UNIVERSITY OF MINNESOTA

This is to certify that I have examined this bound copy of a doctoral thesis by

Anshul Gupta

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Vipin Kumar

Name of Faculty Adviser

Signature of Faculty Adviser

Date

GRADUATE SCHOOL

Analysis and Design of Scalable Parallel Algorithms for Scientific Computing

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Anshul Gupta

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

July 31, 1995

ABSTRACT

This dissertation presents a methodology for understanding the performance and scalability of algorithms on parallel computers and the scalability analysis of a variety of numerical algorithms. We demonstrate the analytical power of this technique and show how it can guide the development of better parallel algorithms. We present some new highly scalable parallel algorithms for sparse matrix computations that were widely considered to be poorly suitable for large scale parallel computers.

We present some laws governing the performance and scalability properties that apply to all parallel systems. We show that our results generalize or extend a range of earlier research results concerning the performance of parallel systems. Our scalability analysis of algorithms such as fast Fourier transform (FFT), dense matrix multiplication, sparse matrix-vector multiplication, and the preconditioned conjugate gradient (PCG) provides many interesting insights into their behavior on parallel computers. For example, we show that a commonly used parallel FFT algorithm that was thought to be ideally suited for hypercubes has a limit on the achievable efficiency that is determined by the ratio of CPU speed and communication bandwidth of the hypercube channels. Efficiencies higher than this threshold value can be obtained if the problem size is increased very rapidly. In the case of parallel PCG algorithm, we found that the use of a truncated Incomplete Cholesky (IC) preconditioner, which was considered unsuitable for parallel computers, can actually improve the scalability over a parallel CG with diagonal or no preconditioning. As a result, a parallel formulation of the PCG algorithm with this IC preconditioner may execute faster than that with a simple diagonal preconditioner even if the latter runs faster in a serial implementation for a given problem.

We have developed a highly parallel sparse Cholesky factorization algorithm that substantially improves the state of the art in parallel direct solution of sparse linear systems—both in terms of scalability and overall performance. It is a well known fact that dense matrix factorization scales well and can be implemented efficiently on parallel computers. However, it had been a challenge

to developing efficient and scalable parallel formulations of sparse matrix factorization. Our new parallel sparse factorization algorithm is asymptotically as scalable as the best dense matrix factorization algorithms for a wide class of problems that include all two- and three-dimensional finite element problems. This algorithm incurs less communication overhead than any previously known parallel formulation of sparse matrix factorization. It is equally scalable on parallel architectures based on 2-D mesh, hypercube, fat-tree, and multistage networks. In addition, it is the only known sparse factorization algorithm that can deliver speedups in proportion to an increasing number of processors while requiring almost constant memory per processor.

We have successfully implemented this algorithm for Cholesky factorization on nCUBE2 and Cray T3D parallel computers. An implementation of this algorithm on the T3D delivers up to 20 GFlops on 1024 processors for medium-size structural engineering and linear programming problems. To the best of our knowledge, this is the highest performance ever obtained for sparse Cholesky factorization on any supercomputer.

Numerical factorization is the most time consuming of the four phases involved in obtaining a direct solution of a sparse system of linear equations. In addition to Cholesky factorization, we present efficient parallel algorithms for two other phases—symbolic factorization and for forward and backward substitution to solve the triangular systems resulting from sparse matrix factorization. These algorithms are designed to work in conjunction with our sparse Cholesky factorization algorithm and incur less communication overhead than parallel sparse Cholesky factorization. Along with some recently developed parallel ordering algorithms, the algorithms presented in this thesis make it possible to develop complete scalable parallel direct solvers for sparse linear systems. Although our current implementations work for Cholesky factorization, the algorithm can be adapted for solving sparse linear least squares problems by QR factorization and for Gaussian elimination of matrices that do not require pivoting, thus paving the way for scalable parallel solution to an even wider class of problems.

ACKNOWLEDGMENTS

In completing this thesis, I am indebted to a number of mentors, colleagues, friends, family members, and institutions—too many to list comprehensively. First and foremost, my sincere gratitude goes to my advisor Professor Vipin Kumar, who gave me a unique educational opportunity and working environment. I owe my academic and other achievements during my graduate student years to his guidance, support, enthusiasm, and an extremely helpful and generous nature.

I would like to thank Professors Shantanu Dutt, David Lilja, Matthew O'Keefe, Youcef Saad, Ahmed Sameh, and Shang-Hua Teng for taking the time to serve on my preliminary and final examination committees. Some parts of the thesis contain results of research done in collaboration with George Karypis and Ahmed Sameh. I am grateful to Dr. Fred Gustafson at IBM T. J. Watson Research Center and Prof. Michael Heath at University of Illinois, Urbana for their input and interest in my work. I thank Professors P. C. P. Bhatt, K. K. Biswas, A. K. Gupta, S. K. Gupta, A. Kumar, S. Kaushik, B. B. Madan, and S. N. Maheshwari at the Computer Science Department, Indian Institute of Technology for helping me acquire the background necessary to pursue a doctoral degree. In an administrative capacity, I thank Professors Sameh, Stein, and Tsai for their efforts. I would also like to thank the cooperative and helpful staff at the Department of Computer Science and the Army High Performance Computing Research Center (AHPCRC) at the University of Minnesota. Other institutions that supported this research through grants and supercomputer time are the Army Research Office, Minnesota Supercomputer Institute, Sandia National Labs, Cray Research Inc., and Pittsburgh Supercomputer Center.

I thank my colleagues George Karypis, Dan Challou, Ananth Grama, and Tom Nurkkala for their friendship, advice, and help. I also consider myself lucky to have such dear friends as Ajit, Deepak, Julianna, Kalpana, Lekha, Sanjay, Sarika, Suranjan, Vivek and many others who made my six years' stay in Minneapolis a rich and rewarding experience. Last, but not the least, I am indebted to my parents, without whose love, efforts, and sacrifice this work would not have been possible.

LIST OF FIGURES

LIST OF TABLES

Contents

Chapter 1

INTRODUCTION

Parallel computers consisting of thousands of processors are now commercially available. These computers provide many orders of magnitude more raw computing power than traditional supercomputers at a much lower cost. They open up new frontiers in the application of computers—many previously unsolvable problems can be solved if the power of these machines is used effectively. The availability of massively parallel computers has created a number of challenges, for example: How should parallel computers be programmed? What algorithms and data structures should be used? How can the quality of the algorithms be analyzed? Which algorithms are suitable for particular parallel computer architectures? This dissertation addresses some of these issues in the design and analysis of parallel algorithms.

Analyzing the performance of a given parallel algorithm/architecture calls for a comprehensive method that accounts for scalability: a measure of a parallel system's capacity to effectively utilize an increasing number of processors. There has been extensive work in investigating the performance and scalability properties of large scale parallel systems and several laws governing their behavior have been proposed. Amdahl's law one of the earliest examples. In Chapter 2, we survey a number of techniques and formalisms that have been developed for studying the performance and scalability issues in parallel systems, and discuss their interrelationships. We present a methodology for understanding these issues based on isoefficiency metric of scalability, which relates problem size to the number of processors required to maintain a fixed efficiency. We present some laws governing the performance and scalability properties that parallel systems must obey and show that our results generalize or extend a range of earlier research results concerning the performance of parallel systems. For example, we show that instances of a problem with increasing size can be solved in a constant parallel run time by employing an increasing number of processors if and only if the isoefficiency function of the parallel system is linear with respect to the number of processors. We show that for a wide class of parallel systems, the relationship between the problem size and the number of processors that minimize the run time for that problem size is given by an isoefficiency curve.

Isoefficiency analysis can be used to determine scalability with respect to the number of processors, their speed, and the communication bandwidth of the interconnection network. It succinctly captures the characteristics of a particular algorithm/architecture combination in a single expression; thus, allowing a comparison among various combinations for a range of problem and machine sizes. In Chapter 3, we demonstrate the analytical power of the isoefficiency function by obtaining many useful insights into the behavior of several numerical algorithms on different parallel architectures. For example, we show that a commonly used parallel FFT algorithm that was thought to be ideally suited for hypercubes has a limit on the achievable efficiency that is determined by the ratio of CPU speed and communication bandwidth of the hypercube channels. Efficiencies higher than this threshold value can be obtained if the problem size is increased very rapidly. In the context of dense matrix multiplication, we show that special hardware permitting simultaneous communication on all the ports of the processors does not improve the overall scalability on a hypercube. In the case of parallel PCG algorithm, we found that the use of a truncated Incomplete Cholesky (IC) preconditioner, which was considered unsuitable for parallel computers, can actually improve the scalability over a parallel CG with diagonal or no preconditioning. As a result, a parallel formulation of the PCG algorithm with this IC preconditioner may execute faster than that with a simple diagonal preconditioner even if the latter runs faster in a serial implementation for a given problem.

In Chapter 4, we present a highly parallel sparse Cholesky factorization algorithm that substantially improves the state of the art in parallel direct solution of sparse linear systems—both in terms of scalability and overall performance. This chapter shows how isoefficiency analysis can guide the development of better parallel algorithms by aiding in identifying and eliminating or reducing the scalability bottlenecks in a parallel system. Through an analysis of this and other parallel sparse factorization algorithms, we have shown that our algorithm is the first and only parallel algorithm for this problem that is optimally scalable for a wide class of practical problems. It is a well known fact that dense matrix factorization scales well and can be implemented efficiently on parallel computers. However, it had been a challenge to developing efficient and scalable parallel formulations of sparse matrix factorization. Our new parallel sparse factorization algorithm is asymptotically as scalable as the best dense matrix factorization algorithms on a variety of parallel architectures for a wide class of problems that include all two- and three-dimensional finite element problems. This algorithm incurs less communication overhead than any previously known parallel formulation of sparse matrix factorization. It is equally scalable on parallel architectures based on 2-D mesh, hypercube, fat-tree, and multistage networks. In addition, it is the only known sparse factorization algorithm that can deliver speedups in proportion to an increasing number of processors while requiring almost constant memory per processor.

The performance and scalability analysis of our algorithm is supported by experimental results on up to 1024 processors of the nCUBE2 parallel computer. We have been able to achieve speedups of up to 364 on 1024 processors and 230 on 512 processors over a highly efficient sequential implementation for moderately sized problems from the Harwell-Boeing collection. An implementation of this algorithm on a 1024-processor Cray T3D delivers up to 20 GFLOPS on medium-size structural engineering and linear programming problems [53, 78]. To the best of our knowledge, this is the highest performance ever achieved on any supercomputer for sparse matrix factorization.

Numerical factorization is the most time consuming of the four phases involved in obtaining a direct solution of the sparse system of linear equation. Although direct methods are used extensively in practice, their use for solving large sparse systems has been mostly confined to big vector supercomputers due to the high time and memory requirements of the factorization phase. Parallel processing offers the potential to tackle both these problems; however, only limited success had been achieved until recently in developing scalable parallel formulations of sparse matrix factorization. By using our algorithm, large sparse systems can be solved efficiently on large scale parallel computers. Given the highly scalable nature of our parallel numerical factorization algorithm, it is imperative that the remaining phases of the solution process be parallelized effectively in order to scale the performance of the overall solver. Furthermore, without an overall parallel solver, the size of the sparse systems that can be solved may be severely restricted by the amount of memory available on a uniprocessor system. In Chapter 4, we also present efficient parallel algorithms for two other phases—symbolic factorization and for forward and backward substitution to solve the triangular systems resulting from sparse matrix factorization. These algorithms are designed to work in conjunction with our sparse Cholesky factorization algorithm and incur less communication overhead than parallel sparse Cholesky factorization. Along with some recently developed parallel ordering algorithms [76], the algorithms presented in this thesis make it possible to develop complete scalable parallel direct solvers for sparse linear systems. Although our current implementations work for Cholesky factorization, the algorithm can be adapted for solving sparse linear least squares problems by QR factorization and for Gaussian elimination of matrices that do not require pivoting, thus paving the way for scalable parallel solution to an even wider class of problems.

Chapter 2

PERFORMANCE AND SCALABILITY METRICS FOR PARALLEL SYSTEMS

At the current state of technology, it is possible to construct parallel computers that employ hundreds or thousands of processors. The availability of such computers has created a number of challenges. Determining the best parallel algorithm to solve a problem on a given architecture is considerably more complex that determining the best sequential algorithm. A parallel algorithm that solves a problem well using a fixed number of processors on a particular architecture may perform poorly if either of these parameters changes. Therefore, analyzing the performance of a given parallel algorithm/architecture calls for a rather comprehensive method. In this chapter, we present a methodology for understanding the performance and scalability of algorithms on parallel computers and present some laws governing the performance and scalability properties that parallel systems must obey.

When solving a problem in parallel, it is reasonable to expect a reduction in execution time that is commensurable with the amount of processing resources employed to solve the problem. The scalability of a parallel algorithm on a parallel architecture is a measure of its capacity to effectively utilize an increasing number of processors. Scalability analysis of a parallel algorithm-architecture combination can be used for a variety of purposes. It may be used to select the best algorithm-architecture combination for a problem under different constraints on the growth of the problem size and the number of processors. It may be used to predict the performance of a parallel algorithm and a parallel architecture for a large number of processors from the known performance on fewer processors. For a fixed problem size, it may be used to determine the optimal number of processors to be used and the maximum possible speedup that can be obtained. The scalability analysis can also predict the impact of changing hardware technology on the performance and thus help design better parallel architectures for solving various problems.

In this chapter, we discuss in detail a popular and useful scalability metric, the *isoefficiency function*, first proposed by Kumar and Rao [85] in the context of depth-first search. We present results that generalize this metric to subsume many others proposed in the literature. We also survey some properties of the common performance metrics, such as, parallel run time, speedup, and efficiency. A number of properties of these metrics have been studied in the literature. For example, it is a well known fact that given a parallel architecture and a problem of a fixed size, the speedup of a parallel algorithm does not continue to increase with increasing number of processors. It usually tends to saturate or peak at a certain limit. Thus it may not be useful to employ more than an optimal number of processors for solving a problem on a parallel computer. This optimal number of processors depends on the problem size, the parallel algorithm and the parallel architecture. In this chapter, we study the impact of parallel processing overheads and the degree of concurrency of a parallel algorithm on the optimal number of processors to be used when the criterion for optimality is minimizing the parallel execution time. We then study a more general criterion of optimality and show how operating at the optimal point is equivalent to operating at a unique value of efficiency which is characteristic of the criterion of optimality and the properties of the parallel system under study. We put the technical results derived in this chapter in perspective with similar results that have appeared in the literature before and show how these results generalize or extend the earlier results.

2.1 Definition and Assumptions

In this section, we formally describe the terminology used in the remainder of the dissertation.

Parallel System : The performance of a parallel algorithm cannot be studied in isolation from the parallel architecture it is implemented on. For the purpose of performance evaluation we define a **parallel system** as a combination of a parallel algorithm and a parallel architecture that is a homogeneous ensemble of processors; i.e., all processors and communication channels are identical in speed.

Problem Size W **:** The size of a problem is a measure of the number of basic operations needed to solve the problem. There can be several different algorithms to solve the same problem. To keep the problem size unique for a given problem, we define it as the number of basic operations required by the fastest known sequential algorithm to solve the problem on a single processor. Problem size is a function of the size of the input. For example, for the problem of computing an N-point FFT, $W = \Theta(N \log N)$.

According to our definition, the sequential time complexity of the fastest known serial algorithm to solve a problem determines the size of the problem. If the time taken by an optimal (or the fastest known) sequential algorithm to solve a problem of size W on a single processor is T_S, then $T_S \propto W$, or

$$T_S = t_c W, \tag{2.1}$$

where t_c is a machine dependent constant. Often, we assume $t_c = 1$ and normalize the other constants with respect to t_c. As a result, we can use W and T_S interchangeably in such cases.

Parallel Execution Time T_P : The time elapsed from the moment a parallel computation starts, to the moment the last processor finishes execution. For a given parallel system, T_P is normally a function of the problem size (W) and the number of processors (p), and we will sometimes write it as $T_P(W, p)$.

Cost: The cost of a parallel system is defined as the product of parallel execution time and the number of processors utilized. A parallel system is said to be cost-optimal if and only if the cost is asymptotically of the same order of magnitude as the serial execution time (i.e., $pT_P = \Theta(W)$). Cost is also referred to as **processor-time product**.

Speedup S : The ratio of the serial execution time of the fastest known serial algorithm (T_S) to the parallel execution time of the chosen algorithm (T_P).

Total Parallel Overhead T_o: The sum total of all the overhead incurred due to parallel processing by all the processors. It includes communication costs, non-essential work and idle time due to synchronization and serial components of the algorithm. Mathematically,

$$T_o = pT_P - T_S. \tag{2.2}$$

In order to simplify the analysis, we assume that T_o is a non-negative quantity. This implies that speedup is always bounded by p. For instance, speedup can be superlinear and T_o can be negative if the memory is hierarchical and the access time increases (in discrete steps) as the memory used by the program increases. In this case, the effective computation speed of a large program will be slower on a serial processor than on a parallel computer employing similar processors. The reason is that a sequential algorithm using M bytes of memory will use only M/p bytes on each processor of a p-processor parallel computer. The core results of in this dissertation are still valid with hierarchical memory, except that the scalability and performance metrics will have discontinuities, and their expressions will be different in different ranges of problem sizes. The flat memory assumption helps us to concentrate on the characteristics of the parallel algorithm and architectures, without getting into the details of a particular machine.

For a given parallel system, T_o is normally a function of both W and p and we will often write it as $T_o(W, p)$.

Efficiency E **:** The ratio of speedup (S) to the number of processors (p). Thus,

$$E = \frac{T_S}{pT_P} = \frac{1}{1 + \frac{T_o}{T_S}}. \tag{2.3}$$

Serial Fraction s**:** The ratio of the serial component of an algorithm to its execution time on one processor. The serial component of the algorithm is that part of the algorithm which cannot be parallelized and has to be executed on a single processor.

Degree of Concurrency $C(W)$**:** The maximum number of tasks that can be executed simultaneously at any given time in the parallel algorithm. Clearly, for a given W, the parallel algorithm can not use more than $C(W)$ processors. $C(W)$ depends only on the parallel algorithm, and is independent of the architecture. For example, for multiplying two $N \times N$ matrices using Fox's parallel matrix multiplication algorithm [37], $W = N^3$ and $C(W) = N^2 = W^{2/3}$. It is easily seen that if the processor-time product [5] is $\Theta(W)$ (i.e., the algorithm is *cost-optimal*), then $C(W) = O(W)$.

Maximum Number of Processors Usable, p_{max}**:** The number of processors that yield maximum speedup S^{max} for a given W. This is the maximum number of processors one would like to use because using more processors will not increase the speedup.

2.2 The Isoefficiency Metric of Scalability

If a parallel system is used to solve a problem instance of a fixed size, then the efficiency decreases as p increases. The reason is that T_o increases with p. For many parallel systems, if the problem size W is increased on a fixed number of processors, then the efficiency increases because T_o grows slower than W. For these parallel systems, the efficiency can be maintained at some fixed value (between 0 and 1) for increasing p, provided that W is also increased. We call such systems **scalable**[1] parallel systems. This definition of scalable parallel algorithms is similar to the definition of parallel effective algorithms given by Moler [102].

For different parallel systems, W should be increased at different rates with respect to p in order to maintain a fixed efficiency. For instance, in some cases, W might need to grow as an exponential function of p to keep the efficiency from dropping as p increases. Such parallel systems are poorly scalable. The reason is that on these parallel systems, it is difficult to obtain good speedups for a large number of processors unless the problem size is enormous. On the other hand, if W needs to grow only linearly with respect to p, then the parallel system is highly scalable. This is because it can easily deliver speedups proportional to the number of processors for reasonable problem sizes.

The rate at which W is required to grow w.r.t. p to keep the efficiency fixed can be used as a measure of scalability of the parallel algorithm for a specific architecture. If W must grow as $f_E(p)$ to maintain an efficiency E, then $f_E(p)$ is defined to be the **isoefficiency function** for efficiency E and the plot of $f_E(p)$ vs. p is called the isoefficiency curve for efficiency E. Equivalently, if the relation $W = f_E(p)$ defines the isoefficiency curve for a parallel system, then p should not grow faster that $f_E^{-1}(W)$ if an efficiency of at least E is desired.

Given that $E = 1/(1 + T_o(W, p)/(t_c W))$, in order to maintain a fixed efficiency, W should be proportional to $T_o(W, p)$. In other words, the following relation must be satisfied in order to maintain a fixed efficiency:

$$W = K T_o(W, p), \tag{2.4}$$

where $K = E/(t_c(1 - E))$ is a constant depending on the efficiency to be maintained. Equation

[1] For some parallel systems (e.g., some of the ones discussed in [125] and [88]), the maximum obtainable efficiency E^{max} is less than 1. Even such parallel systems are considered scalable if the efficiency can be maintained at a desirable value between 0 and E^{max}.

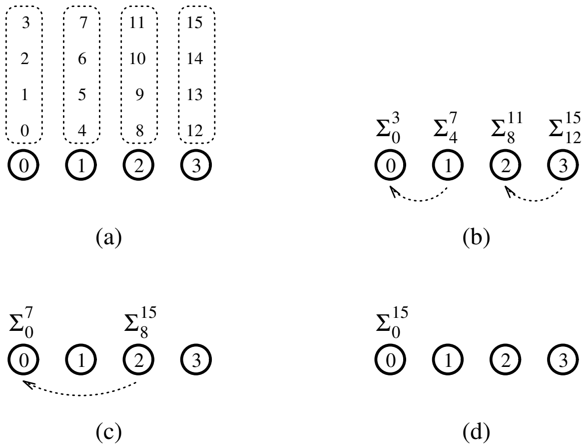


Figure 2.1: Computing the sum of 16 numbers on a 4-processor hypercube.

2.4 is the central relation that is used to determine the isoefficiency function of a parallel algorithm-architecture combination. This is accomplished by abstracting W as a function of p through algebraic manipulations on Equation 2.4.

For example, consider the problem of adding n numbers. For this problem the number of operations, and hence the problem size W is equal to n. If we assume that each addition takes unit time, then $T_S = n = W$ (in reality the number of basic operations, W, is $n - 1$; however, for large values of n, W can be approximated by n). Now consider a parallel algorithm for adding n numbers using a p-processor hypercube. This algorithm is shown in Figure 2.1 for $n = 16$ and $p = 4$. Each processors is allocated n/p numbers. In the first step of this algorithm, each processor locally adds its n/p numbers in $\Theta(n/p)$ time. The problem is now reduced to adding the p partial sums on p processors. These can be done by propagating and adding the partial sums as shown in Figure 2.1. A single step consists of one addition and one nearest neighbor communication of a single word, each of which is a constant time operation. For the sake of simplicity, let us assume that it takes one unit of time to add two numbers and also to communicate a number between two processors. Therefore, n/p time is spent in adding the n/p local numbers at each processor. After the local addition, the p partial sums are added in $\log p$ steps, each step consisting of one addition and one communication. Thus, the total parallel execution time T_P is $n/p + 2\log p$. The same task can be accomplished sequentially in n time units. Thus, out of the $n/p + 2\log p$ time units that each processor spends in parallel execution, n/p time is spent in performing useful work. The remaining

$2 \log p$ units of time per processor contribute to a total overhead of

$$T_o = 2p \log p. \tag{2.5}$$

Substituting the value of T_o in Equation 2.4, we get

$$W = 2Kp \log p. \tag{2.6}$$

Thus the asymptotic isoefficiency function for this parallel system is $\Theta(p \log p)$. This means that if the number of processors is increased from p to p', the problem size (in this case, n) will have to be increased by a factor of $p' \log p' / (p \log p)$ to get the same efficiency as on p processors. In other words, increasing the number of processors by a factor of p'/p requires n to be increased by a factor of $p' \log p' / (p \log p)$, in order to increase the speedup by a factor of p'/p.

In the simple example of adding n numbers, the communication overhead is a function of only p. In general, it can depend on both the problem size and the number of processors. A typical overhead function may have several different terms of different orders of magnitude with respect to p and W. When there are multiple terms of different orders of magnitude in the overhead function, it may be impossible or cumbersome to obtain the isoefficiency function as a closed form function of p. For instance, consider a hypothetical parallel system, for which $T_o = p^{3/2} + p^{3/4}W^{3/4}$. In this case Equation 2.4 will be $W = Kp^{3/2} + Kp^{3/4}W^{3/4}$. It is difficult to solve for W in terms of p. Recall that the condition for constant efficiency is that the ratio of T_o and W should remain fixed. As p and W increase in a parallel system, the efficiency is guaranteed not to drop if none of the terms of T_o grow faster than W. Therefore, if T_o has multiple terms, we balance W against each individual term of T_o to compute the respective isoefficiency function. The component of T_o that causes the problem size to grow at the fastest rate with respect to p determines the overall asymptotic isoefficiency function of the computation.

For example, consider a hypothetical parallel algorithm-architecture combination for which $T_o = p^{3/2} + p^{3/4}W^{3/4}$. If we ignore the second term of T_o and use only the first term in Equation 2.4, we get

$$W = Kp^{3/2}. \tag{2.7}$$

Now consider only the second term of the overhead function and repeat the above analysis. Equation 2.4 now takes the form

$$\begin{aligned} W &= K p^{3/4} W^{3/4}, \\ W^{1/4} &= K p^{3/4}, \\ W &= K^4 p^3. \end{aligned} \tag{2.8}$$

In order to ensure that the efficiency does not decrease as the number of processors increase, the first and the second term of the overhead function require the problem size to grow as $\Theta(p^{3/2})$ and $\Theta(p^3)$, respectively. The asymptotically higher of the two rates should be regarded as the overall asymptotic isoefficiency function. Thus, the isoefficiency function is $\Theta(p^3)$ for this parallel system. This is because if the problem size W grows as $\Theta(p^3)$, then T_o would remain of the same order as W.

Isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel systems [53, 70, 58, 85, 86, 88, 118, 125, 144, 143, 56, 54, 87, 52, 83]. In a single expression, the isoefficiency function captures the characteristics of a parallel algorithm as well as the parallel architecture on which it is implemented. After performing the isoefficiency analysis, we can test the performance of a parallel program on a few processors and then predict its performance on a larger number of processors. However, the utility of isoefficiency analysis is not limited to predicting the impact on performance of an increasing number of processors. In [51], we show how the isoefficiency function characterizes the amount of parallelism inherent in a parallel algorithm. We will see in later (for example in the context of parallel FFT in Chapter 3) that isoefficiency analysis can be used also to study the behavior of a parallel system with respect to changes in hardware parameters such as the speed of processors and communication channels.

The reader should note that in the framework described in this section, a parallel system is considered scalable if its isoefficiency function exists; otherwise the parallel system is unscalable. The isoefficiency function of a scalable system could, however, be arbitrarily large; i.e., it could dictate a very high rate of growth of problem size w.r.t. the number of processors. In practice, the problem size can be increased asymptotically only at a rate permitted by the amount of memory available at each processor. If the memory constraint does not allow the size of the problem to increase at the rate necessary to maintain a fixed efficiency, then the parallel system should be considered unscalable from a practical point of view.

2.3 Relationship between Isoefficiency and Other Metrics

A number of scalability metrics have been proposed by various researchers [22, 34, 36, 55, 61, 59, 60, 75, 80, 98, 108, 131, 130, 132, 137, 142, 146, 149]. We present a detailed survey of these metrics in [84]. After reviewing these various measures of scalability, one may ask whether there exists one measure that is better than all others [66]? The answer to this question is no, as different measures are suitable for different situations.

One situation arises when the problem at hand is fixed and one is trying to use an increasing number of processors to solve it. In this case, the speedup is determined by the serial fraction in the program as well as other overheads such as those due to communication and due to redundant work. In this situation choosing one parallel system over the other can be done using the standard speedup metric. Note that for any fixed problem size W, the speedup on a parallel system will saturate or peak at some value $S^{max}(W)$, which can also be used as a metric. Scalability issues for the fixed problem size case are addressed in [36, 75, 55, 105, 134, 146].

Another possible scenario is that in which a parallel computer with a fixed number of processors is being used and the best parallel algorithm needs to be chosen for solving a particular problem. For a fixed p, the efficiency increases as the problem size is increased. The rate at which the efficiency increases and approaches one (or some other maximum value) with respect to increase in problem size may be used to characterize the quality of the algorithm's implementation on the given architecture.

The third situation arises when the additional computing power due to the use of more processors is to be used to solve bigger problems. Now the question is how should the problem size be increased with the number of processors?

For many problem domains, it is appropriate to increase the problem size with the number of processors so that the total parallel execution time remains fixed. An example is the domain of weather forecasting. In this domain, the size of the problem can be increased arbitrarily provided that the problem can be solved within a specified time (e.g., it does not make sense to take more than 24 hours to forecast the next day's weather). The scalability issues for such problems have been explored by Worley [146], Gustafson [61, 59], and Sun and Ni [131].

Another extreme in scaling the problem size is to try as big problems as can be handled in the memory. This is investigated by Worley [145, 146, 147], Gustafson [61, 59] and by Sun and Ni [131], and is called the memory-constrained case. Since the total memory of a parallel computer increases with increasing p, it is possible to solve bigger problems on parallel computer with bigger p. It should also be clear that any problem size for which the memory requirement exceeds the total available memory cannot be solved on the system.

An important scenario is that in which one is interested in making efficient use of the parallel system; i.e., it is desired that the overall performance of the parallel system increases linearly with p. Clearly, this can be done only for scalable parallel systems, which are exactly the ones for which a fixed efficiency can be maintained for arbitrarily large p by simply increasing the problem size. For such systems, it is natural to use isoefficiency function or related metrics [85, 80, 22]. The analyses in [148, 149, 36, 97, 105, 134, 34] also attempt to study the behavior of a parallel system with some concern for overall efficiency.

Although different scalability measures are appropriate for rather different situations, many of them are related to each other. For example, from the isoefficiency analysis, one can reach a number of conclusions regarding the time-constrained case (i.e., when bigger problems are solved on larger parallel computers with some upper-bound on the parallel execution time). It can be shown that for cost-optimal algorithms, the problem size can be increased linearly with the number of processors while maintaining a fixed execution time if and only if the isoefficiency function is $\Theta(p)$. The proof is as follows. Let $C(W)$ be the degree of concurrency of the algorithm. Thus, as p is increased, W has to be increased at least as $\Theta(p)$, or else p will eventually exceed $C(W)$. Note that $C(W)$ is upper-bounded by $\Theta(W)$ and p is upper-bounded by $C(W)$. T_P is given by $(T_S + T_o(W, p))/p = (t_c W + T_o(W, p))/p$. Now consider the following two cases. Let the first case be when $C(W)$ is smaller than $\Theta(W)$. In this case, even if as many as $C(W)$ processors are used, the term $t_c W/C(W)$ of the expression for T_P will diverge with increasing W, and hence, it is not possible to continue to increase the problem size and maintain a fixed parallel execution time. At the same time, the overall isoefficiency function grows faster than $\Theta(p)$ because the isoefficiency due to concurrency exceeds $\Theta(p)$. In the second case in which $C(W) = \Theta(W)$, as many as $\Theta(W)$ processors can be used. If $\Theta(W)$ processors are used, then the first term in T_P can be maintained at a constant value irrespective of W. The second term in T_P will remain constant if and only if $T_o(W, p)/p$ remains constant when $p = \Theta(W)$ (in other words, T_o/W remains constant while p and W are of the same order). This condition is necessary and sufficient for linear isoefficiency.

A direct corollary of the above result is that if the isoefficiency function is greater than $\Theta(p)$, then the minimum parallel execution time will increase even if the problem size is increased as slowly as linearly with the number of processors. Worley [145, 146, 147] has shown that for many algorithms used in the scientific domain, for any given T_P, there will exist a problem size large enough so that it cannot be solved in time T_P, no matter how many processors are used. Our above analysis shows that for these parallel systems, the isoefficiency curves have to be worse than linear. It can be easily shown that the isoefficiency function will be greater than $\Theta(p)$ for any algorithm-architecture combination for which $T_o > \Theta(p)$ for a given W. The latter is true when any algorithm with a global operation (such as broadcast, and one-to-all and all-to-all personalized communication [18, 71]) is implemented on a parallel architecture that has a message passing latency or message startup time. Thus, it can be concluded that *for any cost-optimal parallel algorithm involving global communication, the problem size cannot be increased indefinitely without increasing the execution time on a parallel computer having a startup latency for messages, no matter how many processors are used (up to a maximum of W).* This class of algorithms includes some fairly important algorithms such as matrix multiplication (all-to-all/one-to-all broadcast) [54], vector dot products (single node accumulation) [58], shortest paths (one-to-all broadcast) [88], and FFT (all-to-all personalized communication) [56], etc. The readers should note that the presence of a global communication operation in an algorithm is a sufficient but not a necessary condition for non-linear isoefficiency on an architecture with message passing latency. Thus, the class of algorithms having the above mentioned property is not limited to the algorithms with global communication.

If the isoefficiency function of a parallel system is greater than $\Theta(p)$, then given a problem size W, there is as lower-bound on the parallel execution time. This lower-bound (lets call it $T_P^{min}(W)$) is a non-decreasing function of W. The rate at which the $T_P^{min}(W)$ for a problem (given arbitrarily many processors) must increase with the problem size can also serve as a measure of scalability of the parallel system. In the best case, $T_P^{min}(W)$ is constant; i.e., larger problems can be solved in a fixed amount of time by simply increasing the number of processors. In the worst case, $T_P^{min}(W) = \Theta(W)$. This happens when the degree of effective parallelism is constant. The slower $T_P^{min}(W)$ grows as a function of the problem size, the more scalable the parallel system is. T_P^{min} is closely related to $S^{max}(W)$. For a problem size W, these two metrics are related by $W = S^{max}(W) \times T_P^{min}(W)$.

Let $\xi(W)$ be the number of processors that should be used for obtaining the minimum parallel execution time $T_P^{min}(W)$ for a problem of size W. Clearly, $T_P^{min}(W) = (W + T_o(W, \xi(W)))/\xi(W)$. Using $\xi(W)$ processors leads to optimal parallel execution time $T_P^{min}(W)$, but may not lead to minimum pT_P product (or the cost of parallel execution). Now consider the cost-optimal implementation of the parallel system (i.e., when the number of processors used for a given problem size is governed by the isoefficiency function). In this case, if $f(p)$ is the isoefficiency function, then T_P is given by $(W + T_o(W, f^{-1}(W)))/f^{-1}(W)$ for a fixed W. Let us call this $T_P^{iso}(W)$. Clearly, $T_P^{iso}(W)$ can be no better than $T_P^{min}(W)$.

Several researchers have proposed to use an operating point where the value of $p(T_P)^r$ is minimized for some constant r and for a given problem size W [36, 34, 134]. It can be shown [134] that this corresponds to the point where ES^{r-1} is maximized for a given problem size. Note that the location of the minima of $p(T_P)^r$ (with respect to p) for two different algorithm-architecture combinations can be used to choose one between the two.

In the following subsections, we will show the relationship between the isoefficiency function and operating at the point of minimum T_P or minimum $p(T_P)^r$ for a wide class of parallel systems.

2.3.1 Minimizing the Parallel Execution Time

In this section we relate the behavior of the T_P verses p curve to the nature of the overhead function T_o. As the number of processors is increased, T_P either asymptotically approaches a minimum value, or attains a minimum and starts rising again. We identify the overhead functions which lead to one case or the other. We show that in either case, the problem can be solved in minimum time by using a certain number of processors which we call p_{max}. Using more processors than p_{max} will either have no effect or will degrade the performance of the parallel system in terms of parallel execution time.

Most problems have a serial component W_s, which is the part of W that has to be executed sequentially. We do not consider the sequential component of an algorithm as a separate entity because it can be subsumed in T_o. While one processor is working on the sequential component, the remaining $p - 1$ are ideal and contribute $(p - 1)W_s$ to T_o. Thus for any parallel algorithm with a nonzero W_s, the analysis can be performed by assuming that T_o includes a term equal to $(p - 1)W_s$. Under this assumption, the parallel execution time T_P for a problem of size W on p processors is given by the following relation:

$$T_P = \frac{W + T_o(W, p)}{p}. \tag{2.9}$$

We now study the behavior of T_P under two different conditions.

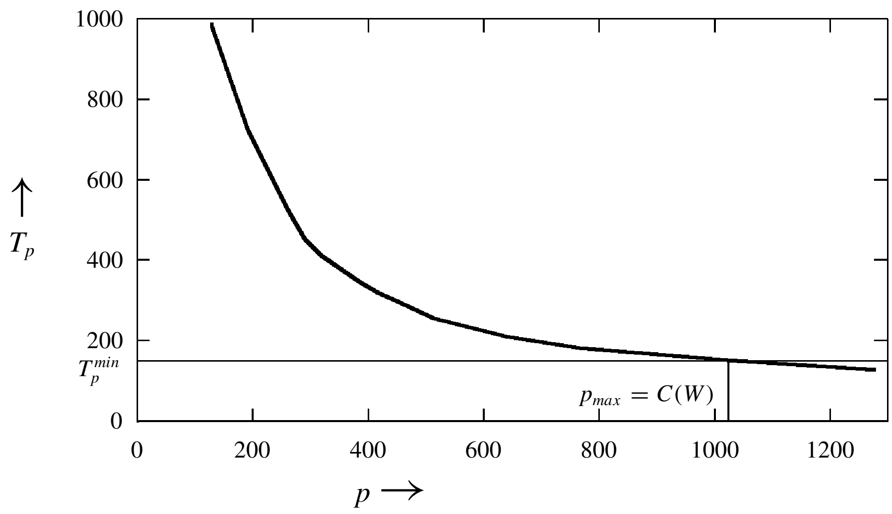


Figure 2.2: A typical T_P verses p curve for $T_o \leq \Theta(p)$.

Case I: $T_o \leq \Theta(p)$

From Equation 2.9 it is clear that if $T_o(W, p)$ grows slower than $\Theta(p)$, then the overall power of p in the R.H.S. of Equation 2.9 is negative. In this case it would appear that if p is increased, then

T_P will continue to decrease indefinitely. If $T_o(W, p)$ grows as fast as $\Theta(p)$ then there will be a lower bound on T_P, but that will be a constant independent of W. But we know that for any parallel system, the maximum number of processors that can be used for a given W is limited by $C(W)$. So the maximum speedup is bounded by $WC(W)/(W + T_o(W, C(W)))$ for a problem of size W and the efficiency at this point of peak performance is given by $W/(W + T_o(W, C(W)))$. Figure 1 illustrates the curve of T_P for the case when $T_o \leq \Theta(p)$.

