larger than the elapsed time of iterations in
the middle. Moreover, the sudden increases of one iteration which is close to the end of the measurement are interesting in our experiments of cage13.rua and cage24.rua matrices. But in the largest matrix cage26.rua we could not see this result.

The reasons for these issues are unclear. However, it is reasonable that we choose the best elapsed time of the first 100 iterations as the experimental result. The best elapsed time of a given matrix lies basically on the middle of the first 100 iterations which is the most stable portion in terms of the graphs shown below. It is simply alleviate the effects of the barrier as well as the synchronization among processes. The different frequencies of different input matrices could be looked as the cache effects of the HPCx system.

Figure 8 (a): The first 100 iteration elapsed time of cage13.rua matrix

Figure 8 (b): The first 100 iteration elapsed time of cage24.rua matrix
Figure 8 (c): The first 100 iteration elapsed time of cage26.rua matrix

Figure 9: The summary of the elapsed time of the first 100 iteration of cage24.rua, cage13.rua and cage26.rua matrices
To summarise this chapter, the timing issue is critical for profiling parallel programs. The original timing problem of IterSol package has to be solved before investigating the performance of IterSol on larger input matrices which will consume a large number of consuming times of the HPCx system. The work done in this chapter makes it possible to merely execute the iterative solver using much less iteration but obtain the roughly same result as 10,000 iterations. Furthermore, the reason of the timing problem has been addressed. It is clear that the barrier and the synchronization mechanism are essential in the solver module of IterSol. However, it is not the permanent solution of the timing problem. The unstable elapsed time of per iteration is interesting and it seems periodic based on the size of the matrix. After this chapter, all tests are based on the new timing mechanism.
Chapter 5

Optimisation of IterSol’s Decomposition Module on HPCx Phase 2

IterSol is a modular software package with three main modules which can be executed respectively. The figure 11 from Dimitris’ dissertation in 2002 shows the profile structure of the IterSol. The controller module is the interface between the users and the package itself and controls most required input parameters of the decomposer module and the solver module. The decomposer module has three main functions. First, read the matrix file from the users (Reader). The matrix formats can be read is either H/B collection or RAW format. This project enables the IterSol read the Matrix Market format and read the self-generated sparse matrix at runtime rather than just from fixed matrix files. Secondly, since the decomposer module implement the Chaco library as the main toolkit for matrix decomposition, the graph for the Chaco library might be constructed. Thirdly, simply invoking the Chaco library and obtain the map file as the main output of the decomposer module. The major results of the decomposition are stored in A.map as well as CSR(Compressed Sparse ROW) format matrix A.csr. The solver module invokes processors which are customized in the controller module and allocates the matrix in terms of the A.map file. Since the number of matrix nonzero entries is balanced in the map file, calculating the matrix-vector multiplication parallel in the solver module and reducing the results of each processor can get the solution which is stored in A.sol file.

Figure 11: The profile structure of IterSol (from Dimitris’ dissertation in 2002)
5.1 The problem of the decomposition module

Recursive Spectral Bisection (RSB), Linear, Random and Block-Striped decomposition algorithms have been implemented as the main decomposition methods by Dimitris in 2002. All of them except Block-Striped algorithm are implemented by simply calling the third-party Chaco [9] library. In 2003, Xing emphasized the optimization on the decomposition module. He found that the invert_cs and construct_graph were not efficient because the complexity of them are \(O(n^2)\), where \(n\) is the size (matrix row/column number) of the input matrix. Therefore, Xing investigated the invert_cs and construct_graph, optimizing the kernel of the invert process from CSC format to CSR format. The current code has been efficient in inverting the format from CSC format to CSR format which is very important while the CSC format is the major storage format of H/B collection.

Xing also investigated bigger matrices than Dimitris did in 2002. Cage12.rua (130,228 x 130,228 with 2,032,536 non-zero entries) and Cage13.rua (445,315 X 445,315 with 7,479,343 non-zero entries) can be decomposed by Chaco successfully on HPCx system. The decomposition results of cage12.rua and cage13.rua have been shown in Chapter 3. Xing decomposed his biggest sparse matrix, cage14.rua (1,505,785 X 1,505,785 with 27,130,349 non-zero entries) on Lomond (SUN Fire15K). It took him around 3.5 hours to decompose the cage14.rua with RSB algorithm on Lomond.

