THE SPEEDUP OF DISTRIBUTED ITERATIVE SOLUTION OF SYSTEMS OF LINEAR EQUATIONS

T. DILANI P. PERERA

The speedup of distributed iterative solution of systems of linear equations

by

© T.Dilani P.Perera

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

April 2006

St. John's Newfoundland

Your file Votre référence
ISBN: 978-0-494-19387-7
Our file Notre référence
ISBN: 978-0-494-19387-7

Abstract

The main objective of this research is to study the performance of distributed solvers for large sparse systems of linear equations. The relationship between the speedup and the number of processors is the main characteristic of distributed computing for this study.

Systems of linear equations can be solved using either direct or iterative methods. For large and sparse systems of equations, iterative methods are often more attractive than direct methods. In distributed implementations of iterative solvers, the number of operations are equally divided among the available processors with the intention that all the sections are processed concurrently (i.e., by different processors).

The iterative approach repeats the following sequence of steps until the required convergence condition is satisfied:

1. distribute the current approximation to all the processors,
2. determine a new approximation to the solution,
3. collect parts of the new approximation and check the convergence conditions.

The implementation is based on the message passing paradigm, which is used widely on certain classes of multiprocessor machines, especially systems with distributed memory.

It is expected that this study will determine the optimal number of processors for distributed linear solvers.

Acknowledgements

I am grateful to acknowledge the valuable assistance of several persons who generously contributed their time to make my study at Memorial University an enormous success.

First of all I would like to thank to my supervisor, Dr. Wlodek Zubrek, for his kind support, guidance and encouragement throughout my program. He always made valuable suggestions regarding my research work and the presentation of the material of this thesis.

I thank my family for their moral support during my studies, especially my sister and brother-in-law, Aruni and Pujitha, for helping me to find this opportunity to study at Memorial University of Newfoundland.

Many thanks to Dr. Banzhaf Wolfgang, Dr. George Miminis, Ms. Malgosia Zuberek, Nolan White, Donald Craig and Ulf Schunemann for their great support in various ways. I would also like to thank all the academic and non-academic staff members in the department of Computer Science for their kind support.

It is a pleasure to express my gratitude to the Natural Sciences and Engineering Research Council of Canada and School of Graduate Studies at Memorial university of Newfoundland for financial support.

Finally, thanks to all my friends at Memorial for making my stay in St. John's enjoyable and unforgettable.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1

2 Systems of linear equations and their solution methods 6

2.1 Sparse matrices 7

2.2 Representation of sparse matrices 8

2.2.1 Coordinate Storage (CS) format 9

2.2.2 Compressed Sparse Row format 9

2.2.3 Compressed Sparse Column format 9

2.2.4 Block Compressed Sparse Row format 9

2.3 Solving systems of linear equations 10

2.3.1 Direct methods 10

2.3.2 Iterative methods 12

2.4 Convergence criteria 16

2.5 Distributed iterative solvers 18

2.6 Performance of distributed solvers 19

3 Implementation **24**

3.1 Representation of sparse matrices 24

3.2 Solution of the systems of linear equations 25

3.2.1 Workload distribution 26

3.2.2 Distributed Iterations 27

3.3 Algorithms 29

3.3.1 Serial algorithm 29

3.3.2 Distributed algorithm 30

3.4 Program Implementation 32

4 Experimental results **34**

4.1 Data 34

4.2 Results and Discussion 36

4.2.1 Speedup of distributed solvers 37

4.2.2 Speedup and the density of data 38

4.2.3 Speedup and the size of the systems of equations 43

4.3 Number of Iterations 45

5 Conclusions **49**

List of Figures

3.1 A 5x5 sparse matrix **A**. 25

3.2 Representation of the matrix form in Figure 3.1 using the sparse format. 25

4.1 Sparsity patterns 36

4.2 Speedup plots for System (1) 37

4.3 Speedup plots for System (2) 38

4.4 Speedup plots for System (3) 39

4.5 Speedup plots for System (4) 39

4.6 Speedup plots for System (5) 40

4.7 Speedup plots for System (6) 40

4.8 The ratio of speedup for System (2) and System (1), as a function of the number of processors. 41

4.9 The ratio of speedup for System (4) and System (3), as a function of the number of processors. 42

4.10 The ratio of speedup for System (6) and System (5), as a function of the number of processors. 42

4.11 Percentage increase of speedup values for System (2) 43

4.12 Percentage increase of speedup values for System (6) 44
4.13 Speedup plot for System (2), when the number of processors is 4, 8, 20 and 32 44
4.14 Number of iterations as a function of number of processors for System (1) 46
4.15 Number of iterations as a function of number of processors for System (3) 46
4.16 The number of iterations of distributed iterative solvers of 100 equations for System (5) 48

Chapter 1

Introduction

It is believed that the performance of processors, that has been doubling every eighteen months (the so called Moore's law [21]), will be improving more slowly in the coming years as the shrinking dimensions of basic electronic elements are approaching their physical limits [21]. Therefore, further significant improvements of computational performance are expected by using parallel and distributed computing [21] rather than more powerful uniprocessors. Also current computer technology favors multiprocessor systems because they are more economical [14]. On one hand, research concentrates on multiprocessor systems implemented on a single chip [10], on the other – an increasing number of large scale applications is migrating to distributed systems, with SETI@home [3] and Climate Prediction [2] projects as just two more popular examples.

Distributed systems can have many different forms which include clusters of workstations (COW) and networks of workstations (NOW) [1]. Such systems are often considered as less expensive and more easily available alternatives to paral-

lel systems [30]. A recent survey of most powerful supercomputing systems shows that seven out of ten most powerful systems are clusters, which indicates that the cluster architecture has a top place among most powerful computers [24]. For the purpose of this project, any collection of processors (i.e., PCs or workstations) connected by a communication medium (e.g., LAN, Internet or a high-performance interconnecting network) is considered a distributed system. The increasing popularity of such systems is due to two factors:

1. Easily available, inexpensive but quite powerful PCs and workstations as well as high-bandwidth communication networks, and

2. Communication libraries (such as MPI [25],[20] and PVM [15]) which provide high-level operations needed for communication among the processors.

In distributed applications, the total workload is divided among the processors of the system. One of the main performance characteristics of any distributed application is its speedup [35], which is usually defined as the ratio of the application's execution time on a single processor, $T(1)$, to the execution time of the same workload on a system composed of N processors, $T(N)$:

$$S(N) = \frac{T(1)}{T(N).}$$

The speedup depends upon a number of factors which include the number of processors and their performances, the communications between the processors, the algorithm used for the distribution of the workload, etc. Some of these factors may be difficult to take into account when estimating the speedup of a distributed application. Therefore, in many cases, a simplified analysis is used to characterize

the steady-state behavior of an application. This simplified analysis is based on a number of assumptions, such as uniform distribution of workload among the processors, constant communication times, and so on.

Although there are some spectacular examples of distributed applications (e.g., SETI@home project), the number of practical applications of distributed computing is still somewhat limited [11]. It appears that in some cases, the migration to a distributed platform is quite straightforward, and the speedup is almost a linear function of the number of processors used, while in other cases, the straightforward distribution of the workload among the processors of the system results in a poor speedup, and an increased number of processors can even slow down the execution of an application.

The goal of this project is to study the performance of distributed solvers for large and sparse systems of linear equations.The relationship between the speedup and the number of processors is the main characteristic of distributed computing for this study.

Large sparse systems of linear equations arise in many areas. Below is a list of areas which are represented in the Harwell-Boeing sparse matrix collection [10]:

Acoustic scattering (4),

Chemical engineering (6),

Laser optics (16),

Chemical kinetics (14),

Petroleum engineering (19),

Economics (11),

Structural engineering (95),

Electric power (18),

Electrical engineering (50),

Survey data (11),

Structural engineering (95).

Also, most of the large matrices arising in the solution of ordinary and partial differential equations are sparse [22].

In many applications, the solution of linear systems of equations is the most computationally intensive step. Time, speed, efficient use of available storage and the accuracy of the results are major factors to be considered when solving such large systems.

Two main classes of methods for solving systems of linear equations are known as direct methods and iterative methods. Gaussian elimination is the most popular direct method. Two types of iterative methods include stationary iterative methods and non-stationary iterative methods.

Iterative methods are often superior to direct methods if the matrix is large and sparse [4],[19],[18]. Iterative methods preserve the sparsity of the system of equations during computations, so memory requirements are low compared to direct methods.

Iterative methods begin with an initial approximation of the solution and calculate the next approximation based on the current one. This process is continued

until convergence criteria are satisfied. Sometimes special techniques are needed to reduce the number of iterations needed to reach the solution.

The remaining chapters of this thesis are organized as follows. Chapter 2 provides a brief overview of sparse matrices and their storage methods, methods of solving systems of linear equations and convergence criteria. Chapter 3 describes the implementation including algorithms used in both the serial and the distributed versions of the program. Chapter 4 presents some experimental results. Finally, Chapter 5 contains the discussion and conclusions.

Chapter 2

Systems of linear equations and their solution methods

A system of simultaneous linear algebraic equations is usually expressed in a general form as $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a coefficient matrix of size $N \times N$ with elements $\mathbf{a_{i,j}}$, $i = 1, 2, ..., N$; $j = 1, 2, ..., N$; $\mathbf{b}$ is a vector of size N with elements $\mathbf{b_i}$, $i = 1, 2, ..., N$; and $\mathbf{x}$ is a vector of unknowns of size N with elements $\mathbf{x_i}$, $i = 1, 2, ..., N$.

In many applications, the coefficient matrix $\mathbf{A}$ contains many zero elements. In large Markov chains, each node is often connected to only a few other nodes; if such a chain is represented by a coefficient matrix, each row contains only a few non-zero elements. In electrical power systems, the ratio between the number of branches and the number of nodes is about 1.5 or less, which means only 0.1% elements in the coefficient matrix are non-zero elements [12]. Matrices, which contain mostly zero elements, are called sparse matrices. In many practical applications, the number of non-zero elements per row in a sparse matrix is 5 to 10. For example,

according to [4], the number of non-zero entries in the coefficient matrix of linear systems that arise in elliptic partial differential equation problems in two and three dimensions is proportional to N, such as $5N$ or $7N$, which means that each row of **A** contains only 5 or 7 non-zero elements.

2.1 Sparse matrices

There is no precise definition of sparse matrices. A matrix that consists of a high proportion of zero elements is considered a sparse matrix. A few definitions of sparse matrices are as follows.

Definition [22] A sparse matrix is one in which most of the element are zero.