There are many important natural parallel systems for which the overhead function does not grow faster than $\Theta(p)$. Such systems typically arise while using shared memory or SIMD machines which do not have a message startup time for data communication. For example, consider a parallel implementation of the FFT algorithm [56] on a SIMD hypercube connected machine (e.g., the CM-2 [72]). If an N point FFT is being attempted on such a machine with p processors, N/p units of data will be communicated among directly connected processors in $\log p$ of the $\log N$ iterations of the algorithm. For this parallel system $W = N \log N$. As shown in [56], $T_o = t_w \times (N/p) \log p \times p = t_w N \log p$, where t_w is the message communication time per word. Clearly, for a given W, $T_o < \Theta(p)$. Since $C(W)$ for the parallel FFT algorithm is N, there is a lower bound on parallel execution time which is given by $(1 + t_w) \log N$. Thus, p_{max} for an N point FFT on a SIMD hypercube is N and the problem cannot be solved in less than $\Theta(\log N)$ time.

Case II: $T_o > \Theta(p)$

When $T_o(W, p)$ grows faster than $\Theta(p)$, a glance at Equation 2.9 will reveal that the term W/p will keep decreasing with increasing p, while the term T_o/p will increase. Therefore, the overall T_P will first decrease and then increase with increasing p, resulting in a distinct minimum. Now we derive the relationship between W and p such that T_P is minimized. Let p_0 be the value of p for which the mathematical expression on the R.H.S of Equation 2.9 for T_P attains its minimum value.

At $p = p_0$, T_P is minimum and therefore $\frac{d}{dp} T_P = 0$.

$$\begin{aligned}
\frac{d}{dp}\left(\frac{W + T_o(W, p)}{p}\right) &= 0, \\
\frac{-W}{p^2} - \frac{T_o(W, p)}{p^2} + \frac{\frac{d}{dp} T_o(W, p)}{p} &= 0, \\
\frac{d}{dp} T_o(W, p) &= \frac{W}{p} + \frac{T_o(W, p)}{p},
\end{aligned}$$

$$\frac{d}{dp}T_o(W,p) \quad = \quad T_P. \tag{2.10}$$

For a given W, we can solve the above equation to find p_0. A rather general form of the overhead is one in which the overhead function is a sum of terms where each term is a product of a function of W and a function of p. In most real life parallel systems, these functions of W and p are such that T_o can be written as $\Sigma_{i=1}^{i=n} c_i W^{y_i} (\log W)^{u_i} p^{x_i} (\log p)^{z_i}$, where c_i's are constants and $x_i \geq 0$ and $y_i \geq 0$ for $1 \leq i \leq n$, and u_i's and z_i's are 0's or 1's. The overhead functions of all architecture-algorithm combinations that we have come across fit this form [85, 88, 125, 56, 54, 58, 144, 143, 52]. As illustrated by a variety of examples in this chapter (these include important algorithms such as Matrix Multiplication, FFT, Parallel Search, finding Shortest Paths in a graph, etc.), on almost all parallel architectures of interest.

For the sake of simplicity of the following analysis, we assume $z_i = 0$ and $u_i = 0$ for all i's. Analysis similar to that presented below can be performed even without this assumption and similar results can be obtained (Appendix B). Substituting $\Sigma_{i=1}^{i=n} c_i W^{y_i} p^{x_i}$ for $T_o(W,p)$ in Equation 2.10, we obtain the following equation:

$$\begin{aligned} \Sigma_{i=1}^{i=n} c_i x_i W^{y_i} p^{x_i - 1} \quad &= \quad \frac{W + \Sigma_{i=1}^{i=n} c_i W^{y_i} p^{x_i}}{p}. \\ W \quad &= \quad \Sigma_{i=1}^{i=n} c_i (x_i - 1) W^{y_i} p^{x_i}. \end{aligned} \tag{2.11}$$

For the overhead function described above, Equation 2.11 determines the relationship between W and p for minimizing T_P provided that T_o grows faster than $\Theta(p)$. Because of the nature of Equation 2.11, it may not always be possible to express p as a function of W in a closed form. So we solve Equation 2.11, considering one R.H.S. term at a time and ignoring the rest. If the ith term is being considered, the relation $W = c_i(x_i - 1)W^{y_i} p^{x_i}$ yields

$$p = \left(\frac{W^{1-y_i}}{c_i(x_i - 1)}\right)^{\frac{1}{x_i}} = \Theta(W^{\frac{1-y_i}{x_i}}). \tag{2.12}$$

It can be shown (Appendix C) that among all the i solutions for p obtained in this manner, the speedup is maximum for any given W when $p = \Theta(W^{(1-y_j)/x_j})$ where $(1 - y_j)/x_j \leq (1 - y_i)/x_i$ for all i $(1 \leq i \leq n)$. We call the jth term of T_o the **dominant term** if the value of $(1 - y_j)/x_j$ is the least among all values $(1 - y_i)/x_i$ $(1 \leq i \leq n)$ because this is the term that determines the order of p_0, or the asymptotic the solution to Equation 2.10 for large values of W and p. If jth term is the dominant term of T_o, then solving Equation 2.11 with respect to the jth term on the R.H.S. yields the following approximate expression for p_0 for large values of W:

$$p_0 \approx (\frac{W^{1-y_j}}{c_j(x_j-1)})^{\frac{1}{x_j}}. \tag{2.13}$$

The value of p_0 thus obtained can be used in the expression for T_P to determine the minimum parallel execution time for a given W. The value of p_0, when plugged in the expression for efficiency, yields the following:

$$\begin{aligned} E_0 &= \frac{W}{W+T_o}, \\ E_0 &\approx \frac{W}{W+c_jW^{y_j}(\frac{W^{\frac{1-y_j}{x_j}}}{(c_jx_j-c_j)^{\frac{1}{x_j}}})^{x_j}}, \\ E_0 &\approx 1-\frac{1}{x_j}. \end{aligned} \tag{2.14}$$

Note that the above analysis holds *only* if x_j, the exponent of p in the dominant term of T_o is greater than 1. If $x_j \leq 1$, then the asymptotically highest term in T_o (i.e., $c_jW^{y_j}p^{x_j}$) is less than or equal to $\Theta(p)$ and the results for the case when $T_o \leq \Theta(p)$ apply.

Equations 2.13 and 2.14 yield the mathematical values of p_0 and E_0 respectively. But the derived value of p_0 may exceed $C(W)$. So in practice, at the point of peak performance (in terms of maximum speedup or minimum execution time), the number of processors p_{max} is given by $min(p_0, C(W))$ for a given W. Thus it is possible that $C(W)$ of a parallel algorithm may determine the minimum execution time rather than the mathematically derived conditions. For example, consider the implementation of Floyd's algorithm described in [88] for finding shortest paths in a graph. In this algorithm, the $N \times N$ adjacency matrix of the graph is striped among p processors such that each processor stores N/p full rows of the matrix. The problem size W here is given by N^3 for finding all to all shortest paths on an N-node graph. In each of the N iterations of this algorithm, a processor broadcasts a row of length N of the adjacency matrix of the graph to every other processor. As shown in [88], if the p processor are connected in a mesh configuration with cut-through routing, the total overhead due to this communication is given by $T_o = t_sNp^{1.5} + t_w(N+\sqrt{p})Np$. Here t_s and t_w are constants related to message startup time and the speed of message transfer respectively. Since t_w is often very small compared to t_s,

$$T_o = (t_s+t_w)Np^{1.5} + t_wN^2p,$$

$$\begin{aligned} T_o &\approx t_s N p^{1.5} + t_w N^2 p, \\ T_o &\approx t_s W^{1/3} p^{1.5} + t_w W^{2/3} p. \end{aligned}$$

From Equation 2.11, p_0 is equal to $(\frac{W^{2/3}}{.5t_s})^{2/3} \approx \frac{1.59N^{4/3}}{t_s^{2/3}}$. But since at most N processors can be used in this algorithm, $p_{max} = min(C(W), p_0) = N$. The minimum execution time for this parallel system is therefore $N^2 + t_s N^{1.5} + t_w N^2$ for $p_{max} = N$.

If working on a 100 node graph, then the speedup will peak at $p = N = 100$ and for $t_s = 1$ and $t_w = 0.1$, the speedup will be 83.33 resulting in an efficiency of 0.83 at the point of peak performance.

It is also possible for two parallel systems to have the same T_o (and hence the same p_0) but different $C(W)$s. In such cases, an analysis of the overhead function might mislead one into believing that the two parallel systems are equivalent in terms of maximum speedup and minimum execution time. For example, consider a different parallel system consisting of another variation of Floyd's algorithm discussed in [88] and a wrap-around mesh with store-and-forward routing. In this algorithm, the $N \times N$ adjacency matrix is partitioned into p sub-blocks of size $N/\sqrt{p} \times N/\sqrt{p}$ each, and these sub-blocks are mapped on a p processor mesh. In this version of Floyd's algorithm, a processor broadcasts $N/\sqrt{p}$ elements among $\sqrt{p}$ processors in each of the N iterations. As shown in [88], this results in a total overhead of $T_o = t_s N p^{1.5} + t_w N^2 p$. Since the expression for T_o is same as that in the previous example, $p_0 = 1.59N^{4/3}/t_s^{2/3}$ again. But $C(W)$ for the checkerboard version of the algorithm is $W^{2/3} = N^2$. Therefore $p_{max} = p_0$ in this case as $p_0 < C(W)$.

For $t_s = 1$ and $t_w = 0.1$, Equation 2.11 yields a value of $p_o = 738$ for a 100 node graph. The speedup peaks with 738 processors at a value of 246, but the efficiency at this peak speedup is only 0.33.

The above example illustrates the case when the speedup peaks at $p = p_o$. The algorithm in the earlier example with striped partitioning on a cut-through mesh has exactly the same T_o and hence the same p_o, but the speedup peaks at $p = C(W)$ because $C(W) < p_o$. Thus the two parallel systems described in these examples are significantly different in terms of their peak performances, although their overhead functions are the same.

Figures 2.3.1 and 2.3.1 graphically depict T_P as a function of p corresponding to Floyd's algorithm with stripe partitioning on a cut-through mesh and with checkerboard partitioning on a

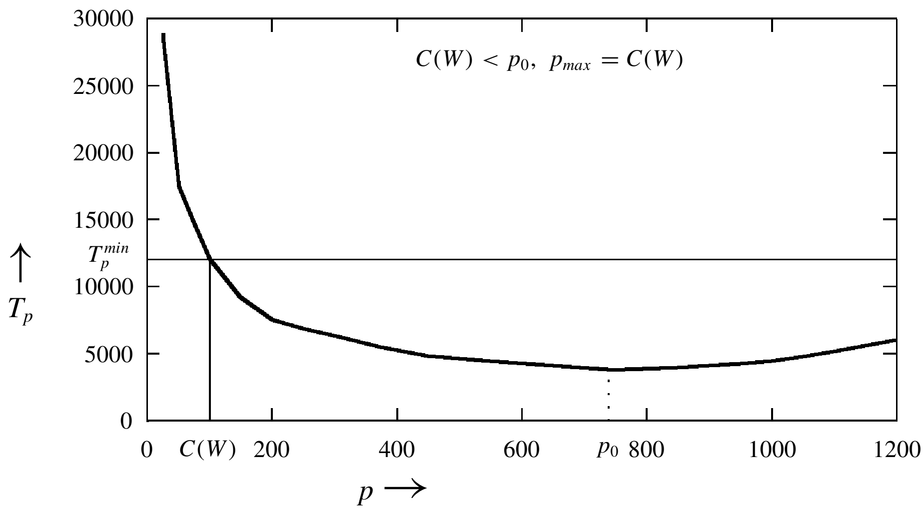


Figure 2.3: T_P verses p curve for $T_o > \Theta(p)$ showing T_P^{min} when $C(W) < p_0$.

store-and-forward mesh, respectively.

2.3.2 Minimizing T_P and the Isoefficiency Function

In this section we show that for a wide class of overhead functions, studying a parallel system at its peak performance in terms of the speedup is equivalent to studying its behavior at a fixed efficiency. The isoefficiency metric [81, 51, 84] comes in as a handy tool to study the fixed efficiency characteristics of a parallel system. The isoefficiency function relates the problem size to the number of processors necessary for an increase in speedup in proportion to the number of processors used. If a parallel system incurs a total overhead of $T_o(W, p)$ while solving a problem of size W on p processors, the efficiency of the system is given by $E = 1/(1 + T_o(W, p)/W)$. In order to maintain a constant efficiency, $W \propto T_o(W, p)$ or $W = KT_o(W, p)$ must be satisfied, where $K = E/(1 - E)$ is a constant depending on the efficiency to be maintained. This is the central relation that is used to determine isoefficiency as a function of p. From this equation, the problem size W can usually be obtained as a function of p by algebraic manipulations. If the problem size W needs to grow as fast as $f_E(p)$ to maintain an efficiency E, then $f_E(p)$ is defined to be the isoefficiency function of

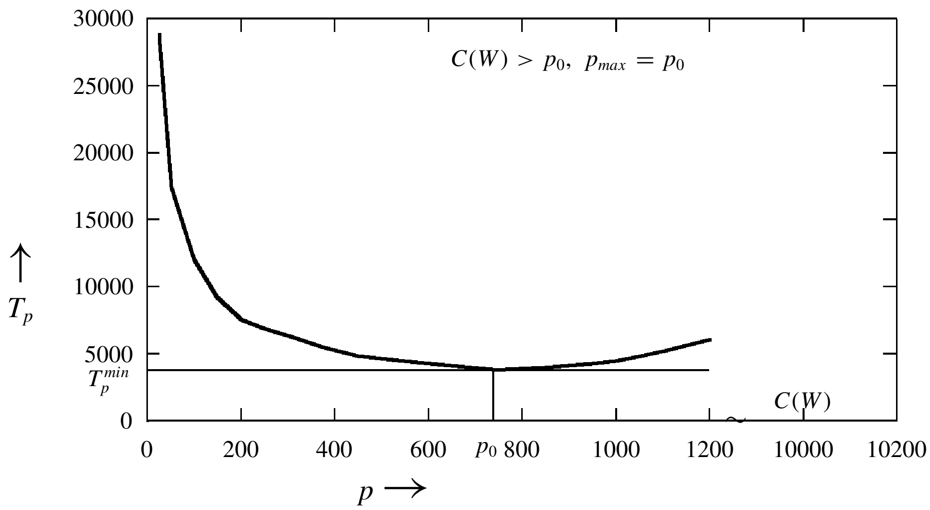


Figure 2.4: T_P verses p curve for $T_o > \Theta(p)$ showing T_P^{min} when $C(W) > p_0$.

the parallel algorithm-architecture combination for efficiency E.

We now show that unless $p_{max} = C(W)$ for a parallel system, a unique efficiency is attained at the point of peak performance. This value of E depends only on the characteristics of the parallel system (i.e., the type of overhead function for the algorithm-architecture combination) and is independent of W or T_P. For the type of overhead function assumed in Case II, the following relation determines the isoefficiency function for an efficiency E:

$$W = \frac{E}{1-E}\Sigma_{i=1}^{i=n} c_i W^{y_i} p^{x_i}. \tag{2.15}$$

Clearly, the above equation has the same form as Equation 2.11, but has different constants. The dominant term on the R.H.S. will yield the relationship between W and p in a closed form in both the equations. If this is the jth term, then both the equations will become equivalent asymptotically if their jth terms are same. This amounts to operating at an efficiency that is given by the following relation obtained by equating the coefficients of the jth terms of Equations 2.11 and 2.15.

$$\begin{aligned} \frac{E}{1-E}c_j &= c_j(x_j - 1), \\ E &= 1 - \frac{1}{x_j}. \end{aligned}$$

The above equation is in conformation with Equation 2.14. Once we know that working at the point of peak performance amounts to working at an efficiency of $1 - 1/x_j$, then, for a given W, we can find the number of processors at which the performance will peak by using the relation $1 - 1/x_j = W/(W + T_o(W, p))$.

As discussed in [81, 51], the relation between the problem size and the maximum number of processors that can be used in a cost-optimal fashion for solving the problem is given by the isoefficiency function. Often, using as many processors as possible results in a non-cost-optimal system. For example, adding n numbers on an n-processor hypercube takes $\Theta(\log n)$ time, which is the minimum execution time for this problem. This is not a cost optimal parallel system because $W = \Theta(N) < pT_P = \Theta(n \log n)$. An important corollary of the result presented in this section is that for the parallel systems for which the relationship between the problem size and the number of processors for maximum speedup (minimum execution time) is given by the isoefficiency function, the asymptotic minimum execution time can be attained in a cost-optimal fashion. For instance, if $\Theta(n/\log n)$ processors are used to add n numbers on a hypercube, the parallel system will be cost-optimal and the parallel execution time will still be $\Theta(\log n)$.

Note that the correspondence between the isoefficiency function and the relation between W and p for operating at minimum T_P will fail if the x_j in the dominant term is less than or equal to 1. In this case, a term other than the one that determines the isoefficiency function will determine the condition for minimum T_P.

Summary of Results

At this point we state the important results of this section.

- *For parallel algorithms with $T_o \leq \Theta(p)$, the maximum speedup is obtained at $p = C(W)$ and for algorithms with $T_o > \Theta(p)$, the maximum speedup is obtained at $p = min(p_o, C(W))$, where p_0 for a given W is determined by solving Equation 2.11.*

- *For the parallel algorithms with T_o of the form described in Case II, if the jth term is the dominant term in the expression for T_o and $x_j > 1$, then the efficiency at the point of maximum speedup always remains the same irrespective of the problem size, and is* *given by $E = 1 - 1/x_j$.*

- *For the parallel algorithms satisfying the above conditions, the relationship between the problem size and the number of processors at which the speedup is maximum for that problem size, is given by the isoefficiency function for $E = 1 - 1/x_j$, unless $p_{max} = C(W)$.*

2.3.3 Minimizing $p(T_P)^r$

From the previous sections, it is clear that operating at a point where T_P is minimum might not be a good idea because for some parallel systems the efficiency at this point might be low. On the other hand, the maximum efficiency is always attained at $p = 1$ which obviously is the point of minimum speedup. Therefore, in order to achieve a balance between speedup and efficiency, several researchers have proposed to operate at a point where the value of $p(T_P)^r$ is minimized for some constant r ($r \geq 1$) and for a given problem size W [36, 34, 134]. It can be shown [134] that this corresponds to the point where ES^{r-1} is maximized for a given problem size.

$$p(T_P)^r = pT_P(\frac{W}{S})^{r-1} = \frac{W^r}{ES^{r-1}}.$$

Thus $p(T_P)^r$ will be minimum when ES^{r-1} is maximum for a given W and by minimizing $p(T_P)^r$, we are choosing an operating point with a concern for both speedup and efficiency, their relative weights being determined by the value of r. Now let us locate the point where $p(T_P)^r$ is minimum.

$$p(T_P)^r = p(\frac{T_e + T_o}{p})^r = p^{1-r}(T_e + T_o)^r.$$

Again, as in the previous section, the following two cases arise:

Case I: $T_o \leq \Theta(p^{(r-1)/r})$

Since $p(T_P)^r = p^{1-r}(T_e + T_o)^r = (T_e p^{(1-r)/r} + T_o p^{(1-r)/r})^r$, if $T_o \leq \Theta(p^{(r-1)/r})$ then the overall power of p in the expression for $p(T_P)^r$ will become negative and hence its value will mathematically tend to some lower bound as $p \rightsquigarrow \infty$. Thus using as many processors as are feasible will lead to minimum $p(T_P)^r$. In other words, for this case, $p(T_P)^r$ is minimum when $p = C(W)$.

Case II: $T_o > \Theta(p^{(r-1)/r})$

If T_o grows faster than $\Theta(p^{(r-1)/r})$, then we proceed as follows. In order to minimize $p(T_P)^r$, $\frac{d}{dp}\, p(T_P)^r$ should be equal to zero; i.e.,

$$
\begin{aligned}
(1-r)p^{-r}(T_e + T_o)^r + rp^{(1-r)}(T_e + T_o)^{(r-1)}\frac{d}{dp}T_o &= 0, \\
\frac{d}{dp}T_o &= \frac{r-1}{r}T_P.
\end{aligned}
\tag{2.16}
$$

We choose the same type of overhead function as in Case II of Section 2.3.1. Substituting $\Sigma_{i=1}^{i=n} c_i W^{y_i} p^{x_i}$ for T_o in Equation 2.16, we get the following equation:

$$
\begin{aligned}
\Sigma_{i=1}^{i=n} c_i x_i W^{y_i} p^{x_i - 1} &= \frac{r-1}{rp}(W + \Sigma_{i=1}^{i=n} c_i W^{y_i} p^{x_i}), \\
W &= \Sigma_{i=1}^{i=n} c_i (\frac{r x_i}{r-1} - 1) W^{y_i} p^{x_i}.
\end{aligned}
\tag{2.17}
$$

Now even the number of processors for which $p(T_P)^r$ is minimum could exceed the value of p that is permitted by the degree of concurrency of the algorithm. In this case the minimum possible value for $p(T_P)^r$ will be obtained when $C(W)$ processors are used. For example, consider a simple algorithm described in [54] for multiplying two $N \times N$ matrices on a $\sqrt{p} \times \sqrt{p}$ wrap-around mesh. As the first step of the algorithm, each processor acquires all those elements of both the matrices that are required to generate the N^2/p elements of the product matrix which are to reside in that processor. For this parallel system, $W = N^3$ and $T_o = t_s p\sqrt{p} + t_w N^2 \sqrt{p}$. For determining the operating point where $p(T_P)^2$ is minimum, we substitute $n = 2$, $r = 2$, $c_1 = t_s$, $c_2 = t_w$, $x_1 = 1.5$, $x_2 = 0.5$, $y_1 = 0$ and $y_2 = 2/3$ in Equation 2.17. This substitution yields the relation $W = 2t_s p^{1.5}$ for determining the required operating point. In other words, the number of processors p_0 at which $p(T_P)^2$ is minimum is given by $p_0 = (W/2t_s)^{2/3} = N^2/(2t_s)^{2/3}$. But the maximum number of processors that this algorithm can use is only N^2. Therefore, for $t_s < .5$, $p_0 > C(W)$ and hence $C(W)$ processors should be used to minimize pT_P^2.

2.3.4 Minimizing $p(T_P)^r$ and the Isoefficiency Function

In this subsection we show that for a wide class of parallel systems, even minimizing $p(T_P)^r$ amounts to operating at a unique efficiency that depends only on the overhead function and the value of r. In other words, for a given W, $p(T_P)^r$ is minimum for some value of p and the relationship between W and this p for the parallel system is given by its isoefficiency function for a unique value of efficiency that depends only on r and the type of overhead function. Equation 2.17, which gives the relationship between W and p for minimum $p(T_P)^r$, has the same form as Equation 2.15 that determines the isoefficiency function for some efficiency E. If the jth terms of the R.H.S.s of Equations 2.15 and 2.17 dominate (and $x_j > (r-1)/r$), then the efficiency at minimum $p(T_P)^r$ can be obtained by equating the corresponding constants; i.e., $Ec_j/(1-E)$ and $c_j(rx_j/(r-1)-1)$. This yields the following expression for the value of efficiency at the point where $p(T_P)^r$ is minimum:

$$E = 1 - \frac{r-1}{rx_j}. \tag{2.18}$$

We now give an example that illustrates how the analysis of Section 2.3.3 can be used for chosing an appropriate operating point (in terms of p) for a parallel algorithm to solve a problem instance of a given size. It also confirms the validity of Equation 2.18. Consider the implementation of the FFT algorithm on an MIMD hypercube using the binary-exchange algorithm. As shown in [56], for an N point FFT on p processors, $W = N \log N$ and $T_o = t_s p \log p + t_w N \log p$ for this algorithm. Taking $t_s = 2$, $t_w = 0.1$ and rewriting the expression for T_o in the form described in Case II in Section 2.3.1, we get the following:

$$T_o \approx 2p \log p + 0.1 \frac{W}{\log W} \log p.$$

Now suppose it is desired to minimize $p(T_P)^2$, which is equivalent to maximizing the ES product. Clearly, the first term of T_o dominates and hence putting $r = 2$ and $x_j = 1$ in Equation 2.18, an efficiency of 0.5 is predicted when $p(T_P)^2$ is minimized. An analysis similar to that in Case II in Section 2.3.3 will show that $p(T_P)^r$ will be minimum when $p \approx N/2$ is used.

If a 1024 point FFT is being attempted, then Table 2.1 shows that at $p = 512$ the ES product is indeed maximum and the efficiency at this point is indeed 0.5.

Again, just like in Section 2.3.2, there are exceptions to the correspondence between the isoefficiency function and the condition for minimum $p(T_P)^r$. If the jth term in Equation 2.15 determines the isoefficiency function and in Equation 2.17, $x_j < (r-1)/r$, then the coefficient of the jth term in Equation 2.17 will be zero or negative and some other term in Equation 2.17 will determine the relationship between W and p for minimum $p(T_P)^r$.

The following subsection summarizes the results of this section.

p	T_P	S	E	$E \times S$
128	99.6	103	.80	82.4
256	59.2	173	.68	117.6
384	46.1	222	.58	128.4
512	**39.8**	**257**	**.50**	**129.3**
640	36.1	284	.44	124.9
768	33.8	303	.39	116.2
896	32.2	318	.35	112.8
1024	31.0	330	.32	105.5

Table 2.1: Performance of FFT on a hypercube with $N = 1024$, $t_s = 2$ and $t_w = 0.1$.

Summary of Results

- *For parallel algorithms with $T_o \leq \Theta(p^{(r-1)/r})$, the minimum value for the expression $p(T_P)^r$ is attained at $p = C(W)$ and for algorithms with $T_o > \Theta(p^{(r-1)/r})$, it is attained at $p = min(C(W), p_0)$, where p_0 for a given W is obtained by solving Equation 2.17.*

- *For the parallel algorithms with T_o of the form described in Case II, if the jth term dominates in the expression for T_o and $x_j > (r-1)/r$, then the efficiency at the point of minimum $p(T_P)^r$ always remains same irrespective of the problem size and is given by $E = 1 - (r-1)/rx_j$.*

- *For the parallel algorithms satisfying the above conditions, the relationship between the problem size and the number of processors at which $p(T_P)^r$ is minimum for that problem size, is given by the isoefficiency function for $E = 1 - (r-1)/rx_j$, provided $C(W) > p_0$ determined from Equation 2.17.*

In fact the results pertaining to minimization of T_P are special cases of the above results when $r \rightsquigarrow \infty$; i.e., the weight of p is zero with respect to T_P or the weight of E is zero with respect to S. Equation 2.11 can be derived from Equation 2.17 and Equation 2.14 from Equation 2.18 if $(r-1)/r$ is replaced by $lim_{r \rightsquigarrow \infty}(r-1)/r = 1$.

2.3.5 Significance in the Context of Related Research

In this section we discuss how this chapter encapsulates several results that have appeared in the literature before and happen to be special cases of the more general results presented here.

Flatt and Kennedy [36, 35] show that if the overhead function satisfies certain mathematical properties, then there exists a unique value p_0 of the number of processors for which T_P is minimum for a given W. A property of T_o on which their analysis depends heavily is that $T_o > \Theta(p)$.[2] This assumption on the overhead function limits the range of the applicability of their analysis. As seen in the example in Section 2.2 (Equation 2.5), there exist parallel systems that do not obey this condition, and in such cases the point of peak performance is determined by the degree of concurrency of the algorithm being used.

Flatt and Kennedy show that the maximum speedup attainable for a given problem is upper-bounded by $1/(\frac{d}{dp}(pT_P))$ at $p = p_0$. They also show that the better a parallel algorithm is (i.e., the slower T_o grows with p), the higher is the value of p_0 and the lower is the value of efficiency obtained at this point. Equations 2.13 and 2.14 provide results similar to Flatt and Kennedy's. But the analysis in [36] tends to conclude the following - (i) if the overhead function grows very fast with respect to p, then p_0 is small, and hence parallel processing cannot provide substantial speedups; (ii) if the overhead function grows slowly (i.e., closer to $\Theta(p)$), then the overall efficiency is very poor at $p = p_0$. Note that if we keep improving the overhead function, the mathematically derived value of p_0 will ultimately exceed the limit imposed by the degree of concurrency on the number of processors that can be used. Hence, in practice no more than $C(W)$ processors will be used. Thus, in this situation, the theoretical value of p_0 and the efficiency at this point does not serve a useful purpose because the point of peak performance efficiency cannot be worse than $W/(W + T_o(W, C(W)))$. For instance, Flatt and Kennedy's analysis will predict identical values of p_{max} and efficiency at this operating point for the parallel systems described in the examples in

---

[2] T_o, as defined in [36], is the overhead incurred per processor when all costs are normalized with respect to $W = 1$. So in the light of the definition of T_o in this chapter, the actual mathematical condition of [36], that T_o is an increasing nonnegative function of p, has been translated to the condition that T_o grows faster than $\Theta(p)$.

Section 2.3.1 because their overhead functions are identical. But as we saw in these examples, this is not the case because the the value of $C(W)$ in the two cases is different.

In [98], Marinescu and Rice develop a model to describe and analyze a parallel computation on a MIMD machine in terms of the number of threads of control p into which the computation is divided and the number events $g(p)$ as a function of p. They consider the case where each event is of a fixed duration θ and hence $T_o = \theta g(p)$. Under these assumptions on T_o, they conclude that with increasing number of processors, the speedup saturates at some value if $T_o = \Theta(p)$, and it asymptotically approaches zero if $T_o = \Theta(p^m)$, where $m \geq 2$. The results of Section 2.3.1 are generalizations of these conclusions for a wider class of overhead functions. Case I in this section shows that the speedup saturates at some maximum value if $T_o \leq \Theta(p)$, and Case II shows that speedup will attain a maximum value and then it will drop monotonically with p if $T_o > \Theta(p)$.

Usually, the duration of an event or a communication step θ is not a constant as assumed in [98]. In general, both θ and T_o are functions of W and p. If T_o is of the form $\theta g(p)$, Marinescu and Rice [98] derive that the number of processors that will yield maximum speedup will be given by $p = (W/\theta + g(p))/g'(p)$, which can be rewritten as $\theta g'(p) = (W + \theta g(p))/p$. It is easily verified that this is a special case of Equation 2.10 for $T_o = \theta g(p)$.

Worley [146] showed that for certain algorithms, given a certain amount of time T_P, there will exist a problem size large enough so that it cannot be solved in time T_P, no matter how many processors are used. In Section 2.3.1, we describe the exact nature of the overhead function for which a lower bound exists on the execution time for a given problem size. This is exactly the condition for which, given a fixed time, an upper bound will exist on the size of the problem that can be solved within this time. We show that for a class of parallel systems, the relation between problem size W and the number of processors p at which the parallel execution time T_P is minimized, is given by the isoefficiency function for a particular efficiency.

Several other researchers have used the minimum parallel execution time of a problem of a given size for analyzing the performance of parallel systems [105, 97, 108]. Nussbaum and Agarwal [108] define scalability of an architecture for a given algorithm as the ratio of the algorithm's asymptotic speedup when run on the architecture in question to its corresponding asymptotic speedup when run on an EREW PRAM. The asymptotic speedup is the maximum obtainable speedup for a given problem size if an unlimited number of processors is available. For a fixed problem size, the scalability of the parallel system, according to their metric, depends directly on the minimum T_P for the system. For the class of parallel systems for which the correspondence between the isoefficiency function and the relation between W and p for minimizing T_P exists, Nussbaum and Agarwal's scalability metric will yield results identical to those predicted by the isoefficiency function on the behavior of these parallel systems.

Eager et al. [34] and Tang and Li [134] have proposed a criterion of optimality different from optimal speedup. They argue that the optimal operating point should be chosen so that a balance is struck between efficiency and speedup. It is proposed in [34] that the "knee" of the execution time verses efficiency curve is a good choice of the operating point because at this point the incremental benefit of adding processors is roughly 0.5 per processor, or, in other words, efficiency is 0.5. Eager *et. al.* and Tang and Li also conclude that for $T_o = \Theta(p)$, this is also equivalent to operating at a point where the ES product is maximum or $p(T_P)^2$ is minimum. This conclusion in [34, 134] is a special case of the more general case that is captured in Equation 2.18. If we substitute $x_j = 1$ in Equation 2.18 (which is the case if $T_o = \Theta(p)$), it can seen that we indeed get an efficiency of 0.5 for $r = 2$. In general, operating at the optimal point or the "knee" referred to in [34] and [134] for a parallel system with $T_o = \Theta(p^{x_j})$ will be identical to operating at a point where $p(T_P)^r$ is minimum, where $r = 2/(2 - x_j)$. This is obtained from Equation 2.18 for $E = 0.5$. Minimizing $p(T_P)^r$ for $r > 2/(2 - x_j)$ will result in an operating point with efficiency lower than 0.5 but a higher speedup. On the other hand, minimizing $p(T_P)^r$ for $r < 2/(2 - x_j)$ will result in an operating point with efficiency higher than 0.5 and a lower speedup.

Chapter 3

SCALABILITY ANALYSIS OF SOME NUMERICAL ALGORITHMS

In this chapter, we present a comprehensive scalability analysis of parallel algorithms for fast Fourier transform (FFT), dense matrix multiplication, sparse matrix-vector multiplication, and the preconditioned conjugate gradient (PCG) algorithm. Wherever applicable, we analyze the impact of algorithmic features, as well as, hardware dependent parameters on scalability of parallel systems. We discuss the cost-performance tradeoffs and the cost-effectiveness of various architectures. In most cases, we present experimental results on commercially available parallel computers such as the nCUBE and CM-5 to support our analysis.

Our scalability analysis of these numerical algorithms provides many interesting insights into their behavior on parallel computers. For example, we show that a commonly used parallel FFT algorithm that was thought to be ideally suited for hypercubes has a limit on the achievable efficiency that is determined by the ratio of CPU speed and communication bandwidth of the hypercube channels. Efficiencies higher than this threshold value can be obtained if the problem size is increased very rapidly. If the hardware supports cut-through routing, then this threshold can also be overcome by using a series of successively less scalable parallel formulations. In the context of dense matrix multiplication, we show that special hardware permitting simultaneous communication on all the ports of the processors does not improve the overall scalability on a hypercube. We discuss the dependence of scalability on technology dependent factors such as communication and computation speeds and show that under certain conditions, it may be better to use a parallel computer with k-fold as many processors rather than one with the same number of processors, each k-fold as fast. In the case of parallel PCG algorithm, we found that the use of a truncated Incomplete Cholesky (IC) preconditioner, which was considered unsuitable for parallel computers, can actually improve the scalability over a parallel CG with diagonal or no preconditioning. As a result, a parallel formulation of the PCG algorithm with this IC preconditioner may execute faster than that with a simple diagonal preconditioner even if the latter runs faster in a serial implementation for a given problem.

3.1 Fast Fourier Transform

Fast Fourier Transform plays an important role in several scientific and technical applications. Some of the applications of the FFT algorithm include Time Series and Wave Analysis, solving Linear Partial Differential Equations, Convolution, Digital Signal Processing and Image Filtering, etc. Hence, there has been a great interest in implementing FFT on parallel computers [11, 17, 29, 56, 72, 106, 133, 13, 74, 19, 25, 3].

3.1.1 The FFT Algorithm

Figure 3.1 outlines the serial Cooley-Tukey algorithm for an n point single dimensional unordered radix-2 FFT adapted from [4, 116]. $\mathbf{X}$ is the input vector of length n ($n = 2^r$ for some integer r) and $\mathbf{Y}$ is its Fourier Transform. ω^k denotes the complex number $e^{j2\pi/nk}$, where $j = \sqrt{-1}$. More generally, ω is the primitive nth root of unity and hence ω^k could be thought of as an element of the finite commutative ring of integers modulo n. Note that in the lth ($0 \leq l < r$) iteration of the loop starting on Line 3, those elements of the vector are combined whose indices differ by 2^{r-l-1}. Thus the pattern of the combination of these elements is identical to a butterfly network.

The computation of each $R[i]$ in Line 8 is independent for different values of i. Hence p processors ($p \leq n$) can be used to compute the n values on Line 8 such that each processor computes n/p values. For the sake of simplicity, assume that p is a power of 2, or more precisely, $p = 2^d$ for some integer d such that $d \leq r$. To obtain good performance on a parallel machine, it is important to distribute the elements of vectors R and S among the processors in a way that keeps the interprocess communication to a minimum. In a parallel implementation, there are two main contributors to the data communication cost—the message startup time t_s and the per-word transfer time t_w. In the following subsections, we present two parallel formulations of the Cooley-Tukey algorithm. As the analysis of Section 3.1.2 will show, each of these formulations minimizes the cost due to one of these constants.

1. **begin**
2.   **for** i := 0 **to** n - 1 **do** R[i] := $\mathbf{X}_i$;
3.   **for** l := 0 **to** r - 1 **do**
4.   **begin**
5.   **for** i := 0 **to** n - 1 **do** S[i] := R[i];
6.   **for** i := 0 **to** n - 1 **do**
7.   **begin**

(* Let $(b_0 b_1 \cdots b_{r-1})$ be the binary representation of i *)

8. $\mathrm{R}[(b_0 \cdots b_{r-1})] := \mathrm{S}[(b_0 \cdots b_{l-1} 0 b_{l+1} \cdots b_{r-1})] + \omega^{(b_l b_{l-1} \cdots b_0 0 \cdots 0)} \mathrm{S}[(b_0 \cdots b_{l-1} 1 b_{l+1} \cdots b_{r-1})]$;
9.   **end**;
10.   **end**;
11. **end**.

Figure 3.1: The Cooley-Tukey algorithm for single dimensional unordered FFT.

The Binary-Exchange Algorithm

In the most commonly used mapping that minimizes communication for the binary-exchange algorithm [81, 5, 11, 17, 29, 72, 106, 133, 116, 94], if $(b_0 b_1 \cdots b_{r-1})$ is the binary representation of i, then for all i, $R[i]$ and $S[i]$ are mapped to processor number $(b_0 \cdots b_{d-1})$.

With this mapping, processors need to communicate with each other in the first d iterations of the main loop (starting at line 3) of the algorithm. For the remaining $r - d$ iterations of the loop, the elements to be combined are available on the same processor. Also, in the lth $(0 \leq l < d)$ iteration, all the n/p values required by a processor are available to it from a single processor; i.e., the one whose number differs from it in the lth most significant bit.