Why did Xing use Lomond rather than the bigger machine HPCx to decompose the cage14.rua? Chaco library is originally compiled with 32-bit mode which means that the maximum allocated memory space is roughly 4Gbytes. However, the test running cage14.rua matrix on HPCx system shows that Chaco cannot allocate enough memory space for both graph file and the map file of the input data and the output data. We intended to re-compile the Chaco with 64-bit mode which could provide much more memory space for the graph and map file. The story is not the same as predicted. Chaco can be compiled with –q64 flag on HPCx system, but the object code does not work properly in the experiments. Therefore, we cannot use Chaco as the decomposition library for the matrix which is roughly larger than 1 million by 1 million. The original biggest library Chaco can execute is cage13.rua. (give the biggest self-generated sparse matrix can run chaco properly)

5.2 Solutions of the decomposer module problem

5.2.1 Parallel decomposition algorithms for the IterSol

Two methods could be introduced to solve the lack of the memory space problem. First and the best solution might be to replace the Chaco library with other more reliable third-party decomposition library. ParMetis [8] is the parallel decomposition library which is suitable for
IterSol’s *decomposer* module. However, implementing parallel decomposition algorithm in the IterSol’s *decomposer* module and analysing its performance are out of the scope of this project. First, we need to fix the setting of the decomposition algorithm to compare the performance of the *solver* module of the IterSol between the HPCx phase 1 and phase 2. Second, *ParMetis* is relatively new parallel decomposition algorithm. Therefore, it will take a lot of risks to change the critical algorithms in a time-intensive MSc project in terms of the software engineering theory. More information about the parallel decomposition algorithm can be referred to Appendix A.3.

### 5.2.2 A possible solution for the IterSol using Chaco library

The second solution is not the direct method although it can produce the bigger map file simulating the output of the *Chaco* library as well as the input of the *solver* module. The general idea of this solution is that if we can decompose a matrix using *Chaco* library with P processor, we can also decompose a matrix which both the size of column and row are doubled the original matrix with Chaco library but using 2P processors [15]. After read the matrix file, the code constructs the graph as the input for Chaco. The biggest matrix library for *Chaco* currently is cage13.rua matrix, so the graph file and the map file generated by *Chaco* can be called cage26.rua.graph and cage26.rua.map. The map file indicates that the number of rows as well as the row identifications which belong to a specific processor. A very simple map file is shown in figure 8. It indicates that a 10-row matrix is decomposed by 2 processors. Each processor will take 5 rows of the matrix and calculate in the *solver* module separately. Figure 12 shows the text map file.
The map file can be duplicated as shown in figure 13. Hereafter, we obtain a larger map file and if the corresponding CSR format matrix is also duplicated. We in fact obtain the same result of the matrix which is 4 times larger than the original matrix. In another word, this method can produce a matrix and can be executed on Chaco library. Bearing in mind that the biggest matrix file Xing could decompose on HPCx system is cage13.rua with 445,315 X 445,315 rows and columns as well as 7,479,343 non-zero entries, we currently can create a matrix, called cage26.rua which is 890,630 X 890,630 rows and columns as well as 14,958,686 non-zero entries. Compared with the cage14.rua, the size of cage26.rua we produce is only the one fourth of the size of cage14.rua. However, in this chapter, we merely focus on the solution of the doubling up map files of the current executable matrices on phase 2. How to get larger matrices and the difficulties of generating very large matrices as well as executing them on phase 2 will be addressed in next chapter.

The doubling up code will double a specific matrix CSR format file and the Chaco’s output map file. For instance, we currently have cage12.rua matrix CSR format file and Chaco map file, what we require is the cage24.rua matrix which the number of rows and columns is double the size of the rows and column as well as the non-zero entries of cage12.rua matrix, so does the map file.
Figure 13: The doubling up map file

Top: Processor 1 and Processor 2 calculate pre-doubling up part of the matrix and Processor 3 and Processor 4 do the new part of the matrix.

Below: Text format map file of the doubling up matrix with double up number of processors

It is predictable that the communication is also doubled while using this code. Table 7 shows the communications of cage12.rua and cage24.rua matrices. The inter-node communication of cage24.rua matrix with RSB4+RSB32 is 79,074 which is exactly twice as much as that of cage12.rua matrix which RSB2+RSB32. Again, the inter-node communications of cage24.rua matrix with RSB8+RSB32 and RSB16+RSB32 are exactly double of the cage12.rua matrix. The same story happens in intra-node communication on both cage12.rua and cage24.rua matrices. It is obviously strong evidence that the doubling up code has been developed and implemented successfully. We could use the doubling up code and the random sparse matrix generator (describe in the Appendix) to benchmark the IterSol’s performance on the HPCx phase 2 system using the original Chaco decomposition package.