Definition [12] A matrix is said to be sparse if it has sufficiently many zero elements for it to be worthwhile to use special techniques to avoid storing or operating with the zeros.

Definition [4] A matrix is sparse, if we can save space and/or computer time, whichever is more important, by employing methods that utilize sparsity.

The last definition is an example of a good operational definition. Sparsity of a matrix **A** is determined by the number of non-zero elements:

$$d(\mathbf{A}) = \frac{Z_N(\mathbf{A})}{N * N},$$

where $Z_N(\mathbf{A})$ is the number of non-zero elements of matrix **A**. There is no borderline to divide matrices into sparse and dense matrices. Sparse matrices should save space and computational time in practical applications. In sparse matrices the

number of non-zero entries is small compared to the total number of entries. The matrices associated with a large class of man-made systems are sparse [34].

2.2 Representation of sparse matrices

Any efficient representation of sparse matrices stores only the non-zero elements, making a significant saving of required storage. A number of different schemes can be used to represent and process large sparse matrices more economically and effectively than in the dense case, by eliminating unnecessary arithmetic operations on zero elements. Generally, sparse techniques are appropriate for matrices with more than 80% zeros [12].

There are a large variety of storage schemes which can be used to represent sparse matrices. Most of these schemes make use of two main storage components [23]:

1. Storing either the non-zero elements or an area of the matrix which includes all of the non-zero elements. This is usually a one-dimensional array, which will be called a primary array.

2. A means of recognizing which elements of the matrix are stored in the primary array. This usually takes the form of one or more one-dimensional arrays of integer identifiers, known as the secondary array.

Popular methods of sparse matrix representation include Coordinate Storage (CS) format, Compressed Sparse Row (CSR) format, Block Compressed Sparse Row (BCSR) format, Skyline Storage (SKS), Jagged Diagonal Storage (JDS) and

Compressed Diagonal Storage (CDS). The space requirement for a full matrix is $O(N^2)$ where N is the matrix order. The space requirement for compact sparse matrices is $O(Z_N(\mathbf{A}))$ where $Z_N(\mathbf{A})$ is the number of non-zero elements. $O(Z_N(\mathbf{A}))$ becomes $O(N^2)$ in the worst case, i.e., the case of dense matrices.

2.2.1 Coordinate Storage (CS) format

This method uses one array to store the non-zero entries in any order (row order or column order) and two other arrays to store the row and column indices of the non-zero elements in the same order as the non-zero elements.

2.2.2 Compressed Sparse Row format

Non-zero elements of the matrix are stored in one array in row order. A second array is used to store the column indices of the non-zero elements and a third array is used to store pointers to the beginning of each row of non-zero elements.

2.2.3 Compressed Sparse Column format

This format is similar to the compressed sparse row format, but instead of column indices, it stores the row indices of the non-zero elements in the second array and a pointer to the beginning of each column of non-zero elements in the third array.

2.2.4 Block Compressed Sparse Row format

The non-zero blocks belonging to successive blocks of the matrix are stored in one rectangular array in row-wise fashion. Column indices of elements in all non-zero

blocks are stored in a second array and the pointers to the beginning of each block row in the first and the second array are stored in a third array.

It should be observed that if the data structures used for sparse matrices are used for representation of dense matrices, more storage would be actually needed than in the dense case, so the density of the matrix needs to be checked carefully.

2.3 Solving systems of linear equations

Two main classes of methods used to solve systems of linear equations are known as direct methods and iterative methods. Direct methods include Gaussian elimination, Gauss-Jordan elimination, LU-decomposition, Cramer's rule, etc. Examples of iterative techniques include the Jacobi method, Gauss-Siedel method, Relaxation methods, Krylov subspace methods, and so on [28].

2.3.1 Direct methods

Direct methods are general and robust because they do not assume special properties (other than linear independence) of systems of equations. These methods are based on elimination methods and are characterized by a fixed number of operations that yield the solution. Direct elimination methods are generally used when the number of equations is rather small (100 or less) and most of the coefficients in the equations are non-zero [22].

The Gaussian elimination method is the most commonly used direct method for obtaining the solution of a system of linear equations. The main objective is to convert the original system into an equivalent triangular form. Methods such as

the Gauss-Jordan method, LU decomposition method, matrix inverse method and Thomas algorithm are extensions of the Gauss elimination method. The number of required arithmetic operations is approximately $N^3/3 + N^2 - N/3$ [22]. Techniques such as using more significant figures during the computations, using partial or complete pivoting during the computations, using scaling during the calculations and using an error correction method after finding the solution can be used to improve the solution of the Gauss elimination method.

In the Gauss-Jordan elimination method, the elimination is done in such a way that off diagonal elements in both upper and lower parts of the coefficient matrix are set to zero. The method transforms the matrix **A** into the identity matrix so that the transformed **b** vector is the solution vector. The number of arithmetic operations in this method is $N^3/2 + N^2 - N/2$, which is about 50 percent higher than the Gaussian elimination method [22]. Theoretically, the Gauss elimination method and Gauss-Jordan method can be applied to even bigger systems of equations, but in practice accuracy of the solution suffers from round-off errors [17] when $N > 50$ and hence the solution may not be accurate.

The LU decomposition method decomposes the matrix **A** into a product of its lower triangular and upper triangular matrices, **L** and **U** respectively (**A** = **LU**). When the unity elements are on the main diagonal of **L**, the method is called the Doolittle method, and when the unity elements are on the major diagonal of **U**, the method is called the Crout method [22]. The solution is done in two steps, using the so-called forward and backward substitution **Ly** = **b** and **Ux** = **y**.

The widely used Thomas algorithm is a special algorithm designed for systems which have the tri-diagonal structure arising naturally in many applications.

One such application is obtaining the numerical solution of differential equations by implicit methods. This is a widely used algorithm in a large number of applications. Although Cramer's rule is not an elimination method, it belongs to direct methods. This method is not used for larger systems of equations since it requires evaluation of numerous determinants which is practically impossible and thus highly inefficient.

For sparse matrices, direct methods lead to fill-ins during the computations. Since one fill-in entry can cause new fill-in entries, there is a tendency to increase matrix density during the solution process. Sometimes fill-ins can be eliminated by reordering the matrix. In certain cases, however, it is impossible to decrease the amount of fill-ins even using matrix re-ordering methods. Intelligent use of data structures and special pivoting techniques to minimize the number of fill-ins are two strategies to improve the direct methods applied to sparse linear systems.

2.3.2 Iterative methods

An iterative method uses a repetitive procedure to find subsequent approximations to the solution $\hat{\mathbf{x}}$. Starting with an initial approximation, $\mathbf{x}^{(0)}$, it determines a sequence of approximations $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ...,$\mathbf{x}^{(k)}$ such that:

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = \hat{\mathbf{x}}.$$

For iterative methods, different initial approximations might result in different solutions or no solution at all. Usually, if the initial approximation is close to the solution, the convergence occurs after a few iteration steps [26]. According to [4], Gauss stated "I recommend this modus operandi. You will hardly eliminate di-

rectly anymore, at least not when you have more than two unknowns. The indirect method can be pursued while half asleep or while thinking about other things." Also the iterative approach does not suffer from the fill-in effects and often produces more accurate results. However, a drawback of iterative methods is that the rate of convergence can be low or the method can diverge [36],[10].

Two basic types of iterative methods are known as stationary iterative methods [33] and non-stationary iterative methods [27]. According to [27], stationary methods are older, simpler to understand and implement, but usually not as effective as the non-stationary iterative methods. Iterative methods perform the same operations on the current iteration vectors in each iteration step. Stationary iterative methods can be represented as:

$$\mathbf{x}^{(\mathbf{k})} = f(\mathbf{A}, \mathbf{b}, \mathbf{x}^{(\mathbf{k-1})}).$$

The four main stationary iterative methods are the Jacobi method, the Gauss-Seidel method, the Successive Over Relaxation (SOR) method and the Symmetric Successive Over Relaxation (SSOR) method [27].

The Jacobi method is also well known as the method of simultaneous displacements. The order in which the equations are evaluated is irrelevant, so updates can be done concurrently:

$$\mathbf{x}_{\mathbf{i}}^{(\mathbf{k+1})} = \frac{1}{\mathbf{a_{ii}}} \left(\mathbf{b_i} - \sum_{j=1, j \neq i}^{N} \mathbf{a_{ij}} \mathbf{x}_{\mathbf{j}}^{(\mathbf{k})} \right).$$

In the Jacobi method, both the current and previous approximations to $\mathbf{x}$ need to be stored since the values $\mathbf{x}_{\mathbf{i}}^{(\mathbf{k+1})}$ depends on the previous approximation $\mathbf{x}_{\mathbf{i}}^{(\mathbf{k})}$. The sufficient condition for the convergence of the Jacobi method is $|\mathbf{a_{ii}}| > \sum_{j=1, j \neq i}^{n} |\mathbf{a_{ij}}|$, which corresponds to the case when equations are diagonally dominant.

The Gauss-Seidel method, also known as the method of successive iterations, is better than the Jacobi method in terms of storage requirements and the rate of convergence. It uses the available values of $\mathbf{x^{(k+1)}}$ to find the other values to $\mathbf{x^{(k+1)}}$:

$$\mathbf{x_i^{(k+1)}} = \frac{1}{\mathbf{a_{ii}}}\left(\mathbf{b_i} - \sum_{j=1}^{i-1} \mathbf{a_{ij}x_j^{(k+1)}} - \sum_{j=i+1}^{N} \mathbf{a_{ij}x_j^{(k)}}\right).$$

The method converges to a correct solution if the system is diagonally dominant and even in many cases when the system is weakly diagonally dominant [26]. According to [27], larger diagonally dominant systems converge usually twice as fast as for the Jacobi method.

An improved version of the Gauss-Seidel method is known as the Relaxation method. The method allows selecting the finest equation for $\mathbf{x^{(k)}}$ to achieve a faster convergence [26]. This is derived from the Gauss-Seidel method by introducing the extrapolation parameter ω. This method converges faster than Gauss-Seidel by an order of magnitude [4].

The relaxation iterative process is described by the equation:

$$\mathbf{x_i^{(k+1)}} = \frac{1}{\mathbf{a_{ii}}}\left(\mathbf{b_i} - \sum_{j=1}^{i-1} \mathbf{a_{ij}x_j^{(k+1)}} - \sum_{j=i+1}^{N} \mathbf{a_{ij}x_j^{(k)}}\right) + (1-\omega)\mathbf{x_i^{(k)}};$$

where i = 1,2,...,N ; k = 0,1,2,....