The Transpose Algorithm

Let the vector $\mathbf{X}$ be arranged in an $\sqrt{n} \times \sqrt{n}$ two dimensional array in row major order. An unordered Fourier Transform of $\mathbf{X}$ can be obtained by performing an unordered radix-2 FFT over all the rows of this 2-D array followed by an unordered radix-2 FFT over all the columns. The row FFT corresponds to the first $\log n/2$ iterations of the FFT over the entire vector $\mathbf{X}$ and the column FFT corresponds to the remaining $\log n/2$ iterations. In a parallel implementation, this $\sqrt{n} \times \sqrt{n}$ can be mapped on to p processors ($p \leq \sqrt{n}$) such that each processor stores $\sqrt{n}/p$ rows of the array. Now the FFT over the rows can be performed without any inter-processor communication. After this step, the 2-D array is transposed and an FFT of all the rows of the transpose is computed. The only step that requires any inter-processor communication is transposing an $\sqrt{n} \times \sqrt{n}$ array on p processors.

The algorithm described above is a two-dimensional transpose algorithm because the data is arranged in a two-dimensional array mapped onto a one-dimensional array of processors. In general, a q-dimensional transpose algorithm can be formulated along the above lines by mapping a q-dimensional array of data onto a $(q-1)$-dimensional array of processors. The binary-exchange algorithm is nothing but a a $(\log p + 1)$-dimensional algorithm. In this chapter, we confine our discussion to the two extremes (2-D transpose and binary-exchange) of this sequence of algorithms. More detailed discussion can be found in [81].

3.1.2 Scalability Analysis of the Binary-Exchange Algorithm for Single Dimensional Radix-2 Unordered FFT

We assume that the cost of one unit of computation (i.e., the cost of executing line 8 in Figure 3.1) is t_c. Thus for an n point FFT, $W = t_c n \log n$. As discussed in Section 3.1.1, the parallel formulation of FFT can use at most n processors. As p is increased, the additional processors will not have any work to do after p exceeds n. So in order to prevent the efficiency to diminish with increasing p, n must grow at least as p so that no processor remains idle. If n increases linearly with p, then W ($= t_c n \log n$) must grow in proportion to $t_c p \log p$. This gives us a lower bound of $\Omega(p \log p)$ on the isoefficiency function for the FFT algorithm. This figure is independent of the parallel architecture and is a function of the inherent parallelism in the algorithm. The overall isoefficiency function of this algorithm can be worse depending upon how the overall overhead T_o increases with p.

Several factors may contribute to T_o in a parallel implementation of FFT. The most significant of these overheads is due to data communication between processors. As discussed in Section 3.1.1, the p processors communicate in pairs in d ($d = \log p$) of the r ($r = \log n$) iterations of the loop starting on Line 3 of Figure 3.1. Let z_l be the distance between the communicating processors in the lth iteration. If the distances between all pairs of communicating processors are not the same, then z_l is the maximum distance between any pair. In this subsection, assume that no part of the various data paths coincides. Since each processor has n/p words, the total communication cost for a parallel computer with store-and-forward routing is given by the following equation:

$$T_o = p \times \Sigma_{l=0}^{l=d-1}(t_s + t_w \frac{n}{p} z_l). \tag{3.1}$$

Isoefficiency on a Hypercube

As discussed in Section 3.1.1, in the lth iteration of the loop beginning on Line 3 of Figure 3.1, data messages containing n/p words are exchanged between the processors whose binary representations are different in the lth most significant bit position ($l < d = \log p$). Since all these pairs of processors with addresses differing in one bit position are directly connected in a hypercube configuration, Equation 3.1 becomes:

$$\begin{aligned} T_o &= p \times \Sigma_{l=0}^{l=(\log p)-1}(t_s + t_w \frac{n}{p}), \\ T_o &= t_s p \log p + t_w n \log p. \end{aligned} \tag{3.2}$$

If p increases, then in order to maintain the efficiency at some value E, W should be equal to KT_o, where $K = E/(1 - E)$. Since $W = t_c n \log n$, n must grow such that

$$t_c n \log n = K(t_s p \log p + t_w n \log p). \tag{3.3}$$

Clearly, the isoefficiency function due to the first term in T_o, is given by:

$$W = K t_s p \log p. \tag{3.4}$$

The requirement on the growth of W (to maintain a fixed efficiency) due to the second term in T_o is more complicated. If this term requires W to grow at a rate less than $\Theta(p \log p)$, then it can be ignored in favor of the first term. On the other hand, if this term requires W to grow at a rate higher than $\Theta(p \log p)$, then the first term of T_o can be ignored.

Balancing W against the second term only yields the following:

$$
\begin{aligned}
&nt_c \log n = K t_w n \log p,\\
&\log n = K \tfrac{t_w}{t_c} \log p,\\
&n = p^{K \frac{t_w}{t_c}}.
\end{aligned}
$$

This leads to the following isoefficiency function (due to the second term of T_o):

$$W = K t_w \times p^{K t_w / t_c} \times \log p. \tag{3.5}$$

This growth is less than $\Theta(p \log p)$ as long as $K t_w / t_c < 1$. As soon as this product exceeds 1, the overall isoefficiency function is given by Equation 3.5. Since the binary-exchange algorithm involves only nearest neighbor communication on a hypercube, the total overhead T_o, and hence the scalability, cannot be improved by using cut-through routing.

Efficiency Threshold

The isoefficiency function given by Equation 3.5 deteriorates very rapidly with the increase in the value of $K t_w / t_c$.

In fact the efficiency corresponding to $K t_w / t_c = 1$, (i.e., $E = t_c/(t_c + t_w)$) acts somewhat as a threshold value. For a given hypercube with fixed t_c and t_w, efficiencies up to this values can be obtained easily. But efficiencies much higher than this threshold can be obtained only if the problem size is extremely large. For example, consider the computation of an n point FFT on a p-processor hypercube on which $t_w = t_c$. The isoefficiency function of the parallel FFT on this machine is $K t_w \times p^K \times \log p$. Now for $K < 1$ (i.e., $E \leq 0.5$) the overall isoefficiency is $\Theta(p \log p)$, but for $E > 0.5$, the isoefficiency function is much worse. If $E = 0.9$, then $K = 9$ and hence the isoefficiency function becomes $\Theta(p^9 \log p)$. As another example, consider the computation of an n point FFT on a p-processor hypercube on which $t_w = 2t_c$. Now the threshold efficiency is 0.33. The isoefficiency function for $E = 0.5$ is $\Theta(p^2 \log p)$ and for $E = 0.9$, it becomes $\Theta(p^{18} \log p)$.

The above examples show how the ratio of t_w and t_c effects the scalability and how hard it is to obtain efficiencies higher than the threshold determined by this ratio.

Isoefficiency on a Mesh

Assume that an n point FFT is being computed on a p-processor simple mesh ($\sqrt{p}$ rows $\times$ $\sqrt{p}$ columns) such that $\sqrt{p}$ is a power of 2. For example consider $p = 64$ such that processor 0,1,2,3,4,5,6,7 form the first row and processors 0,8,16,24,32,40,48,56 form the first column. Now during the execution of the algorithm, processor 0 will need to communicate with processors 1,2,4,8,16,32. All these communicating processors lie in the same row or the same column. More precisely, in $\log \sqrt{p}$ of the $\log p$ steps that require data communication, the communicating processors are in the same row, and in the remaining $\log \sqrt{p}$ steps, they are in the same column. The distance between the communicating processors in a row grows from one hop to $\sqrt{p}/2$ hops, doubling in each of the $\log \sqrt{p}$ steps. The communication pattern is similar in case of the columns. The reader can verify that this is true for all the communicating processors in the mesh. Thus, from Equation 3.1 we get:

$$\begin{aligned}
T_o &= p \times 2\Sigma_{l=0}^{l=(\log \sqrt{p})-1}(t_s + t_w \frac{n}{p} 2^l), \\
T_o &= 2p(t_s \log \sqrt{p} + t_w \frac{n}{p}(\sqrt{p} - 1)), \\
T_o &\approx t_s p \log p + 2t_w n \sqrt{p}. \qquad (3.6)
\end{aligned}$$

Balancing W against the first term yields the following equation for the isoefficiency function:

$$W = K t_s p \log p. \qquad (3.7)$$

Balancing W against the second term yields the following:

$$t_c n \log n = 2K t_w \times n \times \sqrt{p}.$$
$$\log n = 2K \tfrac{t_w}{t_c} \times \sqrt{p}.$$
$$n = 2^{2K \frac{t_w}{t_c} \times \sqrt{p}}.$$

Since the growth in W required by the third term in T_o is much higher than that required by the first two terms (unless p is very small), this is the term that determines the overall isoefficiency function which is given by the following equation:

$$W = 2K t_w \sqrt{p} \times 2^{2K \frac{t_w}{t_c} \sqrt{p}}. \qquad (3.8)$$

From Equation 3.8, it is obvious that the problem size has to grow exponentially with the number of processors to maintain a constant efficiency; hence the algorithm is not very scalable on a simple mesh. Any different mapping of input vector X on the processors does not reduce the communication overhead. It has been shown [135] that in any mapping, there will be at least one iteration in which the pairs of processors that need to communicate will be at least $\sqrt{p}/2$ hops apart. Hence the expression for T_o used in the above analysis cannot be improved by more than a factor of two.

3.1.3 Scalability Analysis of the Transpose Algorithm for Single Dimensional Radix-2 Unordered FFT

As discussed earlier in Section 3.1.1, the only data communication involved in this algorithm is the transposition of an $\sqrt{n} \times \sqrt{n}$ two dimensional array on p processors. It is easily seen that this involves the communication of a chunk of unique data of size n/p^2 between every pair of processors. This communication (known as *all-to-all personalized communication*) can be performed by executing the following code on each processor:

for $i = 1$ **to** p **do**
    send data to processor number ($self\_address \oplus i$)

It is shown in [71], that on a hypercube, in each iteration of the above code, each pair of communicating processors have a contention-free communication path. On a hypercube with store-and-forward routing, this communication will take $t_w(n/p)\log p + t_s p$ time. This communication term yields an overhead function which is identical to the overhead function of the binary exchange algorithm and hence this scheme does not offer any improvement over the binary exchange scheme. However, on a hypercube with cut-through routing, this can be done in time $t_w n/p + t_s p$, leading to an overhead function T_o given by the following equation:

$$T_o = t_w n + t_s p^2. \tag{3.9}$$

The first term of T_o is independent of p and hence, as p increases, the problem size must increase to balance the second communication term. For an efficiency E, this yields the following isoefficiency function, where $K = E/(1 - E)$:

$$W = K t_s p^2. \tag{3.10}$$

In the transpose algorithm, the mapping of data on the processors requires that $\sqrt{n} \geq p$. Thus, as p increases, n has to increase as p^2, or else some processors will eventually be out of work. This requirement imposes and isoefficiency function of $\Theta(p^2 \log p)$ due to the limited concurrency of the transpose algorithm. Since the isoefficiency function due to concurrency exceeds the isoefficiency function due to communication, the former (i.e., $\Theta(p^2 \log p)$) is also the overall isoefficiency function of the transpose algorithm on a hypercube.

As mentioned in Section 3.1.1, in this chapter we have confined our discussion of the transpose algorithm to the two-dimensional case. A generalized transpose algorithm and the related performance and scalability analysis can be found in [81].

3.1.4 Impact of Architectural and Algorithmic Variations on Scalability of FFT

In [56], we analyze the scalability of multidimensional FFTs, ordered FFT, and FFTs with radix higher than 2, and survey some other variations of the Cooley-Tukey algorithm. We find that within a small constant factor, the isoefficiency functions are the same as the ones derived in this chapter for the simplified case of unordered, radix-2, single dimensional FFT. The analysis in this chapter assumes store-and-forward routing on the mesh. In [56] we show that due to message contention, the expressions for the communication overhead (and hence, for the isoefficiency function too) on the mesh do not improve if cut-through or worm-hole routing is used. In [56] and [81], we also discuss the additional overhead that a parallel FFT algorithm might incur due to redundant computation of twiddle-factors. Our analysis shows that the asymptotic isoefficiency term due to this overhead is $\Theta(p \log p)$ and is subsumed by the isoefficiency function due to communication overhead.

3.1.5 Comparison between Binary-Exchange and Transpose Algorithms

As discussed earlier in this section, an overall isoefficiency function of $\Theta(p \log p)$ can be realized by using the binary exchange algorithm if the efficiency of operation is such that $K t_w / t_c \leq 1$. If the desired efficiency is such that $Kt_w/t_c = 2$, then the overall isoefficiency functions of both the binary-exchange and the transpose schemes are $\Theta(p^2 \log p)$. When $Kt_w/t_c > 2$, the transpose algorithm is more scalable than the binary-exchange algorithm and should be the algorithm of choice provided that $n \geq p^2$.

In the transpose algorithm described in Section 3.1.1, the data of size n is arranged in an $\sqrt{n} \times \sqrt{n}$ two dimensional array and is mapped on to a linear array of p processors[1] with $p = \sqrt{n}/k$, where k is a positive integer between 1 and $\sqrt{n}$. In a generalization of this method [81], the vector **X** can be arranged in an m-dimensional array mapped on to an $(m-1)$-dimensional logical array of p processors, where $p = n^{(m-1)/m}/k$. The 2-D transpose algorithm discussed in this section is a special case of this generalization with $m = 2$ and the binary-exchange algorithm is a special case for $m = (\log p + 1)$. A comparison of Equations 3.2 and 3.9 shows that the binary exchange algorithm minimizes the communication overhead due to t_s, whereas the 2-D transpose algorithm minimizes the overhead due to t_w. Also, the binary-exchange algorithm is highly concurrent and can use as many as n processors, whereas the concurrency of the 2-D transpose algorithm is limited to $\sqrt{n}$ processors. By selecting values of m between 2 and $(\log p + 1)$, it is possible to derive algorithms whose concurrencies and communication overheads due to t_s and t_w have intermediate values between those for the two algorithms described in this section. Under certain circumstances, one of these algorithms might be the best choice in terms of both concurrency and communication overheads.

3.1.6 Cost-Effectiveness of Mesh and Hypercube for FFT Computation

The scalability of a certain algorithm-architecture combination determines its capability to use increasing number of processors effectively. Many algorithms may be more scalable on costlier architectures. In such situations, one needs to consider whether it is better to have a larger parallel computer of a cost-wise more scalable architecture that is underutilized (because of poor efficiency), or to have a smaller parallel computer of a cost-wise less scalable architecture that is better utilized. For a given amount of resources, the aim is to maximize the overall performance which is proportional to the number of processors and the efficiency obtained on them. From the

[1] It is a logical linear array of processors which are physically connected in a hypercube network.

scalability analysis of Section 3.1.2, it can be predicted that the FFT algorithm will perform much poorly on a mesh as compared to a hypercube. On the other hand constructing a mesh multicomputer is cheaper than constructing a hypercube with the same number of processors. In this section we show that in spite of this, it might be more cost-effective to implement FFT on a hypercube rather than on a mesh.

Suppose that the cost of building a communication network for a parallel computer is directly proportional to the number of communication links. If we neglect the effect of the length of the links (i.e., $t_h = 0$) and assume that $t_s = 0$, then the efficiency of an n point FFT computation using the binary-exchange scheme is approximately given by $(1 + t_w \log p/(t_c \log n))^{-1}$ on a p processor hypercube and by $(1 + 2t_w\sqrt{p}/(t_c \log n))^{-1}$ on a p processor mesh. It is assumed here that t_w and t_c are same for both the computers. Now it is possible to obtain similar performance on both the computers if we make each channel on the mesh w wide (thus effectively reducing the per-word communication time to t_w/w), choosing w such that $2\sqrt{p}/w = \log p$. The cost of constructing these hypercube and mesh networks will be $p \log p$ and $4wp$ respectively, where $w = 2\sqrt{p}/\log p$. Since $8p\sqrt{p}/\log p$ is greater than $p \log p$ for all p (it is easier to see that $8\sqrt{p}/(\log p)^2 > 1$ for all p), it will be cheaper to obtain the same performance for FFT computation on a hypercube than on a mesh. If the comparison is based on the transpose algorithm, then the hypercube will turn out to be even more cost effective, as the factor w by which the bandwidth of the mesh channels will have to be increased to match its performance with that of a hypercube will now be $\sqrt{p}$. Thus the relative costs of building a mesh and a hypercube with identical performance for the FFT computation will be $8p\sqrt{p}$ and $p \log p$, respectively.

However, if the cost of the network is considered to be a function of the bisection width of the network, as may be the case in VLSI implementations [26], then the picture improves for the mesh. The bisection widths of a hypercube and a mesh containing p processors each are $p/2$ and $\sqrt{p}$ respectively. In order to match the performance of the mesh with that of the hypercube for the binary-exchange algorithm, each of its channels has to made wider by a factor of $w = 2\sqrt{p}/\log p$. In this case, the bisection width of the mesh network becomes $2p/\log p$. Thus the costs of the hypercube and mesh networks with p processors each, such that they yield similar performance on the FFT, will be functions of $p/2$ and $2p/\log p$, respectively. Clearly, for $p > 256$, such a mesh network is cheaper to build than a hypercube. However, for the transpose algorithm the relative costs of the mesh and the hypercube yielding same throughput will be $p/2$ and $2p$, respectively. Hence the hypercube is still more cost effective by a constant factor.

The above analysis shows that the performance of the FFT algorithm on a mesh can be improved considerably by increasing the bandwidth of its communication channels by a factor of $\sqrt{p}/2$. But the enhanced bandwidth can be fully utilized only if there are at least $\sqrt{p}/4$ data items to be transferred in each communication step. Thus the input data size n should be at least $p\sqrt{p}/4$. This leads to an isoefficiency term of $\Theta(p^{1.5}\log p)$ due to concurrency, but is still a significant improvement for the mesh from $\Theta(\sqrt{p}2^{2Kt_w/t_c})$ with channels of constant bandwidth. In fact $\Theta(p^{1.5}\log p)$ is the best possible isoefficiency for FFT on a mesh even if the channel width is increased arbitrarily with the number of processors. It can be shown that if the channel bandwidth grows as $\Theta(p^x)$, then the isoefficiency function due to communication will be $\Theta(p^{.5-x}2^{2K(t_w/t_c)p^{.5-x}})$ and the isoefficiency function due to concurrency will be $\Theta(p^{1+x}\log p)$. If $x < 0.5$, then the overall isoefficiency is determined by communication overheads, and is exponential. If $x \geq 0.5$, then the overall isoefficiency is determined by concurrency. Thus, the best isoefficiency function of $\Theta(p^{1.5}\log p)$ can be obtained at $x = .5$.

Many researchers [33, 124, 2, 1] prefer to compare architectures while keeping the number of communication ports per processor (as opposed to bisection width) the same across the architectures. Dutt and Trinh [33] show that for FFT-like computations, hypercubes are more cost-effective even with this cost criterion.

3.1.7 Experimental Results

We implemented the binary-exchange algorithm for unordered single dimensional radix-2 FFT on a 1024-node nCUBE1 hypercube. Experiments were conducted for a range of problem sizes and a range of machine sizes; i.e., number of processors. The length of the input vector was varied between 4 and 65536, and the number of processors was varied between 1 and 1024. The required twiddle factors were precomputed and stored at each processor. Speedups and efficiencies were computed w.r.t. the run time of sequential FFT running on one processor of the nCUBE1. A unit

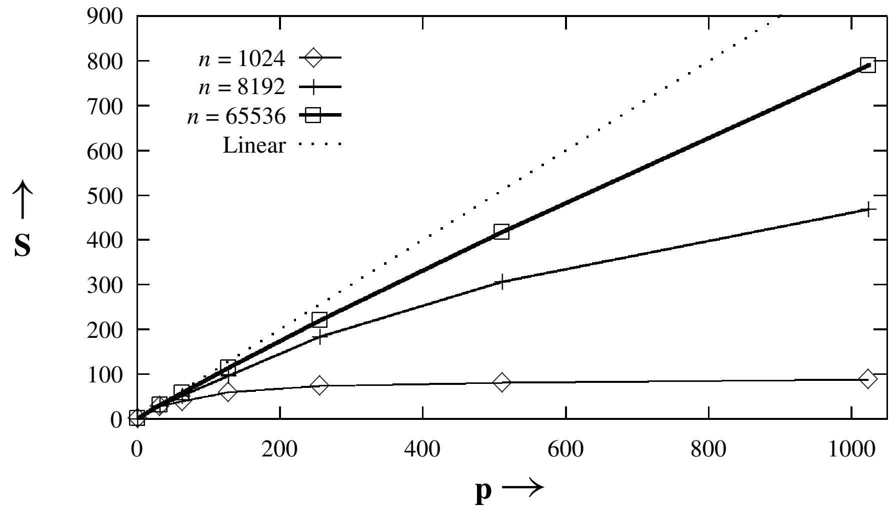


Figure 3.2: Speedup curves on a hypercube for various problem sizes.

FFT computation takes[2] approximately 80 microseconds; i.e., $t_c \approx 80$ microseconds. Figures 3.2 through 3.4 summarize the results of these experiments.

Figure 3.2 shows the speedup curves for 3 different problem sizes. As expected, for a small problem size (input vector length = 1024), the speedup reaches a saturation point for a small number of processors. Beyond this point, an increase in the number of processors does not result in additional speedup. On the other hand, the speedup curve is nearly linear for a larger problem size (length of input vector = 65536).

From the timings obtained in the experiments, we determined t_w, the time to transfer a word (8 bytes), to be 16 microseconds. From the experimentally obtained values of t_w and t_c, the value of Kt_w/t_c was found to exceed 1 at $E = .83$ and the isoefficiency curves for $E > .83$ should be non-linear. We selected those sets of data points from our experiment that correspond to approximately

[2] In our actual FFT program written in C a unit of computation took approximately 400 microseconds. Given that each FFT computation requires four 32-bit additions/subtractions and four 32 bit multiplications, this corresponds to a Mega-FLOP rating of 0.02 which is far lower than those obtained from FFT benchmarks written in Fortran or assembly language. This is perhaps due to our inefficient C-compiler. Since CPU speed has a tremendous impact on the overall scalability of FFT, we artificially increased the CPU speed to a more realistic rating of 0.1 Mega-FLOP. This is obtained by replacing the actual complex arithmetic of the inner loop of the FFT computation by a dummy loop that takes 80 microseconds to execute.

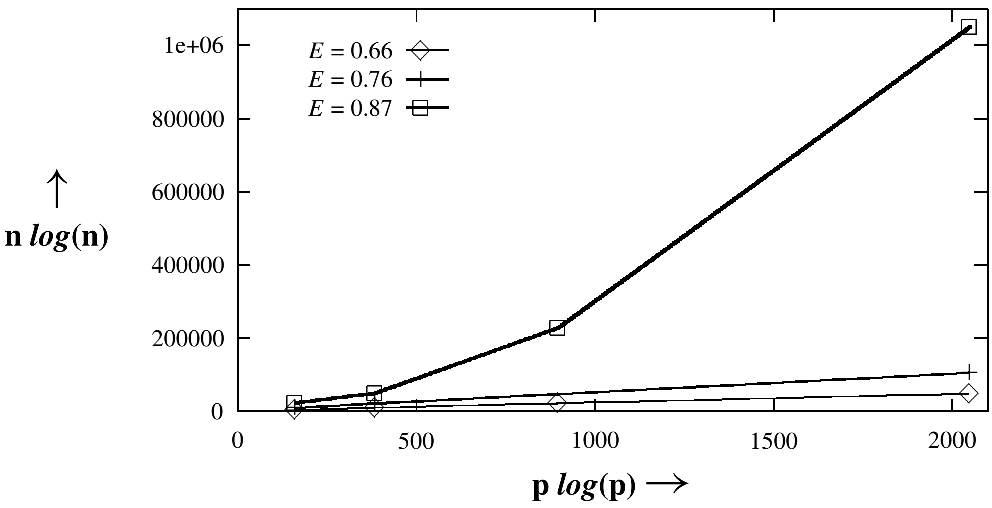


Figure 3.3: Isoefficiency curves for 3 different values of E on a hypercube.

the same efficiencies and used these to plot the three isoefficiency curves given in Figure 3.3. In order to make it easier to see the relationship between problem size and the number of processor, we plot $n \log n$ on the X-axis and $p \log p$ on the Y-axis. The plot is nearly linear for $E = .76$ and $E = .66$, thus, showing an isoefficiency of $\Omega(p \log p)$ which conforms to our analysis. The third curve corresponding to $E = .87$ shows poor isoefficiency. This is in agreement with our analytical results as 0.87 is greater than the break-even efficiency of 0.83.

Using the run times on the hypercube, the corresponding results for a mesh connected computer having identical processor speed and identical communication costs per link were projected. Figure 3.4 shows the isoefficiency curves for hypercube and mesh connected computers for the same efficiency of 0.66. It is clear that the problem size has to grow much more rapidly on a mesh than on a hypercube to maintain the same efficiency.

Table 3.1 illustrates the effect of t_w/t_c ratio on the scalability of the FFT algorithm on hypercubes. The entries in the table show the slope of $\log n$ to $\log p$ curve for maintaining different efficiencies on 4 different machines, namely, M_1, M_2, M_3 and M_4, for which the t_w/t_c ratios are 0.2, 1.28, 0.18 and 10.7 respectively. This table also serves to predict the maximum practically achievable efficiencies

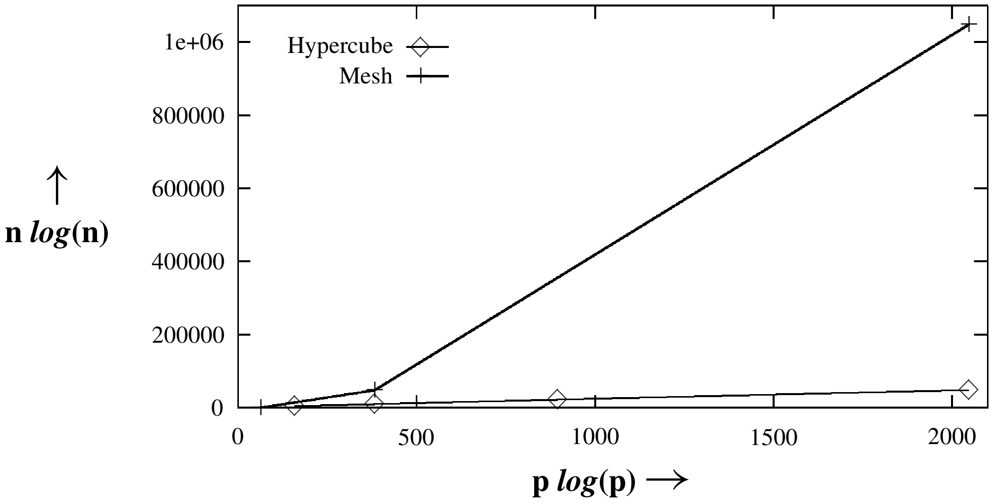


Figure 3.4: Isoefficiency curves on mesh and hypercube for E = 0.66.

on these machines for the FFT computation. A machine can easily obtain those efficiencies for which the entry in table is small; i.e., around 1. For example, the entry for machine M_2 corresponding to an efficiency of 0.5 is 1.28. Thus it can obtain an efficiency of 0.5 even on more than 10,000 processors ($p \approx 2^{17}$) on a FFT problem with $n = 2^{17\times1.28} \approx 2^{22}$, which is only moderately large. A machine for which the entry in the table is large cannot obtain those efficiencies except on a very small number of processors. For example in order to maintain an efficiency of 0.7 on M_4, n will have to grow asymptotically as p^{25}. In other words, a problem with 2^{25} input data elements is required to get an efficiency of 0.7 even on a 2 processors machine for which the t_w/t_c ratio is equal to that of M_4. Note that as discussed in Section 3.1.2, when this figure is less than one, n has to grow as p (or $n \log n$ has to grow asymptotically in proportion with $p \log p$) due to concurrency and other factors such as t_s.

Table 3.2 shows the efficiencies obtainable on a hypercube of type M_4 as a function of number of processors and the size of the input. This table gives an idea as to how large the problem size has to be to obtain reasonable efficiencies on hypercubes of various sizes of type M_4. Clearly, except for unreasonably large problem sizes (with $n > 2^{30}$), the efficiencies obtained will be small (< 0.2)

E	M_1	M_2	M_3	M_4
0.1	.022	.143	.020	1.19
0.2	.050	.320	.045	2.68
0.3	.085	.548	.076	4.59
0.4	.133	.853	.119	7.14
0.5	.200	1.28	.178	10.7
0.6	.300	1.92	.267	16.1
0.7	.466	2.99	.415	25.0
0.8	0.80	5.12	.712	42.9
0.9	1.80	11.5	1.60	96.4
.95	3.80	24.3	3.38	203

Table 3.1: Scalability of FFT algorithm on four different hypercubes for various efficiencies. Each entry denotes the ratio of $\log n$ to $\log p$.

for large hypercubes (having a thousand or more nodes) of type M_4.

The reader should note that the t_w/t_c ratios for M_1, M_2, M_3 and M_4 roughly correspond to those of four commercially available machines; i.e., nCUBE1, nCUBE2, Intel IPSC/2 and IPSC/860 respectively. Their communication channel bandwidths are roughly 0.5, 2.5, 2.8 and 2.8 Megabytes per second and the individual processor speeds are roughly 0.1, 3.2, 0.5 and 30 Mega-FLOPS respectively for FFT computation[3].

3.2 Dense Matrix Multiplication

Matrix multiplication is widely used in a variety of applications and is often one of the core components of many scientific computations. Since the matrix multiplication algorithm is highly computation intensive, there has been a great deal of interest in developing parallel formulations of this algorithm and testing its performance on various parallel architectures [5, 16, 21, 23, 24, 28, 39,

[3] The processor speeds for nCUBE2, Intel IPSC/2 and IPSC/860 are quoted by the respective manufacturers for FFT benchmarks.

Input Size → / No. of Processors ↓	2^{12}	2^{20}	2^{30}	2^{40}	2^{50}
1	1.00	1.00	1.00	1.00	1.00
16	0.22	0.32	0.41	0.48	0.54
64	0.19	0.24	0.32	0.38	0.44
256	0.12	0.19	0.26	0.32	0.37
1024	0.10	0.16	0.22	0.27	0.32
4096	0.09	0.13	0.19	0.24	0.28

Table 3.2: Efficiencies as a function of input size and number of processors on a hypercube of type M_4.

67, 68, 136, 27]. A number of parallel formulations of dense matrix multiplication algorithm have been developed. For arbitrarily large number of processors, any of these algorithms or their variants can provide near linear speedup for sufficiently large matrix sizes and none of the algorithms can be clearly claimed to be superior than the others. In this section, we analyze the performance and scalability of a number of parallel formulations of the matrix multiplication algorithm and predict the conditions under which each formulation is better than the others.

3.2.1 Parallel Matrix Multiplication Algorithms

In this section we briefly describe some well known parallel matrix multiplication algorithms give their parallel execution times.

A Simple Algorithm

Consider a logical two dimensional mesh of p processors (with $\sqrt{p}$ rows and $\sqrt{p}$ columns) on which two $n \times n$ matrices A and B are to be multiplied to yield the product matrix C. Let $n \geq \sqrt{p}$. The matrices are divided into sub-blocks of size $n/\sqrt{p} \times n/\sqrt{p}$ which are mapped naturally on the processor array. The algorithm can be implemented on a hypercube by embedding this processor mesh into it. In the first step of the algorithm, each processor acquires all those elements of both the matrices that are required to generate the n^2/p elements of the product matrix which are to reside in that processor. This involves an all-to-all broadcast of n^2/p elements of matrix A among the $\sqrt{p}$ processors of each row of processors and that of the same sized blocks of matrix B among $\sqrt{p}$ processors of each column which can be accomplished in $2t_s \log p + 2t_w n^2/\sqrt{p}$ time. After each processor gets all the data it needs, it multiplies the $\sqrt{p}$ pairs of sub-blocks of the two matrices to compute its share of n^2/p elements of the product matrix. Assuming that an addition and multiplication takes a unit time, the multiplication phase can be completed in n^3/p units of time. Thus the total parallel execution time of the algorithm is given by the following equation:

$$T_p = \frac{n^3}{p} + 2t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}. \tag{3.11}$$

This algorithm is memory-inefficient. The memory requirement for each processor is $\Theta(n^2/\sqrt{p})$ and thus the total memory requirement is $\Theta(n^2\sqrt{p})$ words as against $\Theta(n^2)$ for the sequential algorithm.

Cannon's Algorithm

A parallel algorithm that is memory efficient and is frequently used is due to Cannon [21]. Again the two $n \times n$ matrices A and B are divided into square submatrices of size $n/\sqrt{p} \times n/\sqrt{p}$ among the p processors of a wrap-around mesh (which can be embedded in a hypercube if the algorithm was to be implemented on it). The sub-blocks of A and B residing with the processor (i, j) are denoted by A^{ij} and B^{ij} respectively, where $0 \leq i < \sqrt{p}$ and $0 \leq j < \sqrt{p}$. In the first phase of the execution of the algorithm, the data in the two input matrices is aligned in such a manner that the corresponding square submatrices at each processor can be multiplied together locally. This is done by sending the block A^{ij} to processor $(i, (j + i) mod \sqrt{p})$, and the block B^{ij} to processor $((i + j) mod \sqrt{p}, j)$. The copied sub-blocks are then multiplied together. Now the A sub-blocks are rolled one step to the left and the B sub-blocks are rolled one step upward and the newly copied sub-blocks are multiplied and the results added to the partial results in the C sub-blocks. The multiplication of A and B is complete after $\sqrt{p}$ steps of rolling the sub-blocks of A and B leftwards and upwards, respectively, and multiplying the in coming sub-blocks in each processor. In a hypercube with cut-through routing, the time spent in the initial alignment step can be ignored with respect to the $\sqrt{p}$ shift operations during the multiplication phase, as the former is a simple one-to-one communication along non-conflicting paths. Since each sub-block movement in the second phase takes $t_s + t_w n^2/p$ time, the total parallel execution time for all the movements of the sub-blocks of both the matrices is given by the following equation:

$$T_p = \frac{n^3}{p} + 2t_s\sqrt{p} + 2t_w\frac{n^2}{\sqrt{p}}. \tag{3.12}$$

Fox's Algorithm

This algorithm is due to Fox et al. and is described in detail in [39] and [38]. The input matrices are initially distributed among the processors in the same manner as in the simple algorithm in Section 3.2.1. The algorithm works in $\sqrt{p}$ iterations, where p is the number of processors being used. The data communication in the algorithm involves successive broadcast of the the sub-blocks of A in a horizontal direction so that all processors in the ith row receive the sub-block $A^{i(i+j)}$ in the jth iteration (iterations are numbered from 0 to $j - 1$). After each broadcast the sub-blocks of A are multiplied by the sub-blocks of B currently residing in each processor and are accumulated in the sub-blocks of S. The last step of each iteration is the shifting of the sub-blocks of B in all the processors to their respective North neighbors in the wrap-around mesh, the sub-blocks of the topmost row being rolled into the bottommost row. Thus, for the mesh architecture, the algorithm takes $(t_s + t_w n^2/p)\sqrt{p}$ time in communication in each of the $\sqrt{p}$ iterations, resulting in a total parallel execution time of $n^3/p + t_w n^2 + t_s p$. By sending the sub-blocks in small packets in a pipelined fashion, Fox et al. show the run time of this algorithm to be as follows:

$$T_p = \frac{n^3}{p} + 2t_w\frac{n^2}{\sqrt{p}} + t_s p. \tag{3.13}$$

Clearly the parallel execution time of this algorithm is worse than that of the simple algorithm or Cannon's algorithm. On a hypercube, it is possible to employ a more sophisticated scheme for one-to-all broadcast [71] of sub-blocks of matrix A among the rows. Using this scheme, the parallel execution time can be improved to $n^3/p + 2t_w n^2/\sqrt{p} + t_s\sqrt{p}\log p + 2n\sqrt{t_s t_w logp}$, which is still worse than Cannon's algorithm. However, if the procedure is performed in an asynchronous manner (i.e., in every iteration, a processor starts performing its computation as soon as it has all the required data, and does not wait for the entire broadcast to finish) the computation and communication of sub-blocks can be interleaved. It can be shown that if each step of Fox's algorithm is not synchronized and the processors work independently, then its parallel execution time can be reduced to almost a factor of two of that Cannon's algorithm.

Berntsen's Algorithm

Due to nearest neighbor communications on the $\sqrt{p} \times \sqrt{p}$ wrap-around array of processors, Cannon's algorithm's performance is the same on both mesh and hypercube architectures. In [16], Berntsen describes an algorithm which exploits greater connectivity provided by a hypercube. The algorithm uses $p = 2^{3q}$ processors with the restriction that $p \leq n^{3/2}$ for multiplying two $n \times n$ matrices A and B. Matrix A is split by columns and B by rows into 2^q parts. The hypercube is split into 2^q subcubes, each performing a submatrix multiplication between submatrices of A of size $n/2^q \times n/2^{2q}$ and submatrices of B of size $n/2^{2q} \times n/2^q$ using Cannon's algorithm. It is shown in [16] that the time spent in data communication by this algorithm on a hypercube is $2t_s p^{1/3} + (t_s \log p)/3 + 3t_w n^2/p^{2/3}$, and hence the total parallel execution time is given by the following equation:

$$T_p = \frac{n^3}{p} + 2t_s p^{1/3} + \frac{1}{3}t_s \log p + 3t_w \frac{n^2}{p^{2/3}}. \tag{3.14}$$

The terms associated with both t_s and t_w are smaller in this algorithm than the algorithms discussed earlier in this section. It should also be noted that this algorithm, like the simple algorithm, is not memory efficient as it requires storage of $2n^2/p + n^2/p^{2/3}$ matrix elements per processor.

The DNS Algorithm

One Element Per Processor Version An algorithm that uses a hypercube with $p = n^3 = 2^{3q}$ processors to multiply two $n \times n$ matrices was proposed by Dekel, Nassimi and Sahni in [28, 118]. The p processors can be visualized as being arranged in an $2^q \times 2^q \times 2^q$ array. In this array, processor p_r occupies position (i, j, k) where $r = i2^{2q} + j2^q + k$ and $0 \leq i, j, k < 2^q$. Thus if the binary representation of r is $r_{3q-1}r_{3q-2}...r_0$, then the binary representations of i, j and k are $r_{3q-1}r_{3q-2}...r_{2q}$, $r_{2q-1}r_{2q-2}...r_q$ and $r_{q-1}r_{q-2}...r_0$ respectively. Each processor p_r has three data registers a_r, b_r and c_r, respectively. Initially, processor p_s in position (0,j,k) contains the element $a(j, k)$ and $b(j, k)$ in a_s and b_s respectively. The computation is accomplished in three stages. In the first stage, the elements of the matrices A and B are distributed over the p processors. As a result, a_r gets $a(j, i)$ and b_r gets $b(i, k)$. In the second stage, product elements $c(j, k)$ are computed and stored in each c_r. In the final stage, the sums $\Sigma_{i=0}^{n-1} c_{i,j,k}$ are computed and stored in $c_{0,j,k}$.