<table>
<thead>
<tr>
<th></th>
<th>Cage12.rua</th>
<th>Cage24.rua</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inter</td>
<td>Intra</td>
</tr>
<tr>
<td>RSB2+RSB32</td>
<td>39,537</td>
<td>201,188</td>
</tr>
<tr>
<td>RSB4+RSB32</td>
<td>105,511</td>
<td>211,992</td>
</tr>
<tr>
<td>RSB8+RSB32</td>
<td>164,999</td>
<td>233,499</td>
</tr>
<tr>
<td>RSB16+RSB32</td>
<td>-----------</td>
<td>-----------</td>
</tr>
</tbody>
</table>

Table 7: Verification of the doubling up code by comparing the communication between cage12 and cage24 matrices
To summarise this chapter, we get a possible solution to solve the memory limitation of the Chaco library on the HPCx system and doubling up the large matrices to larger matrices which cannot be obtained by merely invoking the Chaco library. In addition, a simple larger matrix can be generated by the sparse matrix generator. The scalability of IterSol on the HPCx phase 2 with very large matrices will be addressed in the next chapter to indicate the value of the doubling up implementation and extend the scope of the IterSol using the Chaco library.
Chapter 6

Scalability of the IterSol on HPCx Phase 2 with very large matrices

We have solved the decomposition problem in chapter 5 and simulate the output map files of Chaco library successfully. The simulated map files and doubled up CSR format matrices can be used as the inputs of solver module of IterSol. Since the memory limitation problem of Chaco library has been solved, it is interesting to address larger matrices with IterSol’s solver module on the HPCx phase 2.

6.1 The memory limitation of the IterSol input matrices

When large matrices are investigated in this project, Chaco library is not the unique memory limitation dilemma. The IterSol uses MPI_Bsend() and MPI_Buffer_attach() mechanism in the solver module. Dimitris customized the buffer size of the buffer message passing as the column of the input matrix by the number of available processors. It is expected this value will increase dramatically when large matrices are executed by a number of processors. This would not be an issue if the size of input matrices is small. However, when the size of input matrices and the number of available processors are large, the ANSI C malloc subroutine will fail because of the memory limitation.

The maximum allocatable memory space for malloc subroutine on the HPCx phase 2 is around 267Mbytes in terms of the malloc experimental program. The original buffer size comprises the size of the input matrix column by the number of the processors. Table 8 shows the maximum number of processors for different experimental cases. For cage26.rua matrix, the maximum number of processors which is not out of the limited memory space is 32. It is undoubtable that we cannot satisfy with merely one node to calculate the biggest matrix cage26.rua. Refinement work for the MPI_Buffer_attach should be introduced to get rid of this dilemma. One possible method is to adjust the buffer size of the malloc subroutine.

The modified code is shown in figure 14 (a) (b). The updated version has nothing to do with the number of processors which will be deployed. Since input matrices are always sparse matrices, the number of non-zero entries should be much smaller than the number of column by the number of available processors even in large problem size.
<table>
<thead>
<tr>
<th>Matrix</th>
<th>Non-zero entries</th>
<th>Row or Column</th>
<th>Max Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cage12.rua</td>
<td>2,032,536</td>
<td>130,228</td>
<td>256</td>
</tr>
<tr>
<td>Cage24.rua</td>
<td>4,065,072</td>
<td>260,456</td>
<td>128</td>
</tr>
<tr>
<td>Cage13.rua</td>
<td>7,479,343</td>
<td>445,315</td>
<td>64</td>
</tr>
<tr>
<td>Cage26.rua</td>
<td>14,958,686</td>
<td>890,630</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 8: Matrices and the number of maximum processors for `MPI_Buffer_attach()` on HPCx system

Bufsize = `sizeof(precision) * A->ncol * nprocs;`

If (bufsize< MINBUFSIZE)
Bufsize = MINBUFSIZE;
Buffer = (void *)smalloc(bufsize);
MPI_Buffer_attach(buffer, bufsize);

Figure 14 (a): The original buffer size for `MPI_Buffer_attach()`

Bufsize = `sizeof(precision) * A->nonzero;`

If (bufsize< MINBUFSIZE)
Bufsize = MINBUFSIZE;
Buffer = (void *)smalloc(bufsize);
MPI_Buffer_attach(buffer, bufsize);