When $\omega = 1$, this method becomes the Gauss-Seidel method. When $0 < \omega < 1$, the method is known as the Successive Under-Relaxation method which is widely used for the solution of nonlinear algebraic equations. When $1 < \omega < 2$, it is called the Successive Over-Relaxation method. The optimum value of the over-relaxation factor depends on the size of the system of equations, the nature of the equations such as the strength of the diagonal dominance, and the structure of the

coefficient matrix. Iterative methods diverge if $\omega = 2.0$ [22]. The optimum value of the relaxation factor, ω, which yields the fastest convergence, is not known and it is usually determined by a trial and error process [26].

Successive over-relaxation methods and their variants have been extremely popular in areas of nuclear reaction diffusion, oil reservoir modeling and weather prediction, and were the methods of choice in computer codes for large practical problems until the emergence of more powerful techniques such as Krylov methods [4].

Although the analysis of non-stationary iterative methods is more complex than stationary methods, they are highly effective. These methods consist of iteration dependent coefficients, which means that the computations involve information that changes at each iteration step [10].

Krylov subspace solution methods come under the non-stationary methods [10]. In these methods, the solution of $\mathbf{Ax} = \mathbf{b}$ starts with an initial approximation $\mathbf{x}^{(0)}$ and, at each iterative step k, generates an approximate solution $\mathbf{x}^{(k)}$ from the linear variety $\mathbf{x}^{(0)} + \mathbf{span}\{\mathbf{r}^{(0)}, \mathbf{Ar}^{(0)}, ..., \mathbf{A}^{k-1}\mathbf{r}^{(0)}\}$, where $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{Ax}^{(0)}$ is the initial residual [16]. Based on this, the four projection-type approaches used to find the suitable approximation to solution $\mathbf{x}$ of $\mathbf{Ax} = \mathbf{b}$ are the Ritz-Galerkin approach, Minimum Residual approach, Petrov-Galerkin approach and the Minimum Error approach [10].

The Ritz-Galerkin approach includes well-known popular techniques such as Conjugate Gradient method (CG), Lanczos method, Full Orthogonal Method (FOM) and Generalized Conjugate Gradient method (GENCG). Generalized Minimum Residual method (GMRES), and Minimize Residual (MINRES) method come

under Minimum Residual approach. The Petro-Galekin approach includes Bi-Conjugate Gradient method (Bi-CG) and Quasi-Minimal Residual (QMR) method. Symmetric Matrix LQ method (SYMMLQ) and Generalized Minimal Error method (GMERR) belong to the Minimum Error approach. The most recent developments include hybrids of these methods such as Conjugate Gradient Square method (CGS), Bi-Conjugate Gradient Stabilized method (Bi-CGSTAB) and so on [10],[8].

The rate of convergence of the Krylov subspace methods depends on the spectral properties of the given matrix, commonly unknown to the user [10]. Preconditioners chosen in various ways improve the convergence of the iteration method by transforming the matrix $\mathbf{A}$. The main idea is to create a preconditioning matrix $\mathbf{K}$ which is a good approximation to $\mathbf{A}$ in some sense. The cost of the construction of $\mathbf{K}$ is not prohibitive and the system $\mathbf{Kx} = \mathbf{b}$ is much easier to solve than the original system of equations [10]. Using a trial and error method is the best way to choose the best preconditioner [10]. Solving the preconditioned system using a Krylov subspace method will create subspaces different from the original system. The aim of the preconditioning is that the chosen iterative methods will converge much faster [10],[13]. Left-preconditioning, right-preconditioning and two-sided preconditioning are three different implementations of the preconditioning method [10],[6].

2.4 Convergence criteria

A convergence criterion (accuracy criterion) is used to terminate the iterations in the iterative process of solving linear equations. When the method produces in-

significant changes in the solution vector, it is assumed that the desired accuracy is reached and the iteration ends. The criterion should be robust and consistent, otherwise the convergence criteria can either lead to poor results or to excessive computational times.

Different criteria can be used for the convergence. Since the accuracy can be measured in terms of the error, the absolute error or the relative error can be used as a criterion of convergence. Absolute error is the difference between the approximate value and the exact value, while the relative error is the ratio of the absolute error and the exact value. And since the exact value is normally not known, that ablute error is approximated in practice by the difference of the results obtained in two consecutive iterations. Choosing the relative error as the criterion is usually preferable.

There are no universal guidelines as to how to choose the best convergence criterion, and the trial and error method is often the best way to get good results. The dominance of the diagonal coefficients, the method of iteration, the initial solution vector and the criteria for convergence are factors that affect the number of required iterations [22]. The convergence criteria depend on the problem to be solved.

Traditionally used convergence criteria, which continue the computations until some norm of the residual vector or the scaled residual vector becomes smaller than a quantity prescribed in advance, are not sufficient to ensure robust and reliable control of the error arising when solving systems of linear algebraic equations [9]. Other stopping criteria which are based on principles such as an attempt to evaluate the rate of convergence and use it in the stopping criteria or a check of the

variability of some important parameters, which are calculated and used at each iteration of the iterative process, can be used instead [9]. It can be shown that the convergence of the iterative process is discouraged when these parameters vary too much from one iteration to another; the parameters increase or decrease too quickly [9].

2.5 Distributed iterative solvers

In distributed computing [14],[30], the main objective is to improve the performance of the solvers by minimizing its total execution time. The overall performance of the system is not limited to computation time, it is composed of computation time, waiting times and communication times. Therefore, to improve the performance, one must consider minimizing all the above mentioned times. In the ideal case, the use of N processors reduces the execution time N times, but such an ideal case is rarely encountered in practice. Some important factors to be considered when developing distributed programs are minimizing the communication cost, load balancing and the overlapping of communication and computations.

During the execution of a distributed program, each processor should be assigned the same workload to reduce the idle times. In the distributed implementation of iterative linear solvers, workload is divided among the processors by dividing the number of operations into a number of equal groups, each group assigned to one of processors. Processing of the groups is done concurrently by the processors. During each iteration, the following sequence of steps is executed until the given convergence criterion is satisfied [36]:

1. Distribute the current approximation of the solution to all the processors.

2. Determine parts of the next approximation of the solution by all processors.

3. Collect parts of the new approximation and check the convergence.

One of processors, called the root processor, controls the distributed iterative process. It initially sends the data values of matrix **A** and the vector **b**, as well as the initial approximation to the solution $\mathbf{x}^{(0)}$ to all other processors. Once the data is received, each of the processors starts the computation of its part of the vector **x**. Once the computation is finished, each processor sends the results back to the root processor. The root processor waits until all the computed segments of the vector **x** arrive from all the other processors, and then performs the necessary computations and continues the iteration until the convergence criterion is satisfied.

2.6 Performance of distributed solvers

Multiprocessor performance of any system can be measured in terms of the elapsed time, speedup and efficiency. There are many relationships relating these performance measures.

The speedup is one of the most intuitive and important metrics in performance analysis. The speedup of an N-processor system is usually defined as [35]:

$$S(N) = \frac{T(1)}{T(N)},$$

where $T(1)$ is the execution time using one processor (or uni-processor) system, and $T(N)$ is the execution time of the same workload on an N-processor system.

For distributed iterative solvers, the execution time $T(N)$ for one iteration step can be expressed in terms of simpler operations, i.e.:

T_b: time to broadcast the current approximation to the solution to all processors,

T_c: time for computation using only one processor,

T_r: time to send results from one processor back to the root processor.

It is assumed that T_b does not depend on the number of processors. T_r is the time required to transfer the results from a single processor to the root processor which performs the convergence check. This resending operation is performed in a sequential manner, so the execution time for this total transfer is equal to $(N-1)T_r$. Although T_r depends upon N, the dependence is not very strong [31], and is ignored here.

Consequently, $T(N) = T_b + \frac{T_c}{N} + (N-1)T_r$, and then the speedup is:

$$S(N) = \frac{T_c}{T_b + \frac{T_c}{N} + (N-1)T_r}.$$

Assuming that $T_b = T_r$, the speedup becomes:

$$S(N) = \frac{T_c}{NT_r + \frac{T_c}{N}}.$$

Let the computation-to-communication ratio $r_{comp/comm}$ be the ratio of T_c, the computation time of a single iteration on a single processor, to T_r, then:

$$S(N) = \frac{r_{comp/comm}}{N + \frac{r_{comp/comm}}{N}}.$$

If N unicast operations are used in the first step instead of one broadcast operation (i.e., sending the data from the root processor to all the other processors is done using a sequence of N unicast send operations), it is expected that the speedup

is not changed significantly. This is due to the staggered execution of different operations. All the operations, namely unicast, computation and resending the results back to the root processor are staggered one after another during execution phase.

In each iteration step, all the processors have to wait until they receive the data from the root processor. Once they receive the current approximation to the solution, they start the computation. Finally, the results are sent back to the root processor for further processing. These steps are the same for all the processors and thus the operations are staggered one after another.

If the time of sending the approximation of the solution is represented by T_a, the execution time of one distributed iteration is given by:

$$T(N) = (N-1) * max(T_a, T_r) + \frac{T_c}{N} + min(T_a, T_r).$$

When $T_a \approx T_r$, then:

$$T(N) = N * T_r + \frac{T_c}{N}.$$

This is similar to the execution time of one distributed iteration when the broadcast operation is used to send the initial approximation from the root processor to all the other processors instead of using N unicast operations. So, the equation for the speedup is similar even if N unicast operations are used instead of the broadcast operation.

Figure 2.1 shows the values of speedup for N = 2, 4,..., 32 and for $r_{comp/comm}$= 10,...,120. According to Figure 2.1, better speedups can be obtained when the computation-to-communication ratio is higher. The best speedup is achieved when the number of processors is between 10 to 20. With an increase in the num-

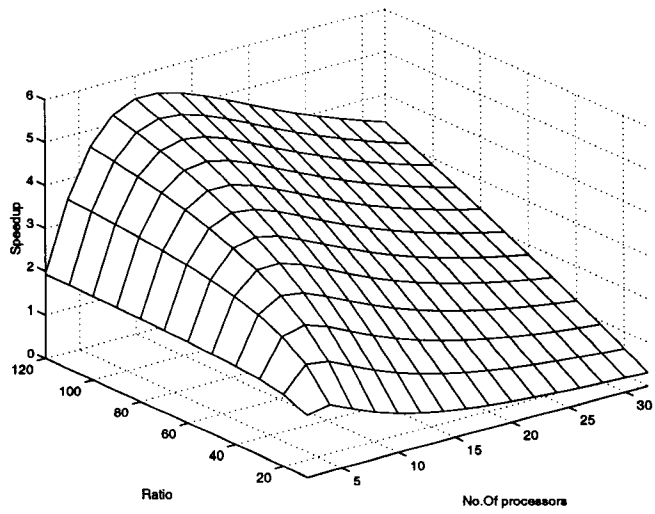

Figure 2.1: Speedup of distributed iterative solvers

ber of processors, the execution time increases gradually due to the dominating communication time. This reduces the expected speedup.