The above algorithm accomplishes the $\Theta(n^3)$ task of matrix multiplication in $\Theta(\log n)$ time using n^3 processors. Since the processor-time product of this parallel algorithm exceeds the sequential time complexity of the algorithm, it is not processor-efficient. This algorithm can be made processor-efficient by using fewer that n^3 processors; i.e., by putting more than one element of the matrices on each processor. There are more than one ways to adapt this algorithm to use fewer than n^3 processors. The method proposed by Dekel, Nassimi, and Sahni in [28, 118] is as follows.

Variant with More than One Element Per Processor This variant of the DNS algorithm can work with $n^2 r$ processors, where $1 < r < n$, thus using one processor for more than one element of each of the two $n \times n$ matrices. The algorithm is similar to the one above except that a logical processor array of r^3 (instead of n^3) superprocessors is used, each superprocessor comprising of $(n/r)^2$ hypercube processors. In the second step, multiplication of blocks of $(n/r) \times (n/r)$ elements instead of individual elements is performed. This multiplication of $(n/r) \times (n/r)$ blocks is performed according to Fox's on $n/r \times n/r$ subarrays (each such subarray is actually a subcube) of processors using Cannon's algorithm for one element per processor. This step will require a communication time of $2(t_s + t_w)n/r$.

In the first stage of the algorithm, each data element is broadcast over r processors. In order to place the elements of matrix A in their respective positions, first the buffer $a_{(0,j,k)}$ is sent to $a_{(k,j,k)}$ in $\log r$ steps and then $a_{(k,j,k)}$ is broadcast to $a_{(k,j,l)}$, $0 \le l < r$, again in $\log r$ steps. By following a similar procedure, the elements of matrix B can be transmitted to their respective processors. In the second stage, groups of $(n/r)^2$ processors multiply blocks of $(n/r) \times (n/r)$ elements each processor performing n/r computations and $2n/r$ communications. In the final step, the elements of matrix C are restored to their designated processors in $\log r$ steps. The communication time can thus be shown to be equal to $(t_s + t_w)(5 \log r + 2n/r)$ resulting in the parallel run time given by the following equation:

$$T_p = \frac{n^3}{p} + (t_s + t_w)(5\log(\frac{p}{n^2}) + 2\frac{n^3}{p}). \tag{3.15}$$

If $p = n^3/\log n$ processors are used, then the parallel execution time of the DNS algorithm is $\Theta(\log n)$. The processor-time product is now $\Theta(n^3)$, which is same as the sequential time complexity of the algorithm.

Our Variant of the DNS Algorithm

Here we present another scheme to adapt the single element per processor version of the DNS algorithm to be able to use fewer than n^3 processors on a hypercube. Henceforth, we shall refer to this algorithm as the GK variant of the DNS algorithm. As shown later in Section 3.2.3, this algorithm performs better than the DNS algorithm for a wide range of n and p. Also, unlike the DNS algorithm which works only for $n^2 \leq p \leq n^3$, this algorithm can use any number of processors from 1 to n^3. In this variant, we use $p = 2^{3q}$ processors where $q < (\log n)/3$. The matrices are divided into sub-blocks of $n/2^q \times n/2^q$ elements and the sub-blocks are numbered just the way the single elements were numbered in the one-element-per-processor version of the DNS algorithm. Now, all the single element operations of this algorithm are replaced by sub-block operations; i.e., matrix sub-blocks are multiplied, communicated, and added.

Let t_{mult} and t_{add} is the time to perform a single floating point multiplication and addition respectively. Also, according to the earlier assumption, $t_{mult} + t_{add} = 1$. In the first stage of this algorithm, $n^2/p^{2/3}$ data elements are broadcast over $p^{1/3}$ processors for each matrix. In order to place the elements of matrix A in their respective positions, first the buffer $a_{(0,j,k)}$ is sent to $a_{(k,j,k)}$ in $\log p^{1/3}$ steps and then $a_{(k,j,k)}$ is broadcast to $a_{(k,j,l)}$, $0 \leq l < p^{1/3}$, again in $\log p^{1/3}$ steps. By following a similar procedure, the elements of matrix B can be sent to the processors where they are to be utilized in $2\log p^{1/3}$ steps. In the second stage of the algorithm, each processor performs $(n/p^{1/3})^3 = n^3/p$ multiplications. In the third step, the corresponding elements of $p^{1/3}$ groups of $n^2/p^{2/3}$ elements each are added in a tree fashion. The first stage takes $4t_s \log p^{1/3} + 4t_w(n^2/p^{2/3})\log p^{1/3}$ time. The second stage contributes $t_{mult}n^3/p$ to the parallel execution time and the third stage involves $t_s \log p^{1/3} + t_w(n^2/p^{2/3})\log p^{1/3}$ communication time and $t_{add}n^3/p$ computation time for calculating

Algorithm	Total Overhead Function, T_o	Asymptotic Isoeff. Function	Range of Applicability
Berntsen's	$2t_s p^{4/3} + \frac{1}{3}t_s p \log p + 3t_w n^2 p^{1/3}$	$\Theta(p^2)$	$1 \leq p \leq n^{3/2}$
Cannon's	$2t_s p^{3/2} + 2t_w n^2 \sqrt{p}$	$\Theta(p^{1.5})$	$1 \leq p \leq n^2$
GK	$\frac{5}{3}t_s p \log p + \frac{5}{3}t_w n^2 p^{1/3} \log p$	$\Theta(p(\log p)^3)$	$1 \leq p \leq n^3$
Improved GK	$t_w n^2 p^{1/3} + \frac{1}{3}t_s p \log p + 2np^{2/3}\sqrt{\frac{1}{3}t_s t_w \log p}$	$\Theta(p(\log p)^{1.5})$	$1 \leq p \leq (\frac{n}{\sqrt{\frac{t_s}{t_w} \log n}})^3$
DNS	$(t_s + t_w)(\frac{5}{3}p \log p + 2n^3)$	$\Theta(p \log p)$	$n^2 \leq p \leq n^3$

Table 3.3: Communication overhead, scalability and range of application of the four algorithms on a hypercube.

the sums. The total parallel execution time is therefore given by the following equation:

$$T_p = \frac{n^3}{p} + \frac{5}{3}t_s \log p + \frac{5}{3}t_w \frac{n^2}{p^{2/3}} \log p. \tag{3.16}$$

This execution time can be further reduced by using a more sophisticated scheme for one-to-all broadcast on a hypercube [71]. This is discussed in detail later while analyzing the scalability of the GK algorithm.

3.2.2 Scalability Analysis

Following the technique described in Section 2.2, in [54] we have analyzed the scalability of all the algorithms discussed in Section 3.2.1 on the hypercube architecture using the isoefficiency metric. The asymptotic scalabilities and the range of applicability of these algorithms is summarized in Table 3.3. The isoefficiency functions in Table 3.3 reflect the impact of communication overheads, as well as, the degree of concurrency. For example, the isoefficiency term for Berntsen's algorithm is $\Theta(p^{4/3})$ due to communication overhead [54]. However, for this algorithm, p cannot exceed $n^{3/2}$. This restriction $p \leq n^{3/2}$ implies that $n^3 = W = \Omega(p^2)$. Hence, the overall asymptotic isoefficiency function for this algorithm is $\Theta(p^2)$. In this section and the rest of this chapter, we skip the discussion of the simple algorithm and Fox's algorithm because the expressions for their iso-efficiency functions differ with that for Cannon's algorithm by small constant factors only [54].

Note that Table 3.3 gives only the asymptotic scalabilities of the four algorithms. In practice, none of the algorithms is strictly better than the others for all possible problem sizes and number of processors. Further analysis is required to determine the best algorithm for a given problem size and a certain parallel machine depending on the number of processors being used and the hardware parameters of the machine. A detailed comparison of these algorithms based on their respective total overhead functions is presented in the next section.

3.2.3 Relative Performance of the Four Algorithms on a Hypercube

The isoefficiency functions of the four matrix multiplication algorithms predict their relative performance for a large number of processors and large problem sizes. But for moderate values of n and p, a seemingly less scalable parallel formulation can outperform the one that has an asymptotically smaller isoefficiency function. In this subsection, we derive the exact conditions under which each of these four algorithms yields the best performance.

We compare a pair of algorithms by comparing their total overhead functions (T_o) as given in Table 3.3. For instance, while comparing the GK algorithm with Cannon's algorithm, it is clear that the t_s term for the GK algorithm will always be less than that for Cannon's algorithm. Even if $t_s = 0$, the t_w term of the GK algorithm becomes smaller than that of Cannon's algorithm for $p > 130$ million. Thus, $p = 130$ million is the cut-off point beyond which the GK algorithm will perform better than Cannon's algorithm irrespective of the values of n. For $p < 130$ million, the performance of the GK algorithm will be better than that of Cannon's algorithm for values of n less than a certain threshold value which is a function of p and the ration of t_s and t_w. A hundred and thirty million processors is clearly too large, but we show that for reasonable values of t_s, the GK algorithm performs better than Cannon's algorithm for very practical values of p and n.

In order to determine ranges of p and n where the GK algorithm performs better than Cannon's algorithm, we equate their respective overhead functions and compute n as a function of p. We call this $n_{Equal-T_o}(p)$ because this value of n is the threshold at which the overheads of the two algorithms will be identical for a given p. If $n > n_{Equal-T_o}(p)$, Cannon's algorithm will perform better and if $n < n_{Equal-T_o}(p)$, the GK algorithm will perform better.

$$T_o^{(Cannon)} = 2t_s p^{3/2} + 2t_w n^2 \sqrt{p} = T_o^{(GK)} = \frac{5}{3} t_s p \log p + \frac{5}{3} t_w n^2 p^{1/3} \log p,$$

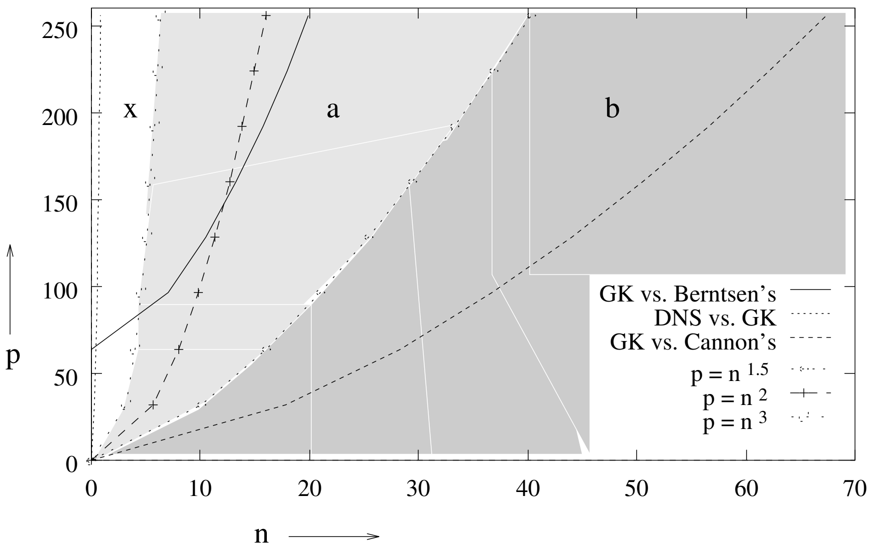


Figure 3.5: A comparison of the four algorithms for $t_w = 3$ and $t_s = 150$.

$$n_{Equal-T_o}(p) = \sqrt{\frac{(5/3p\log p - 2p^{3/2})t_s}{(2\sqrt{p} - 5/3p^{1/3}\log p)t_w}}. \tag{3.17}$$

Similarly, equal overhead conditions can be determined for other pairs of algorithms too and the values of t_w and t_s can be plugged in depending upon the machine in question to determine the best algorithm for a give problem size and number of processors. We have performed this analysis for three practical sets of values of t_w and t_s. In the rest of the section we demonstrate the practical importance of this analysis by showing how any of the four algorithms can be useful depending on the problem size and the parallel machine available.

The plain lines represent equal overhead conditions for pairs of algorithms. For a curve marked "X vs Y" in a figure, algorithm X has a smaller value of communication overhead to the left of the curve, algorithm Y has smaller communication overhead to the right side of the curve, while the two algorithms have the same value of T_o along the curve. The lines with symbols $\diamond$, $+$, and $\square$ plot the functions $p = n^{3/2}$, $p = n^2$ and $p = n^3$, respectively. These lines demarcate the regions of applicabilities of the four algorithms (see Table 3.3) and are important because an algorithm might not be applicable in the region where its overhead function T_o is mathematically superior than others. In all the figures in this section, the region marked with an **x** is the one where $p > n^3$ and none of the algorithms is applicable, the region marked with an **a** is the one where the GK algorithm

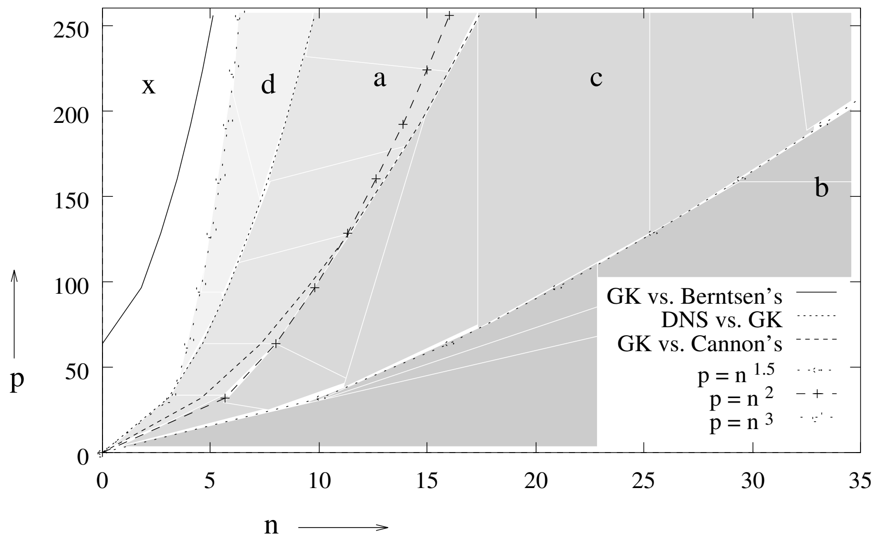


Figure 3.6: A comparison of the four algorithms for $t_w = 3$ and $t_s = 10$.

is the best choice, the symbol **b** represents the region where Berntsen's algorithm is superior to the others, the region marked with a **c** is the one where Cannon's algorithm should be used and the region marked with a **d** is the one where the DNS algorithm is the best.

Figure 3.5 compares the four algorithms for $t_w = 3$ and $t_s = 150$. These parameters are very close to that of a currently available parallel computer like the nCUBE2. In this figure, since the $n_{Equal-T_o}$ curve for the DNS algorithm and the GK algorithm lies in the **x** region, and the DNS algorithm is better than the GK algorithm only for values of n smaller than $n_{Equal-T_o}(p)$. Hence the DNS algorithm will always[4] perform worse than the GK algorithm for this set of values of t_s and t_w and the latter is the best overall choice for $p > n^2$ as Berntsen's algorithm and Cannon's algorithm are not applicable in this range of p. Since the $n_{Equal-T_o}$ curve for GK and Cannon's algorithm lies below the $p = n^{3/2}$ curve, the GK algorithm is the best choice even for $n^{3/2} \leq p \leq n^2$. For $p < n^{3/2}$, Berntsen's algorithm is always better than Cannon's algorithm, and for this set of t_s and t_w, also than the GK algorithm. Hence it is the best choice in that region in Figure 3.5.

In Figure 3.6, we compare the four algorithms for a hypercube with $t_w = 3$ and $t_s = 10$. Such a machine could easily be developed in the near future by using faster CPU's (t_w and t_s represent

[4] Actually, the $n_{Equal-T_o}$ curve for DNS vs GK algorithms will cross the $p = n^3$ curve for $p = 2.6 \times 10^{18}$, but clearly this region has no practical importance.

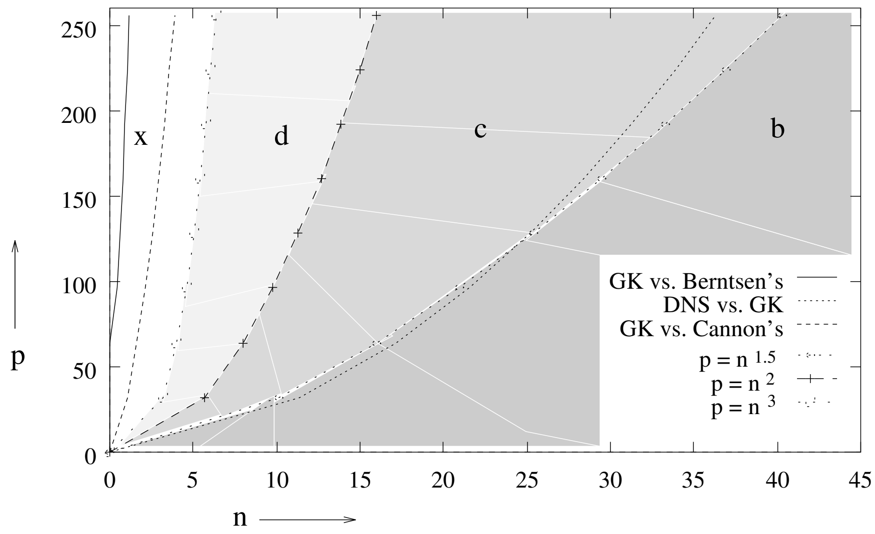


Figure 3.7: A comparison of the four algorithms for $t_w = 3$ and $t_s = 0.5$.

relative communication costs with respect to the unit computation time) and reducing the message startup time. By observing the $n_{Equal-T_o}$ curves and the regions of applicability of these algorithms, the regions of superiority of each of the algorithms can be determined just as in case of Figure 3.5. It is noteworthy that in Figure 3.6 each of the four algorithms performs better than the rest in some region and all the four regions **a**, **b**, **c** and **d** contain practical values of p and n.

In Figure 3.7, we present a comparison of the four algorithms for $t_w = 3$ and $t_s = 0.5$. These parameters are close to what one can expect to observe on a typical SIMD machine like the CM-2. For the range of processors shown in the figure, the GK algorithm is inferior to the others[5]. Hence it is best to use the DNS algorithm for $n^2 \leq p \leq n^3$, Cannon's algorithm for $n^{3/2} \leq p \leq n^2$ and Berntsen's algorithm for $p < n^{3/2}$.

3.2.4 Scalabilities of Different Algorithms with Simultaneous Communication on All Hypercube Channels

On certain parallel machines like the nCUBE2, the hardware supports simultaneous communication on all the channels. This feature of the hardware can be utilized to significantly reduce the

[5] The GK algorithm does begin to perform better than the other algorithms for $p > 1.3 \times 10^8$, but again we consider this range of p to be impractical.

communication cost of certain operations involving broadcasting and personalized communication [71]. In this section we investigate as to what extent can the performance of the algorithms described in Section 3.2.1 can be improved by utilizing simultaneous communication on all the $\log p$ ports of the hypercube processors.

Cannon's algorithm, Berntsen's algorithm and the pipelined version of Fox's algorithm employ only nearest neighbor communication and hence can benefit from simultaneous communication by a constant factor only as the subbocks of matrices A and B can now be transferred simultaneously. The DNS algorithm can also gain only a constant factor in its communication terms as all data messages are only one word long. Hence, among the matrix multiplication algorithms discussed here, the ones that can potentially benefit from simultaneous communications on all the ports are the simple algorithm (or its variations [68]) and the GK algorithm.

The Simple Algorithm with All-Port Communication

This algorithm requires an all-to-all broadcast of the sub-blocks of the matrices A and B among groups of $\sqrt{p}$ processors. The best possible scheme utilizing all the channels of a hypercube simultaneously can accomplish an all-to-all broadcast of blocks of size n^2/p among $\sqrt{p}$ processors in time $2t_w n^2\sqrt{p}/(p\log p) + (t_s \log p)/2$. Moreover, the communication of the sub-blocks of both A and B can proceed simultaneously. Thus the parallel execution time of this algorithm on a hypercube with simultaneous communication is given by the following equation:

$$T_p = \frac{n^3}{p} + 2t_w\frac{n^2}{\sqrt{p}\log p} + \frac{1}{2}t_s \log p. \tag{3.18}$$

Recall that the simple algorithm is not memory efficient. Ho et al. [68] give a memory efficient version of this algorithm which has somewhat higher execution time than that given by Equation 3.18. It can be shown that the isoefficiency function due to communication overheads is only $\Theta(p\log p)$ now, which is a significant improvement over the $\Theta(p^{1.5})$ isoefficiency function of this algorithm when communication on only one of the $\log p$ ports of a processor was allowed at a time.

However, as mentioned in [68], the lower limit on the message size imposes the condition that $n \geq (\sqrt{p}\log p)/2$. This requires that $n^3 = W \geq p^{1.5}(\log p)^3/8$. Thus the rate at which the the problem size is required to grow with respect to the number of processors in order to utilize all the communication channels of the hypercube is higher than the isoefficiency function of the algorithm implemented on a simple hypercube with one port communication at a time.

The GK Algorithm with All-Port Communication

Using the one-to-all broadcast scheme of [71] for a hypercube with simultaneous all-port communication, the parallel execution time of the GK algorithm can be reduced to the following:

$$T_p = \frac{n^3}{p} + t_s \log p + 9t_w \frac{n^2}{p^{2/3} \log p} + 6\frac{n}{p^{1/3}}\sqrt{t_s t_w}. \tag{3.19}$$

The communication terms now yield an isoefficiency function of $\Theta(p \log p)$, but it can be shown that lower limit on the message size entails the problem size to grow as $\Theta(p(\log p)^3)$ with respect to p which is not any better that the isoefficiency function of this algorithm on a simple hypercube with one port communication at a time.

Discussion

The gist of the analysis in this section is that allowing simultaneous on all the ports of a processor on a hypercube does not improve the overall scalability of matrix multiplication algorithms. The reason is that simultaneous communication on all channels requires that each processor has large enough chunks of data to transfer to other processors. This imposes a lower bound on the size of the problem that will generate such large messages. In case of matrix multiplication algorithms, the problem size (as a function of p) that can generate large enough messages for simultaneous communication to be useful, turns out to be larger than what is required to maintain a fixed efficiency with only one port communication at a time. However, there will be certain values of n and p for which the modified algorithm will perform better.

3.2.5 Isoefficiency as a Function of Technology Dependent Factors

The isoefficiency function can be used not only to determine the rate at which the problem size should grow with respect to the number of processors, but also with respect to a variation in other hardware dependent constants such as the communication speed and processing power of the processors used etc. In many algorithms, these constants contribute a multiplicative term to the isoefficiency function, but in some others they effect the asymptotic isoefficiency of a parallel system (e.g., parallel FFT). For instance, a multiplicative term of $(t_w)^3$ appears in most isoefficiency functions of matrix multiplication algorithms described in this chapter. As discussed earlier, t_w depends on the ratio of the data communication speed of the channels to the computation speed of the processors used in the parallel architecture. This means that if the processors of the multicomputer are replaced by k times faster processors, then the problem size will have to be increased by a factor of k^3 in order to obtain the same efficiency. Thus the isoefficiency function for matrix multiplication is very sensitive to the hardware dependent constants of the architecture. For example, in case of Cannon's algorithm, if the number of processors is increased 10 times, one would have to solve a problem 31.6 times bigger in order to get the same efficiency. On the other hand, for small values of t_s (as may be the case with most SIMD machines), if p is kept the same and 10 times faster processors are used, then one would need to solve a 1000 times larger problem to be able to obtain the same efficiency. Hence for certain problem sizes, it may be better to have a parallel computer with k-fold as many processors rather than one with the same number of processors, each k-fold as fast (assuming that the communication network and the bandwidth etc. remain the same). This should be contrasted with the conventional wisdom that suggests that better performance is always obtained using fewer faster processors [15].

3.2.6 Experimental Results

We verified a part of the analysis of Section 3.2.3 through experiments on the CM-5 parallel computer. On this machine, the fat-tree [89] like communication network on the CM-5 provides simultaneous paths for communication between all pairs of processors. Hence the CM-5 can be viewed as a fully connected architecture which can simulate a hypercube connected network. We implemented Cannon's and the algorithm and our (GK) variant of the DNS algorithm.

On the CM-5, the time taken for one floating point multiplication and addition was measured to be 1.53 microseconds on our implementation. The message startup time for our program was observed to be about 380 microseconds and the per-word transfer time for 4 byte words was observed to be about 1.8 microseconds[6]. Since the CM-5 can be considered as a fully connected network

[6] These values do not necessarily reflect the communication speed of the hardware but the overheads observed for our

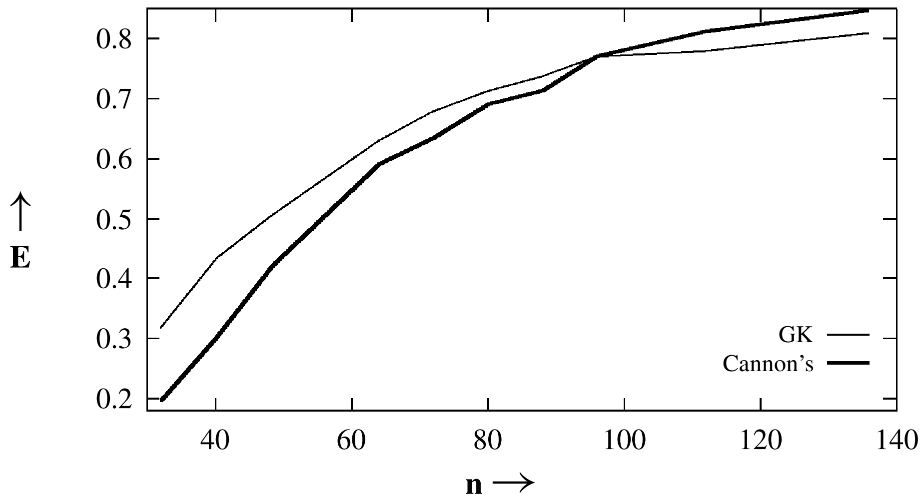


Figure 3.8: Efficiency as a function of matrix size for Cannon's algorithm and GK algorithm for 64 processors.

of processors, the expression for the parallel execution time for the algorithm of Section 3.2.1 will have to be modified slightly. The first part of the procedure to place the elements of matrix A in their respective positions, requires sending the buffer $a_{(0,j,k)}$ to $a_{(k,j,k)}$. This can be done in one step on the CM-5 instead of $\log(p^{1/3})$ steps on a conventional hypercube. The same is true for matrix B as well. It can be shown that the following modified expression gives the parallel execution time for this algorithm on the CM-5:

$$T_p = \frac{n^3}{p} + t_s(\log p + 2) + t_w \frac{n^2}{p^{2/3}}(\log p + 2). \tag{3.20}$$

Computing the condition for equal T_o for this and Cannon's algorithm by deriving the respective values of T_o from Equations 3.20 and 3.12, it can be shown that for 64 processors, Cannon's algorithm should perform better that our algorithm for $n > 83$. Figure 3.8 shows the efficiency vs n curves for the two algorithms for $p = 64$. It can be seen that as predicted, our algorithm performs better for smaller problem sizes. The experimental cross-over point of the two curves

implementation. For instance, a function call in the program associated with sending or receiving a message could contribute to the message startup overhead.

is at $n = 96$. A slight deviation from the derived value of 83 can be explained due to the fact that the values of t_s and t_w are not exactly the same for the two programs. For 512 processors, the predicted cross-over point is for $n = 295$. Since the number of processors has to be a perfect square for Cannon's algorithm on square matrices, in Figure 3.9, we draw the efficiency vs n curve for $p = 484$ for Cannon's algorithm and for $p = 512$ for the GK algorithm[7]. The cross-over point again closely matches the predicted value. These experiments suggest that the algorithm of Section 3.2.1 can outperform the classical algorithms like Cannon's for a wide range of problem sizes and number of processors. Moreover, as the number of processors is increased, the cross-over point of the efficiency curves of the GK algorithm and Cannon's algorithm corresponds to a very high efficiency. As seen in Figure 3.9, the cross-over happens at $E \approx 0.93$ and Cannon's algorithm cannot outperform the GK algorithm by a wide margin at such high efficiencies. On the other hand, the GK algorithm achieves an efficiency of 0.5 for a matrix size of 112×112, whereas Cannon's algorithm operates at an efficiency of only 0.28 on 484 processors on 110×110 matrices. In other words, in the region where the GK algorithm is better than Cannon's algorithm, the difference in the efficiencies is quite significant.

3.3 Performance and Scalability of Preconditioned Conjugate Gradient Methods on Parallel Computers

Solving large sparse systems of linear equations is an integral part of mathematical and scientific computing and finds application in a variety of fields such as fluid dynamics, structural analysis, circuit theory, power system analysis, surveying, and atmospheric modeling. With the availability of large-scale parallel computers, iterative methods such as the Conjugate Gradient method for solving such systems are becoming increasingly appealing, as they can be parallelized with much greater ease than direct methods. As a result there has been a great deal of interest in implementing the Conjugate Gradient algorithm on parallel computers [6, 12, 62, 73, 79, 101, 122, 138, 139]. In this section, we study performance and scalability of parallel formulations of an iteration of the Preconditioned Conjugate Gradient (PCG) algorithm [50] for solving large sparse linear systems of equations of the form $\mathbf{A}\ \mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a symmetric positive definite matrix. Although,

[7] This is not an unfair comparison because the efficiency can only be better for smaller number of processors.

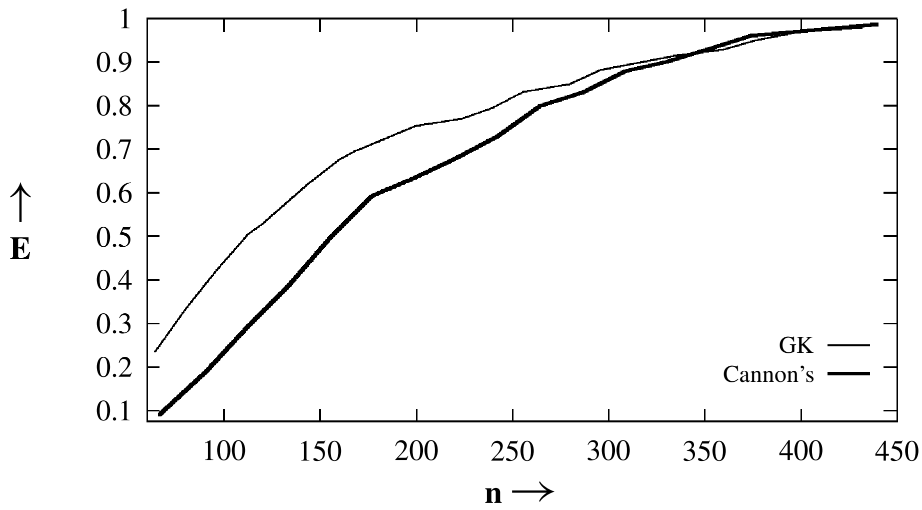


Figure 3.9: Efficiency vs matrix size for Cannon's algorithm (p = 484) and the GK algorithm (p = 512).

we specifically deal with the Preconditioned CG algorithm only, the analysis of the diagonal preconditioner case applies to the non-preconditioned method also. In fact the results of this section can be adapted to a number of iterative methods that use matrix-vector multiplication and vector inner product calculation as the basic operations in each iteration.

3.3.1 The Serial PCG Algorithm

Figure 3.10 illustrates the PCG algorithm [50] for solving a linear system of equations $\mathbf{A}\ \mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a sparse symmetric positive definite matrix. The PCG algorithm performs a few basic operations in each iteration. These are matrix-vector multiplication, vector inner product calculation, scalar-vector multiplication, vector addition and solution of a linear system $\mathbf{M}\ \mathbf{z} = \mathbf{r}$. Here $\mathbf{M}$ is the preconditioner matrix, usually derived from $\mathbf{A}$ using certain techniques. We will consider two kinds of preconditioner matrices $\mathbf{M}$ - *(i)* when $\mathbf{M}$ is chosen to be a diagonal matrix, usually derived from the principal diagonal of $\mathbf{A}$, and *(ii)* when $\mathbf{M}$ is obtained through a truncated Incomplete Cholesky (IC) factorization [73, 138] of $\mathbf{A}$. In the following subsections, we determine begin

$i := 0$; $\mathbf{x}_0 := 0$; $\mathbf{r}_0 := \mathbf{b}$; $\rho_0 := ||\mathbf{r}_0||_2^2$;

while ($\sqrt{\rho_i} > \epsilon ||\mathbf{r}_0||_2$) and ($i < i_{max}$) do

begin

Solve $\mathbf{M}\ \mathbf{z}_i = \mathbf{r}_i$;

$\gamma_i := \mathbf{r}_i^T\ \mathbf{z}_i$;

$i := i + 1$;

if (i = 1) $\mathbf{p}_1 := \mathbf{z}_0$

else begin

$\beta_i := \gamma_{i-1}/\gamma_{i-2}$;

$\mathbf{p}_i := \mathbf{z}_{i-1} + \beta_i \mathbf{p}_{i-1}$;

end;

$\mathbf{w}_i := \mathbf{A}\ \mathbf{p}_i$;

$\alpha_i := \gamma_{i-1}/\mathbf{p}_i^T\ \mathbf{w}_i$;

$\mathbf{x}_i := \mathbf{x}_{i-1} + \alpha_i \mathbf{p}_i$;

$\mathbf{r}_i := \mathbf{r}_{i-1} - \alpha_i \mathbf{w}_i$;

$\rho_i := ||\mathbf{r}_i||_2^2$;

end; $\{while\}$

$\mathbf{x} := \mathbf{x}_i$;

end.

Figure 3.10: The Preconditioned Conjugate Gradient algorithm.

the serial execution time for each iteration of the PCG algorithm with both preconditioners.

Diagonal Preconditioner

During a PCG iteration for solving a system of N equations, the serial complexity of computing the vector inner product, scalar-vector multiplication and vector addition is $\Theta(N)$. If $\mathbf{M}$ is a diagonal matrix, the complexity of solving $\mathbf{M}\ \mathbf{z} = \mathbf{r}$ is also $\Theta(N)$. If there are m non-zero elements in each row of the sparse matrix $\mathbf{A}$, then the matrix-vector multiplication of a CG iteration can be performed in $\Theta(mN)$ time by using a suitable scheme for storing $\mathbf{A}$. Thus, with the diagonal preconditioner, each CG iteration involves a few steps of $\Theta(N)$ complexity and one step of $\Theta(mN)$ complexity. As a result, the serial execution time for one iteration of the PCG algorithm with a diagonal preconditioner can be expressed as follows:

$$W = c_1 N + c_2 m N. \tag{3.21}$$

Here c_1 and c_2 are constants depending upon the floating point speed of the computer and m is the number non-zero elements in each row of the sparse matrix.

The IC Preconditioner

In this section, we only consider the case when $\mathbf{A}$ is a block-tridiagonal matrix of dimension N resulting from the discretization of a 2-dimensional self-adjoint elliptic partial differential equation via finite differences using natural ordering of grid points. Besides the principal diagonal, the matrix $\mathbf{A}$ has two diagonals on each side of the principal diagonal at distances of 1 and $\sqrt{N}$ from it. Clearly, all the vector operations can be performed in $\Theta(N)$ time. The matrix-vector multiplication operation takes time proportional to $5N$. When $\mathbf{M}$ is an IC preconditioner, the structure of $\mathbf{M}$ is identical to that of $\mathbf{A}$.

A method for solving $\mathbf{M}\ \mathbf{z} = \mathbf{r}$, originally proposed for vector machines [138], is briefly described below. A detailed description of the same can be found in [81]. As shown in Section 3.3.2, this method is perfectly parallelizable on CM-5 and other architectures ranging from mesh to hypercube. In fact, this method leads to a parallel formulation of the PCG algorithm that is somewhat more scalable and efficient (in terms of processor utilization) than a formulation using a simple diagonal preconditioner.

The matrix $\mathbf{M}$ can be written as $\mathbf{M} = (\mathbf{I} - \mathbf{L})\mathbf{D}(\mathbf{I} - \mathbf{L}^T)$, where $\mathbf{D}$ is a diagonal matrix and $\mathbf{L}$ is a strictly lower triangular matrix with two diagonals corresponding to the two lower diagonals of $\mathbf{M}$. Thus, the system $\mathbf{Mz} = \mathbf{r}$ may be solved by the following steps:

solve $(\mathbf{I} - \mathbf{L})\mathbf{u} = \mathbf{r}$

solve $\mathbf{Dv} = \mathbf{u}$

solve $(\mathbf{I} - \mathbf{L}^T)\mathbf{z} = \mathbf{v}$

Since $\mathbf{L}$ is strictly lower triangular (i.e., $\mathbf{L}^N = 0$), $\mathbf{u}$ and $\mathbf{z}$ may be written as $\mathbf{u} = \Sigma_{i=0}^{N-1}\mathbf{L}^i\mathbf{r}$ and $\mathbf{z} = \Sigma_{i=0}^{N-1}(\mathbf{L}^i)^T\mathbf{v}$. These series may be truncated to $(k+1)$ terms where $k \ll N$ because $\mathbf{M}$ is diagonally dominant [73, 138]. In our formulation, we form the matrix $\tilde{\mathbf{L}} = (\mathbf{I} + \mathbf{L} + \mathbf{L}^2 + \ldots + \mathbf{L}^k)$ explicitly. Thus, solving $\mathbf{Mz} = \mathbf{r}$ is equivalent to

(i) $\mathbf{u} \approx \tilde{\mathbf{L}}\mathbf{r}$

(ii) $\mathbf{v} \approx \mathbf{D}^{-1}\mathbf{u}$

(iii) $\mathbf{z} \approx \tilde{\mathbf{L}}^T\mathbf{v}$

The number of diagonals in the matrix $\tilde{\mathbf{L}}$ is equal to $(k+1)(k+2)/2$. These diagonals are distributed in $k+1$ clusters at distances of $\sqrt{N}$ from each other. The first cluster, which includes the principal diagonal, has $k+1$ diagonals, and then the number of diagonals in each cluster decreases by one. The last cluster has only one diagonal which is at a distance of $k\sqrt{N}$ from the principal diagonal. Thus solving the system $\mathbf{M}\ \mathbf{z} = \mathbf{r}$, in case of the IC preconditioner, is equivalent to performing one vector division (step (ii)) and two matrix-vector multiplications (steps (i) and (iii)), where each matrix has $(k+1)(k+2)/2$ diagonals. Hence the complexity of solving $\mathbf{M}\ \mathbf{z} = \mathbf{r}$ for this case is proportional to $(k+1)(k+2)N$ and the serial execution time of one complete iteration is given by the following equation:

$$W = (c_1 + (5 + (k+1)(k+2))c_2) \times N.$$

Here c_1 and c_2 are constants depending upon the floating point speed of the computer. The above equation can be written compactly as follows by putting $\eta(k) = c_1 + c_2(5 + (k+1)(k+2))$.

$$W = \eta(k)N. \tag{3.22}$$

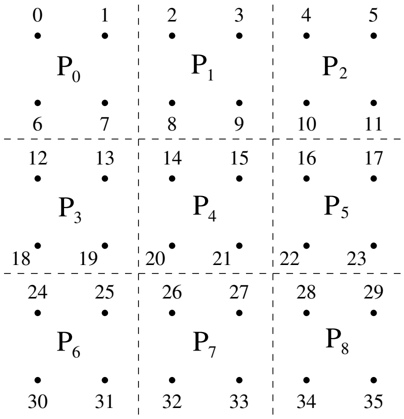


Figure 3.11: Partitioning a finite difference grid on a processor mesh.