Figure 14(b): The refinement buffer size for `MPI_Buffer_attach()`

### 6.2 Performance of IterSol solver module on phase 2 with very large matrices

The cage24.rua and cage26.rua matrices are doubled up matrices from existing cage12.rua and cage13.rua matrices. We have proved that the Hierarchical Recursive Spectral Bisection 32 algorithm is currently the best decomposition algorithm for IterSol package. In figure 15, the scalabilities of the cage24.rua and cage26.rua matrices are shown and compared. The cage24.rua matrix obtains the highest speedup roughly 17.5 in 128 processors and then tails off on phase 2. On the other hand, cage26.rua matrix can get generally higher speedup than cage24.rua matrix did. The performance of 2-node, 4-node of cage26.rua is better and it is expected that the curve goes toward higher value of speedup after 128 processors which indicates that the scalability of cage26.rua would be much better than cage24.rua matrix.
It is interesting to compare the performance of cage24.rua and cage26.rua matrices against the performance of their source matrices. In Figure 16, the new curves are compared with the curves of cage12.rua and cage13.rua matrices which have been expressed in chapter 4. Bearing in mind that the order of the sizes of the testing examples is: cage12 < cage24 < cage13 < cage26, it is obvious that the larger the matrix is, the better speedup we can get on the phase 2 system. It is reasonable to expect that larger matrices can get better performance and will show higher curves in the graph than any curves shown in Figure 16.
To summarise this chapter, the performance of doubling up matrices are investigated on the HPCx phase 2. The cage26.rua matrix is the largest matrix we used in the experiments. The behaviour of cage26.rua matrix is basically similar to its source cage13.rua matrix, but the speedup is higher than cage13.rua matrix. The memory limitation problem of the `smalloc` subroutine which is critical in the IterSol package and inevitable problem while large matrices are feed into the solver has been highlighted and quantified in this chapter.
Chapter 7

Conclusion

IterSol, an iterative solver for sparse matrix application on high-end computers, was developed and improved in the last two years. First developed in 2002 by Dimitris on Sun Fire 6800 system as an iterative solver for sparse matrix solution on small high-end computer, it shows computational power and elegant design. The UK’s national supercomputer HPCx system has made the IterSol more interesting and computationally powerful since Xing Chen ported and optimized the IterSol on HPCx system. However, as we always require extra computational power, more work is worthwhile to investigate the IterSol on the upgraded HPCx system which is named phase 2. This project focuses on investigating and optimising the IterSol package on the HPCx phase 2.

The differences between the HPCx phase 2 and phase 1 are the motivation to re-address the general performance and unsolved problems of the IterSol package. The overall optimised performance benefits of phase 2 are above the clock rate ratio between phase 2 and phase 1. Other factors such as communication hardware and cache effects cannot be ignored while comparing the performance of different size of matrices between phase 1 and phase 2. Since cache effects are general factors which are relevant to the sparse matrix-vector multiplication but significantly difficult to investigate, the upgraded communication factor becomes the focus of the project this year.

The decomposition module is always the most interesting portion of the effort for the IterSol package. Dimitris and Xing put plenty of efforts on their dissertations because the overall performance of the solver is significantly affected by the decomposition of the coefficient matrix. The Hierarchical Recursive Spectral Bisection algorithm has been proved that it is the best decomposition algorithm in the IterSol package because it can decrease the demand of both intra-node communication volume and the inter-node communication volume. Both the intra-node communication and the inter-node communication are increasing when the size of the matrix is getting larger. The speedup will tail off when the increase of the inter-node communication volume overrides the increase of the computational power of the machine.

The timing method is critical in the parallel program. When the IterSol is getting complex, the timing method is necessary to be investigated again for our future experiments. Bugs have been found in the timing process. Barrier and upgraded timing mechanisms have been introduced to the solver module of the IterSol package. The required running time of the IterSol on HPCx system for performance studies has been decreased to a new level. The 10,000 iterations of the BiCGSTAB algorithm on HPCx system are not necessary to get the convincing results. Instead, 200 iterations of BiCGSTAB can get satisfactory results which mean that we save a lot of HPCx time after this effort. However, the unstable iteration problem of the 100 first iterations is still
under investigation.

The IterSol package cannot decompose large matrices because of the memory limitation. The conclusion has been drawn that the Chaco decomposition library cannot execute on 64-bit mode which will incur the failure of memory space allocation. Two possible solutions are modification of the decomposition library and simulating the outputs of the Chaco library. The first choice is difficult and introduces risks in the project. The doubling up code which is the latter choice for sparse matrices can simulate the outputs of the Chaco library properly. The diagonal non-zero sparse matrix generator and the doubling up code provide a simple method to generate a random sparse matrix and enlarge its size as required. These codes make the IterSol reliable for huge size matrices instead of changing the decomposition library and algorithm.

For the purpose of benchmark suite of the IterSol on the HPCx system, the doubling up code provides a convenient and efficient method to double up a small matrix to a larger matrix as required. Since the Chaco library has memory limitation to decompose matrices, the doubling up code is important for producing the decomposition results of huge matrices for the solver module of the IterSol. The diagonal sparse matrix generator and the doubling up code provide an extra method to generate sparse matrices and simulate the outputs of Chaco library as required.