Organizing the collection of results in a hierarchical manner can be used to improve the speedup values and is an efficient way to get better results. One such example is collecting the results in a 2-level hierarchical way, so that first, the results of computations are collected in groups of K processors, and then the results of these groups are combined together. It can be shown that minimal total communication time is obtained when K is equal to $\sqrt{N}$. Then the speedup becomes:

$$S(N) = \frac{r_{comp/comm}}{\frac{r_{comp/comm}}{N} + 2 * \sqrt{N} + 1}.$$

Figure 2.2 shows the speedup values for N = 2, 4, ..., 32 and for $r_{comp/comm}$= 10,...,120 for this hierarchical approach. According to this Figure, better speedup values can be achieved than in the previous case, and the best speedup can be achieved for a higher number of processors than before.

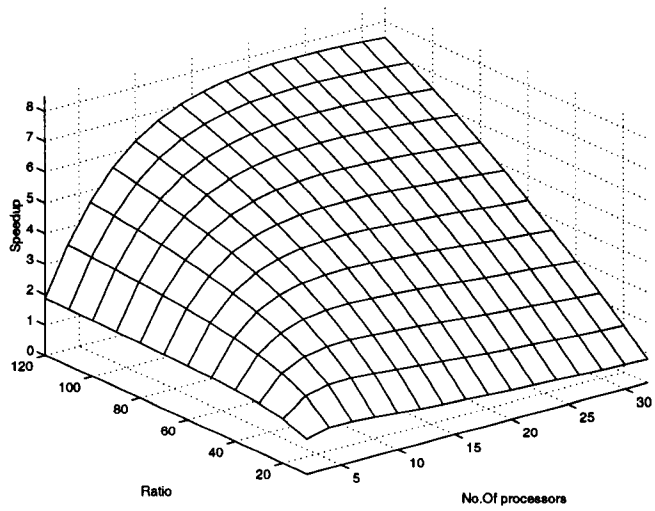

Figure 2.2: Speedup of modified distributed iterative solvers

For large numbers of processors, this approach can be further improved by using additional levels of the hierarchical collection of results.

Chapter 3

Implementation

This chapter provides an overview of the implementation of a distributed iterative linear solver. Section 3.1 describes the format used to represent the sparse matrices for distributed iterative solvers. Section 3.2 outlines the method used to solve the systems of linear equations and the stopping criteria for the iterative method. It also describes how the workload is distributed and how the distributed iterations work. Section 3.3 contains the pseudo-code of algorithms for both uni-processor and distributed cases. The last section discusses the program implementation of the distributed solver.

3.1 Representation of sparse matrices

The storage structure used to represent the sparse matrix $\mathbf{A}$ of the linear system $\mathbf{Ax} = \mathbf{b}$ is similar to the method described in Section 2.2.2, but instead of storing the pointers, the third array is used to store the number of non-zero elements in

$$\begin{pmatrix} 2 & 0 & 0 & 1 & 0 \\ -1 & 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3.1: A 5x5 sparse matrix **A**.

$A_{i,j}$	2	1	-1	3	2	1	1	-2	1
j	1	4	1	2	2	4	1	4	5
count	2		2		2		2		1

Figure 3.2: Representation of the matrix form in Figure 3.1 using the sparse format.

each row, in row major order. An example of sparse matrix representation is shown in Figure 3.1 and Figure 3.2.

3.2 Solution of the systems of linear equations

Section 2.3.2 provided an overview of iterative methods for solving linear algebraic equations. Out of these methods, the Relaxation method is used in the distributed implementation. Factors such as higher rate of convergence than for the Jacobi and Gauss-Seidel methods, and simplicity of implementation make the Relaxation method more attractive than the recently introduced preconditioned methods.

The optimum value of the relaxation factor is determined by a trial and error

approach based on how fast the iterative process converges. It is observed that the best relaxation factor depends on the size of the system as well as on the structure of the coefficient matrix. The initial approximation to the solution was obtained by multiplying the solution vector by a suitable ratio.

3.2.1 Workload distribution

In a distributed system, the workload should be distributed among the processors as uniformly as possible. Workload distribution can be done in many ways, depending upon the characteristics of the problem being solved. One way is by dividing the total number of equations approximately equally among the processors. When the system is diagonal or uniform or when most of the rows contain approximately the same number of non-zero elements this approach is satisfactory. But when the system is non-uniform, irregular or when some of the rows contain a large number of non-zeros, such an approach can cause a significant load imbalance because some processors may get more computing intensive segments than others. This can negatively impact the expected performance.

Load imbalance can be eased by allocating approximately the same numbers of non-zero elements (i.e., the same number of operations) among the available processors. The number of operations performed by each processor is calculated by dividing the total number of non-zero elements by the number of processors (and rounding the result to deal with complete equations if possible).

Assigning the number of operations to each processor and allocating the starting element of the sparse matrix of coefficients is shown below:

Begin Workload distribution
N_Z := the total number of non-zeros;
P := the number of processors;
Next := 1;
For i:=1 **To** P **Do**
    K := round(N_Z/P);
    Noper[i] := K;
    N_Z := N_Z - K;
    Start[i] := Next;
    Next := Next + K
End For
End Division of tasks by the root processor.

For each processor, the vector "**Start**" indicates the first element of the sparse matrix of coefficients processed by this processor, and "**Noper**" the number of nonzero elements assigned to this processor.

3.2.2 Distributed Iterations

In the distributed implementation of the iterative algorithm, the root processor is the "master" processor which controls all other processors. The master processor distributes the workload to other processors as described in the previous section. Each processor starts its calculations when it receives the information from the master processor. Once each processor completes its calculation, it sends the results back to the root processor.

In each iteration, the root processor receives the results from all other processors. The order of getting these results back from other processors can change from one iteration to another. The root processor collects the results, finalizes the calculation of the **xnew**-vector and determines the relative error. If the convergence

criteria are not satisfied, the root processor sends the **xnew**-vector back to all other processors and the calculation is resumed. When the convergence criteria are satisfied, the root processor sets a flag to True and sends it to all the other processors. Then all the other processors end their computations.

It is assumed that the convergence is satisfied when the relative error is less than the required tolerance. The relative error indicates how close the approximate solution is to the optimal solution.

Computation of the relative error, RelErr, is carried out as follows (N is the total number of equations):

```
Begin Computation of the Relative Error(xold,xnew)
RelErr := 0;
Tot := 0.0;
Sum := 0.0;
For i:=1 To N Do
    V[i]:= xnew[i] - xold[i] ;
    Tot := Tot + V[i] * V[i] ;
    Sum := Sum + xnew[i] * xnew[i]
End For;
RelErr := Sqrt(Tot/Sum)
End Computation of the Relative Error(xold,xnew).
```

For very large sets of data, the straightforward computation of a 2-norm can result in an overflow which can be avoided by a simple modification of the computations that involves scaling [32].

3.3 Algorithms

The performance of a uniprocessor system is based on the algorithm presented in Section 3.3.1. The distributed version of the algorithm is outlined in Section 3.3.2. The following notations are used in the algorithms shown in Sections 3.3.1, 3.3.2:

a - the coefficient matrix,

b - the right hand side vector,

xold - the previous approximation to the solution vector,

xnew - the new approximation to the solution vector,

Iter - the number of iterations.

3.3.1 Serial algorithm

The following algorithm shows how the program works for a uniprocessor system. For each iteration the relative error is calculated and the iterations terminate when the convergence criteria is satisfied.

Begin SerialAlgorithm
N := the total number of equations;
Read data from a file;
xnew := the initial approximation $x^{(0)}$;
Tol := the required relative tolerance;
ω := the relaxation coefficient;
Iter := 0;
Converged := False;
While not Converged **Do**
    For i := 1 **to** N **Do**
        xold[i] := **xnew**[i]
    End For;

```
    For Row := 1 to N Do
        Sum := b[Row];
        For Column := 1 to N Do
            If Row ≠ Column Then
                Sum := Sum - A[Row,Column] * xnew[Column]
            End If
        End For;
        Sum := Sum/A[Row,Row];
        xnew[i] = Sum + (1 - ω) * xnew[i]
    End For;
    RelErr := relative error(xold,xnew);
    If RelErr < Tol Then
        Converged := True
    End If;
    Iter := Iter + 1
End While
End SerialAlgorithm.
```

3.3.2 Distributed algorithm

The distributed algorithm uses MPI for communication between the master processor and all other processors. The algorithm shows how the data are passed back and forth between the master processor (root processor) and the other processors in a distributed system. The main task of the root processor is to calculate the relative error and to check the convergence of the iterative process. All processors (including the master processor) participate in the calculation process.

```
Begin DistributedAlgorithm
P := the number of processors;
N := the total number of equations;
Converged := False;
If the current processor = root processor Then
    Read the data from a file;
    Tol := the required relative tolerance;
    Iter := 0;
    xnew := the initial approximation x(0);
    Perform workload distribution;
```

```
        Division of tasks by the root processor
        Send the initial data A,b to other processors;
Else
        Receive the data A,b from the root processor
End If;
While not Converged Do
        If the current processor = root processor Then
                For i := 1 to N Do
                    xold[i] := xnew[i]
                End For;
                For i := 2 to P Do
                    Send xold to a processor
                End For;
                Calculate of a segment of the vector xnew;
                For i := 2 to P Do
                    Receive a segment of xnew from one of processors
                End For;
                Iter := Iter + 1;
                Calculate the final value of xnew;
                RelErr := relative error(xold,xnew);
                If RelErr < Tol Then
                    Converged := True;
                    send "terminate" to other processors
                End If
        Else
                Receive xold from the root processor;
                If received("terminate") Then
                    Converged := True
                Else
                    Calculate a segment of xnew;
                    Send the segment of xnew to the root processor
                End If
        End If
End While
End DistributedAlgorithm.
```

In order to collect the measurements for different numbers of processors in a single run, the distributed program is implemented in such a way that the number of processors P can be changed during program execution. The average execution time for a single iteration is obtained by dividing the total execution time by the

number of iterations required to find the solution.