3.3.2 Scalability Analysis: Block-Tridiagonal Matrices

In this section we consider the parallel formulation of the PCG algorithm with block-tridiagonal matrix of coefficients resulting from a square two dimensional finite difference grid with natural ordering of grid points. Each point on the grid contributes one equation to the system $\mathbf{A}\ \mathbf{x} = \mathbf{b}$; i.e., one row of matrix $\mathbf{A}$ and one element of the vector $\mathbf{b}$.

The Parallel Formulation

The points on the finite difference grid can be partitioned among the processors of a mesh connected parallel computer as shown in Figure 3.11. Since a mesh can be mapped onto a hypercube or a fully connected network, a mapping similar to the one shown in Figure 3.11 will work for these architectures as well.

In the PCG algorithm, the scalar-vector multiplication and vector addition operations do not involve any communication overhead, as all the required data is locally available on each processor and the results are stored locally as well. If the diagonal preconditioner is used, then even solving the system $\mathbf{M}\ \mathbf{z} = \mathbf{r}$ does not require any data communication because the resultant vector $\mathbf{z}$ can be obtained by simply dividing each element of $\mathbf{r}$ by the corresponding diagonal element of $\mathbf{M}$, both of which are locally available on each processor. Thus, the operations that involve any communication overheads are computation of inner products, matrix-vector multiplication, and, in case of the IC preconditioner, solving the system $\mathbf{M}\ \mathbf{z} = \mathbf{r}$.

In the computation of the inner product of two vectors, the corresponding elements of each vector are multiplied locally and these products are summed up. The value of the inner product is the sum of these partial sums located at each processor. The data communication required to perform this step involves adding all the partial sums and distributing the resulting value to each processor.

In order to perform parallel matrix-vector multiplication involving the block-tridiagonal matrix, each processor has to acquire the vector elements corresponding to the column numbers of all the matrix elements it stores. It can be seen from Figure 3.11 that each processor needs to exchange information corresponding to its $\sqrt{N/p}$ boundary points with each of its four neighboring processors. After this communication step, each processor gets all the elements of the vector it needs to multiply with all the matrix elements it stores. Now the multiplications are performed and the resultant products are summed and stored locally on each processor.

A method for solving $\mathbf{M}\ \mathbf{z} = \mathbf{r}$ for the IC preconditioner $\mathbf{M}$ has been described in Section 3.3.1. This computation involves multiplication of a vector with a lower triangular matrix $\tilde{\mathbf{L}}$ and an upper triangular matrix $\tilde{\mathbf{L}}^T$, where each triangular matrix has $(k+1)(k+2)/2$ diagonals arranged in the fashion described in Section 3.3.1. If the scheme of partitioning $\mathbf{A}$ among the processors (every processor stores $\sqrt{N/p}$ clusters of $\sqrt{N/p}$ matrix rows each) is extended to $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{L}}^T$, then it can be shown that for $\sqrt{N/p} > k$ the data communication for performing these matrix-vector multiplications requires each processor in the mesh to send $k\sqrt{N/p}$ vector elements to its immediate north, east, south and west neighbors.

Communication Overheads

In this section we determine the overall overhead due to parallel processing in a PCG iteration. As discussed in the previous subsection, the operations that incur communication overheads are computation of inner products, matrix-vector multiplication and solving the system $\mathbf{M}\ \mathbf{z} = \mathbf{r}$. Let these three components of T_o be $T_o^{Inner-Prod}$, $T_o^{Matrix-Vector}$, and $T_o^{PC-solve}$, respectively. In order to compute each component of T_o, first we compute the time spent by each processor in performing data communication for the operation in question. The product of this time with p gives the total time spent by all the processors in performing this operation and T_o is the sum of each of these individual components.

Overhead in Computing the Inner Products The summation of p partial sums (one located at each processor) can be performed through recursive doubling in $(t_s + t_w) \log p$ time on a hypercube, and in $(t_s + t_w) \log p + t_h \sqrt{p}$ time on a two dimensional mesh. It takes the same amount of time to broadcast the final result to each processor. On a machine like the CM-5, such operations (known as *reduction* operations) are performed by the control network in a small constant time which can be ignored in comparison with the overhead incurred in other parts of the algorithm, such as, matrix-vector multiplication. The following equations give the expressions for the total overhead for each iteration over all the processors for computing the three inner products (lines 6, 14 and 17 in Figure 3.10) on different architectures. In these equations t_w is ignored in comparison with t_s as the latter is much larger in most practical machines.

$$T_o^{Inner-Prod} \approx 6(t_s \log p + t_h \sqrt{p}) \times p \qquad (2-D\ mesh). \qquad (3.23)$$

$$T_o^{Inner-Prod} \approx 6t_s \log p \times p \qquad (Hypercube). \qquad (3.24)$$

$$T_o^{Inner-Prod} \approx 0 \qquad (CM-5). \qquad (3.25)$$

Overhead Due to Matrix-Vector Multiplication During matrix-vector multiplication, each processor needs to exchange vector elements corresponding to its boundary grid points with each of its four neighboring processors. This can be done in $4t_s + 4t_w\sqrt{N/p}$ time on a mesh, hypercube or a virtually fully connected network like that of the CM-5. The total overhead for this step is given by the following equation:

$$T_o^{Matrix-Vector} = 4(t_s + t_w\sqrt{\frac{N}{p}}) \times p. \qquad (3.26)$$

Overhead in Solving M z = r If a simple diagonal preconditioner is used, then this step does not require any communication. For the case of the IC preconditioner, as discussed in Section 3.3.1, the communication pattern for this step is identical to that for matrix-vector multiplication, except that $k\sqrt{N/p}$ vector elements are now exchanged at each processor boundary. The required expression for the overall overhead for this step is as follows.

$$T_o^{PC-solve} = 0 \qquad (diagonal\ preconditioner). \qquad (3.27)$$

$$T_o^{PC-solve} = 4(t_s + t_w k\sqrt{\frac{N}{p}}) \times p \qquad (IC\ preconditioner). \qquad (3.28)$$

Total Overhead Now, having computed each of its components, we can write the expressions for the overall T_o using Equations 3.23 through 3.28. The following equations give the required approximate expressions (after dropping the insignificant terms, if any) for T_o.

- **The CM-5**

$$T_o = 4(t_s p + t_w\sqrt{pN}) \qquad (diagonal\ preconditioner). \qquad (3.29)$$

$$T_o = 4(2t_s p + t_w(k+1)\sqrt{pN}) \qquad (IC\ preconditioner). \qquad (3.30)$$

- **Hypercube**

$$T_o = 6t_s p \log p + 4t_w\sqrt{pN} \qquad (diagonal\ preconditioner). \qquad (3.31)$$

$$T_o = 6t_s p \log p + 4(k+1)t_w\sqrt{pN} \qquad (IC\ preconditioner). \qquad (3.32)$$

- **Mesh**

$$T_o = 6t_s p \log p + 4t_w\sqrt{pN} + 6t_h p\sqrt{p} \qquad (diagonal\ preconditioner). \qquad (3.33)$$

$$T_o = 6t_s p \log p + 4(k+1)t_w\sqrt{pN} + 6t_h p\sqrt{p} \qquad (IC\ preconditioner). \qquad (3.34)$$

Isoefficiency Analysis

Since we perform the scalability analysis with respect to one PCG iteration, the problem size W will be considered to be $\Theta(N)$ and we will study the rate at which N needs to grow with p for a fixed efficiency as a measure of scalability. If $T_o(W, P)$ is the total overhead, the efficiency E is given by $W/(W + T_o(W, p))$. Clearly, for a given N, if p increases, then E will decrease because $T_o(W, p)$ increases with p. On the other hand, if N increases, then E increases because the rate of increase of T_o is slower than that of W for a scalable algorithm. The isoefficiency function for a certain efficiency E can be obtained by equating W with $T_o E/(1-E)$ (Equation 2.4) and then solving this equation to determine N as a function of p. In our parallel formulation, T_o has a number of different terms due to t_s, t_w, t_h, etc. When there are multiple terms of different orders of magnitude with respect to p and N in W or T_o, it is often impossible to obtain the isoefficiency function as a closed form function of p. For a parallel algorithm-architecture combination, as p and W increase, efficiency is guaranteed not to drop if none of the terms of T_o grows faster than W. Therefore, if T_o has multiple terms, we balance W against each individual term of T_o to compute the respective isoefficiency function. The component of T_o that requires the problem size to grow at the fastest rate with respect to p determines the overall asymptotic isoefficiency function of the entire computation.

Diagonal Preconditioner Since the number of elements per row (m) in the matrix of coefficients is five, from Equation 3.21, we obtain the following expression for W:

$$W = N(c_1 + 5c_2). \tag{3.35}$$

Now we will use Equations 3.29, 3.31, 3.33, and 3.35 to compute the isoefficiency functions for the case of diagonal preconditioner on different architectures.

- **The CM-5**

  According to Equation 2.4, in order to determine the isoefficiency term due to t_s, W has to be proportional to $4et_s p$ (see Equation 3.29), where $e = E/(1-E)$ and E is the desired efficiency that has to be maintained. Therefore,

$$\begin{aligned} N(c_1 + 5c_2) &\propto 4et_s p, \\ N &\propto p\frac{4et_s}{c_1 + 5c_2}. \end{aligned} \tag{3.36}$$

  The term due to t_w in T_o is $4t_w\sqrt{pN}$ (see Equation 3.29) and the associated isoefficiency condition is determined as follows:

$$\begin{aligned} N(c_1 + 5c_2) &\propto 4et_w\sqrt{pN}, \\ \sqrt{N} &\propto \sqrt{p}\frac{4et_w}{c_1 + 5c_2}, \\ N &\propto p(\frac{4et_w}{c_1 + 5c_2})^2. \end{aligned} \tag{3.37}$$

According to both Equations 3.36 and 3.37, the overall isoefficiency function for the PCG algorithm with a simple diagonal preconditioner is $\Theta(p)$; i.e, it is a highly scalable parallel system which requires only a linear growth of problem size with respect to p to maintain a certain efficiency.

- **Hypercube**

Since the terms due to t_w are identical in the overhead functions for both the hypercube and the CM-5 architectures, the isoefficiency condition resulting from t_w is still determined by Equation 3.37. The term associated with t_s yields the following isoefficiency function:

$$N \propto \frac{6et_s}{c_1 + 5c_2} p \log p. \tag{3.38}$$

Since Equation 3.38 suggests a higher growth rate for the problem size with respect to p to maintain a fixed E, it determines the overall isoefficiency function which is $\Theta(p \log p)$. Also, t_s has a higher impact on the efficiency on a hypercube than on the CM-5.

- **Mesh**

The isoefficiency term due to t_s will be the same as in Equation 3.38 because the terms due to t_s in the overhead functions for the hypercube and the mesh are identical. Similarly, the isoefficiency term due to t_w will be the same as in Equation 3.37. Balancing W against the term due to t_h in Equation 3.33, we get

$$\begin{aligned} N(c_1 + 5c_2) &\propto 6et_h p\sqrt{p}, \\ N &\propto \frac{6et_h}{c_1 + 5c_2} p^{1.5}. \end{aligned} \tag{3.39}$$

Now N has to grow as $\Theta(p)$ to balance the overheads due to t_w (Equation 3.37), as $\Theta(p \log p)$ to balance the overhead due to t_s (Equation 3.38), and as $\Theta(p^{1.5})$ to balance the overhead due to t_h (Equation 3.39). Clearly, Equation 3.39 determines the asymptotic isoefficiency function for the mesh.

IC Preconditioner The following overall isoefficiency functions can be derived for the case of the IC preconditioner using the analysis similar to that in the case of rge diagonal preconditioner by taking the expression for W from Equation 3.22 and expressions for T_o from Equations 3.30, 3.32, and 3.34 for various architectures:

$$N \propto p(\frac{4et_w(k+1)}{\eta(k)})^2 \qquad (CM-5). \tag{3.40}$$

$$N \propto \frac{6et_s}{\eta(k)} p \log p \qquad (hypercube). \tag{3.41}$$

$$N \propto \frac{6et_h}{\eta(k)} p\sqrt{p} \qquad (mesh). \tag{3.42}$$

The isoefficiency functions given by the above equations are asymptotically the same as those for the diagonal preconditioner (Equations 3.36 through 3.39), but with different constants. The impact of these constants on the overall efficiency and scalability of the PCG iteration is discussed in the next section.

Discussion

A number of interesting inferences can be drawn from the scalability analysis performed in Section 3.3.2. For a typical MIMD mesh or hypercube with $t_s \gg t_w$, matrix-vector multiplication and solution of $\mathbf{M}\ \mathbf{z} = \mathbf{r}$ with the preconditioner $\mathbf{M}$ incur relatively small communication overheads compared to the computation of inner-product of vectors. For these architectures, the inner-products calculation contributes the overhead term that determines the overall isoefficiency function and the total communication cost is dominated by the message startup time t_s. In contrast, on the CM-5, the communication overhead in the inner product calculation is minimal due to the control network. As a result, the CM-5 is ideally scalable for an iteration of this algorithm; i.e., speedups proportional to the number of processors can be obtained by simply increasing N linearly with p. Equivalently, bigger instances of problems can be solved in a fixed given time by using linearly increasing number of processors. In the absence of the control network, even on the CM-5 the overhead due to message startup time in the inner product computation would have dominated and the isoefficiency function of the PCG algorithm would have been greater than $\Theta(p)$. Thus, for this application, the control network is highly useful.

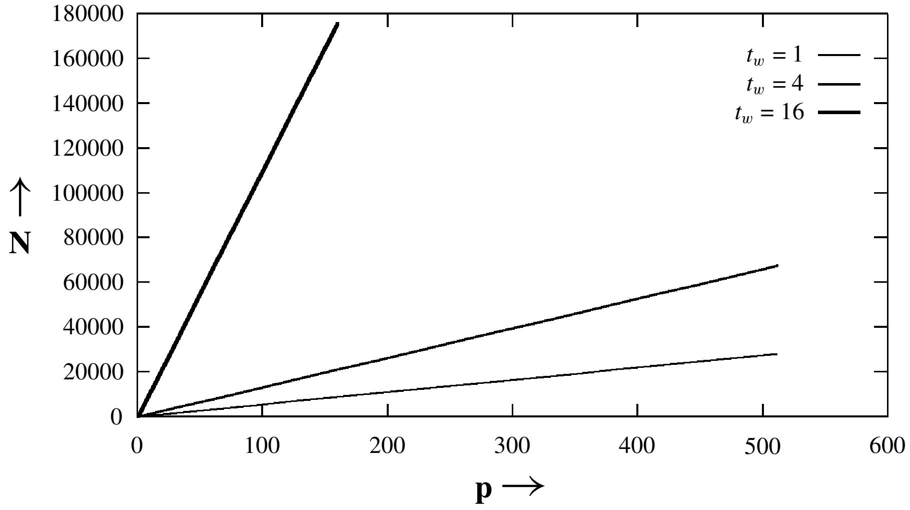


Figure 3.12: Isoefficiency curves for $E = 0.5$ with a fixed processor speed and different values of channel bandwidth.

There are certain iterative schemes, like the Jacobi method [50], that require inner product calculation only for the purpose of performing a convergence check. In such schemes, the parallel formulation can be made almost linearly scalable even on mesh and hypercube architectures by performing the convergence check once in a few iterations. For instance, the isoefficiency function for the Jacobi method on a hypercube is $\Theta(p \log p)$. If the convergence check is performed once every $\log p$ iterations, the amortized overhead due to inner product calculation will be $\Theta(p)$ in each iteration, instead of $\Theta(p \log p)$ and the isoefficiency function of the modified scheme will be $\Theta(p)$. Similarly, performing the convergence check once in every $\sqrt{p}$ iterations on a mesh architecture will result in linear scalability.

Equations 3.36 and 3.37 suggest that the PCG algorithm is highly scalable on a CM-5 type architecture and a linear increase in problem size with respect to the number of processors is sufficient to maintain a fixed efficiency. However, we would like to point out as to how hardware related parameters other than the number of processors affect the isoefficiency function. According to Equation 3.37, N needs to grow in proportion to the square of the ratio of t_w to the unit computation

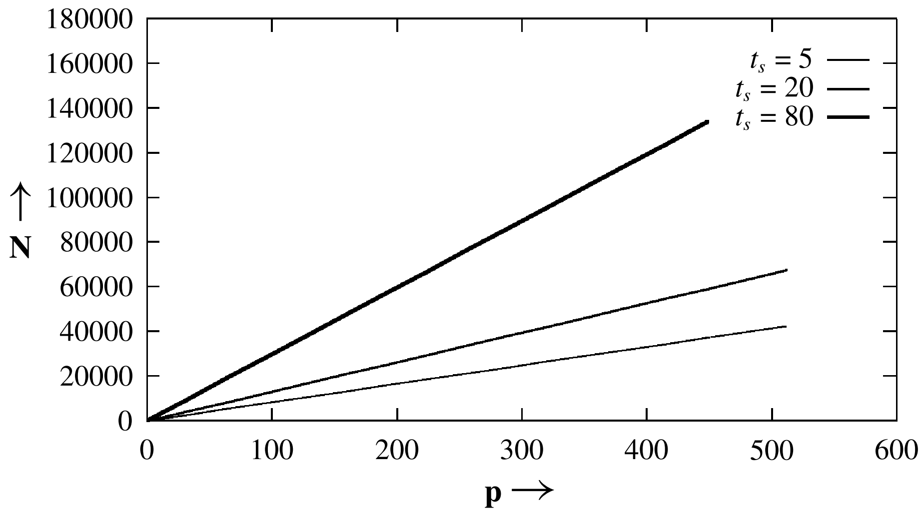


Figure 3.13: Isoefficiency curves for $E = 0.5$ with a fixed processor speed and different values of message startup time.

time on a processor. According to Equation 3.36, N needs to grow in proportion to the ratio of t_s to the unit computation time. Thus isoefficiency is also a function of the ratio of communication and computation speeds. Figure 3.12 shows theoretical isoefficiency curves for different values of t_w for a hypothetical machine with fixed processor speed ($(c_1 + 5c_2)$ = 2 microseconds[8]) and message startup time (t_s = 20 microseconds). Figure 3.13 shows isoefficiency curves for different values of t_s for the same processor speed with t_w = 4 microseconds. These curves show that the isoefficiency function is much more sensitive to changes in t_w than t_s. Note that t_s and t_w are normalized with respect to CPU speed. Hence, effectively t_w could go up if either the CPU speed increases or inter-processor communication bandwidth decreases. Figure 3.12 suggests that it is very important to have a balance between the speed of the processors and the bandwidth of the communication channels, otherwise good efficiencies will be hard to obtain; i.e., very large problems will be required.

---

[8] This corresponds to a throughput of roughly 10 MFLOPS. On a fully configured CM-5 with vector units, a throughput of this order can be achieved very easily.

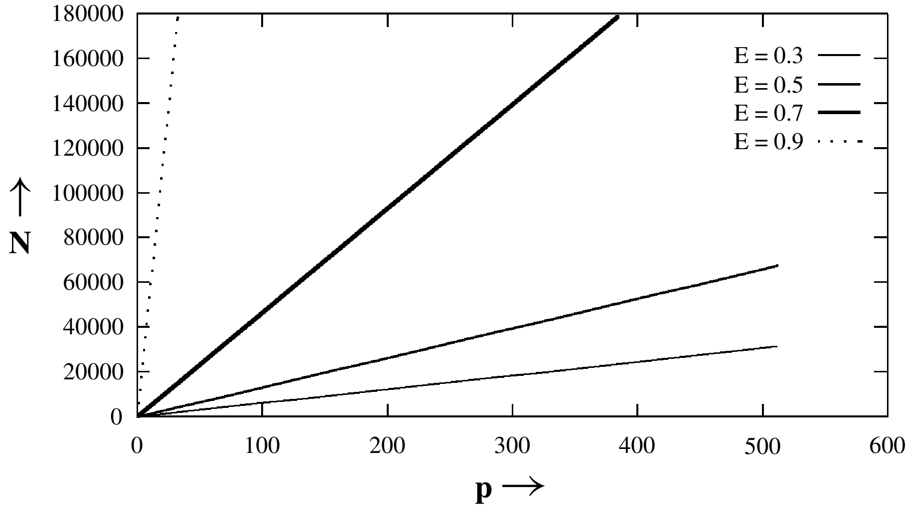


Figure 3.14: Isoefficiency curves for different efficiencies with $t_s = 20$ and $t_w = 4$.

Apart from the computation and communication related constants, isoefficiency is also a function of the efficiency that is desired to be maintained. N needs to grow in proportion to $(E/(1-E))^2$ in order to balance the useful computation W with the overhead due to t_w (Equation 3.37) and in proportion to $E/(1-E)$ to balance the overhead due to t_s (Equation 3.36). Figure 3.14 graphically illustrate the impact of desired efficiency on the scalability of a PCG iteration. The figure shows that as higher and higher efficiencies are desired, it becomes increasingly difficult to obtain them. An improvement in the efficiency from 0.3 to 0.5 takes little effort, but it takes substantially larger problem sizes for a similar increase in efficiency from 0.7 to 0.9. The constant $(et_w/(c_1 + 5c_2))^2$ in Equation 3.37 indicates that a better balance between communication channel bandwidth and the processor speed will reduce the impact of increasing the efficiency on the rate of growth of problem size and higher efficiencies will be obtained more readily.

The isoefficiency functions for the case of the IC preconditioner for different architectures given by Equations 3.40 through 3.41 are of the same order of magnitude as those for the case of the diagonal preconditioner given by Equations 3.36 to 3.39. However, the constants associated with the isoefficiency functions for the IC preconditioner are smaller due to the presence of $\eta(k)$ in the denominator which is $c_1 + 5c_2 + (k+1)(k+2)c_2$. As a result, the rate at which the problem size should increase to maintain a particular efficiency will be asymptotically the same for both preconditioners, but for the IC preconditioner the same efficiency can be realized for a smaller problem size than in the case of the diagonal preconditioner. Also, for given p and N, the parallel implementation with the IC preconditioner will yield a higher efficiency and speedup than one with the diagonal preconditioner. Thus, if enough processors are used, a parallel implementation with the IC preconditioner may execute faster than one with a simple diagonal preconditioner even if the latter was faster in a serial implementation. The reason is that the number of iterations required to obtain a residual whose norm satisfies a given constraint does not increase with the number of processors used. However, the efficiency of execution of each iteration will drop more rapidly in case of the diagonal preconditioner that in case of the IC preconditioner as the number of processors are increased.

It can be shown that the scope of the results of this section is not limited to the type of block-tridiagonal matrices described in Section 3.3.1 only. The results hold for all symmetric block-tridiagonal matrices where the distance of the two outer diagonals from the principal diagonal is N^r $(0 < r < 1)$. Such a matrix will result from a rectangular $N^r \times N^{1-r}$ finite difference grid with natural ordering of grid points. Similar scalability results can also be derived for matrices resulting from three dimensional finite difference grids. These matrices have seven diagonals and the scalability of the parallel formulations of an iteration of the PCG algorithm on a hypercube or the CM-5 is the same as in case of block-tridiagonal matrices. However, for the mesh architecture, the results will be different. On a two dimensional mesh of processors, the isoefficiency due to matrix-vector multiplication will be $\Theta(p^{3/2})$ for the matrices resulting from 3-D finite difference grids. Thus, unlike the block-tridiagonal case, here the overheads due to both matrix vector multiplication and inner-product computation are equally dominant on a mesh.

3.3.3 Scalability Analysis: Unstructured Sparse Matrices

In this section, we consider a more general form of sparse matrices, in which the non-zeros are distributed randomly and do not form a regular pattern that can be utilized effectively. Such matrices occur in some applications, notably in linear programming problems. Often, such systems

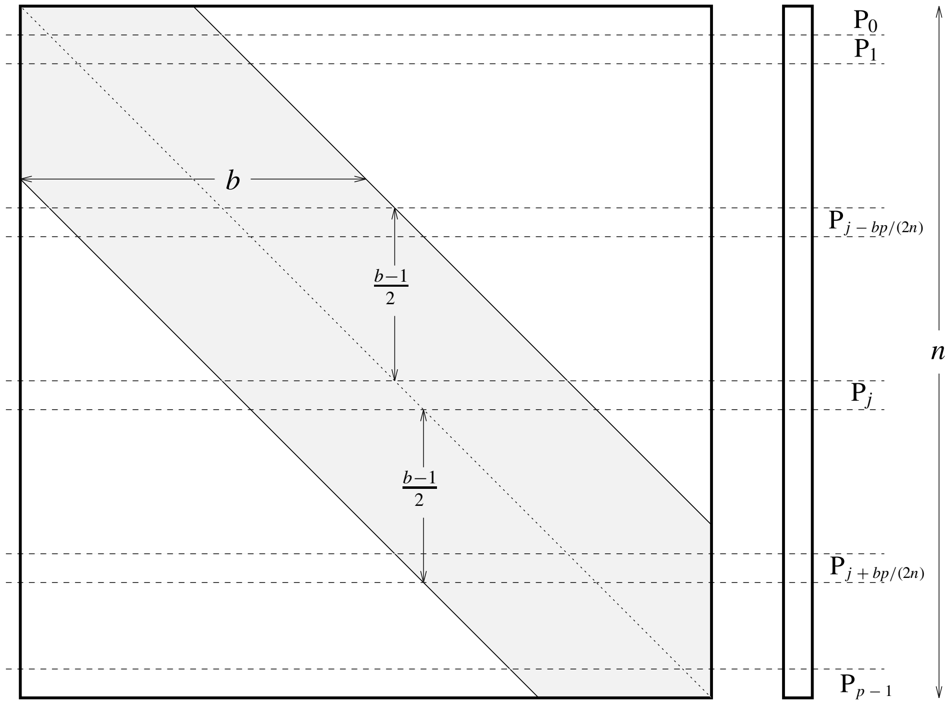


Figure 3.15: Partition of a banded sparse matrix and a vector among the processors.

are encountered where the non-zero elements of matrix **A** occur only within a band around the principal diagonal. Even if the non-zero elements are scattered throughout the matrix, it is often possible to restrict them to a band through certain re-ordering techniques [45, 48]. Such a system is shown in Figure 3.15 in which the non-zero elements of the sparse matrix are randomly distributed within a band along the principal diagonal. Let the width of the band of the $N \times N$ matrix be given by b, and $b = \beta N^y$, and $0 \leq y \leq 1$. Suitable values of the constants β and y can be selected to represent the kind of systems being solved. If $\beta = 1$ and $y = 1$, we have the case of a totally unstructured sparse matrix.

The matrix **A** is stored in the *Ellpack-Itpack* format [121]. In this storage scheme, the non-zero elements of the matrix are stored in an $N \times m$ array while another $N \times m$ integer array stores the column numbers of the matrix elements. It can be shown that *co-ordinate* and the *compressed sparse column* storage formats incur much higher communication overheads, thereby leading to unscalable parallel formulations. Two other storage schemes, namely *jagged diagonals* [121] and *compressed sparse rows* involve communication overheads similar to the *Ellpack-Itpack* scheme, but the latter is the easiest to work with when the number of non-zero elements is almost the same in each row of the sparse matrix. The matrix **A** and the vector **b** are partitioned among p processors as shown in Figure 3.15. The rows and the columns of the matrix and the elements of the vector are numbered from 0 to $N-1$. Processor P_i stores $(N/p)i$ to $(N/p)(i+1)-1$ rows of matrix **A** and elements with indices from $(N/p)i$ to $(N/p)(i+1)-1$ of vector **b**. The preconditioner and all the other vectors used in the computation are partitioned similarly.

We will study the application of only the diagonal preconditioner in this case; hence the serial execution time is given by Equation 3.21. Often the number of non-zero elements per row of the matrix **A** is not constant but increases with N. Let $m = \alpha N^x$, where the constants α and x ($0 \le x \le 1$) can be chosen to describe the kind of systems being solved. A more general expression for W, therefore, would be as follows:

$$W = c_1 N + c_2 \alpha N^{1+x}. \tag{3.43}$$

Communication Overheads

For the diagonal preconditioner, $T_o^{PC-solve} = 0$ as discussed in Section 3.3.2. It can be shown that $T_o^{Matrix-Vector}$ dominates $T_o^{Inner-Prod}$ for most practical cases. Therefore, $T_o^{Matrix-Vector}$ can be considered to represent the overall overhead T_o for the case of unstructured sparse matrices.

If the distribution of non-zero elements in the matrix is unstructured, each row of the matrix could contain elements belonging to any column. Thus, for matrix-vector multiplication, any processor could need a vector element that belongs to any other processor. As a result, each processor has to send its portion of the vector of size N/p to every other processor. If the bandwidth of the matrix **A** is b, then the ith row of the matrix can contain elements belonging to columns $i - b/2$ to $i + b/2$. Since a processor contains N/p rows of the matrix and N/p elements of each vector, it will need the elements of the vector that are distributed among the processors that lie within a distance of $bp/(2N)$ on its either side; i.e., processor P_i needs to communicate with all the processors P_j such that $i - bp/(2N) \le j < i$ and $i < j \le i + bp/(2N)$. Thus the total number of communication steps in which each processor can acquire all the data it needs to perform matrix-vector multiplication will be bp/N. As a result, the following expression gives the value of T_o for $b = \beta N^y$.

$$T_o = (t_s \beta p N^{y-1} + t_w \beta N^y) \times p. \tag{3.44}$$

It should be noted that this overhead is the same for all architectures under consideration in this section from a linear array to a fully connected network.

Isoefficiency Analysis

The size W of the problem at hand is given by Equation 3.43, which may be greater than $\Theta(N)$ for $x > 0$. Strictly speaking, the isoefficiency function is defined as the rate at which the problem size needs to grow as a function of p to maintain a fixed efficiency. However, to simplify the interpretation of the results, in this section we will measure the scalability in terms of the rate at which N (the size of the linear system of equations to be solved) needs to grow with p rather than rate at which the quantity $c_1 N + c_2 \alpha N^{1+x}$ should grow with respect to p.

According to Equation 2.4, the following condition must be satisfied in order to maintain a fixed efficiency E:

$$c_1 N + c_2 \alpha N^{1+x} = \frac{E}{1-E}(t_s \beta p^2 N^{y-1} + t_w \beta p N^y),$$

$$c_2 \alpha N^{x+y} + c_1 N^y - \frac{E}{1-E} t_w \beta p N^{2y-1} - \frac{E}{1-E} t_s \beta p^2 N^{2y-2} = 0, \tag{3.45}$$

$$N = f_E(p, x, y, \alpha, \beta, t_w, t_s, c_1, c_2). \tag{3.46}$$

From Equation 3.45, it is not possible to compute the isoefficiency function f_E in a closed form for general x and y. However, Equation 3.45 can be solved for N for specific values of x and y. We, therefore, compute the isoefficiency functions for a few interesting cases that result from assigning some typical values to the parameters x and y. Table 3.4 gives a summary of the scalability analysis for these cases. In order to maintain the efficiency at some fixed value E, the size N of the system has to grow according to the expression in the third column of the Table 3.4, where $e = E/(1-E)$.

The above analysis shows that a PCG iteration is unscalable for a totally unstructured sparse matrix of coefficients with a constant number of non-zero elements in each row. However, it can be rendered scalable by either increasing the number of non-zero elements per row as a function of the matrix size, or by restricting the non-zero elements into a band of width less than $\Theta(N)$ for an $N \times N$ matrix using some re-ordering scheme. Various techniques for re-ordering sparse systems to yield banded or partially banded sparse matrices are available [45, 48]. These techniques may vary in their complexity and effectiveness. By using Equation 3.45 the benefit of a certain degree of re-ordering can be quantitatively determined in terms of improvement in scalability.

	Parameter values	Isoefficiency function	Interpretation in terms of scalability
1.	$x = 0$, $y = 1$	Does not exist	Unscalable
2.	$x = 0$, $y = \frac{1}{2}$	$N \propto p^2(\frac{e\beta t_w}{c_1+c_2\alpha})^2$	O(p^2) scalability (moderately scalable)
3.	$x = 1$, $y = 1$	$N \propto p^{\frac{et_w\beta+\sqrt{e^2t_w^2\beta^2+4c_2\alpha\beta et_s}}{2c_2\alpha}}$	Linearly scalable with a high constant
4.	$x = 0$, $y = 0$	$N \propto p\frac{et_w\beta}{2(c_1+c_2\alpha)}\left(1+\sqrt{1+\frac{4t_s(c_1+c_2\alpha)}{et_w^2\beta}}\right)$	Linear scalability (highly scalable)
5.	$x = \frac{1}{2}$, $y = \frac{1}{2}$	$N \propto p\frac{et_w\beta}{2c_2\alpha}\left(1+\sqrt{1+\frac{4t_sc_2\alpha}{et_w^2\beta}}\right)$	Linear scalability (highly scalable)

Table 3.4: Scalability of a PCG iteration with unstructured sparse matrices of coefficients. The average number of entries in each row of the $N \times N$ matrix is αN^x and these entries are located within a band of width βN^y along the principal diagonal.

3.3.4 Experimental Results and their Interpretations

The analytical results derived in the earlier sections were verified through experimental data obtained by implementing the PCG algorithm on the CM-5. Both block-tridiagonal and unstructured sparse symmetric positive definite matrices were generated randomly and used as test cases. The degree of diagonal dominance of the matrices was controlled such that the algorithm performed enough number of iterations to ensure the accuracy of our timing results. Often, slight variation in the number of iterations was observed as different number of processors were used. In the parallel formulation of the PCG algorithm, both matrix-vector multiplication and the inner product computation involve communication of certain terms among the processors which are added up to yield a certain quantity. For different values of p, the order in which these terms are added could be different. As a result, due to limited precision of the data, the resultant sum may have a slightly different value for different values of p. Therefore, the criterion for convergence might not be satisfied after exactly the same number of iterations for all p. In our experiments, we normalized

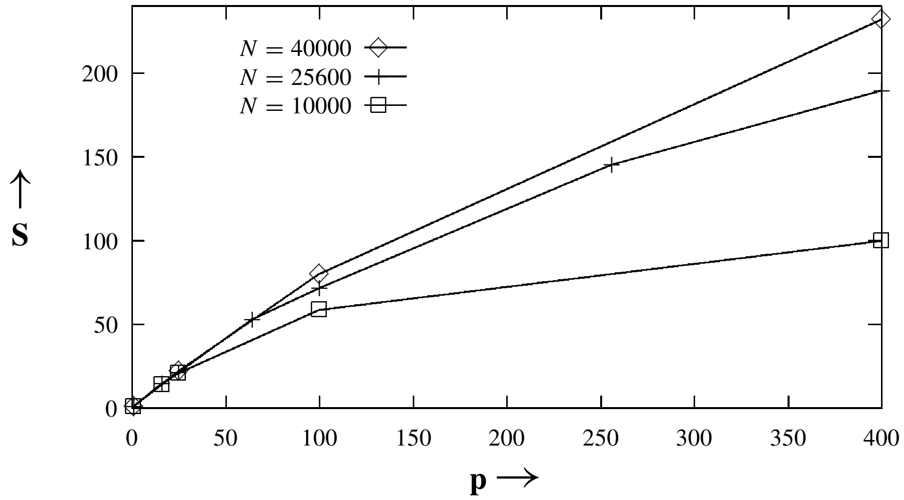


Figure 3.16: Speedup curves for block-tridiagonal matrices with diagonal preconditioner.

the parallel execution time with respect to the number of iterations in the serial case in order to compute the speedups and efficiencies accurately. Sparse linear systems with matrix dimension varying from 400 to over 64,000 were solved using up to 512 processors. For block-tridiagonal matrices, we implemented the PCG algorithms using both the diagonal and the IC preconditioners. For unstructured sparse matrices, only the diagonal preconditioner was used. The configuration of the CM-5 used in our experiments had only a SPARC processor on each node which delivered approximately 1 MFLOPS (double-precision) in our implementation. The message startup time for the program was observed to be about 150 microseconds and the per-word transfer time for 8 byte words was observed to be about 3 microseconds[9].

Figure 3.16 shows experimental speedup curves for solving problems of different sizes using the diagonal preconditioner on block-tridiagonal matrices.

As expected, for a given number of processors, the speedup increases with increasing problem size. Also, for a given problem size, the speedup does not continue to increase with the number of

[9] These values do not necessarily reflect the communication speed of the hardware but the overheads observed for our implementation. For instance copying the data in and out of the buffers in the program contributes to the per-word overhead. Moreover, the machine used in the experiments was still in beta testing phase, hence the performance obtained in our experiments may not be indicative of the achievable performance of the machine.

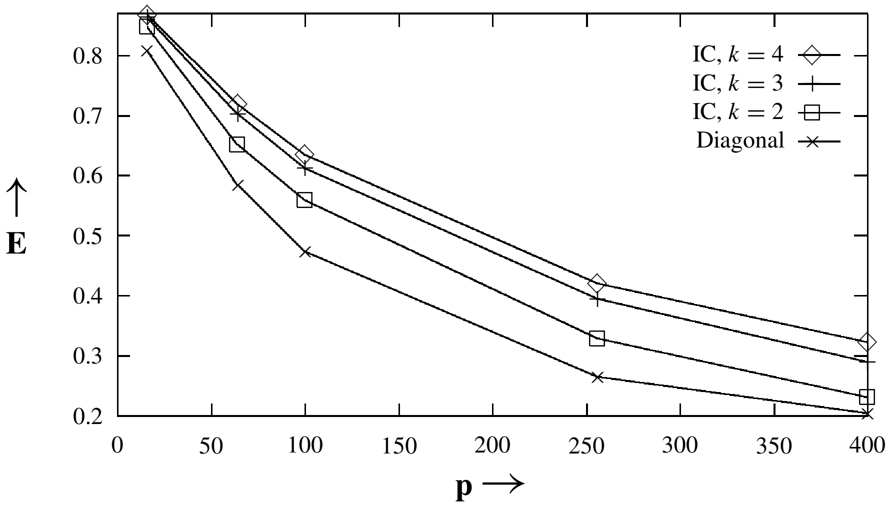


Figure 3.17: Efficiency curves for the diagonal and the IC preconditioner with a 1600×1600 matrix of coefficients.

processors, but tends to saturate.