The generated matrices which are doubled from the original matrices by the doubling up code have been investigated. We draw the conclusion that the IterSol package can only solve a problem which the size of necessary memory space is less than 268Mbytes. The buffer space for the MPI communication mechanism should be refined carefully because 268Mbytes memory space is easy to over-use for large matrices. The patterns of the doubling up matrices are similar to their smaller source matrices. The larger the input matrix is, the higher speedup we can obtain on the HPCx phase 2.

Overall, this thesis is based on the iterative solver for sparse matrix application. The modification of the UK’s national supercomputer HPCx makes the investigation of the performance and reliability of the IterSol package valuable. Bugs have been found in the original IterSol package and some of them have been investigated and solved. The IterSol package was optimised on the phase 2 system. More work for IterSol package are expected to be performed for elegant iterative solve on supercomputer system in the future.
Chapter 8

Future Work

This MSc project focuses on verifying the original IterSol timing function and producing very large matrices as the input for the solver module of the IterSol package. Three major respective efforts are discussed below for the future development and analysis for improving the functionality of the IterSol package.

First of all, for the purpose of development a reliable and robust scientific sparse matrix analysis package, more efforts for the decomposition module of IterSol are necessary if significantly large matrices are to be fed into the solver module. The doubled up matrices code described in Chapter 5 might be a good solution to feed large enough matrices to the solver module while keeping the Chaco library as the main decomposition algorithm routine in the decomposer module. On the other hand, since the HPCx phase 2 has been upgraded to 32-way mode, it is expected that the decomposition module could decompose input matrices in parallel rather than using only one processor in a Regatta. Therefore, a parallel decomposition library might be worthwhile introducing in the future development of the project. Although the project this year has not implemented parallel decomposition library, a few belief work has been done on an up-to-date parallel decomposition library ParMetis which is discussed in Appendix A.3.

Second, the major memory allocation subroutine smalloc() is limited up to roughly 267Mbytes (based on the test of my malloc test code) on the HPCx system which is still not sufficient enough for large sparse matrices to feed in the current version of the IterSol package. Work has been done for this problem and we get the temporary solution which is acceptable. However, more work is worthy investigating deeply to refine the buffer size because it is relevant the size of matrix problem IterSol can calculate and get a permanent solution.

Third, the performance of IBM’s High Performance Switch is interesting and essential while comparing the communication among different nodes. More tests for investigating the inter-node communication should be done in the future after the switch upgrade and benchmarking on 22nd September 2004.

Furthermore, the unstable timing per iteration of BiCGSTAB algorithm is unclear based on the limited experimental cases. More cases might be helpful for investigating the unstable elapsed time. It is believed that the refinement implementation of the BiCGSTAB algorithm in the IterSol is useful to get more stable timing results as well as make the cache effects of the HPCx sensible in the IterSol.

Finally, no effort has been contributed to the kernel, the sparse matrix-vector multiplication operation (A·v), which has been indicated by Xing last year and shown again in Table 9. However,
better kernel performance can be obtained in the experiments on the HPCx Phase 2. Table 10 shows the comparison of MFLOPS per second and the percentage of the nominal peak performance between the HPCx phase 1 and phase 2. The phase 1 system comprises 3.94% of the peak performance while the upgraded phase 2 system increase this value to 4.38% which is roughly 1.4536 times of the phase 1 performance. As the better processors on the phase 2 system are concerned, the IterSol kernel on phase 2 achieves 11.13% benefits above the clock rate. It might be the cache effect which was discussed by S.Toledo in “Improving the memory-system performance of sparse-matrix vector multiplication” [12], as well as the modification from phase 1 system 8-way to phase 2 system 32-way. Although it is not easy to get high performance in the matrix-vector multiplication, it becomes necessary in the future to optimize the kernel of the IterSol while larger and larger matrices are feed. Works have been performed to address possible approaches to optimise the sparse-matrix vector multiplication code. The probably best toolkit is sparsity developed by Eun-Jin in her Ph.D thesis. [6].

```
for (i=0; i<n; i++) {
    res[i] = 0.0;
    for (j=rowptr[i]; j<rowptr[i+1]; j++) {
        index = colind[j];
        res[i] += values[j] * vec[index];
    }
}
```

Table 9: Matrix vector multiplication ---- The kernel of IterSol

<table>
<thead>
<tr>
<th></th>
<th>MFLOP/sec</th>
<th>Float rate / Peak rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPCx Phase 1</td>
<td>205.1128</td>
<td>3.94%</td>
</tr>
<tr>
<td>HPCx Phase 2</td>
<td>298.1504</td>
<td>4.38%</td>
</tr>
</tbody>
</table>