3.4 Program Implementation

The implementation of distributed solvers is based on the message-passing paradigm, which is used widely on distributed systems. MPI [25],[20] is a message passing system which is used to write portable distributed programs in FORTRAN, C or C++. MPI is the leading standard for message passing libraries for parallel computing. MPI implementations exist for heterogeneous networks of workstations and symmetric multiprocessor systems running UNIX or Windows NT operating systems. MPI can be used on Networks of Workstations (NOWs), Scalable Parallel Computers (SPCs) and combinations of the two.

MPI is not a directive-based data parallel language like High-Performance FORTRAN (HPF) or OpenMP. It is a library of communication routines which is attached to the program thereby providing more flexibility than directive-based approaches. Also, MPI can be implemented on both shared-memory and distributed-memory architectures. It is the programmer's task to explicitly divide the data and work among the processors and to manage the communication between them.

The basic communication mechanism of MPI is point-to-point communication, where data is transmitted between a pair of processors. Blocking send and receive operations were used for the implementation. Programs were implemented using the unicast operation.

The MPI program was executed on the LAM/MPI run time environment on the Linux platform. LAM/MPI is frequently used on Linux-based machines [7].

Once launched, the LAM run time environment, or so-called LAM Universe MPI programs, can be successfully executed. LAM/MPI is a high performance, freely available, high-quality open source implementation of the MPI standard which has a rich set of features. It was developed and is maintained by Open Systems Lab at Indiana University.

The convergence criterion used is the relative error approach; the iterative process is stopped when the relative error satisfies the required tolerance (refer to Section 2.4).

Because the order in which the messages are received is not pre-determined, the program was implemented so that the root processor can receive the sections of vector-x from the other processors in any order.

The measurement of individual computation and communication times is rather difficult and requires some support at the operating system level. Therefore a more general approach has been adopted in which only the total execution time T_{ex} of N_{iter} iterations has been measured (using the available system-level procedures), and then the time of a single iteration was simply obtained as

$$T_{it} = \frac{T_{ex}}{N_{iter}}.$$

For a larger number of iterations (in the order of several thousands), the effects of initialization and termination of the iterative process can be neglected.

Chapter 4

Experimental results

This chapter presents experimental results for distributed iterative solvers of large and sparse systems of linear equations. Performance of distributed solvers was analyzed by measuring the total execution time and calculating the speedup of a distributed solver. Speedup is presented for several structures of systems of equations and also for systems of different sizes with the same structure, using from 2 to 32 processors. The purpose of this study is to analyze how the number of processors affects the performance of the system, to study the effect of system size on the speedup, and to check the influence of the number of processors on the number of iterations required for the iterative solution.

4.1 Data

Several different sparsity patterns in the coefficient matrix are possible. These include a matrix with constant band width, band matrix with step, strip matrix, band

matrix with margin, block diagonal matrix with margin and general sparse matrix [29]. The systems of equations chosen for the experiments include several of these sparsity patterns as well as their combinations. Some typical sparsity patterns are shown in Figure 4.1.

Sparse systems used for the experiments are as follows:

1. System (1): Band structure as in Figure 4.1(a); the band is symmetric and contains three diagonals.

2. System (2): The structure is the one in Figure 4.2(b), with six diagonals.

3. System (3): Band matrix with a margin as shown in Figure 4.1(c); the band is symmetric, contains five diagonals and the margin is at the bottom of the matrix.

4. System (4): Variation of System (3), with the margin at the top of the matrix, as shown in Figure 4.1(d).

5. System (5): A block diagonal structure similar to the one in Figure 4.1(e), the size of each block is 10 by 10, and blocks are sparse.

6. System (6): The structure is similar to that in Figure 4.1(f); it is a block diagonal with overlapping blocks; this system is denser than System (5).

Densities of these systems are approximately $5/N$ for N = 500, 1000, 1500 and 2000, which means that the systems of equations are very sparse.

For each sparsity pattern, data was generated by specialized programs, so no data inconsistencies were expected. In all cases, the exact solution was known

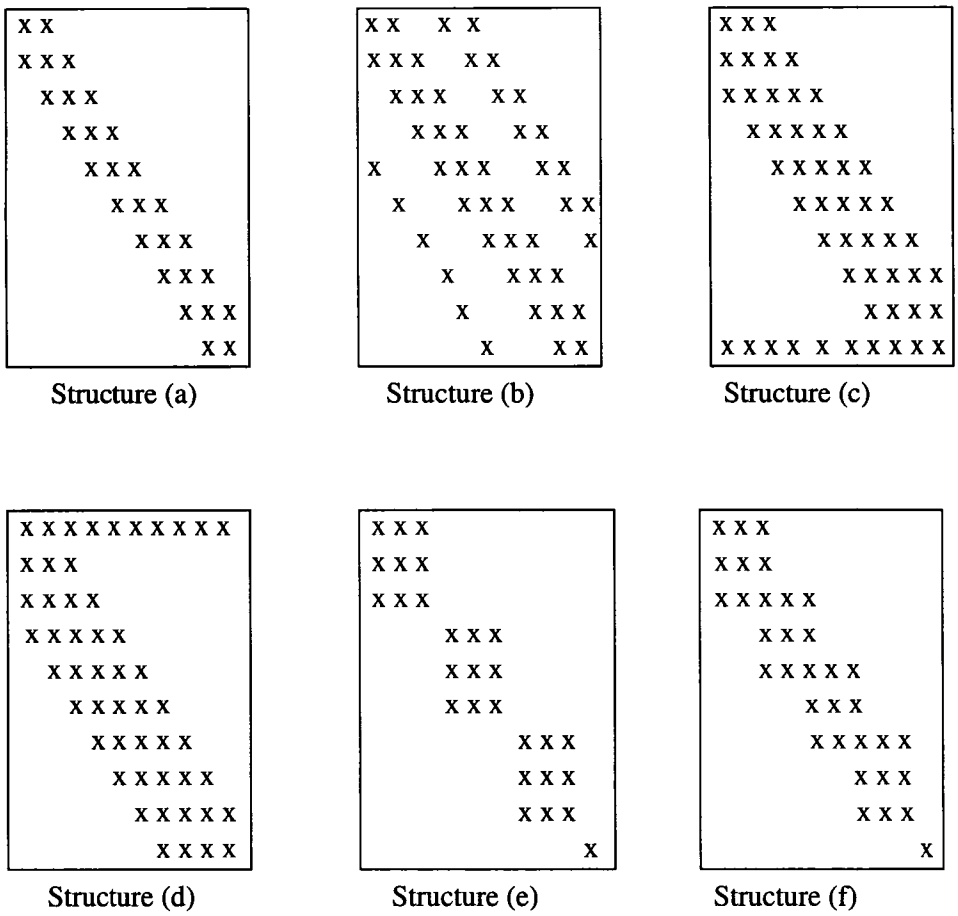

Figure 4.1: Sparsity patterns

and the initial approximation to the solution was obtained by disturbing the exact solution by a fixed ratio.

4.2 Results and Discussion

The purpose of using distributed iterative solvers is to reduce the total execution time and this reduction is represented by the speedup. Speedup was calculated by dividing the time required for the solution using only one processor by the solution

time using N processors.

4.2.1 Speedup of distributed solvers

Figures 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7 show the speedup as a function of the number of processors for the six sparsity patterns discussed in section 4.1, and for four different sizes of the systems of equations.

For System (1), the speedup curves are shown in Figure 4.2. It can be observed that the speedup improves with the size of system of equations. Also, the maximum speedup is obtained for a rather small number of processors, in the range of 6 to 13. For larger number of processors, the speedup decreases as predicted by the simple analysis in Section 2.6. For all sparsity patterns, the speedup plots (Figures 4.2 to 4.7) have the characteristic shapes as shown in Figure 2.1.

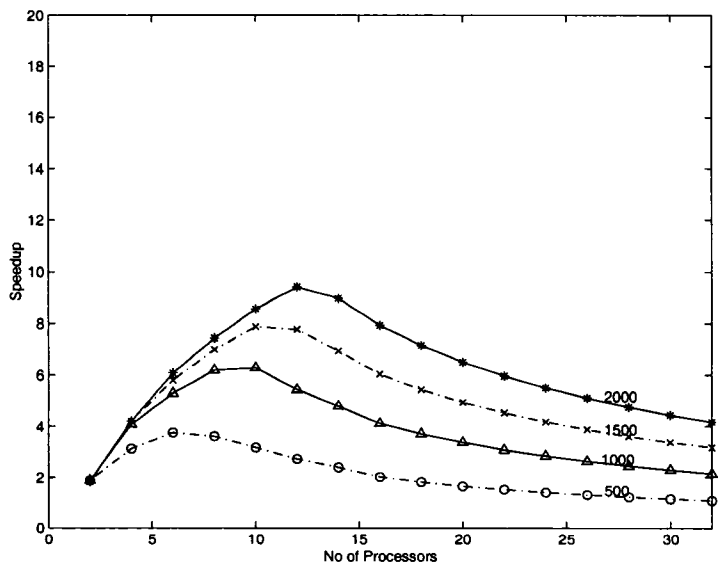

Figure 4.2: Speedup plots for System (1)

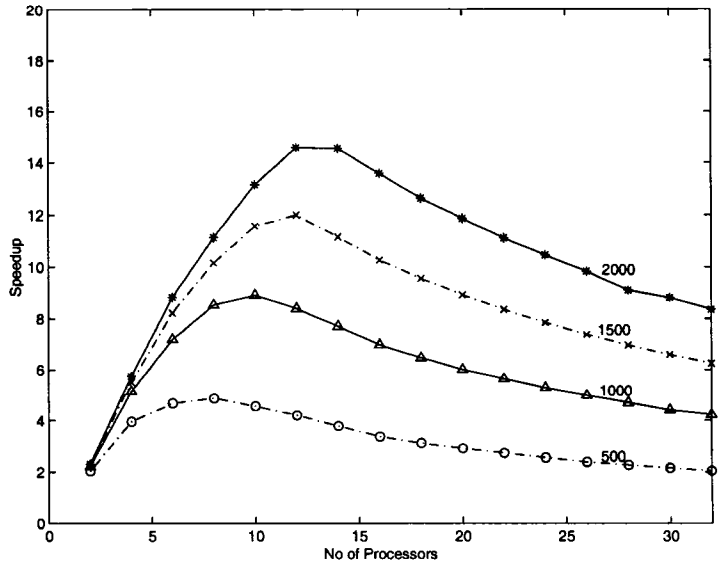

Figure 4.3: Speedup plots for System (2)