Recall that the use of the IC preconditioner involves substantially more computation per iteration of the PCG algorithm over a simple diagonal preconditioner, but it also reduces the number of iterations significantly. As the approximation of the inverse of the preconditioner matrix is made more accurate (i.e., k is increased, as discussed in Section 3.3.1), the computation per iteration continues to increase, while the number of iterations decreases. The overall performance is governed by the amount of increase in the computation per iteration and the reduction in the number of iterations. As discussed in Section 3.3.3, for the same number of processors, an implementation with the IC preconditioner will obtain a higher efficiency[10] (and hence speedup) than one with the diagonal preconditioner. Even in case of the IC preconditioner, speedups will be higher for higher values of k. Figure 3.17 shows the efficiency curves for the diagonal preconditioner and the IC preconditioner for k = 2, 3 and 4 for a given problem size. From this figure it is clear that one of the IC preconditioning schemes may yield a better overall execution time in a parallel

[10] The efficiency of a parallel formulation is computed with respect to an identical algorithm running on a single processor.

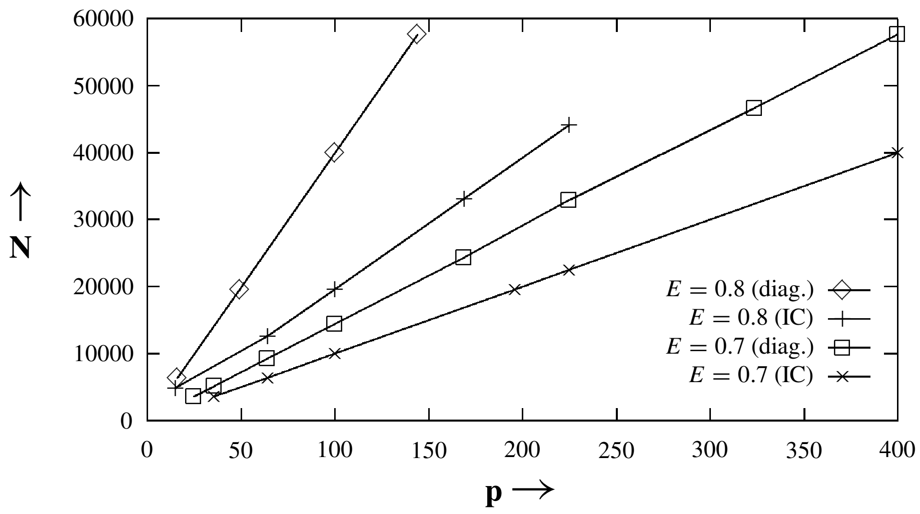


Figure 3.18: Isoefficiency curves for the two preconditioning schemes.

implementation due to a better speedup than the diagonal preconditioning scheme, even if the latter is faster in a serial implementation. For instance, assume that on a serial machine the PCG algorithm runs 1.2 times faster with a diagonal preconditioner than with the IC preconditioner for a certain system of equations. As shown in Figure 3.17, with 256 processors on the CM-5, the IC preconditioner implementation with $k = 3$ executes with an efficiency of about 0.4 for an 80×80 finite difference grid, while the diagonal preconditioner implementation attains an efficiency of only about 0.26 on the same system. Therefore, unlike on a serial computer, on a 256 processor CM-5 the IC preconditioner implementation for this system with $k = 3$ will be faster by a factor of $0.4/0.26 \times 1.0/1.2 \approx 1.3$ than a diagonal preconditioner implementation.

In Figure 3.18, experimental isoefficiency curves are plotted for the two preconditioners by experimentally determining the efficiencies for different values of N and p and then selecting and plotting the (N, P) pairs with nearly the same efficiencies. As predicted by Equations 3.36, 3.37, and 3.40, the N versus p curves for a fixed efficiency are straight lines. These equations, as well as Figure 3.18, suggest that this is a highly scalable parallel system and requires only a linear growth of problem size with respect to p to maintain a certain efficiency. This figure also shows that the IC

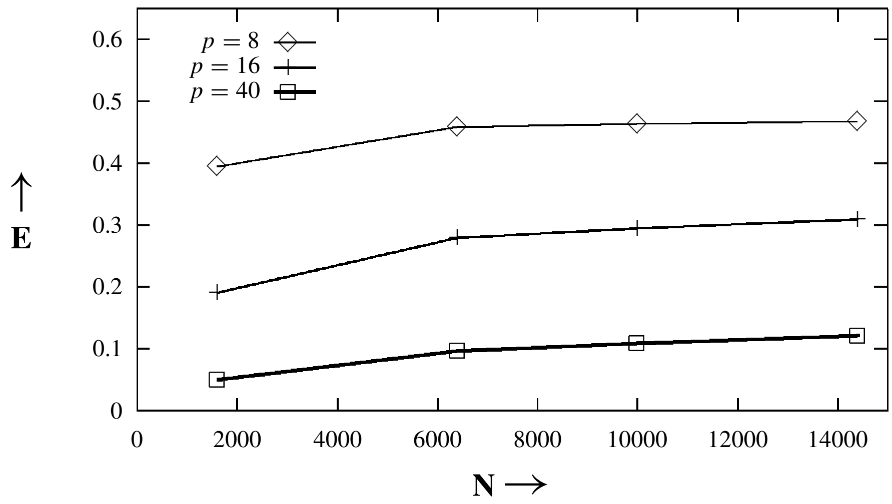


Figure 3.19: Efficiency plots for unstructured sparse matrices with fixed number of non-zero elements per row.

preconditioner needs a smaller problem size than the diagonal preconditioner to achieve the same efficiency. For the same problem size, the IC preconditioner can use more processors at the same efficiency, thereby delivering higher speedups than the diagonal preconditioner.

Figure 3.19 shows plots of efficiency versus problem size for three different values of p for a totally unstructured sparse matrix with a fixed number of non-zero elements in each row. This kind of a matrix leads to an unscalable parallel formulation of the CG algorithm. This fact is clearly reflected in Figure 3.19. Not only does the efficiency drop rapidly as the number of processors are increased, but also an increase in problem size does not suffice to counter this drop in efficiency. For instance, using 40 processors, it does not seem possible to attain the efficiency of 16 processors, no matter how big a problem is being solved.

Figures 3.20 and 3.21 show how the parallel CG algorithm for unstructured sparse matrices can be made scalable by either confining the non-zero elements within a band of width $< O(N)$, or by increasing the number of non-zero elements in each row as a function of N. Figure 3.20 shows the experimental isoefficiency curves for a banded unstructured sparse matrix with a bandwidth of

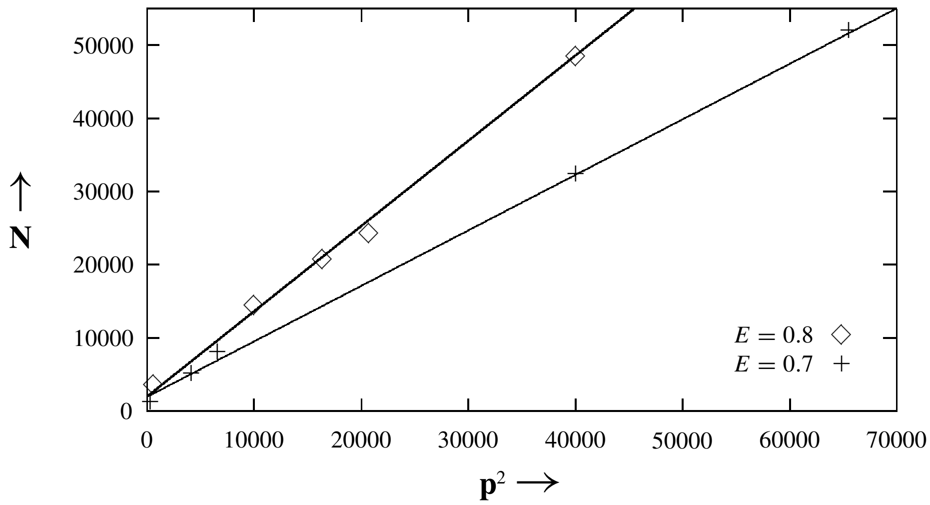


Figure 3.20: Isoefficiency curves for banded unstructured sparse matrices with fixed number of non-zero elements per row.

$2\sqrt{N}$ and 6 non-zero elements per row. The curves were drawn by picking up (N, p) pairs that yielded nearly the same efficiency on the CM-5 implementation, and then plotting N with respect to p^2. As shown in Table 3.4, the isoefficiency function is $\Theta(p^2)$ and a linear relation between N and p^2 in Figure 3.20 confirms this. Figure 3.21 shows the isoefficiency curve for $E = 0.25$ for a totally unstructured $N \times N$ sparse matrix with $0.0015N$ non-zero elements in each row. As shown in Table 3.4, the isoefficiency function is linear in this case, although the constant associated with it is quite large because of the terms $2c_2\alpha$ in the denominator, α being 0.0015.

3.3.5 Summary of Results

We have studied the performance and scalability of an iteration of the Preconditioned Conjugate Gradient algorithm on a variety of parallel architectures.

It is shown that block-tridiagonal matrices resulting from the discretization of a 2-dimensional self-adjoint elliptic partial differential equation via finite differences lead to highly scalable parallel formulations of the CG method on a parallel machine like the CM-5 with an appropriate mapping

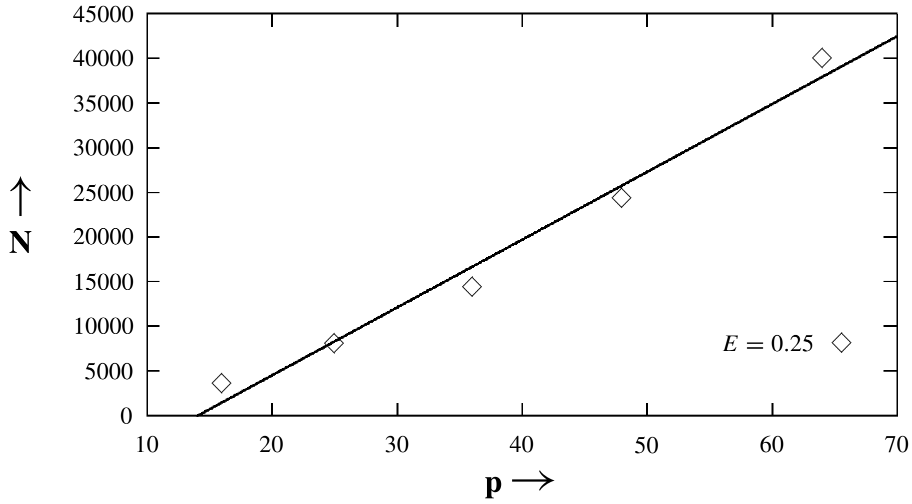


Figure 3.21: An isoefficiency curve for unstructured sparse matrices with the number of non-zero elements per row increasing with the matrix size.

of data onto the processors. On this architecture, speedups proportional to the number of processors can be achieved by increasing the problem size almost linearly with the number of processors. The reason is that on the CM-5, the control network practically eliminates the communication overhead in computing inner-products of vectors, whereas on more conventional parallel machines with significant message startup times, it turns out to be the costliest operation in terms of overheads and affects the overall scalability. The isoefficiency function for a PCG iteration with a block-tridiagonal matrix is $\Theta(p)$ on the CM-5, $\Theta(p \log p)$ on a hypercube, and $\Theta(p\sqrt{p})$ on a mesh. In terms of scalability, if the number of processors is increased from 100 to 1000, the problem size will have to increased 32 times on a mesh, 15 times on a hypercube, and only 10 times on the CM-5 to obtain ten times higher speedups. Also, the effect of message startup time t_s on the speedup is much more significant on a typical hypercube or a mesh than on the CM-5. However, for some iterative algorithms like the Jacobi method, linear scalability can be obtained even on these architectures by performing the convergence check at a suitably reduced frequency.

We have shown that in case of the block-tridiagonal matrices, the truncated Incomplete Cholesky preconditioner can achieve higher efficiencies than a simple diagonal preconditioner if the data mapping scheme of Figure 3.11 is used. The use of the IC preconditioner usually significantly cuts down the number of iterations required for convergence. However, it involves solving a linear system of equations of the form $\mathbf{M}\ \mathbf{z} = \mathbf{r}$ in each iteration. This is a computationally costly operation and may offset the advantage gained by fewer iterations. Even if this is the case for the serial algorithm, in a parallel implementation the IC preconditioner may outperform the diagonal preconditioner as the number of processors are increased. This is because a parallel implementation with IC preconditioner executes at a higher efficiency than one with a diagonal preconditioner.

If the matrix of coefficients of the linear system of equations to be solved is a random unstructured sparse matrix (such matrices often occur in linear programming problems) with a constant number of non-zero elements in each row, a parallel formulation of the PCG method will be unscalable on any practical massage passing architecture unless some ordering is applied to the sparse matrix. The efficiency of parallel PCG with an unordered sparse matrix will always drop as the number of processors is increased and no increase in the problem size is sufficient to counter this drop in efficiency. The system can be rendered scalable if either the non-zero elements of the $N \times N$ matrix of coefficients can be confined in a band whose width is less than $\Theta(N)$, or the number of non-zero elements per row increases with N, where N is the number of simultaneous equations to be solved. The scalability increases as the number of non-zero elements per row in the matrix of coefficients is increased and/or the width of the band containing these elements is reduced. Both the number of non-zero elements per row and the width of the band containing these elements depend on the characteristics of the system of equations to be solved. In particular, the non-zero elements can be organized within a band by using some re-ordering techniques [45, 48]. Such restructuring techniques can improve the efficiency (for a given number of processors, problem size, machine architecture, etc.) as well as the asymptotic scalability of the PCG algorithm.

Chapter 4

SCALABLE PARALLEL ALGORITHMS FOR SOLVING SPARSE SYSTEMS OF LINEAR EQUATIONS

Solving large sparse systems of linear equations is at the core of many problems in engineering and scientific computing. Such systems are typically solved by two different types of methods—**iterative methods** and **direct methods**. The nature of the problem at hand determines which method is more suitable. A direct method for solving a sparse linear system of the form $Ax = b$ involves explicit factorization of the sparse coefficient matrix A into the product of lower and upper triangular matrices L and U. This is a highly time and memory consuming step; nevertheless, direct methods are important because of their generality and robustness. For linear systems arising in certain applications, such as linear programming and structural engineering applications, they are the only feasible solution methods. In many other applications too, direct methods are often preferred because the effort involved in determining and computing a good preconditioner for an iterative solution may outweigh the cost of direct factorization. Furthermore, direct methods provide and effective means for solving multiple systems with the same coefficient matrix and different right-hand side vectors because the factorizations needs to be performed only once.

Although direct methods are used extensively in practice, their use for solving large sparse systems has been mostly confined to big vector supercomputers due to their high time and memory requirements. Parallel processing offers the potential to tackle both these problems; however, despite intensive research, only limited success had been achieved until recently in developing scalable parallel formulations of sparse matrix factorization [63, 123, 44, 47, 46, 95, 96, 10, 114, 115, 127, 41, 69, 63, 64, 117, 8, 7, 104, 140, 49, 123, 103]. We have developed a highly parallel sparse Cholesky factorization algorithm that substantially improves the state of the art in parallel direct solution of sparse linear systems—both in terms of scalability and overall performance. We show that our algorithm is just as scalable as dense matrix factoriztion, which is known to be highly scalable. In addition to providing a fast direct solution, this algorithm significantly increases the range of problem sizes that can be solved. It is the only known sparse factorization algorithm that can deliver speedups in proportion to an increasing number of processors while requiring almost constant memory per processor. By using this algorithm, large problems that could not be solved on serial or small parallel computers due to the high memory and time requirement of direct factorization can now be effectively parallelized to utilize large scale parallel computers.

The performance and scalability analysis of our algorithm is supported by experimental results on up to 1024 processors of the nCUBE2 parallel computer. We have been able to achieve speedups of up to 364 on 1024 processors and 230 on 512 processors over a highly efficient sequential implementation for moderately sized problems from the Harwell-Boeing collection [30]. In [77], we have applied this algorithm to obtain a highly scalable parallel formulation of interior point algorithms and have observed significant speedups in solving linear programming problems. A variation of this algorithm [53] implemented on a 1024-processor Cray T3D delivers up to 20 GFLOPS on medium-size structural engineering and linear programming problems. To the best of our knowledge, this is the first parallel implementation of sparse Cholesky factorization that has delivered speedups of this magnitude and has been able to benefit from several hundred processors.

In this chapter we describe our highly scalable parallel algorithm for sparse matrix factorization. We analyze its performance and scalability on a few different architectures two important classes of sparse matrices. We give detailed experimental results of the implementations of our algorithm on the nCUBE2 and Cray T3D parallel computers. Although our current implementations work for Cholesky factorization of symmetric positive definite matrices, the algorithm can be adapted for solving sparse linear least squares problems and for Gaussian elimination of diagonally dominant matrices that are almost symmetric in structure. In addition to numerical factorization we also present efficient parallel algorithms for two of the three other phases of solving a sparse system of linear equations. These two phases are symbolic factorization and forward and backward substitution to solve the triangular systems resulting from sparse matrix factorization. We show that symbolic factorization and triangular solutions, though less computation-intensive than numerical factorization, can be parallelized sufficiently so that the overall solution process is as scalable as the numerical factorization phase.

Although we focus on Cholesky factorization of symmetric positive definite matrices in this chapter, the methodology developed here can be adapted for performing Gaussian elimination on diagonally dominant matrices that are almost symmetric in structure [32] and for solving sparse linear least squares problems [99].

4.1 Earlier Research in Sparse Matrix Factorization and Our Contribution

Since sparse matrix factorization is the most time consuming phase in the direct solution of a sparse system of linear equations, there has been considerable interest in developing its parallel formulations. It is well known that dense matrix factorization can be implemented efficiently on distributed-memory parallel computers [40, 109, 42, 81]. However, despite inherent parallelism in sparse direct methods, not much success has been achieved to date in developing their scalable parallel formulations [63, 123] and for several years, it has been a challenge to implement efficient sparse linear system solvers using direct methods on even moderately parallel computers. Performance delivered by most existing parallel sparse matrix factorizations had been quite poor. In [123], Schreiber concludes that it is not yet clear whether sparse direct solvers can be made competitive at all for highly ($p \geq 256$) and massively ($p \geq 4096$) parallel computers.

It is difficult to derive analytical expressions for the number of arithmetic operations in factorization and for the size (in terms of number of nonzero entries) of the factor for general sparse matrices. This is because the computation and fill-in during the factorization of a sparse matrix is a function of the the number and position of nonzeros in the original matrix. With the aid of Figure 4.1[1], we summarize the contribution of our work in the context of Cholesky factorization of the important class of sparse matrices that are adjacency matrices of graphs whose n-node subgraphs have $\Theta(\sqrt{n})$-node separators (this class includes sparse matrices arising out of all two-dimensional finite difference and finite element problems). The results for the three-dimensional case are very similar. A simple fan-out algorithm [44] with column-wise partitioning of an $N \times N$ matrix of this type on p processors results in an $\Theta(Np \log N)$ total communication volume [47] (box A). The communication volume of the column-based schemes represented in box A has been improved using smarter ways of mapping the matrix columns onto processors, such as, the subtree-

[1] In [110], Pan and Reif describe a parallel sparse matrix factorization algorithm for a PRAM type architecture. This algorithm is not cost-optimal (i.e., the processor-time product exceeds the serial complexity of sparse matrix factorization) and is not included in the classification given in Figure 4.1.

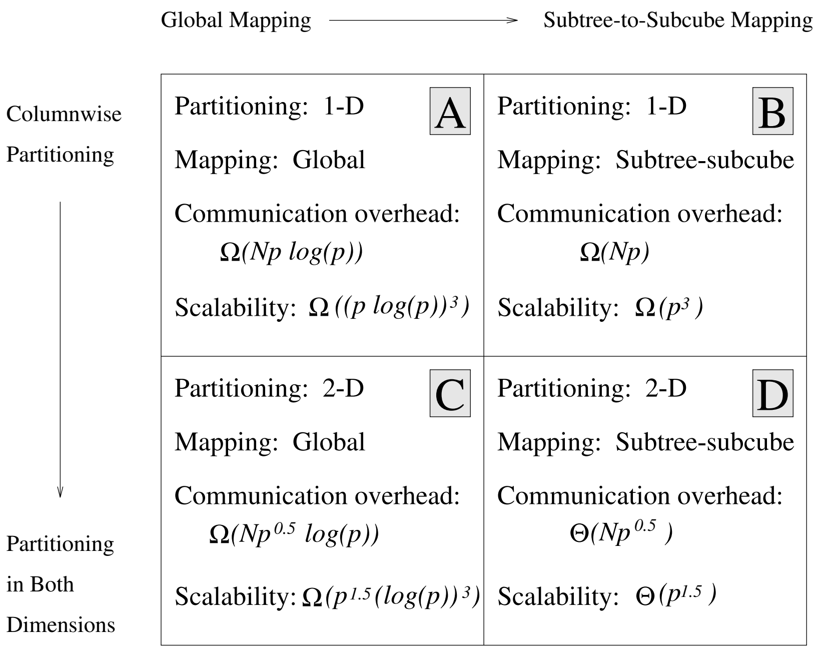


Figure 4.1: An overview of the performance and scalability of parallel algorithms for factorization of sparse matrices resulting from two-dimensional N-node grid graphs. Box D represents our algorithm, which is a significant improvement over other known classes of algorithms for this problem.

to-subcube mapping [46] (box B). A number of column-based parallel factorization algorithms [95, 96, 10, 114, 115, 127, 44, 47, 41, 69, 63, 64, 117, 123, 103] have a lower bound of $\Omega(Np)$ on the total communication volume [47]. Since the overall computation is only $\Theta(N^{1.5})$ [45], the ratio of communication to computation of column-based schemes is quite high. As a result, these column-cased schemes scale very poorly as the number of processors is increased [123, 120]. In [8], Ashcraft proposes a fan-both family of parallel Cholesky factorization algorithms that have a total communication volume of $\Theta(N\sqrt{p}\log N)$. Although the communication volume is less than the other column-based partitioning schemes, the isoefficiency function of Ashcraft's algorithm is still $\Theta(p^3)$ due to concurrency constraints because the algorithm cannot effectively utilize more than $O(\sqrt{N})$ processors for matrices arising from two-dimensional constant node-degree graphs. A few schemes with two-dimensional partitioning of the matrix have been proposed [120, 119, 7, 104, 140, 49], and the total communication volume in the best of these schemes [120, 119] is $\Theta(N\sqrt{p}\log p)$ (box

C).

Most researchers so far have analyzed parallel sparse matrix factorization in terms of the total communication volume. It is noteworthy that, on any parallel architecture, the total communication volume is only a lower bound on the overall communication overhead. It is the total communication overhead that actually determines the overall efficiency and speedup, and is defined as the difference between the parallel processor-time product and the serial run time [55, 81]. The communication overhead can be asymptotically higher than the communication volume. For example, a one-to-all broadcast algorithm based on a binary tree communication pattern has a total communication volume of $m(p-1)$ for broadcasting m words of data among p processors. However, the broadcast takes $\log p$ steps of $\Theta(m)$ time each; hence, the total communication overhead is $\Theta(mp \log p)$ (on a hypercube). In the context of matrix factorization, the experimental study by Ashcraft [10] serves to demonstrate the importance of studying the total communication overhead rather than volume. In [10], the fan-in algorithm, which has a lower communication volume than the distributed multifrontal algorithm, has a higher overhead (and hence, a lower efficiency) than the multifrontal algorithm for the same distribution of the matrix among the processors.

The isoefficiency function for dense matrix factorization is $\Theta(p^{1.5})$ [81]. It is easy to prove that $\Theta(p^{1.5})$ is also the lower bound on the isoefficiency function for factoring the above mentioned class of sparse matrices. None of the classes of algorithms represented by boxes A, B, and C in Figure 4.1 achieve this lower bound. Note that the simple parallel algorithm with $\Theta(Np \log p)$ communication volume (box A) has been improved along two directions—one by improving the mapping of matrix columns onto processors (box B) and the other by splitting the matrix along both rows and columns (box C). Observing the expressions for the communication overhead and isoefficiency functions in boxes A, B, and C of Figure 4.1, it is intuitive that if somehow the benefits of subtree-to-subcube mapping and a two-dimensional partitioning of the sparse matrix could be combined, one could obtain an optimally scalable parallel algorithm with an isoefficiency function of $\Theta(p^{1.5})$.

Based on the above observations, we devised an algorithm that is indeed optimally scalable for the class of matrices being studied. The main features of our algorithm are that it is based on the multifrontal technique, it uses subtree-to-subcube mapping scheme, and it partitions the matrix

Phase	2-D complexity	3-D complexity
Reordering:	O(N)	O(N)
Symbolic Factorization:	O(N log N)	$O(N^{4/3})$
Numerical Factorization:	$O(N^{3/2})$	$O(N^2)$
Triangular Solutions:	O(N log N)	$O(N^{4/3})$

Figure 4.2: The serial computational complexity of the various phases of solving a sparse system of linear equations arising from two- and three-dimensional constant node-degree graphs.

in two dimensions along both rows and columns. In Section 4.4, we describe how the algorithm integrates all these features to minimize communication overhead. A key to this integration is a block-cyclic two-dimensional partitioning based on the bits of the binary representation of row and column indices of the sparse coefficient matrix. The total communication overhead of our algorithm is only $\Theta(N\sqrt{p})$ for factoring an $N \times N$ matrix on p processors if it corresponds to a graph that satisfies the separator criterion. Our algorithm reduces the communication overhead by a factor of at least $\Theta(\log p)$ over the best algorithm [120, 119] implemented to date. It is also significantly simpler in concept as well as in implementation, which helps in keeping the constant factors associated with the overhead term low. We show in Section 4.6, the reduction in communication overhead by a factor of $\Theta(\log p)$ results in an improvement in the scalability of the algorithm by a factor of $\Theta((\log p)^3)$; i.e., the rate at which the problem size must increase with the number of processors to maintain a constant efficiency is lower by a factor of $\Theta((\log p)^3)$. This can make the difference between the feasibility and non-feasibility of parallel sparse factorization on highly parallel ($p \geq 256$) computers. In addition, our algorithm is the only known sparse factorization algorithm that can deliver speedups in proportion to an increasing number of processors while requiring almost constant memory per processor.

4.2 Chapter Outline

The process of obtaining a direct solution of a sparse system of linear equations of the form $Ax = b$ consists of the following four phases: **Ordering**, which determines permutation of the coefficient matrix A such that the factorization incurs low fill-in and is numerically stable; **Symbolic Factorization**, which determines the structure of the triangular matrices that would result from factorizing the coefficient matrix resulting from the ordering step; **Numerical Factorization**, which is the actual factorization step that performs arithmetic operations on the coefficient matrix A to produce a lower triangular matrix L and an upper triangular matrix U; and **Solution of Triangular Systems**, which produces the solution vector x by performing forward and backward eliminations on the triangular matrices resulting from numerical factorization. As shown in Figure 4.2, numerical factorization is the most time-consuming phase. Karypis and Kumar have proposed an efficient parallel algorithm for determining fill-reducing orderings for parallel factorization of sparse matrices [76]. In this chapter, we present efficient and scalable parallel algorithms for symbolic factorization, numerical factorization, and for solving the upper and lower triangular systems.

In Section 4.3, we describe the serial multifrontal algorithm for sparse Cholesky factorization. This algorithm forms the basis of our optimally scalable parallel algorithm described in Section 4.4. Sections 4.5 and 4.6 present the analysis of communication overhead and scalability of the parallel sparse Cholesky factorization algorithm for two widely used classes of sparse matrices on mesh and hypercube architectures. Section 4.7 contains the experimental results of sparse Cholesky factorization implementations based on the algorithm of Section 4.4 on up to 1024 processors of nCUBE2 and Cray T3D parallel computers.

The experimental results in Section 4.7 show that our algorithm can easily speedup Cholesky factorization by a factor of at least a few hundred on up to 1024 processors. With such speedups in numerical factorization, it is imperative that the remaining phases of the solution process be parallelized effectively in order to scale the performance of the overall solver. Furthermore, without an overall parallel solver, the size of the sparse systems that can be solved may be severely restricted by the amount of memory available on a uniprocessor system. In Section 4.8, we address the problem of performing the final phase of forward and backward substitution in parallel on a distributed memory multiprocessor. Our analysis and experiments show that, although not as scalable as the best parallel sparse Cholesky factorization algorithms, parallel sparse triangular solvers can yield reasonable speedups in runtime on hundreds of processors. We also show that for a wide class of problems, the sparse triangular solvers described in this paper are optimal and are asymptotically as scalable as a dense triangular solver. In Section 4.9, we describe and analyze a parallel formulation of symbolic factorization. In this section, we show that the communication overhead of parallel symbolic factorization is asymptotically less than that of the factorization step; hence, this step does not impose any constraints on the performance and scalability of a complete parallel sparse linear system solver.

The algorithm in [76], while performing the ordering in parallel, also distributes the data among the processors in way that the remaining steps can be carried out with minimum data-movement. At the end of the parallel ordering step, the parallel symbolic factorization algorithm described in Section 4.9 can proceed without any redistribution. Since numerical factorization is the step with the highest computational complexity in the entire process, it is critical to have an efficient parallel algorithm for this step. Our algorithms for symbolic factorization and triangular solutions are tailored to work in conjunction with the numerical factorization algorithm described in Section 4.4. Therefore, we describe this algorithm in detail first, before we discuss triangular solution and symbolic factorization (in that order).

4.3 The Serial Multifrontal Algorithm for Sparse Matrix Factorization

The multifrontal algorithm for sparse matrix factorization was proposed independently, and in somewhat different forms, by Speelpening [126] and Duff and Reid [31], and later elucidated in a tutorial by Liu [92]. In this section, we briefly describe a condensed version of multifrontal sparse Cholesky factorization.

Given a sparse matrix and the associated elimination tree, the multifrontal algorithm can be recursively formulated as shown in Figure 4.3. Consider the Cholesky factorization of an $N \times N$ sparse symmetric positive definite matrix A into LL^T, where L is a lower triangular matrix. The algorithm performs a postorder traversal of the elimination tree associated with A. There is a frontal matrix F^k and an update matrix U^k associated with any node k. The row and column indices of F^k correspond to the indices of row and column k of L in increasing order.

In the beginning, F^k is initialized to an $(s + 1) \times (s + 1)$ matrix, where $s + 1$ is the number of nonzeros in the lower triangular part of column k of A. The first row and column of this initial F^k is simply the upper triangular part of row k and the lower triangular part of column k of A. The

/*

A is the sparse $N \times N$ symmetric positive definite matrix to be factored. L is the lower triangular matrix such that $A = LL^T$ after factorization. $A = (a_{i,j})$ and $L = (l_{i,j})$, where $0 \leq i, j < N$. Initially, $l_{i,j} = 0$ for all i, j.

*/

begin function Factor(k)

$$F^k := \begin{pmatrix} a_{k,k} & a_{k,q_1} & a_{k,q_2} & \cdots & a_{k,q_s} \\ a_{q_1,k} & 0 & 0 & \cdots & 0 \\ a_{q_2,k} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{q_s,k} & 0 & 0 & \cdots & 0 \end{pmatrix};$$

    for all i such that Parent(i) = k in the elimination tree of A, **do**

    begin

        Factor(i);

        F^k := Extend_add(F^k, U^i);

    end

/*

At this stage, F^k is a $(t+1) \times (t+1)$ matrix, where t is the number of nonzeros in the sub-diagonal part of column k of L. U^k is a $t \times t$ matrix. Assume that an index i of F^k or U^k corresponds to the index q_i of A and L.

*/

    for $i := 0$ **to** t **do**

        $l_{q_i,k} := F^k(i,0)/\sqrt{F^k(0,0)}$;

    for $j := 1$ **to** t **do**

        for $i := j$ **to** t **do**

            $U^k(i,j) := F^k(i,j) - l_{q_i,k} \times l_{q_j,k}$;

end function Factor.

Figure 4.3: An elimination-tree guided recursive formulation of the multifrontal algorithm for Cholesky factorization of a sparse SPD matrix A into LL^T. If r is the root of the postordered elimination tree of A, then a call to Factor(r) factors the matrix A.

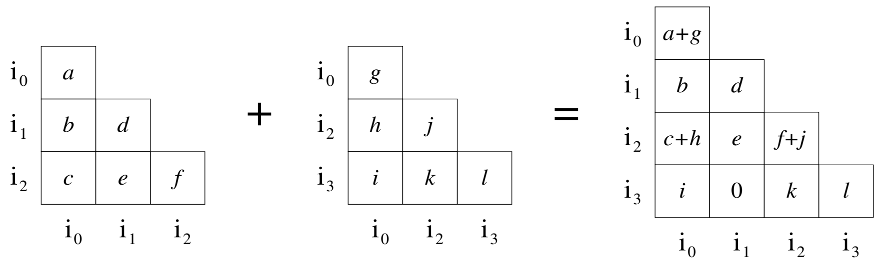


Figure 4.4: The extend-add operation on two 3×3 triangular matrices. It is assumed that $i_0 < i_1 < i_2 < i_3$.

remainder of F^k is initialized to all zeros. Line 2 of Figure 4.3 illustrates the initial F^k.

After the algorithm has traversed all the subtrees rooted at a node k, it ends up with a $(t+1) \times (t+1)$ frontal matrix F^k, where t is the number of nonzeros in the strictly lower triangular part of column k in L. The row and column indices of the final assembled F^k correspond to $t+1$ (possibly) noncontiguous indices of row and column k of L in increasing order. If k is a leaf in the elimination tree of A, then the final F^k is the same as the initial F^k. Otherwise, the final F^k for eliminating node k is obtained by merging the initial F^k with the update matrices obtained from all the subtrees rooted at k via an extend-add operation. The extend-add is an associative and commutative operator on two update matrices such the index set of the result is the union of the index sets of the original update matrices. Each entry in the original update matrices is mapped onto some location in the accumulated matrix. If entries from both matrices overlap on a location, they are added. Empty entries are assigned a value of zero. Figure 4.4 illustrates the extend-add operation.

After F^k has been assembled through the steps of lines 3–7 of Figure 4.3, a single step of the standard dense Cholesky factorization is performed with node k as the pivot (lines 8–12). At the end of the elimination step, the column with index k is removed from F^k and forms the column k of L. The remaining $t \times t$ matrix is called the update matrix U^k and is passed on to the parent of k in the elimination tree.

The multifrontal algorithm is further illustrated in a step-by-step fashion in Figure 4.6 for factoring the matrix of Figure 4.5(a).

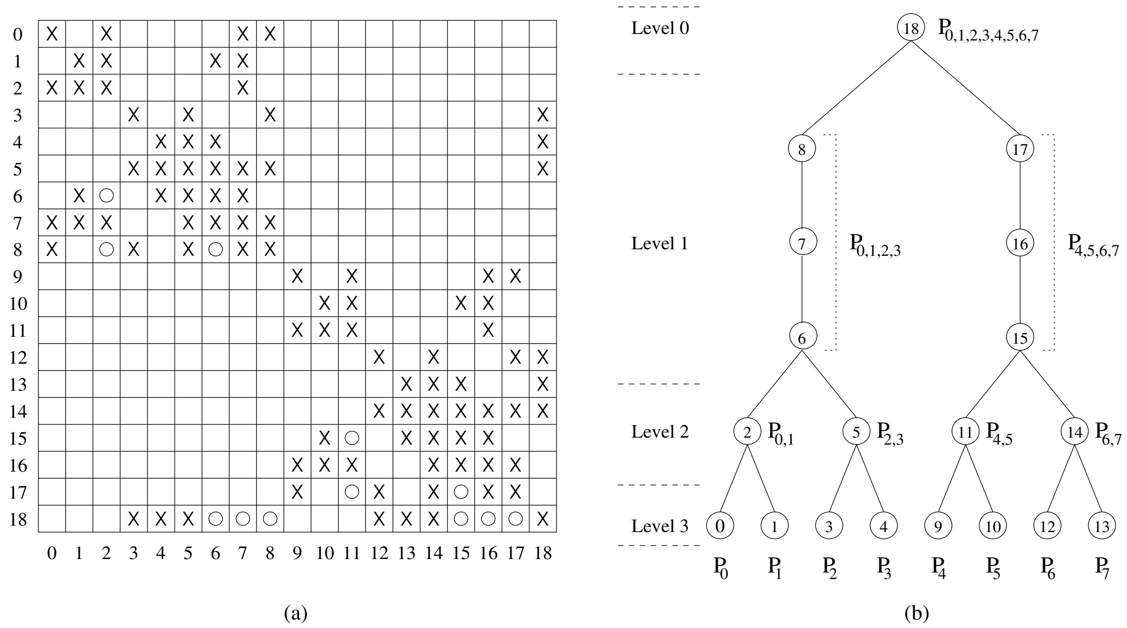


Figure 4.5: A symmetric sparse matrix and the associated elimination tree with subtree-to-subcube mapping onto 8 processors. The nonzeros in the original matrix are denoted by the symbol "$\times$" and fill-ins are denoted by the symbol "$\circ$".

4.4 A Parallel Multifrontal Algorithm

In this section we describe the parallel multifrontal algorithm. We assume a hypercube interconnection network; however, as we will show in a later section, the algorithm also can be adapted for a mesh topology without any increase in the asymptotic communication overhead. On other architectures as well, such as those of the CM-5, Cray T3D, and IBM SP-2, the asymptotic expression for the communication overhead remains the same. In this chapter, we use the term *relaxed supernode* for a group of consecutive nodes in the elimination tree with one child. Henceforth, any reference to the height, depth, or levels of the tree will be with respect to the relaxed supernodal tree. For the sake of simplicity, we assume that the relaxed supernodal elimination tree is a binary tree up to the top $\log p$ relaxed supernodal levels. Any elimination tree can be converted to a binary relaxed supernodal tree suitable for parallel multifrontal elimination by a simple preprocessing step.