Table 10: Performance of Matrix-Vector Multiplication

The figures are based on the experiments of cage12.rua matrix (130,228 X 130,228 with 2,032,536 non-zero entries) with 32 processors. Nominal per processor peak performance of HPCx Phase 1 is 5.2GFLOP/sec; Nominal per processor peak performance of HPCx Phase 2 is 6.8GFLOP/sec.
Appendix A

A.1 The Matrix Market format

A wide range of formats can be considered to store the sparse matrix file. In the Basic Linear Algebra
Subprogram (BLAS) Technical Forum standard, there are nine sparse matrix formats
supported. They are Coordinate (COO), Compressed Sparse Row (CSR), Compressed Sparse
Column (CSC) and Sparse Diagonal (DIA) formats for point entry formats where individual
entries are listed in the storage format. Block Coordinate (BCO), Block Sparse Diagonal (BDI),
Block Compressed Sparse Row (BSR), Block Compressed Sparse Column (BSC) and Variable
Block Compressed Sparse Row (VBR) formats are block entry formats in which the sparse
structure of the matrix can be represented as a series of small dense blocks [6]. On the other hand,
Matrix Market Forum also publicises Matrix Market Format, Harwell-Boeing Format and
Coordinate Text File Format which are widely used and simply understood.

The original IterSol can read the Harwell-Boeing Format matrices and Raw Format matrices.
However, since Matrix Market format has a number of advantages which are helpful when
generating matrices by hand, we describe the Matrix Market Format in this section.

The Matrix Market Format offers a convenient way to facilitate the exchange of matrix data. In the
Matrix Market Forum initial specification, two matrix formats are supported. Firstly, Coordinate
Format is suitable for representing general sparse matrix; only the nonzero elements of the matrix
are stored, as well as the coordinates of each nonzero element is provided explicitly. Secondly,
Array Format is basically designed for storage of dense matrix; all elements are provided in a
pre-defined order. The later format for dense matrix is out of the range of research goal of this
project, we merely discuss the coordinate format of the specification.

We show a 5 X 5 matrix with 8 nonzero entries below:

\[
\begin{bmatrix}
1 & 0 & 0 & 6 & 0 \\
0 & 10 & 0 & 0 & 0 \\
0 & 0 & 1.5 & 0 & 0 \\
0 & 25 & 0 & 28 & 33 \\
0 & 0 & 0 & 0 & 12 \\
\end{bmatrix}
\]

%MatrixMarket matrix coordinate real general
%=====================================================================================================
% This ASCII file represents a sparse MxN matrix with L nonzero elements in the following Matrix Market format:
% +-------------------------------------------------------------+
% | %Matrix Market matrix coordinate real general | <--- header line
% | % | <--- +
The first line is the type code explicating it is the Matrix Market format file. (5,5,8) indicates it is a 5 X 5 matrix containing 8 nonzero elements. The following lines explicit the coordinate as well as the corresponding nonzero entry line by line. Obviously, this format is straightforward to understand and store. In addition, the storage memory space of Matrix Market format file is less than that of Harwell-Beoing file which is essential for the purpose of generating a huge synthetic matrix.

**A.2 MatGen**

One of our goals in this project is to read the synthetic matrix at the runtime rather than only read the matrix from the library. For the purpose of generating a huge sparse matrix, one of the simple ways is duplicating the small sparse matrix to a number of sparse matrices in the diagonal direction. However, first of all, we should generate a small sparse matrix via the help of a matrix generator. Matgen is one of our choices for this purpose.

Matgen is a Unix command line toolkit for generating a matrix. It merely supports random sparse matrix by now [7]. What we need to do to generate a small matrix, is that to edit a specific input file for matgen, telling the number of rows and columns, the sparsity structure and varied properties of the generated matrix. The input file of the is shown as follow,
# Matrix type: 0 - random sparse  
1 - random sparse by row #

0

#Symmetry: 0 - non-symmetric  
1 - symmetric)#

0

#Number of rows#  
5

#Number of cols#  
5

#Non-zero density#  
0.4

#Non-zero diagonal (0 - no / 1 - yes)#  
1

#diagonal dominant (0 - no / 1 - yes)#  
1

#diagonal multiplication factor#  
1.0

#minimum#  
0

#maximum#  
1

#force sparse output format (0 - no / 1 - yes)#  
1

---

Figure A.2: Matgen Input file for generating sparse matrix

In order to execute the matgen to obtain the small sparse matrix, we only type the command  
**Matgen** inputfile.input > matrix.mtx

The **matrix.mtx** is the Matrix Market Format file we need. Different matrices can be easily obtained by only adjusting the parameters in the input file. The Matrix Market format is widely used in the sparse matrix realm currently. However, other self-generated sparse matrix method is also described in Chapter 4.