4.2.2 Speedup and the density of data

The performance results can be used to asses the influence of the density of the data on the speedup. For example, for Systems (1) and (2), for the same size of the systems of equations, System (2) has density approximately twice as large as System (1), which also means that the $r_{comp/comm}$ for System (2) is more than two times greater than that for System (1). Because of that, the speedup values for System (2) are expected to be greater than those for System (1). The ratio of the speedup values of System (2) to that of System (1) is shown in Figure 4.8 as a function of the number of processors. As shown in Figure 4.8, the ratio of the two speedups grows monotonously with the number of processors which also means that for the maximum speedups corresponding to 10 to 15 processors, (Figures 4.2

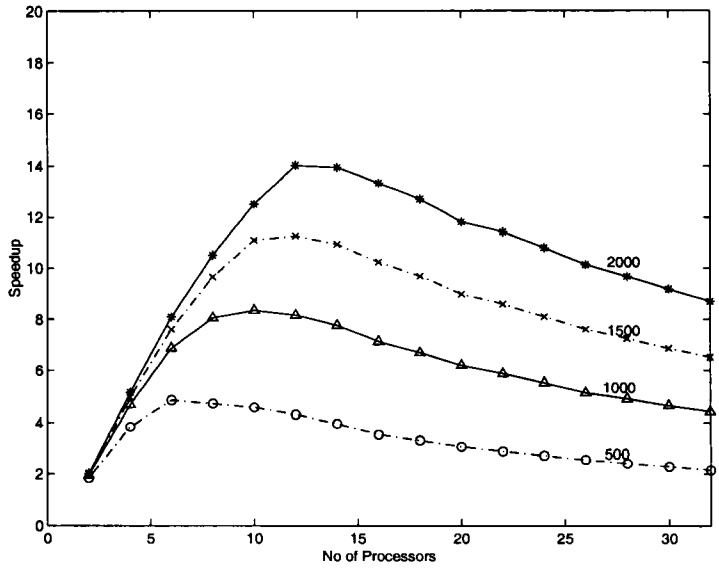

Figure 4.4: Speedup plots for System (3)

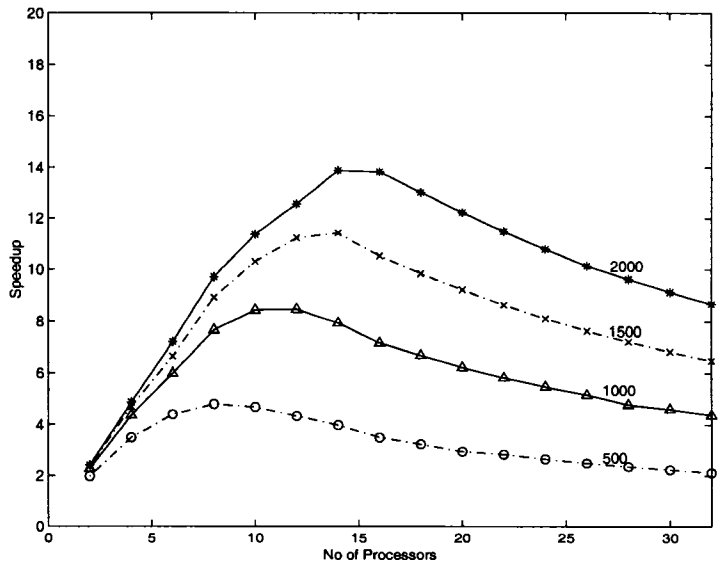

Figure 4.5: Speedup plots for System (4)

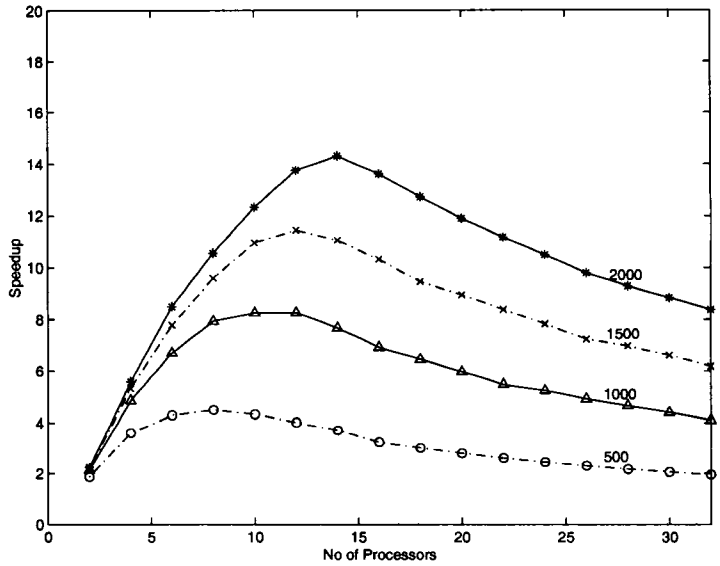

Figure 4.6: Speedup plots for System (5)

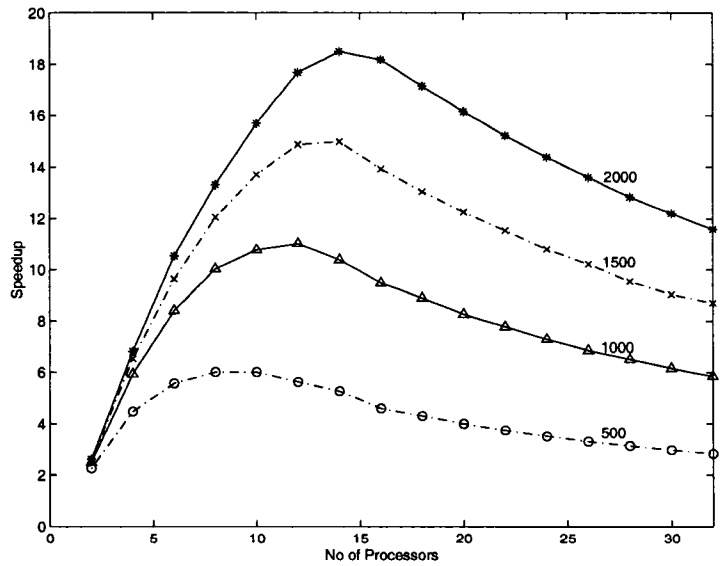

Figure 4.7: Speedup plots for System (6)

and 4.3), this improvement is only about 50%.

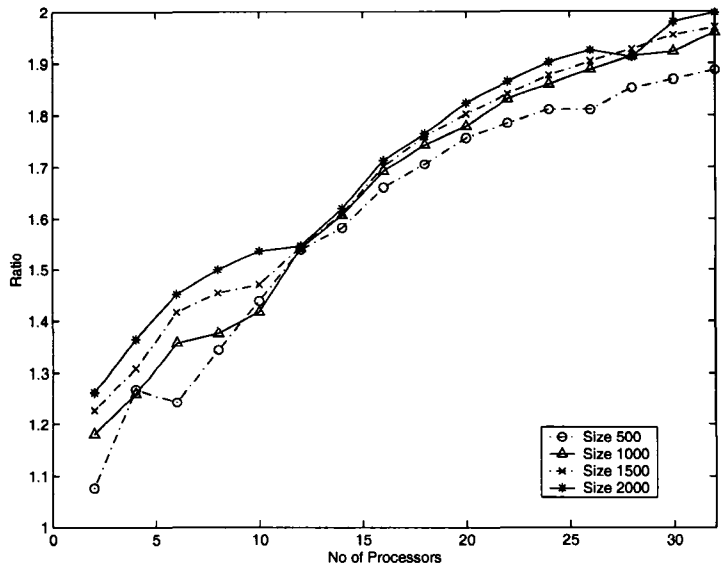

Figure 4.8: The ratio of speedup for System (2) and System (1), as a function of the number of processors.

Systems (3) and (4) have the same densities, and therefore, the same values of $r_{comp/comm}$. Consequently, the speedup values are expected to be the same, as shown in Figures 4.4 and 4.5. This is well illustrated in Figure 4.9.

It can be observed in Figures 4.6 and 4.7 that the speedup values for System (6) are higher than the speedup values for System (5). This is due to the higher density of System (6) and larger values of $r_{comp/comm}$.

Figure 4.10 shows the ratio of the speedup for System (6) to that for System (5) as a function of the number of processors. In this case, the improvement of the speedup is only about 50%.

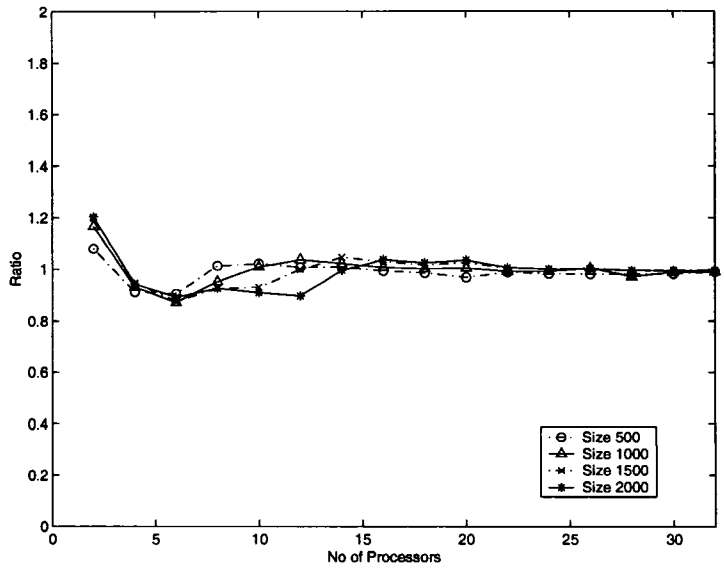

Figure 4.9: The ratio of speedup for System (4) and System (3), as a function of the number of processors.

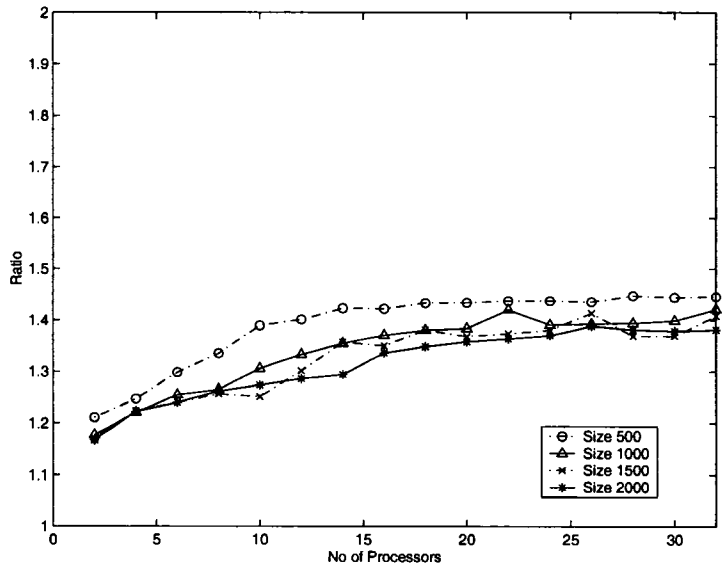

Figure 4.10: The ratio of speedup for System (6) and System (5), as a function of the number of processors.