In order to factorize the sparse matrix in parallel, portions of the elimination tree are assigned to processors using the standard subtree-to-subcube assignment strategy. This assignment is illustrated

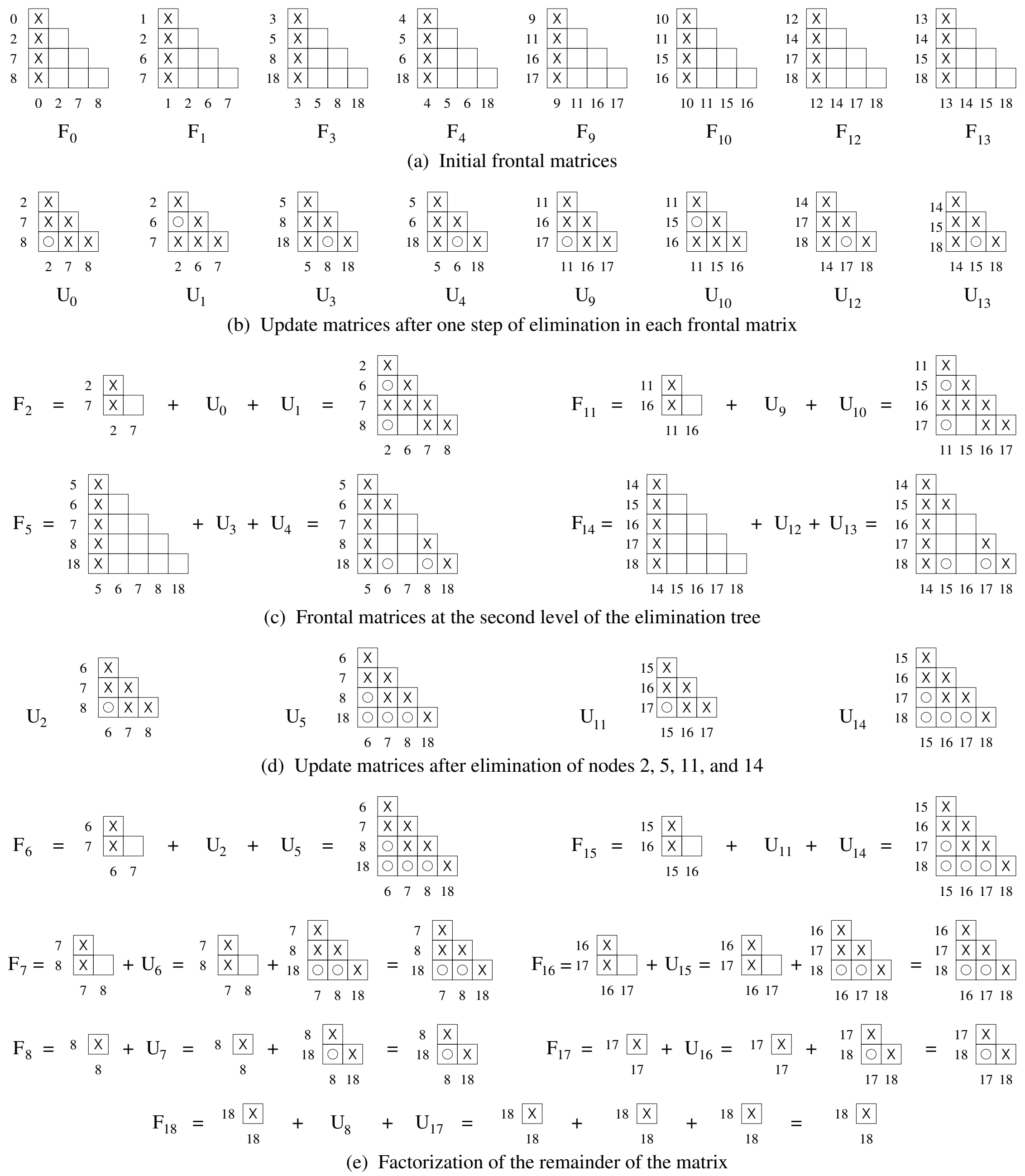


Figure 4.6: Steps in serial multifrontal Cholesky factorization of the matrix shown in Figure 4.5(a). The symbol "+" denotes an extend-add operation. The nonzeros in the original matrix are denoted by the symbol "×" and fill-ins are denoted by the symbol "○".

in Figure 4.5(b) for eight processors. With subtree-to-subcube assignment, all p processors in the system cooperate to factorize the frontal matrix associated with the topmost relaxed supernode of the elimination tree. The two subtrees of the root are assigned to subcubes of $p/2$ processors each. Each subtree is further partitioned recursively using the same strategy. Thus, the p subtrees at a depth of $\log p$ relaxed supernodal levels are each assigned to individual processors. Each processor can work on this part of the tree completely independently without any communication overhead. A call to the function *Factor* given in Figure 4.3 with the root of a subtree as the argument generates the update matrix associated with that subtree. This update matrix contains all the information that needs to be communicated from the subtree in question to other columns of the matrix.

After the independent factorization phase, pairs of processors (P_{2j} and P_{2j+1} for $0 \leq j < p/2$) perform a parallel extend-add on their update matrices, say Q and R, respectively. At the end of this parallel extend-add operation, P_{2j} and P_{2j+1} roughly equally share $Q + R$. Here, and in the remainder of this chapter, the sign "+" in the context of matrices denotes an extend-add operation. More precisely, all even columns of $Q + R$ go to P_{2j} and all odd columns of $Q + R$ go to P_{2j+1}. At the next level, subcubes of two processors each perform a parallel extend-add. Each subcube initially has one update matrix. The matrix resulting from the extend-add on these two update matrices is now merged and split among four processors. To effect this split, all even rows are moved to the subcube with the lower processor labels, and all odd rows are moved to the subcube with the higher processor labels. During this process, each processor needs to communicate only once with its counterpart in the other subcube. After this (second) parallel extend-add each of the processors has a block of the update matrix roughly one-fourth the size of the whole update matrix. Note that, both the rows and the columns of the update matrix are distributed among the processors in a cyclic fashion. Similarly, in subsequent parallel extend-add operations, the update matrices are alternatingly split along the columns and rows.

Assume that the levels of the binary relaxed supernodal elimination tree are labeled starting with 0 at the top as shown in Figure 4.5(b). In general, at level l of the relaxed supernodal elimination tree, $2^{\log p - l}$ processors work on a single frontal or update matrix. These processors form a logical $2^{\lfloor (\log p - l)/2 \rfloor} \times 2^{\lceil (\log p - l)/2 \rceil}$ mesh. All update and frontal matrices at this level are distributed on this mesh of processors. The cyclic distribution of rows and columns of these matrices among the

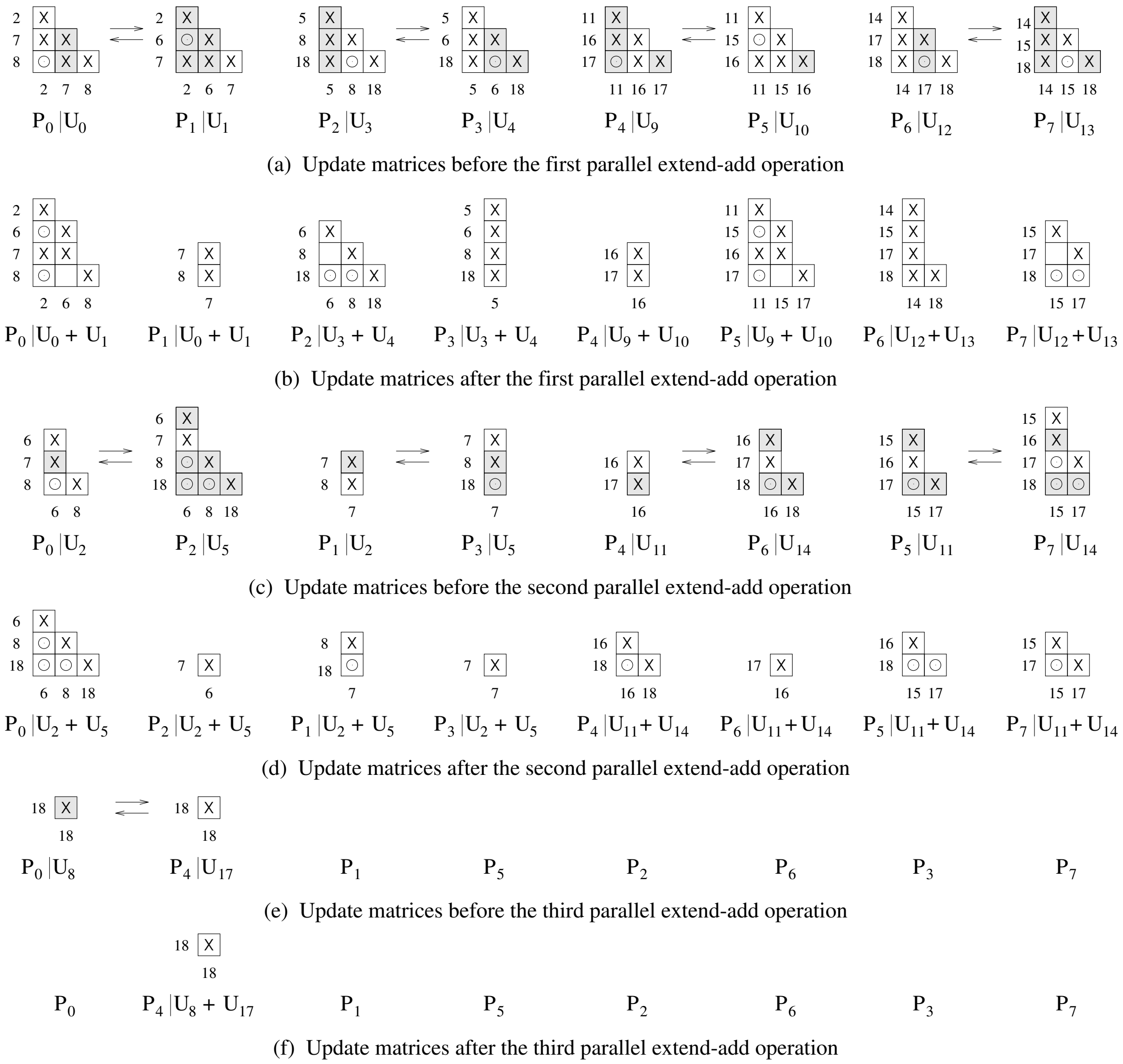


Figure 4.7: Extend-add operations on the update matrices during parallel multifrontal factorization of the matrix shown in Figure 4.5(a) on eight processors. $P_i|M$ denotes the part of the matrix M that resides on processor number i. M may be an update matrix or the result of performing an extend-add on two update matrices. The shaded portions of a matrix are sent out by a processor to its communication partner in that step.

processors helps maintain load-balance. The distribution also ensures that a parallel extend-add operation can be performed with each processor exchanging roughly half of its data only with its counterpart processor in the other subcube. This distribution is fairly straightforward to maintain. For example, during the first two parallel extend-add operations, columns and rows of the update matrices are distributed depending on whether their least significant bit (LSB) is 0 or 1. Indices with LSB = 0 go to the lower subcube and those with LSB = 1 go to the higher subcube. Similarly, in the next two parallel extend-add operations, columns and rows of the update matrices are exchanged among the processors depending on the second LSB of their indices.

Figure 4.7 illustrates all the parallel extend-add operations that take place during parallel multifrontal factorization of the matrix shown in Figure 4.5. The portion of an update matrix that is sent out by its original owner processor is shown in grey. Hence, if processors P_i and P_j with respective update matrices Q and R perform a parallel extend-add, then the final result at P_i will be the add-extension of the white portion of Q and the grey portion of R. Similarly, the final result at P_j will be the add-extension of the grey portion of Q and the white portion of R. Figure 4.8 further illustrates this processes by showing four consecutive extend-add operations on hypothetical update matrices to distribute the result among 16 processors.

Between two successive parallel extend-add operations, several steps of dense Cholesky elimination may be performed. The number of such successive elimination steps is equal to the number of nodes in the relaxed supernode being processed. The communication that takes place in this phase is the standard communication in pipelined grid-based dense Cholesky factorization [109, 81]. If the average size of the frontal matrices is $t \times t$ during the processing of a relaxed supernode with m nodes on a q-processor subcube, then $\Theta(m)$ messages of size $\Theta(t/\sqrt{q})$ are passed through the grid in a pipelined fashion. Figure 4.9 shows the communication for one step of dense Cholesky factorization of a hypothetical frontal matrix for $q = 16$. It is shown in [82] that although this communication does not take place between the nearest neighbors on a subcube, the paths of all communications on any subcube are conflict free with e-cube routing [107, 81] and cut-through or worm-hole flow control. This is a direct consequence of the fact that a circular shift is conflict free on a hypercube with e-cube routing. Thus, a communication pipeline can be maintained among the processors of a subcube during the dense Cholesky factorization of frontal matrices.

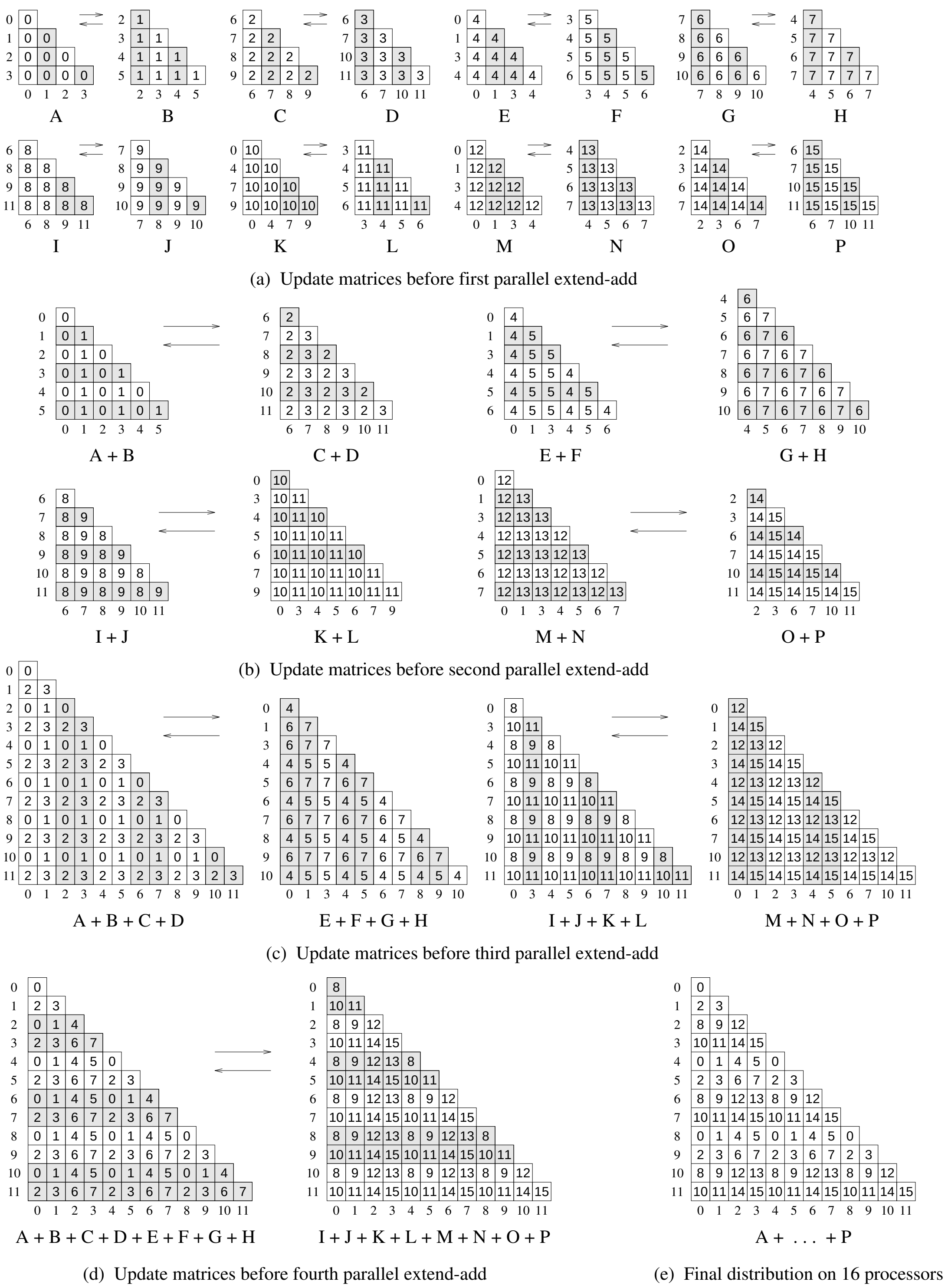


Figure 4.8: Four successive parallel extend-add operations (denoted by "+") on hypothetical update matrices for multifrontal factorization on 16 processors.

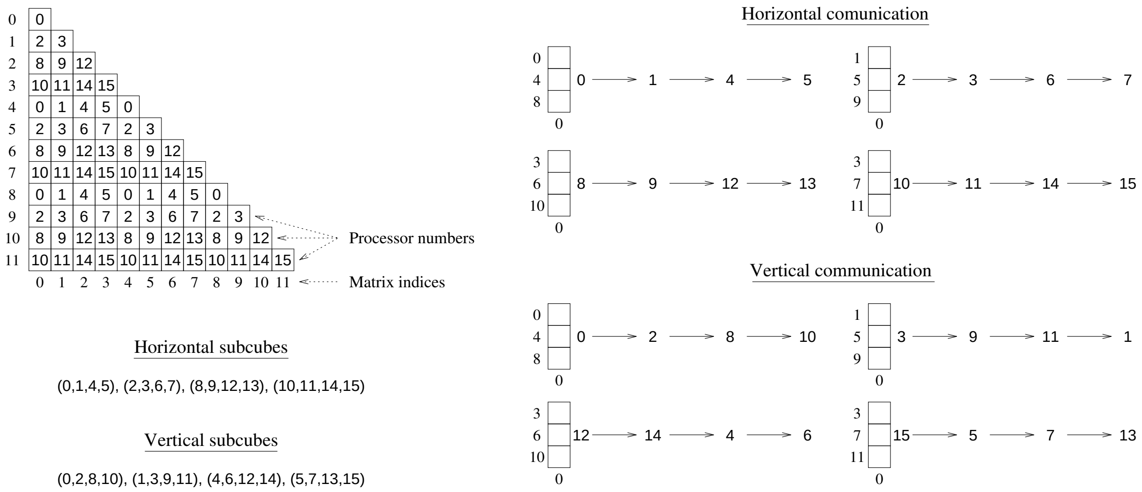
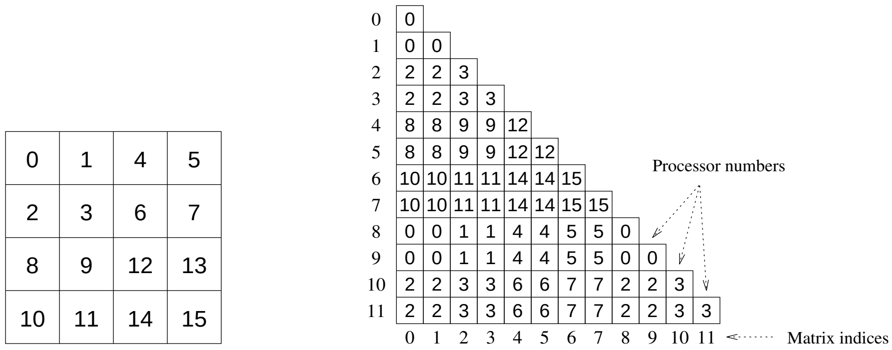


Figure 4.9: The two communication operations involved in a single elimination step (index of pivot = 0 here) of Cholesky factorization on a 12×12 frontal matrix distributed over 16 processors.

4.4.1 Block-Cyclic Mapping of Matrices onto Processors

In the parallel multifrontal algorithm described in this section, the rows and columns of frontal and update matrices are distributed among the processors of a subcube in a cyclic manner. For example, the distribution of a matrix with indices from 0 to 11 on a 16-processor subcube is shown in Figure 4.8(e). The 16 processors form a logical mesh. The arrangement of the processors in the logical mesh is shown in Figure 4.10(a). In the distribution of Figure 4.8(e), consecutive rows and columns of the matrix are mapped onto neighboring processors of the logical mesh. If there are more rows and columns in the matrix than the number of processors in a row or column of the processor mesh, then the rows and columns of the matrix are wrapped around on the mesh.

Although the mapping shown in Figure 4.8(e) results in a very good load balance among the processors, it has a disadvantage. Notice that while performing the steps of Cholesky factorization on the matrix shown in Figure 4.8(e), the computation corresponding to consecutive pivots starts on different processors. For example, pivot 0 on processor 0, pivot 1 on processor 3, pivot 2 on processor 12, pivot 3 on processor 15, and so on. If the message startup time is high, this may lead to significant delays between the stages of the pipeline. Furthermore, on cache-based processors, the use of BLAS-3 for eliminating multiple columns simultaneously yields much higher performance than the use of BLAS-2 for eliminating one column at a time. Figure 4.10(b) shows a variation of



(a) A logical mesh of 16 processors (b) Block-cyclic mapping with 2 X 2 blocks

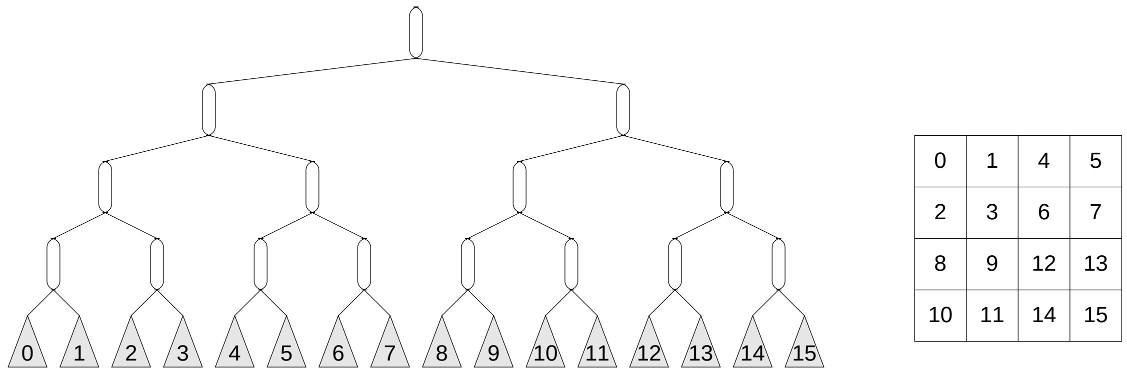
Figure 4.10: Block-cyclic mapping of a 12×12 matrix on a logical processor mesh of 16 processors.

the cyclic mapping, called block-cyclic mapping [81], that can alleviate these problems at the cost of some added load imbalance.

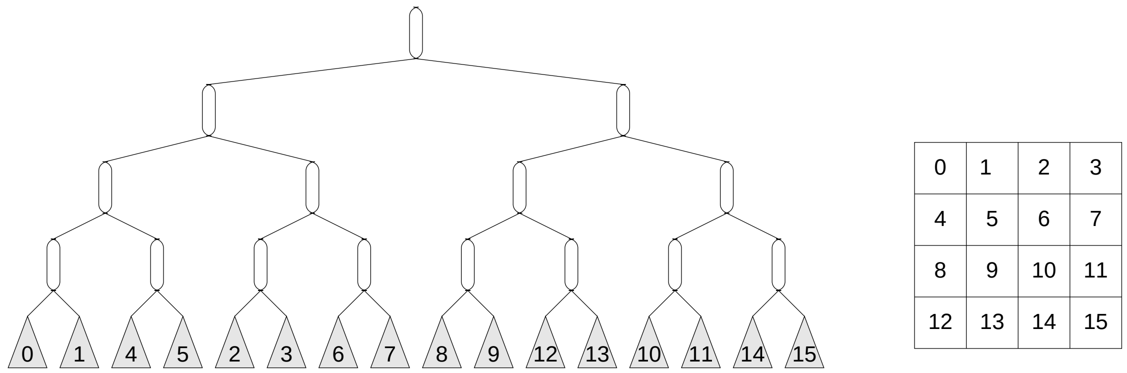
Recall that in the mapping of Figure 4.8(e), the least significant $\lceil \log p/2 \rceil$ bits of a row or column index of the matrix determine the processor to which that row or column belongs. Now if we disregard the least significant bit, and determine the distribution of rows and columns by the $\lceil \log p/2 \rceil$ bits starting with the second least significant bit, then the mapping of Figure 4.10(b) will result. In general, we can disregard the first k least significant bits, and arrive at a block-cyclic mapping with a block size of $2^k \times 2^k$. The optimal value of k depends on the ratio of computation time and the communication latency of the parallel computer in use and may vary from one computer to another for the same problem. In addition, increasing the block size too much may cause too much load imbalance during the dense Cholesky steps and may offset the advantage of using BLAS-3.

4.4.2 Subtree-to-Submesh Mapping for the 2-D Mesh Architecture

The mapping of rows and columns described so far works fine for the hypercube network. At each level, the update and frontal matrices are distributed on a logical mesh of processors (e.g., Figure 4.10(a)) such that each row and column of this mesh is a subcube of the hypercube. However, if the underlying architecture is a mesh, then a row or a column of the logical mesh may not correspond to a row or a column of the physical mesh. This will lead to contention for communication channels during the pipelined dense Cholesky steps of Figure 4.9 on a physical



(a) Subtree-subcube assignment of level-4 subtrees and the corresponding logical mesh for level-0 supernode



(b) Subtree-submesh assignment of level-4 subtrees and the corresponding logical mesh for level-0 supernode

Figure 4.11: Labeling of subtrees in subtree-to-subcube (a) and subtree-to-submesh (b) mappings.

mesh. To avoid this contention for communication channels, we define a subtree-to-submesh mapping in this subsection. The subtree-to-subcube mapping described in Figure 4.5(b) ensures that any subtree of the relaxed supernodal elimination tree is mapped onto a subcube of the physical hypercube. This helps in localizing communication at each stage of factorization among groups of as few processors as possible. Similarly, the subtree-to-submesh mapping ensures that a subtree is mapped entirely within a submesh of the physical mesh.

Note that in subtree-to-subcube mapping for a 2^d-processor hypercube, all level-d subtrees of the relaxed supernodal elimination tree are numbered in increasing order from left to right and a subtree labeled i is mapped onto processor i. For example, the subtree labeling of Figure 4.11(a) results in the update and frontal matrices for the supernodes in the topmost (level-0) relaxed supernode to be distributed among 16 processors as shown in Figure4.10(a). The subtree-to-submesh mapping starts with a different initial labeling of the level-d subtrees. Figure4.11(b) shows this labeling for 16 processors, which will result in the update and frontal matrices of the topmost relaxed supernode being partitioned on a 4×4 array of processors labeled in a row-major fashion.

We now define a function map such that replacing every reference to processor i in subtree-to-subcube mapping by a reference to processor $map(i, m, n)$ results in a subtree-to-submesh mapping on an $m \times n$ mesh. We assume that both m and n are powers of two. We also assume that either $m = n$ or $m = n/2$ (this configuration maximizes the cross-section width and minimizes the diameter of an mn-processor mesh). The function $map(i, m, n)$ is given by the following recurrence:

$$
\begin{array}{rcll}
map(i, m, n) & = & i, & \text{if } i < 2. \\
map(i, m, n) & = & map(i, \frac{m}{2}, n), & \text{if } m = n,\ i < \frac{mn}{2}. \\
map(i, m, n) & = & \frac{mn}{2} + map(i - \frac{mn}{2}, \frac{m}{2}, n), & \text{if } m = n,\ i \geq \frac{mn}{2}. \\
map(i, m, n) & = & m\lfloor map(i, m, \frac{n}{2})/m \rfloor + map(i, m, \frac{n}{2}), & \text{if } m = \frac{n}{2},\ i < \frac{mn}{2}. \\
map(i, m, n) & = & m(\lfloor map(i - \frac{mn}{2}, m, \frac{n}{2})/m \rfloor + 1) + map(i - \frac{mn}{2}, m, \frac{n}{2}), & \text{if } m = \frac{n}{2},\ i \geq \frac{mn}{2}.
\end{array}
$$

The above recurrence always maps a level-l relaxed supernode of a binary relaxed supernodal elimination tree onto an $(mn/2^l)$-processor submesh of the mn-processor two-dimensional mesh.

4.5 Analysis of Communication Overhead

In this section, we derive expressions for the communication overhead of our algorithm for sparse matrices resulting from a finite difference operator on regular two- and three-dimensional grids. Within constant factors, these expressions can be generalized to all sparse matrices that are adjacency matrices of all graphs whose node-degree is bounded by a constant. Such two- and three-dimensional graphs have the property that their n-node subgraphs have $\Theta(\sqrt{n})$-node and $\Theta(n^{2/3})$-node separators, respectively. All two- and three-dimensional finite-element graphs belong to this class. The properties of separators can be generalized from grids to all such graphs within the same order of magnitude bounds [91, 90, 45].

The parallel multifrontal algorithm described in Section 4.4 incurs two types of communication overhead: one during parallel extend-add operations (Figure 4.8) and the other during the steps of dense Cholesky factorization while processing the supernodes (Figure 4.9). Crucial to estimating the communication overhead is estimating the sizes of frontal and update matrices at any level of the supernodal elimination tree.

Consider a $\sqrt{N} \times \sqrt{N}$ regular finite difference grid. We analyze the communication overhead for factorizing the $N \times N$ sparse matrix associated with this grid on p processors. In order to simplify the analysis, we assume a somewhat different form of nested-dissection than the one used in the actual implementation. This method of analyzing the communication complexity of sparse Cholesky factorization has been used in [47] in the context of a column-based subtree-to-subcube scheme. Within very small constant factors, the analysis holds for the standard nested dissection [43] of grid graphs. We consider a cross-shaped separator (described in [47]) consisting of $2\sqrt{N}-1$ nodes that partitions the N-node square grid into four square subgrids of size $(\sqrt{N}-1)/2 \times (\sqrt{N}-1)/2$. We call this the level-0 separator that partitions the original grid (or the level-0 grid) into four level-1 grids. The nodes in the separator are numbered after the nodes in each subgrid have been numbered. To number the nodes in the subgrids, they are further partitioned in the same way, and the process is applied recursively until all nodes of the original grid are numbered. The supernodal elimination tree corresponding to this ordering is such that each non-leaf supernode has four children. The topmost supernode has $2\sqrt{N}-1$ ($\approx 2\sqrt{N}$) nodes, and the size of the supernodes at each subsequent level of the tree is half of the supernode size at the previous level. Clearly, the number of supernodes increases by a factor of four at each level, starting with one at the top (level 0).

The nested dissection scheme described above has the following properties: (1) the size of level-l subgrids is approximately $\sqrt{N}/2^l \times \sqrt{N}/2^l$, (2) the number of nodes in a level-l separator is approximately $2\sqrt{N}/2^l$, and hence, the length of a supernode at level l of the supernodal elimination tree is approximately $2\sqrt{N}/2^l$. It has been proved in [47] that the number of nonzeros that an $i \times i$ subgrid can contribute to the nodes of its bordering separators is bounded by ki^2, where $k = 341/12$. Hence, a level-l subgrid can contribute at most $kN/4^l$ nonzeros to its bordering nodes. These nonzeros are in the form of the triangular update matrix that is passed along from the root of the subtree corresponding to the subgrid to its parent in the elimination tree. The dimensions of a matrix with a dense triangular part containing $kN/4^l$ entries is roughly $\sqrt{2kN}/2^l \times \sqrt{2kN}/2^l$. Thus, the size of an update matrix passed on to level $l-1$ of the supernodal elimination tree from level l is roughly upper-bounded by $\sqrt{2kN}/2^l \times \sqrt{2kN}/2^l$ for $l \geq 1$.

The size of a level-l supernode is $2\sqrt{N}/2^l$; hence, a total of $2\sqrt{N}/2^l$ elimination steps take place while the computation proceeds from the bottom of a level-l supernode to its top. A single elimination step on a frontal matrix of size $(t+1) \times (t+1)$ produces an update matrix of size $t \times t$. Since the size of an update matrix at the top of a level-l supernode is at most $\sqrt{2kN}/2^l \times \sqrt{2kN}/2^l$, the size of the frontal matrix at the bottom of the same supernode is upper-bounded by $(\sqrt{2k}+2)\sqrt{N}/2^l \times (\sqrt{2k}+2)\sqrt{N}/2^l$. Hence, the average size of a frontal matrix at level l of the supernodal elimination tree is upper-bounded by $(\sqrt{2k}+1)\sqrt{N}/2^l \times (\sqrt{2k}+1)\sqrt{N}/2^l$. Let $\sqrt{2k}-1=\alpha$. Then $\alpha\sqrt{N}/2^l \times \alpha\sqrt{N}/2^l$ is an upper bound on the average size of a frontal matrix at level l.

We are now ready to derive expressions for the communication overhead due to the parallel extend-add operations and the elimination steps of dense Cholesky on the frontal matrices.

4.5.1 Overhead in Parallel Extend-Add

Before the computation corresponding to level $l-1$ of the supernodal elimination tree starts, a parallel extend-add operation is performed on lower triangular portions of the update matrices of size $\sqrt{2kN}/2^l \times \sqrt{2kN}/2^l$, each of which is distributed on a $\sqrt{p}/2^l \times \sqrt{p}/2^l$ logical mesh of processors. Thus, each processor holds roughly $(kN/4^l) \div (p/4^l) = kN/p$ elements of an update matrix. Assuming that each processor exchanges roughly half of its data with the corresponding processor of another subcube, $t_s + t_w kN/(2p)$ time is spent in communication, where t_s is the message startup time and t_w is the per-word transfer time. Note that this time is independent of l. Since there are $(\log p)/2$ levels at which parallel extend-add operations take place, the total communication time for these operations is $\Theta(N/p)\log p$ on a hypercube. The total communication overhead due to the parallel extend-add operations is $\Theta(N\log p)$ on a hypercube.

4.5.2 Overhead in Factorization Steps

We have shown earlier that the average size of a frontal matrix at level l of the supernodal elimination tree is bounded by $\alpha\sqrt{N}/2^l \times \alpha\sqrt{N}/2^l$, where $\alpha = \sqrt{341/6}-1$. This matrix is distributed on a $\sqrt{p}/2^l \times \sqrt{p}/2^l$ logical mesh of processors. As shown in Figure 4.9, there are two communication operations involved with each elimination step of dense Cholesky. The average size of a message is $(\alpha\sqrt{N}/2^l) \div (\sqrt{p}/2^l) = \alpha\sqrt{N/p}$. It can be shown [109, 81] that in a pipelined implementation on a $\sqrt{q} \times \sqrt{q}$ mesh of processors, the communication time for s elimination steps with an average message size of m is $\Theta(ms)$. The reason is that although each message must go to $\Theta(\sqrt{q})$ processors, messages corresponding to $\Theta(\sqrt{q})$ elimination steps are active simultaneously in different parts of the mesh. Hence, each message effectively contributes only $\Theta(m)$ to the total communication time. In our case, at level l of the supernodal elimination tree, the number of steps of dense Cholesky is $2\sqrt{N}/2^l$. Thus the total communication time at level l is $\alpha\sqrt{N/p} \times 2\sqrt{N}/2^l = \Theta((N/\sqrt{p})(1/2^l))$. The total communication time for the elimination steps at the top $(\log p)/2$ levels of the supernodal elimination tree is $\Theta((N/\sqrt{p})\Sigma_{l=0}^{\log_4 p-1}(1/2^l))$. This has an upper bound of $\Theta(N/\sqrt{p})$. Hence, the total communication overhead due to the elimination steps is $\Theta(p \times N/\sqrt{p}) = \Theta(N\sqrt{p})$.

The parallel multifrontal algorithm incurs an additional overhead of emptying the pipeline $\log p$ times (once before each parallel extend-add) and then refilling it. It can be easily shown that this overhead is $\Theta(N)$ each time the pipeline restarts. Hence, the overall overhead due to restarting the pipeline $\log p$ time is $\Theta(N \log p)$, which is smaller in magnitude than the $\Theta(N\sqrt{p})$ communication overhead of the dense Cholesky elimination steps.

4.5.3 Communication Overhead for 3-D Problems

The analysis of the communication complexity for the sparse matrices arising out of three-dimensional finite element problems can be performed along the lines of the analysis for the case of two-dimensional grids. Consider an $N^{1/3} \times N^{1/3} \times N^{1/3}$ grid that is recursively partitioned into eight subgrids by a separator that consists of three orthogonal $N^{1/3} \times N^{1/3}$ planes. The number of nonzeros that an $i \times i \times i$ subgrid contributes to the nodes of its bordering separators is $\Theta(i^4)$ [47]. At level l, due to l bisections, i is no more than $N^{1/3}/2^l$. As a result, an update or a frontal matrix at level l of the supernodal elimination tree will contain $\Theta(N^{4/3}/2^{4l})$ entries distributed among $p/8^l$ processors. Thus, the communication time for the parallel extend-add operation at level l is $\Theta(N^{4/3}/(2^l p))$. The total communication time for all parallel extend-add operations is $\Theta((N^{4/3}/p)\Sigma_{l=0}^{\log_8 p-1}(1/2^l))$, which is $\Theta(N^{4/3}/p)$. For the dense Cholesky elimination steps at any level, the message size is $\Theta(N^{2/3}/\sqrt{p})$. Since there are $3N^{2/3}/4^l$ nodes in a level-l separator, the total communication time for the elimination steps is $\Theta((N^{4/3}/\sqrt{p})\Sigma_{l=0}^{\log_8 p-1}(1/4^l))$, which is $\Theta(N^{4/3}/\sqrt{p})$.

Hence, the total communication overhead due to parallel extend-add operations is $\Theta(N^{4/3})$ and that due to the dense Cholesky elimination steps is $\Theta(N^{4/3}\sqrt{p})$. As in the 2-D case, these asymptotic expressions can be generalized to sparse matrices resulting from three-dimensional graphs whose n-node subgraphs have $\Theta(n^{2/3})$-node separators. This class includes the linear systems arising out of three-dimensional finite element problems.

4.5.4 Communication Overhead on a Mesh

The communication overhead due the dense Cholesky elimination steps is the same on both the mesh and the hypercube architectures because the frontal matrices are distributed on a logical mesh of processors. However, the parallel extend operations use the entire cross-section bandwidth of a hypercube, and the communication overhead due to them will increase on a mesh due to channel contention.