### A.3 ParMETIS

ParMETIS is an MPI-based library extended from the well-known serial package METIS. It is particularly suitable for parallel computations and large-scale numerical simulations. A wide range of algorithms have been implemented in the ParMETIS library for partitioning unstructured graphs as well as computing fill-reducing ordering of sparse matrices [8]. As one of excellent parallel libraries for numerical simulations including large unstructured meshes, ParMETIS dramatically reduces the decomposition cost via minimizing the communication among processors
and loading balance as much as possible.

ParMETIS is a multi-functional graph partitioning library. The project this year will not beyond the scope of the unstructured graph partitioning. The routine ParMETIS_V3_PartKway takes a graph and computes a k-way partitioning without any assumption how the graph is initially distributed among the available processors. The advantage of implementing ParMETIS_V3_PartKway is that it can minimize the number of edges that are cut by the partitioning [8]. The details of the implementation and the optimized performance are discussed in Chapter 5.

However, the application of the ParMETIS is too complicated to address within the time limitation of this project, further implementation and investigation are probably worthwhile in the future.
Appendix B

B.1 Generation Synthetic Matrix

Generating synthetic sparse matrix is essential in this project because all matrices used in last two years are based on the collections in computational and engineering research field. Most matrices from these collections have complicated sparse structure and reuse and locality of the data in these matrices are not predicted and difficult to be optimized. As the test cases of the IterSol on large scale clustered computer, synthetic sparse matrix is a good choice as an input of the iterative solver. Two methods described below are able to generate sparse matrices which are needed in the experiments.

B.1.1 Random matrix generator

There are plenty of tools and libraries provided for generating a random matrix on the internet, such as RANDOM which is a numerical linear algebra random sparse matrix generator written in Java and output as Matrix Market format and XLATMR which is written in Fortran while the outputs can be various formats [5]. However, the toolkits and libraries for generating sparse matrix are too complicated to be the test cases in our experiments. The diagonal sparse matrix is simple and controllable in our experiments as well as suitable for decomposition evenly. An example is shown in figure XX.

The random matrix generator in this project is based on the ANSI C random number function srand(). I simply use the current time as the seed of the random function to get the real random number matrix. However, const seed can be used when comparison of the specific sparse matrix with different conditions such as decomposition algorithms or iteration times in solver module of IterSol. The density of the generated sparse matrix can be controlled by simply modifying the bandwidth of the diagonal non-zero entries.

```c
srand((unsigned)time(NULL));

printf("RAND_MAX = %d \n", RAND_MAX);

/* Allocate memory for the original array */
MALLOC(rand_number, sizeof(double *) * ncol);
for(i=0;i<nrow;i++){
    MALLOC(rand_number[i], sizeof(double) * nrow);
}

/* Create the original matrix */
for(i=0; i<nrow; i++){
    for(j=0; j<ncol; j++){
```
if (abs(i-j) <= band){
    rand_number[i][j] = (double)rand()/RAND_MAX;
    if(rand_number[i][j]<0.05){
        rand_number[i][j] = 0.05;
        nonzero++;
    }
    else{
        rand_number[i][j] = 0.0;
    }
}

Figure B.1: Random sparse matrix generator kernel

The generated random sparse matrix method used here is efficient to generate millions by millions large matrices. The elapsed time to generate a matrix with 1,000,000 rows and 1,000,000 and 199,000,000 non-zero entries is roughly 40 seconds which is satisfactory time for generating this kind of big matrices. Bearing in mind that we can customize the rows, column and bandwidth of the original matrix, how to give the values of these parameters is important for the data reuse in the final generated matrix. As the data reuse and data locality effect the performance of the matrix-vector multiplication in the solver module, it is believed that small values of the number of rows and columns can produce more repeated numbers in the matrix so that the performance of the matrix-vector multiplication might be faster. However, in the test cases we use in this project for the IterSol with self-generated sparse matrix, we originally customized the row and column 100 X 100 respectively and the bandwidth = 5. Duplicating the original 100 X 100 small sparse matrix for 10,000 times, we can obtain a millions by millions large sparse matrix with relatively simple sparsity structure.

In addition to generate the CSR format matrix, the random matrix generator can store the generated matrix as Matrix Market format. This format is more straightforward than the CSR format although the same dilemma as self-generated CSR format matrix is the diagonal non-zero values are repeated periodically.