4.2.3 Speedup and the size of the systems of equations

As the size of the system of equations increases, the workload of each processor increases and the values of $r_{comp/comm}$ also increases. Therefore, the speedup for larger systems of equations is expected to be greater than for smaller systems.

The increase of the speedup for System (2) and System (6), when the number of equations are 1000, 1500 and 2000, are shown in Figures 4.11 and 4.12 with respect to the speedup for size 500. It is clear that when the number of processors is between 10 to 15, the increase is significant for all the sizes.

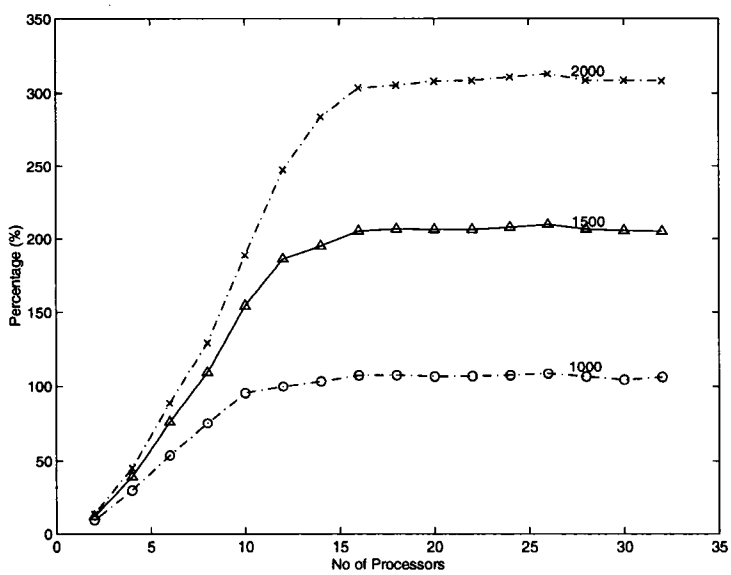

Figure 4.11: Percentage increase of speedup values for System (2)

The speedup as a function of the size of the systems of equations for the same number of processors is shown in Figures 4.13 for System (2) and for a number of processors equal to 4, 8, 20 and 32.

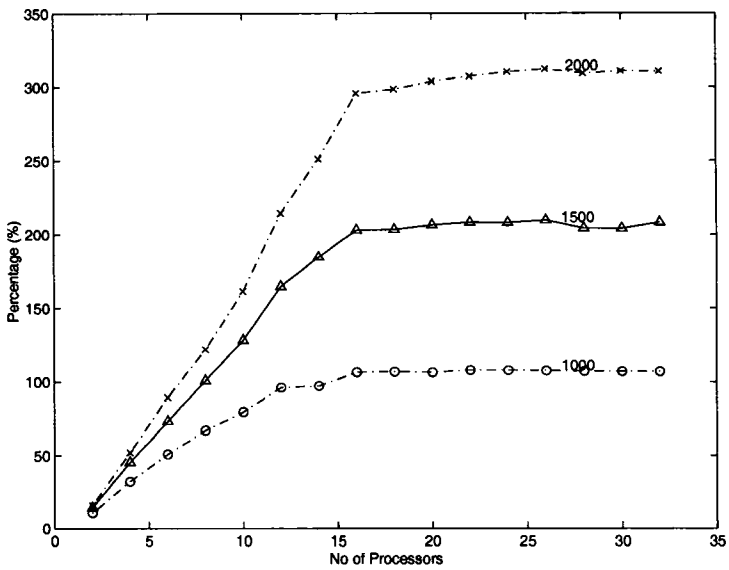

Figure 4.12: Percentage increase of speedup values for System (6)

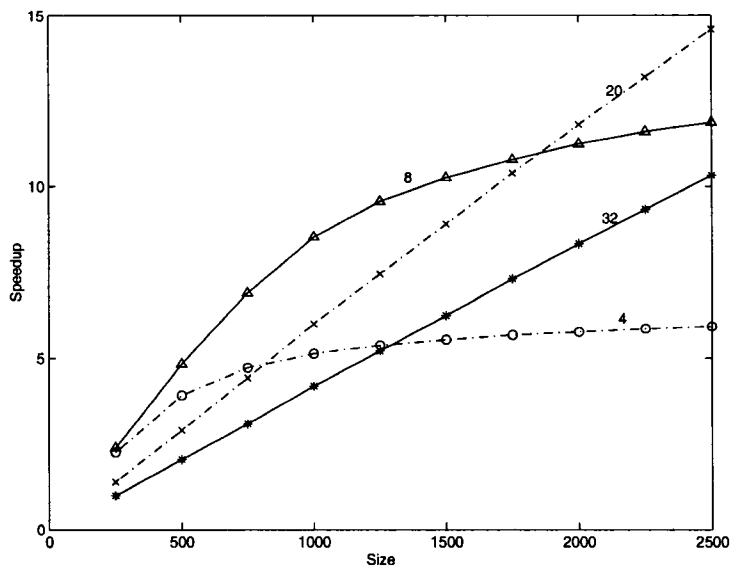

Figure 4.13: Speedup plot for System (2), when the number of processors is 4, 8, 20 and 32

The plots shown in Figure 4.13 can be approximated using the speedup formula:

$$S(N) = \frac{N r_{comp/comm}}{N^2 + r_{comp/comm}}.$$

If the number of processors, N, is large, $N^2 > r_{comp/comm}$, and the speedup becomes:

$$S(N) = \frac{r_{comp/comm}}{N}.$$

Since the value $r_{comp/comm}$ is directly proportional to the size of the system of equations, the plots in Figure 4.13 which correspond to large values of N are practically linear, and the values for N=32 are smaller than those for N=20. For small values of N, the plots in Figure 4.13 reflect the non-linear characteristics of $S(N)$.

4.3 Number of Iterations

The distributed version of the iterative process, presented in Section 3.2, may require a different number of iterations than the corresponding sequential version to provide the same accuracy of the iterated solutions. This difference is a consequence of distributed environment in which, during each iteration, the processors do not have access to results evaluated by other processors. In effect, for a given system of equations, when the number of processors increases, the numbers of operations assigned to each processor become smaller, and the effects of computations – more localized. This, in turn, may affect the convergence properties of the iterative process.

Figures 4.14 and 4.15 show the (total) number of iterations required for distributed iterative solutions of systems of 500, 1000, 1500 and 2000 equations as a

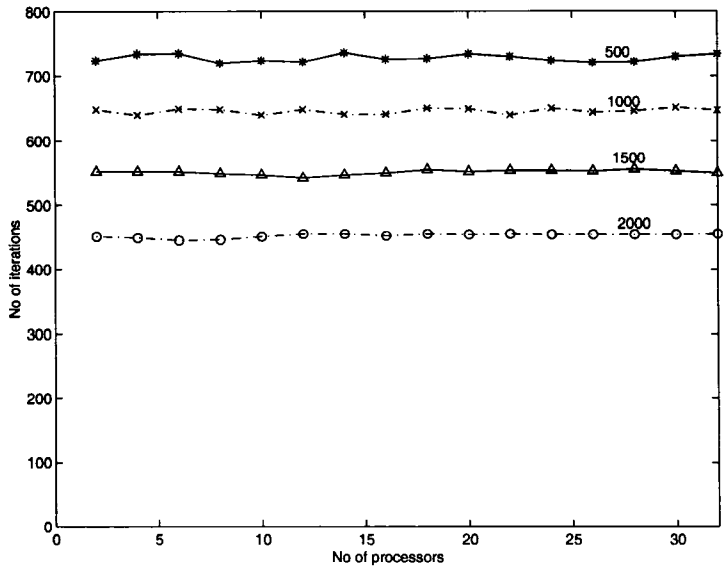

Figure 4.14: Number of iterations as a function of number of processors for System (1)

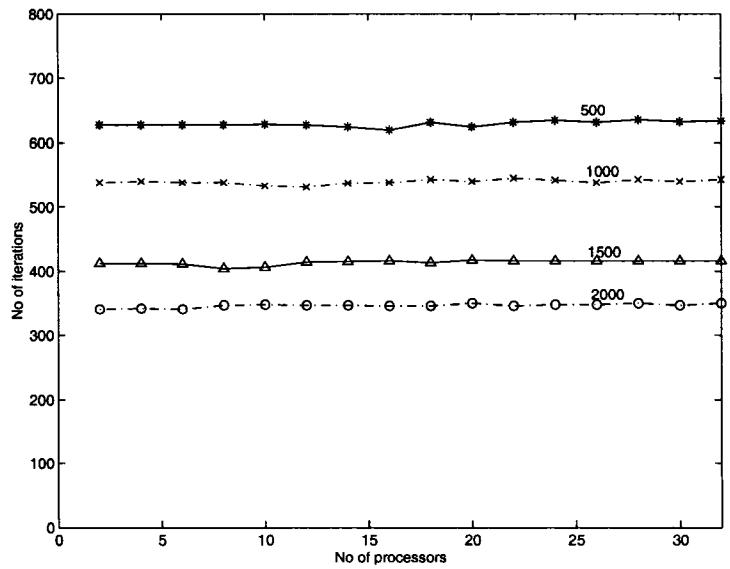

Figure 4.15: Number of iterations as a function of number of processors for System (3)

function of the number of processors used for two different systems of equations.

According to the results shown in Figures 4.14 and 4.15, for System (1) and System (3) the number of processors does not affect the convergence in any significant way if the system is sufficiently large.

Figures 4.14 and 4.15 are somewhat irregular but the number of iterations varies over a small range, so this number does not practically depend upon the number of processors. However, Figures 4.14 and 4.15 show that the number of iterations actually decreases with the increased size of the system of equations; this can be due to improved overall convergence properties of the iterative process when the number of operations assigned to each processor increases. Figures 4.14 and 4.15 also show that the rate of reduction of the number of required iterations depends upon the sparsity structure of the data. When the number of equations increases four times (from 500 to 2000 equations), the number of required iterations decreases by about 35% for System (1) (Figure 4.14) and almost 50% for System (3) (Figure 4.15).

The dependence of the required number of iterations on the number of processors is more pronounced for small systems of equations. Figure 4.16 shows the number of iterations as a function of the number of processors for distributed iterative solvers of System (5) with 100 equations.

The number of required iterations steadily increases in Figure 4.16 as the number of processors changes from 2 to 32. This increase of the number of iterations is due to the same effects as before; as the number of processors increases, the number of operations assigned to each processor decreases, and this affects the convergence properties of the iterative process.

It is anticipated that the effects shown in Figure 4.16 can be more significant for

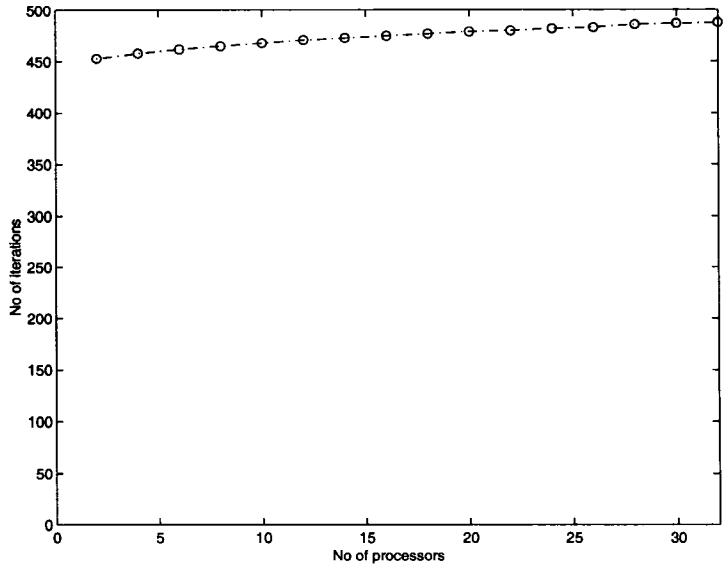

Figure 4.16: The number of iterations of distributed iterative solvers of 100 equations for System (5)

data with other sparsity structures.

Chapter 5

Conclusions

This thesis analyzes the performance of distributed iterative solvers for large and sparse systems of linear equations. Performance of the system was analyzed using the speedup – the most popular metric of the performance of distributed systems.

In this project, the distribution of equations among the processors is straight forward but it introduces a limitation on the speedup. It is observed that the use of a large number of processors may increase the solution time of the system and thus compromise the speedup.

Since distributed systems operate in an unreliable communication environments with a finite-bandwidth, it is obvious that the communication network has a major affect on the systems' performance. In order to reach better performance of the system, communication links should be reliable with small communication delays. Also, other factors such as buffering the data and packing or unpacking the data can reduce system performance. The topology of the interconnection network also plays an important role in distributed systems. By improving the communi-

cation protocols or routing to avoid communication congestion, the performance of distributed systems.

According to both simple analytical considerations and the experimental results, the performance of the distributed solvers increases with the number of processors, but after a certain point, the advantage of distributed computing becomes less significant. Even though the idea of distributed systems is to divide the computation time among the processors, the total execution time is dominated by communication time among the processors exchanging data and results. To achieve better performance, the communication delays must be reduced. This can be done by increasing the bandwidth of the communication medium, or by introducing concurrency at the level of communication, (for example, in the form of parallel channels) or by a combination of the two approaches.

When $r_{comp/comm}$ is high, the communication overhead does not significantly reduce the performance of the system which implies that the higher the ratio of computation to communication, $r_{comp/comm}$, the better speedup.

The discussion of the distribution of computations among the processors of a distributed system (Section 3.2.1) assumes that all processors have similar characteristics (i.e., the system is homogeneous). In this case, the distribution formula is very simple, and the number of non-zero elements assigned to each processor is

$$N_i = \frac{N_Z}{P},$$

where N_Z is the total number of non-zero elements in the linear system and P is the number of processors. For heterogeneous systems, in which the performance of each system can be different, the "load distribution" must take into account the

performance characteristics of processors in such a way that the computation times of all processors are approximately the same, so the more powerful processors should be assigned more workload than the less powerful ones. If M_i denotes the performance (in Mflops per second, for example) of processor i, $i = 1, 2, ..., P$, a simple work allocation formula for a heterogeneous distributed system can be as follows:

$$N_i = \frac{M_i N_Z}{M_P}$$

where $M_P = M_1 + M_2 + ... + M_P$. All remaining aspects of distributed implementation are as described in this thesis.

The approach discussed in this thesis does not apply to shared-memory multiprocessor systems, because the "communication component" in such systems is nonexistent. On the other hand, access to shared memory is typically much slower than to local memory, and concurrent accesses are performed sequentially. Consequently, the performance of iterative solvers on shared-memory systems can be described by the Amdahl's law [5] with the convergence checking section constituting the serial part of code.

The major conclusions from this project are as follows:

1. The experimental results are consistent with analytical predictions.

2. The best speedups are obtained for a rather small number of processors, ranging from 10 to 20 when the system size is average or higher.

3. The speedup improves with the size of the systems of linear equations.

4. Increased density of the system improves the speedup.

5. For a larger number of processors, the performance of the communication network is essential for the overall performance of the solver.

6. When the computation to communication ratio is high, the effect of communication delays is insignificant.

7. If the system is large enough, the number of iterations are practically independent from the number of processors.

Iterative solvers were implemented using the C language and the MPI libraries [25], a message-passing interface. Experimental results were observed on up to 32 processors using the networks of PC's and workstations in the labs of the Department of Computer Science, Memorial University of Newfoundland. Processor configurations were Pentium III with 800 MHZ clock, 256 KB cache memory, 128 MB RAM and 2 GB virtual memory. The operating system used was Linux. The cluster was located at PA-1019 on MUN's campus. All client computers were connected to a 48 port Cisco 100 Megabit switch using Cat5e Ethernet.

All computations were performed in single precision. It is expected that the change of precision will not affect the results presented in this thesis in a significant way.

The experiments were executed on a dedicated cluster of PCs (i.e., no other users were allowed during experiments).

Some further details related to this study can be found in [37] and [36].

Bibliography

[1] Aspen systems home page: "www.aspsys.com/clusters/beowulf/".

[2] Climate prediction home page: "www.climateprediction.net".

[3] Seti@home home page: "setiathome.ssl.berkeley.edu".

[4] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[5] L. Baker and B. J. Smith. *Parallel programming*. McGraw-Hill, 1996.

[6] M. Benzi. *Preconditioning techniques for large linear systems: a survey*. Journalof Computational Physics, Volume 182, Issue 2, pp. 418-477, November 2002.

[7] S. Bhattacharya, P. De Mauro, S. Gundavaram, M. Mamone, K. Sharma, D. Thomas, and S. Whiting. *Beginning Red Hat Linux 9*. Wiley Publishing Inc., Indianapolis, Indiana, 2003.

[8] R. H. Chan, T. F. Chan, and G. H. Golub. *Iterative Methods in Scientific Computing*. Springer-Verlag Singapore Pte. Ltd., 1997.

[9] I. Dimov, I. Lirkov, S. Margenov, and Z. Zlatev (eds). *Numerical Methods and Applications*, volume 2542. Springer-Verlag Berlin Heidelberg, 2003.

[10] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. Van der Vorst. *Numerical Linear Algebra for High performance computers*. Society for Industrial and Applied Mathematics, 1998.

[11] L. Erlander. *Distributed computing: an introduction*. Extreme Tech, April 4, 2002.

[12] D. J. Evans (ed). *Sparsity and its Applications*. Cambridge University Press, 1985.

[13] A. Facius. *Highly accurate verified error bounds for Krylov type linear system solvers*. Applied Numerical Mathematics, vol. 45, no. 1, pp. 41-58, 2003.

[14] V. K. Garg. *Principles of distributed systems*. Kluwer Academic Publ., 1998.

[15] A. I. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, editors. *Parallel Virtual Machine*. The MIT Press, Cambridge, Massachusetts, London, England, 1997.

[16] G. Golub, A. Greenbaum, and M. Luskin. *Recent Advances in Iterative Methods*, volume 60. Springer-Verlag, 1994.

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins Univ. Press, 1983.

[18] A. Greenbaum. *Iterative solution of large sparse systems of equations (Applied Mathematical Sciences 95)*. Springer-Verlag, 1995.

[19] A. Greenbaum. *Iterative methods for solving linear systems (Frontiers in Applied Mathematics 17)*. SIAM, 1997.

[20] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI:portable parallel programming with the message-passing interface*. MIT Press, 1999.

[21] S. Hamilton. *Taking Moore's law into the next century*. IEEE Computer Magazine, vol. 32, no. 1 edition, 1999.

[22] J. D. Hoffman. *Numerical solutions for engineers and scientists*. pub-MH, second edition edition, 1992.

[23] A. Jennings and J. J. McKeown. *Matrix Computations*. John Wiley and Sons, second edition, 1992.

[24] R. Merritt. *Intel, clusters on the rise in 'Top 500 Supercomputer' list*. EE Times Online, November 18, 2003.

[25] P. S. Pacheco (ed). *Parrallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.

[26] S. S. Rao. *Applied Numerical methods for Engineers and Scientists*. Prentice Hall, 2002.

[27] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems*. SIAM, Philadelphia, PA, second edition, 1994.

[28] Y. Saad. *Iterative methods for sparse linear systems*. SIAM 2003, second edition.

[29] U. Schendel. *Sparse Matrices numerical aspects with applications for scientists and engineers*. Ellis Horwood Limited, 1989.

[30] J. A. Stankovic. *Distributed Computing in Distributed computing systems.* IEEE CS Press, 1994.

[31] M. R. Steed and M. J. Clement. *Performance prediction of PVM programs.* Proc. 10-th Int. Parallel processing symposium (IPPS-96), pp. 803-807, 1996.

[32] G. W. Stewart. *Introduction to Matrix Computations.* New York, Academic Press, 1973.

[33] M. Mills Strout, L. Carter, J. Ferrante, and B. Kreaseck. *Tiling for Stationary Iterative methods.* International Journal of High Performance Computing Applications, Volume 18, Issue 1, pp. 95-113, 2004.

[34] R. P. Tewarson. *Sparse Matrices.* Academic Press Inc, New York, 1973.

[35] B. Wilkinson. *Computer Architecture Design and Performance.* Prentice Hall, second edition, 1964.

[36] W. M. Zuberek and T. D. P. Perera. *Speedup of Distributed Iterative Solvers of Large Sparse Systems of Linear Equations.* WSEAS Transactions on Mathematics, vol. 4, no. 3, pp. 281-288, July 2005.

[37] W. M. Zuberek and T. D. P. Perera. *On the Speedup of Distributed Linear Solvers.* 5-th EUROSIM Congress on Modeling and Simulation, pp. 222-223, September 2004.