B.1.2 Using Matgen to generate a sparse matrix

B.2 Random Sparse Matrices

As we have specified in Chapter 2, the matgen can only generate sparse matrix. For this type of matrices, users should indicate the non-zero density of the desired sparse matrices. The matgen generator can produce non-zero elements until the specified density is obtained. Unlike the
self-generated random matrices generator above, this generator takes a relatively long time to
generate a matrix because the program does a strict validation to avoid the periodical repeated
non-zero elements in the matrix. The input file for this type of random sparse matrices has been
indicated in Chapter 2.2

B.2.1 Random Sparse Matrices by Row

The time consuming generated process of generating sparse matrices by matgen is obviously not
satisfied in some time-intensive cases. The random sparse matrix by row is an alternative method
to generate a same size matrix spending less time. For each row, the checking validation to avoid
periodical repeated non-zero values is different from the previous method specified in 4.2.1. The
program generates a random number of non-zero values and checks the repeated elements line by
line, which is much faster than the method checking the validation at the final section of the
program. The drawback of this type is that it cannot implement the symmetric sparse matrix [7].
The belief input file is shown in figure 3.

```
#Type#
    1
#Number of rows#
   100
#Number of cols#
   100
#max non-zeros per row#
    20
#min non-zeros per row#
     5
#Non-zero diagonal (0 - no / 1 - yes)#
    1
#diagonal dominant (0 - no / 1 - yes)#
    1
#diagonal multiplication factor#
    1.3
#minimum#
   0.1
#maximum#
   2.0
#force sparse output format (0 - no / 1 - yes)#
    1
```

Figure B.2: matgen sample input file for random sparse matrix by row type
Appendix C

C.1 User Guide of self sparse matrix generator

The sparse matrix generator in this project is very straightforward to use. It generates a diagonal non-zero sparse matrix with specific number of rows and columns. Bandwidth is the width of the diagonal of the generated matrix. It is a command line sparse matrix generator tested in Sun Solaris Operating System 5.8 and the usage is shown below.

Usage: ./gen_matrix  [row]  [column]  [multi]  [bandwidth]

Row and column is the number of row and column of the generated matrix respectively. Multi stands for how many times the small matrix will be duplicated to generate the destination matrix. Bandwidth is the width of the diagonal of the generated matrix.

The code has been tested in Sun Fire15K with Solaris 5.8 and the elapsed time for generating a 1,000,000 by 1,000,000 with bandwidth = 5 sparse matrix is roughly 30 seconds.

C.2 The doubling up code usage for the specific matrix

The doubling up code for simulating the output of Chaco library is also easy to use on Solaris 5.8. Both the original map file and the CSR format file of the original matrix are required as the parameters in the command line. We need to specify the name of the output map file and the CSR format of the destination matrix. Note that the number of processors in the command line is the number of processors of the input map file used rather than the output map file.

duplicate [input map file] [input csr file] [output map file] [output csr file] [number of process]

The input map file and the CSR format of the original matrix can be obtained in the HPCx system. However, users do not need to execute the doubling up code on HPCx. The doubling up code in this project is implemented and executed on Sun Fire15K system.
Appendix D

D.1 Decomposition Algorithms in Chaco Package

The decomposer module of IterSol implements one of the five available algorithms to decompose the input matrix: Recursive Spectral Bisection (RSB), Hierarchical Recursive Spectral Bisection, Block-Striped, Linear, Random.

Recursive Spectral Bisection (RSB) and Hierarchical Recursive Spectral Bisection algorithms are both heuristic algorithms which are explained deeply in [9]. The difference between the RSB algorithm and the Hierarchical RSB algorithm is the Hierarchical RSB is multi-process of the RSB which means that it uses the RSB algorithm twice for the purpose of minimising the expensive inter-node communication. Two stages of decomposition would be implemented in Hierarchical RSB algorithm. The first stage decomposes across nodes to obtain the minimal inter-node communications. The second stage is straightforward to take these large domains and decompose them independently within a node. Users can customise the number of nodes and the number of processors in each node can be deployed via the command-line. The terminology RSBM + RSBN means that M nodes (stage 1) and N processors per node (stage 2) for a final decomposition into M * N domains. This context is used widely in this dissertation.

The Block-Striped decomposition algorithm does not need Chaco. It will simply allocate \( nrow/p \) lines to every processors ( \( nrow \) is the number of rows in the matrix and \( p \) is the number of processors).

The Linear decomposition algorithm is a weighted version of Block-Striped decomposition, which a consecutive number of rows is allocated to each processor. However, Chaco package can perform a sort of refinement of the produced partitions, trying to increase the number of rows that only require local data. This feature is important for the IterSol package by setting the proper parameters of Chaco always to true, resulting in a decomposition different from the Block-Striped.

Random decomposition algorithm is only supported for timing purpose [1]. It is expected to produce well-balanced partitions but with a very intensive interconnection pattern, even when the internal vertices parameters are switched on.
Bibliography


[15] The doubling-up approach was chosen after very helpful correspondence with Prof Iain Duff of the Rutherford Appleton Laboratory.