Recall that the communication time for parallel extend-add at any level is $\Theta(N/p)$ on a hypercube. The extend-add is performed among groups of $p/4^l$ processors at level l of the supernodal elimination tree. Therefore, at level l, the communication time for parallel extend-add on a $\sqrt{p}/2^l \times \sqrt{p}/2^l$ submesh is $\Theta(N/(2^l\sqrt{p}))$. The total communication time for all the levels is $\Theta((N/\sqrt{p})\Sigma_{l=0}^{\log_4 p-1}(1/2^l))$. This has an upper bound of $\Theta(N/\sqrt{p})$, and the upper bound on the corresponding communication overhead term is $\Theta(N\sqrt{p})$. This is the same as the total communication overhead for the elimination steps. Hence, for two-dimensional problems, the overall asymptotic communication overhead is the same for both mesh and hypercube architectures.

The communication time on a hypercube for the parallel extend-add operation at level l is $\Theta(N^{4/3}/(2^l p))$ for three-dimensional problems. The corresponding communication time on a mesh would be $\Theta(N^{4/3}/(4^l\sqrt{p})$. The total communication time for all the parallel extend-add operations is $\Theta((N^{4/3}/\sqrt{p})\Sigma_{l=0}^{\log_8 p-1}(1/4^l))$, which is $\Theta(N^{4/3}/\sqrt{p})$. As in the case of two-dimensional problems, this is asymptotically equal to the communication time for the elimination steps.

4.6 Scalability Analysis

It is well known [45] that the total work involved in factoring the adjacency matrix of an N-node graph with an $\Theta(\sqrt{N})$-node separator using nested dissection ordering of nodes is $\Theta(N^{1.5})$. We have shown in Section 4.5 that the overall communication overhead of our scheme is $\Theta(N\sqrt{p})$.

From Equation 2.4, a fixed efficiency can be maintained if and only if $N^{1.5} \propto N\sqrt{p}$, or $\sqrt{N} \propto \sqrt{p}$, or $N^{1.5} = W \propto p^{1.5}$. In other words, the problem size must be increased as $\Theta(p^{1.5})$ to maintain a constant efficiency as p is increased. In comparison, a lower bound on the isoefficiency function of Rothberg and Gupta's scheme [120] with a communication overhead of at least $\Theta(N\sqrt{p}\log p)$ is $\Theta(p^{1.5}(\log p)^3)$. The isoefficiency function of any column-based scheme is at least $\Theta(p^3)$ because the total communication overhead has a lower bound of $\Theta(Np)$. Thus, the scalability of our algorithm is superior to that of the other schemes.

It is easy to show that the scalability of our algorithm is $\Theta(p^{1.5})$ even for the sparse matrices arising out of three-dimensional finite element grids. The problem size in the case of an $N \times N$ sparse matrix resulting from a three-dimensional grid is $\Theta(N^2)$ [47]. We have shown in Section 4.5 that the overall communication overhead in this case is $\Theta(N^{4/3}\sqrt{p})$. To maintain a fixed efficiency, $N^2 \propto N^{4/3}\sqrt{p}$, or $N^{2/3} \propto \sqrt{p}$, or $N^2 = W \propto p^{1/5}$.

A lower bound on the isoefficiency function for dense matrix factorization is $\Theta(p^{1.5})$ [81, 82] if the number of rank-1 updates performed by the serial algorithm is proportional to the rank of the matrix. The factorization of a sparse matrix derived from an N-node graph with an $S(N)$-node separator involves a dense $S(N) \times S(N)$ matrix factorization. $S(N)$ is $\Theta(\sqrt{N})$ and $\Theta(N^{2/3})$ for two- and three-dimensional constant node-degree graphs, respectively. Thus, the complexity of the dense portion of factorization for these two types of matrices is $\Theta(N^{1.5})$ and $\Theta(N^2)$, respectively, which is of the same order as the computation required to factor the entire sparse matrix [45, 47]. Therefore, the isoefficiency function of sparse factorization of such matrices is bounded from below by the isoefficiency function of dense matrix factorization, which is $\Theta(p^{1.5})$. As we have shown earlier in this section, our algorithm achieves this lower bound for both two- and three-dimensional cases.

4.6.1 Scalability with Respect to Memory Requirement

We have shown that the problem size must increase in proportion to $p^{1.5}$ for our algorithm to achieve a fixed efficiency. As the overall problem size increases, so does the overall memory requirement. For an N-node two-dimensional constant node-degree graphs, the size of the lower triangular factor L is $\Theta(N \log N)$ [45]. For a fixed efficiency, $W = N^{1.5} \propto p^{1.5}$, which implies $N \propto p$ and

$N \log N \propto p \log p$. As a result, if we increase the number of processors while solving bigger problems to maintain a fixed efficiency, the overall memory requirement increases at the rate of $\Theta(p \log p)$ and the memory requirement per processor increase logarithmically with respect to the number of processors.

In the three-dimensional case, size of the lower triangular factor L is $\Theta(N^{4/3})$ [45]. For a fixed efficiency, $W = N^2 \propto p^{1.5}$, which implies $N \propto p^{3/4}$ and $N^{4/3} \propto p$. Hence, in this case, the overall memory requirement increases linearly with the number of processors and the per-processor memory requirement is constant for maintaining a fixed efficiency. It can be easily shown that for the three-dimensional case, the isoefficiency function should not be of a higher order than $\Theta(p^{1.5})$ if speedups proportional to the number of processors are desired without increasing the memory requirement per processor. To the best of our knowledge, the algorithm described in Section 4.4 is the only parallel algorithm for sparse Cholesky factorization that satisfies this condition.

4.7 Experimental Results of Sparse Cholesky Factorization

We implemented the parallel multifrontal algorithm described in this chapter on the nCUBE2 parallel computer. We have gathered some preliminary speedup results for two classes of sparse matrices, which are summarized in Tables 4.1 and 4.2. The entire code is written in the C programming language augmented with the message passing primitives for nCUBE and compiled using the *ncc* compiler with the *-O* optimization option. The results for 64 processors or less were obtained on the 64-processor nCUBE2 (with 16 MB memory per node) at the CIS department of University of Florida at Gainsville. The results for 128 to 1024 processors were obtained on the 1024-processor nCUBE2 (with 4 MB memory per node) at Sandia National Labs. All timings are for numerical factorization with single precision arithmetic. From an independent set of experiments, we determined the time for a floating point operation to be roughly 0.5μs, the message startup time to be roughly 180μs, and the communication rate with no channel conflicts to be roughly 2 bytes per μs.

We conducted one set of experiments on sparse matrices associated with a 9-point difference operator on rectangular grids. The purpose of these experiments was to compare their results with the scalability analysis in Section 4.6. The dimensions of the grids were chosen such that the elimination trees were as balanced as possible. The standard nested dissection ordering [43] was used for these matrices. Nested dissection has been shown to have optimal fill-in in the case of regular grids [45]. The results of our implementation for some of these grids are summarized in Table 4.1. Matrix GRIDixj in the table refers to the sparse matrix obtained from an $i \times j$ 9-point finite difference grid.

From our experiments on the 2-D grids, we selected a few points of equal efficiency and plotted the W versus p curve, which is shown by the solid line in Figure 4.12 for $E \approx 0.31$. The problem size W is measured in terms of the total number of floating point arithmetic operations performed during factorization; for example, for GRID63x63, $W = 4.1684 \times 10^6$. The dotted curve below the experimental curve corresponds to an isoefficiency function of $\Theta(p^{1.5})$ and the dashed curve at the top corresponds to an isoefficiency function of $\Theta(p^{1.5}(\log p)^3)$. The first point on each curve is that for the matrix GRID63x63 on 64 processors. To obtain the dotted curve, W was increased by a factor of $2\sqrt{2}$ every time the number of processors p was doubled. To obtain the dashed curve, a constant was defined to be $4.1684 \times 10^6/(64 \times 8 \times 6^3) \approx 37.7$. Then, for any $p > 64$, $W = 37.7p(\log p)^3$. Given that for $p = 64$, an efficiency of roughly 31% is obtained for problem size 4.1684×10^6, the dotted and dashed curves indicate the problem sizes that will yield the same efficiency for p = 128, 256, and 512 if the isoefficiency function is $\Theta(p^{1.5})$ and $\Theta(p^{1.5}(\log p)^3)$, respectively.

Figure 4.12 shows that the experimental isoefficiency curve is considerably better than $\Theta(p^{1.5}(\log p)^3)$, which is a lower bound on the isoefficiency function of the previously best known (in terms of total communication volume) parallel algorithm [120] for sparse matrix factorization. However, it is worse than $\Theta(p^{1.5})$, which is the asymptotic isoefficiency function derived in Section 4.6. There are two main reasons for this. First, the $\Theta(p^{1.5})$ isoefficiency function does not take load imbalance into account. It has been shown in [96] that even if a grid graph is perfectly partitioned in terms of the number of nodes, the work load associated with each partition varies. The partitions closer to the center of the grid require more computation than the ones on or closer to the periphery. Another reason that the experimental isoefficiency function appears worse than the prediction is that the efficiencies of the parallel implementation are computed with respect to a very efficient serial implementation. This can be judged from the run times on a single nCUBE2

Matrix: GRID63x63; N = 3969; NNZ = 99.45 thousand; FLOP = 4.1684 million								
p	1	2	4	8	16	32	64	128
Time	8.27	4.22	2.22	1.41	.904	.594	.411	.315
Speedup	1.00	1.96	3.73	5.87	9.15	13.9	20.1	26.2
Efficiency	100.0%	98.0%	93.1%	73.3%	57.2%	43.5%	31.4%	20.5%
Matrix: GRID103x95; N = 9785; NNZ = 288.04 thousand; FLOP = 16.599 million								
p	1	4	8	16	32	64	128	256
Time	28.92	7.500	4.493	2.699	1.596	1.017	.7301	.5516
Speedup	1.00	3.86	6.44	10.7	18.1	28.4	39.6	52.4
Efficiency	100.0%	96.4%	80.5%	66.9%	56.6%	44.4%	31.0%	20.5%
Matrix: GRID127x127; N = 16129; NNZ = 518.58 thousand; FLOP = 36.682 million								
p	1	4	8	16	32	64	128	256
Time	58.86	15.07	8.878	5.155	2.971	1.783	1.171	.8546
Speedup	1.00	3.90	6.63	11.4	19.8	33.0	50.3	68.9
Efficiency	100.0%	97.6%	82.8%	71.0%	61.9%	51.6%	39.3%	26.9%
Matrix: GRID175x127; N = 22225; NNZ = 731.86 thousand; FLOP = 56.125 million								
p	1	8	16	32	64	128	256	512
Time	87.38	12.74	7.420	4.164	2.468	1.570	1.092	.8529
Speedup	1.00	6.86	11.8	21.0	35.4	55.7	80.0	102.4
Efficiency	100.0%	85.7%	73.6%	65.6%	55.3%	43.5%	31.3%	20.0%
Matrix: GRID255x127; N = 32385; NNZ = 1140.6 thousand; FLOP = 100.55 million								
p	1	16	32	64	128	256	512	1024
Time	149.15	12.22	6.651	3.861	2.349	1.557	1.091	.8357
Speedup	1.00	12.2	22.4	38.6	63.5	95.8	136.7	178.5
Efficiency	100.0%	76.3%	70.1%	60.4%	49.6%	37.4%	26.7%	17.4%
Matrix: GRID223x207; N = 46161; NNZ = 1498.7 thousand; FLOP = 179.87 million								
p	1	16	32	64	128	256	512	1024
Time	254.70	20.21	10.85	6.161	3.751	2.380	1.634	1.237
Speedup	1.00	12.6	23.5	41.3	67.9	107.0	155.9	205.9
Efficiency	100.0%	78.8%	73.4%	64.6%	53.0%	41.8%	30.5%	20.1%

Table 4.1: Experimental results for factoring sparse symmetric positive definite matrices associated with a 9-point difference operator on rectangular grids. All times are in seconds.

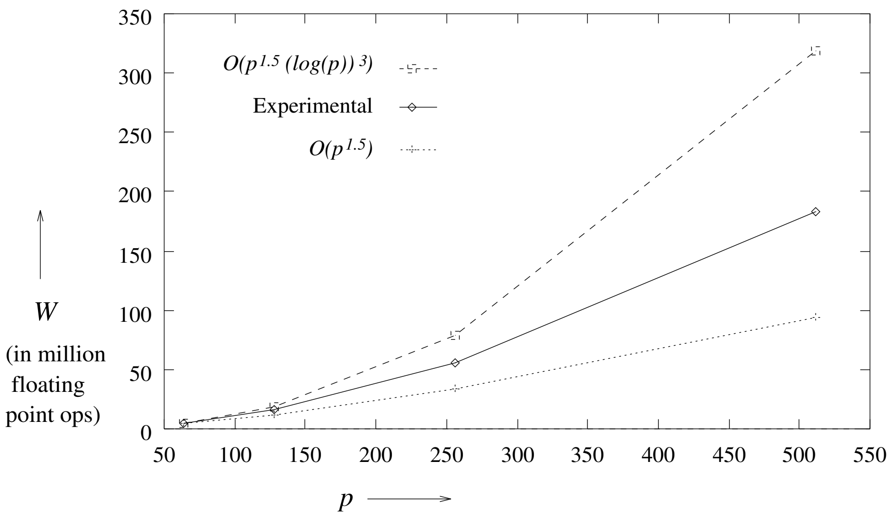


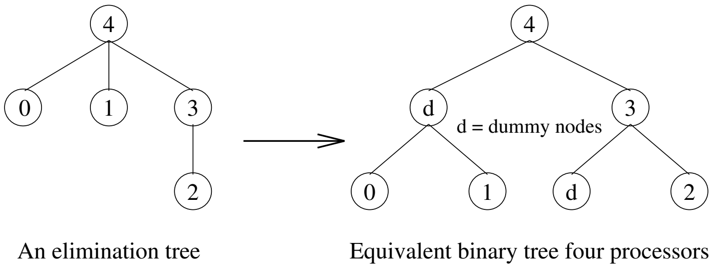
Figure 4.12: Comparison of our experimental isoefficiency curves with $\Theta(p^{1.5})$ curve (theoretical asymptotic isoefficiency function of our algorithm due to communication overhead on a hypercube) and with $\Theta(p^{1.5}(\log p)^3)$ curve (the lower bound on the isoefficiency function of the best known parallel sparse factorization algorithm until now). The four data points on the curves correspond to the matrices GRID63x63, GRID103x95, GRID175x127, and GRID223x207.

processor whose peak performance is rated at roughly 2.5 MFLOPS (double precision) and 8 MIPS. In our implementation, the computation associated with the subtrees below level $\log p$ in the relaxed supernodal elimination tree is handled by the serial code. However, the computation above this level is handled by a separate code. In our preliminary implementation, this part of the code is less efficient than the serial code (disregarding communication) due to additional bookkeeping, which has a potential for optimization. For example, the total time spent by all the processors participating in a parallel extend-add operation besides message passing is more than the time taken to perform extend-add on the same update matrices on a single processor. The same is true for the dense factorization steps too. However, despite these inefficiencies, our implementation is more scalable than a hypothetical ideal implementation (with perfect load balance) of the previously best known parallel algorithm for sparse Cholesky factorization.

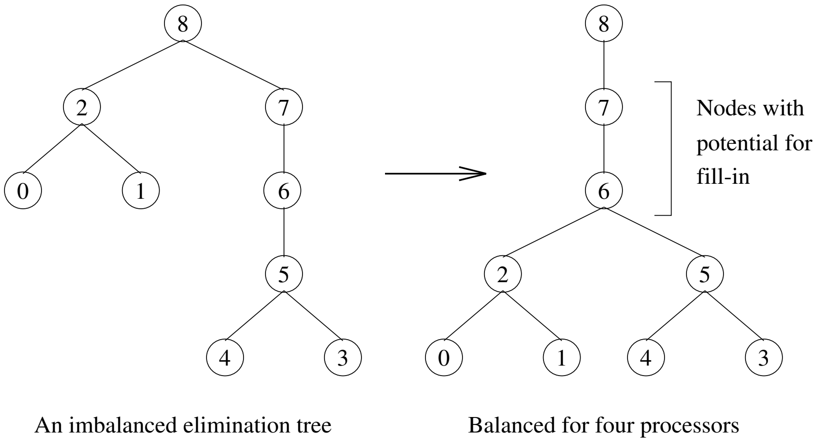
In Table 4.2 we summarize the results of factoring some matrices from the Harwell-Boeing collection of sparse matrices [30]. The purpose of these experiments was to to demonstrate that our algorithm can deliver good speedups on hundreds of processors for practical problems. Spectral

Matrix: BCSSTK15; N = 3948; NNZ = 488.8 thousand; FLOP = 85.55 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	103.73	52.63	26.66	14.88	8.29	4.98	3.20	2.156	1.530		
Speedup	1.00	1.97	3.89	6.97	12.5	20.8	32.4	48.1	67.8		
Efficiency	100.0%	98.5%	97.3%	87.1%	78.2%	65.1%	50.7%	37.6%	26.5%		
Load balance	100%	99%	98%	91%	91%	87%	87%	84%	84%		
Matrix: BCSSTK25; N = 15439; NNZ = 1940.3 thousand; FLOP = 512.88 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	588.5	301.23	184.84	74.71	52.29	30.01	16.66	10.38	6.64	4.53	
Speedup	1.00	1.95	3.18	6.21	11.3	19.6	35.3	56.7	88.6	129.9	
Efficiency	100.0%	97.7%	79.6%	77.7%	70.3%	61.3%	57.0%	44.3%	34.6%	25.4%	
Load balance	100%	98%	80%	78%	71%	63%	62%	62%	62%	61%	
Matrix: BCSSTK29; N = 13992; NNZ = 2174.46 thousand; FLOP = 609.08 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	704.0	359.7	212.9	110.45	55.06	31.36	19.22	12.17	7.667	4.631	3.119
Speedup	1.00	1.96	3.31	6.37	12.8	22.5	36.6	57.9	91.8	152.6	225.6
Efficiency	100.0%	97.9%	82.7%	79.7%	79.9%	70.2%	57.2%	45.2%	35.9%	29.8%	22.0%
Load balance	100%	98%	83%	82%	84%	82%	77%	72%	68%	72%	72%
Matrix: BCSSTK30; N = 28924; NNZ = 5893.59 thousand; FLOP = 2246.0 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	2599.0*	1493.5	1050.8	537.4	256.6	134.5	79.93	43.67	24.73	14.616	11.078
Speedup	1.00	1.74	2.47	4.84	10.1	19.3	32.5	59.5	105.0	177.8	234.6
Efficiency	100.0%	87.0%	61.8%	60.5%	63.3%	60.4%	50.8%	46.5%	41.0%	34.7%	22.9%
Matrix: BCSSTK31; N = 35588; NNZ = 6458.34 thousand; FLOP = 2583.6 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	3358.0*	1690.7	924.6	503.0	262.0	134.3	73.57	42.02	24.58	14.627	9.226
Speedup	1.00	1.99	3.63	6.68	12.8	25.0	45.6	79.9	136.6	229.6	364.2
Efficiency	100.0%	99.3%	90.8%	83.4%	80.1%	78.1%	71.3%	62.4%	53.4%	44.8%	35.6%
Matrix: BCSSTK32; N = 44609; NNZ = 8943.91 thousand; FLOP = 4209.0 million											
p	1	2	4	8	16	32	64	128	256	512	1024
Time	5215.0#					276.17	153.23		46.24	27.25	16.40
Speedup	1.00					18.9	34.0		112.8	191.4	318.0
Efficiency	100.0%					59.0%	53.2%		44.1%	37.4%	31.1%

Table 4.2: Experimental results for factoring some sparse symmetric positive definite matrices resulting from 3-D problems in structural engineering. All times are in seconds. The single processor run times suffixed by "*" and "#" were estimated by timing different parts of factorization on two and 32 processors, respectively.



(a) Converting an elimination tree to a binary tree



(b) Balancing an elimination tree

Figure 4.13: The two functions performed by the tree balancing algorithm.

nested dissection (SND) [111, 112, 113] was used to order the matrices in Table 4.2. This choice of the ordering scheme was prompted by two factors. First, there is increasing evidence that spectral orderings offer a good balance between generality of application and the quality of ordering—both in terms of load balance and fill reduction [20]. Second, the SND algorithm itself can be parallelized efficiently, whereas most other ordering schemes do not appear to be as well-suited for parallelization. Although, at the present time we compute the ordering on a serial computer, SND is our ordering algorithm of choice in a prospective completely parallel implementation of a sparse linear system solver based on our parallel multifrontal algorithm. A drawback of using a serial implementation of SND is that its run time is too high. However, variations of SND such as multilevel SND [14, 113] run much faster without compromising on the quality of ordering.

From the experimental results in Tables 4.1 and 4.2, we can infer that our algorithm can deliver substantial speedups, even on moderate problem sizes. These speedups are computed with respect to a very efficient serial implementation of the multifrontal algorithm. To lend credibility to our speedup figures, we compared the run times of our program on a single processor with the single processor run times given for iPSC/2 in [114] and [127]. The nCUBE2 processors are about 2 to 3 times faster than iPSC/2 processors and our serial implementation, with respect to which the speedups are computed, is 4 to 5 times faster than the one in [114] and [127]. Our single processor run times are four times less than the single processor run times on iPSC/2 reported in [9]. We also found that for some matrices (e.g., that from a 127×127 9-point finite difference grid), our implementation on eight nCUBE2 processors (8.9 seconds) is faster than the 16-processor iPSC/860 implementation (9.7 seconds) reported in [141], although iPSC/860 has much higher computation speeds.

4.7.1 Load Balancing for Factorization

The factorization algorithm as described in this chapter requires a binary relaxed supernodal elimination trees that are fairly balanced. After obtaining the ordered matrix and the corresponding elimination tree, we run the elimination tree through a very fast tree balancing heuristic. This heuristic performs a single pass of depth-first search on the elimination tree and accomplishes two tasks. First, it converts a general tree into a binary tree required by the subtree to subcube mapping. Second, it performs limited reordering of the subtrees within the elimination tree to ensure that the load imbalance at a given level does not exceed a predefined tolerance. The functionality of this algorithm is briefly described in Figure 4.13. If elimination of a column is regarded as a basic subtask in Cholesky factorization, then the elimination tree gives a partial ordering of these subtask for correct factorization [92]. Our tree balancing algorithm is based on the fact that a modified elimination tree that does not violate the partial order specified by the original tree still leads to correct factorization. This algorithm generally improves the load-balance significantly in parallel factorization at the cost of very small increase in fill.

Table 4.2 also gives the percentage load balance for different values of p for BCSSTK15, BCSSTK25, and BCSSTK29 matrices. To evaluate the load balance, we wrote a sequential program that simulates the parallel multifrontal algorithm and reports the maximum achievable efficiency for a given number of processors in the absence of communication overhead. The simulator takes as input the same binary relaxed supernodal elimination tree as the the parallel algorithm. Now the parallel run time to factorize the part of the matrix corresponding to a subtree rooted at level l of the binary relaxed supernodal elimination tree is given by the sum of the parallel run time to process the root of the subtree and the parallel run time to eliminate the subtrees of the root. The parallel run time to processes the root is computed by dividing the serial run time to processes the root by $p/2^m$, where $m = l$ if $0 \leq l < \log p$ and $m = \log p$ if $l \geq \log p$. The parallel run time to eliminate the subtrees is computed as the maximum of the parallel run times for the two subtrees if $0 \leq l < \log p$ and as the sum of the run times of the subtrees if $l \geq \log p$. Thus, by timing the work associated with individual subtrees in the recursive serial implementation of the multifrontal algorithm, the simulator can easily estimate the parallel run time in the absence of communication overhead for a given p. The load balance or the maximum achievable efficiency is then estimated by dividing the serial run time with the product of p and the estimated parallel run time. Note that the inefficiency due to load imbalance of subtree-to-subcube mapping does not continue to increase with the number of processors, but tends to saturate at 64–128 processors.

It is evident from the load balance values given in Table 4.2 that a combination of spectral nested dissection with our tree balancing algorithm results in very respectable load balances for up to 1024 processors. The number of nonzeros in the triangular factor (NNZ) and the number of floating point operations (FLOP) reported in Table 4.2 are for the single processor case. As the number of processors is increased, the tree balancing algorithm is applied to more levels ($\log p$) of the relaxed supernodal elimination tree, and consequently, the total NNZ and FLOP increase. Thus, an efficiency of $x\%$ in Table 4.2 indicates that there is a $(100 - x)\%$ loss, which includes three factors: communication, load imbalance, and extra work. For example, the efficiency for BCSSTK25 on 64 processors is 57%; i.e., there is a 43% overhead. However, only 5% overhead is due to communication and extra work. The remaining 38% overhead is due to load imbalance.

Although we have observed through our experiments that the upper bound on efficiency due to load imbalance does not fall below 60–70% for hundreds of processors, even this bound can be improved further. The subtree-to-subcube mapping can be relaxed [53, 78] to a subforest-to-

					Number of Processors					
Problem	n	$\|A\|$	$\|L\|$	OPC	32	64	128	256	512	1024
PILOT87	2030	122550	504060	240M	0.44	0.73	1.05			
MAROS-R7	3136	330472	1345241	720M	0.83	1.41	2.14	3.02	4.07	4.48
FLAP	51537	479620	4192304	940M	0.75	1.27	1.85	2.87	3.83	4.25
BCSSTK33	8738	291583	2295377	1000M	0.76	1.30	1.94	2.90	4.36	6.02
BCSSTK30	28924	1007284	5796797	2400M		1.48	2.42	3.59	5.56	7.54
BCSSTK31	35588	572914	6415883	3100M	0.80	1.45	2.48	3.97	6.26	7.93
BCSSTK32	44609	985046	8582414	4200M		1.51	2.63	4.16	6.91	8.90
COPTER2	55476	352238	12681357	9200M	1.10	1.94	3.31	5.76	9.55	14.78
CUBE35	42875	124950	11427033	10300M	1.27	2.26	3.92	6.46	10.33	15.70
NUG15	6330	186075	10771554	29670M			4.32	7.54	12.53	19.92

Table 4.3: The performance of sparse Cholesky factorization on Cray T3D (from [53, 78]). For each problem the table contains the number of equations n of the matrix A, the original number of nonzeros in A, the nonzeros in the Cholesky factor L, the number of operations required to factor the nodes, and the performance in gigaflops for different number of processors.

subcube mapping, which reduces load imbalances at the cost of a little increase in communication. A preliminary implementation of this variation yields up to 20 GFLOPS on medium-size problems on a 1024-processor Cray T3D. Table 4.3 and Figure 4.14 show the performance of this implementation on some selected matrices.

4.8 Parallel Algorithms for Forward Elimination and Backward Substitution in Direct Solution of Sparse Linear Systems

A few parallel algorithms for solving triangular systems resulting from parallel factorization of sparse linear systems have been proposed and implemented recently. We present a detailed analysis of the parallel complexity and scalability of parallel algorithm described briefly in [64] to obtain a solution to the system of sparse linear equations of the forms $LY = B$ and $UX = Y$, where L is a lower triangular matrix and U is an upper triangular matrix. Here L and U are obtained from the numerical factorization of a sparse coefficient matrix A of the original system $AX = B$ to be

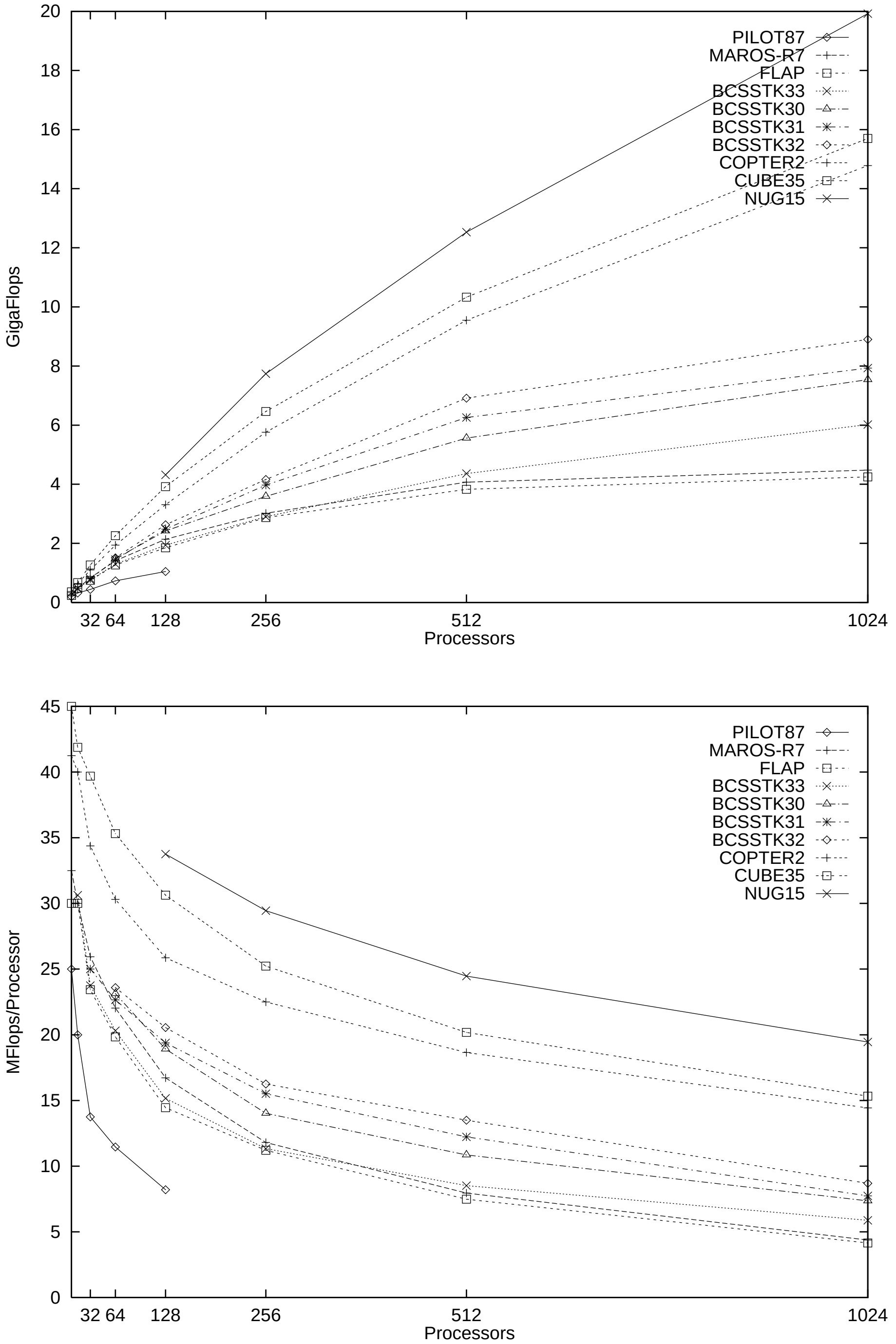


Figure 4.14: Plot of the performance of the parallel sparse multifrontal algorithm for various problems on Cray T3D (from [53, 78]). The first plot shows total Gigaflops obtained and the second one shows Megaflops per processor.

solved. If A, L, and U are $N \times N$ matrices, then X, Y, and B are $N \times m$ matrices, where m is the number of right-hand side vectors for which the solution to the sparse linear system with A as the coefficient matrix is desired. Our analysis and experiments show that, although not as scalable as the best parallel sparse Cholesky factorization algorithms, parallel sparse triangular solvers can yield reasonable speedups in runtime on hundreds of processors. We also show that for a wide class of problems, the sparse triangular solvers described in this chapter are optimal and are asymptotically as scalable as a dense triangular solver.

For a single right-hand side ($m = 1$), our experiments show a 256-processor performance of up to 435 MFLOPS on a Cray T3D, on which the single-processor performance for the same problem is $\approx$ 8.6 MFLOPS. With $m = 30$, the maximum single-processor and 256-processor performance observed in our experiments is $\approx$ 30 MFLOPS and $\approx$ 3050 MFLOPS, respectively. To the best of our knowledge, this is the highest performance and speedup for this problem reported on any massively parallel computer.

In addition to the performance and scalability analysis of parallel sparse triangular solvers, we discuss the redistribution of the triangular factor matrix among the processors between numerical factorization and triangular solution, and its impact on performance. In [53], we describe an optimal data-distribution scheme for Cholesky factorization of sparse matrices. This distribution leaves groups of consecutive columns of L with identical pattern of non-zeros (henceforth called *supernodes*) with a two-dimensional partitioning among groups of processors. However, this distribution is not suitable for the triangular solvers, which are scalable only with a one-dimensional partitioning of the supernodal blocks of L. We show that if the supernodes are distributed in a subtree-to-subcube manner [46] then the cost of converting the two-dimensional distribution to a one-dimensional distribution is only a constant times the cost of solving the triangular systems. From our experiments, we observed that this constant is fairly small on the Cray T3D—at most 0.9 for a single right-hand side vector among the test cases used in our experiments. Of course, if more than one systems need to be solved with the same coefficient matrix, then the one-time redistribution cost is amortized.

4.8.1 Algorithm Description

In this section, we describe parallel algorithms for sparse forward elimination and backward substitution, which have been discussed briefly in [64]. The description in this section assumes a single right-hand side vector; however, the algorithm can easily be generalized to multiple right-hand sides by replacing all vector operations by the corresponding matrix operations.

Forward Elimination

The basic approach to forward elimination is very similar to that of multifrontal numerical factorization [92] guided by an elimination tree [93, 81] with the distribution of computation determined by a subtree-to-subcube mapping [46]. A symmetric sparse matrix, its lower triangular Cholesky factor, and the corresponding elimination tree with subtree-to-subcube mapping onto 8 processors is shown in Figure 4.5. The computation in forward elimination starts with the leaf supernodes of the elimination tree and progresses upwards to terminate at the root supernode. A supernode is a set of columns $i_1, i_2, \ldots, i_t$ of the sparse matrix such that all of them have non-zeros in identical locations and i_{j+1} is the parent of i_j in the elimination tree for $1 \leq j < t$. For example, in Figure 4.5, nodes 6, 7, and 8 form a supernode. The portion of the lower triangular matrix L corresponding to a supernode is a dense trapezoidal block of width t and maximum height n, where t is the number of nodes in the supernode and n is the number of non-zeros in the leftmost column of the supernode. Figure 4.15 outlines the forward elimination process across three levels of the left half of the elimination tree of Figure 4.5. One of the blocks of L shown in Figure 4.15 is the dense trapezoidal supernode consisting of nodes 6, 7, and 8. For this supernode, $n = 4$ and $t = 3$.

As in the case of multifrontal numerical factorization [92], the computation in forward and backward triangular solvers can also be organized in terms of dense matrix operations. In forward elimination (see Figure 4.15), before the computation starts at a supernode, the elements of the right-hand side vector with the same indices as the nodes of the supernode are collected in the first t contiguous locations in a vector of length n. The remaining $n - t$ entries of this vector are filled with zeros. The computation corresponding to a trapezoidal supernode, which starts at the leaves, consists of two parts. The first computation step is to solve the dense triangular system at the top of the trapezoid (above the dotted line in Figure 4.15). The second step is to subtract the product of

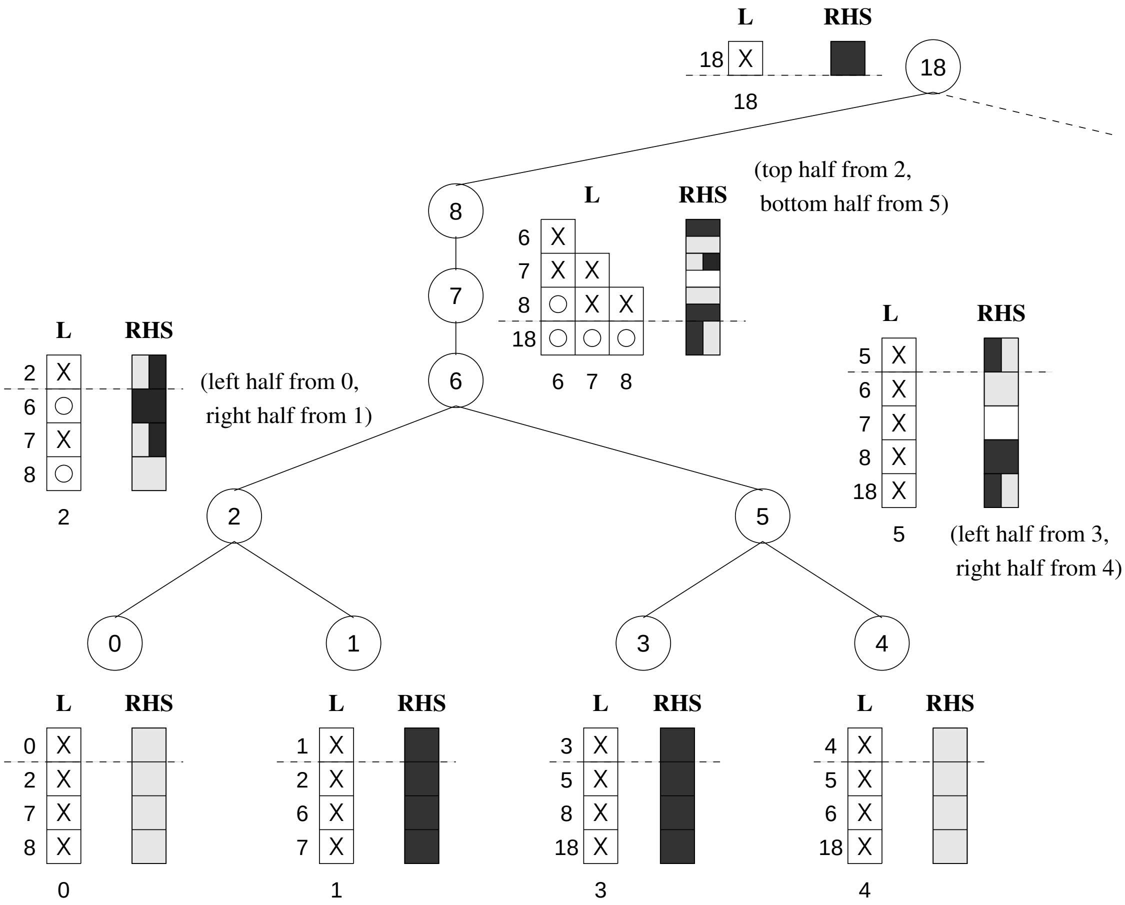


Figure 4.15: Pictorial representation of forward elimination along three levels of an elimination tree. The color of an RHS box is determined by the color(s) of the box(es) at the next lower level that contribute to its value.

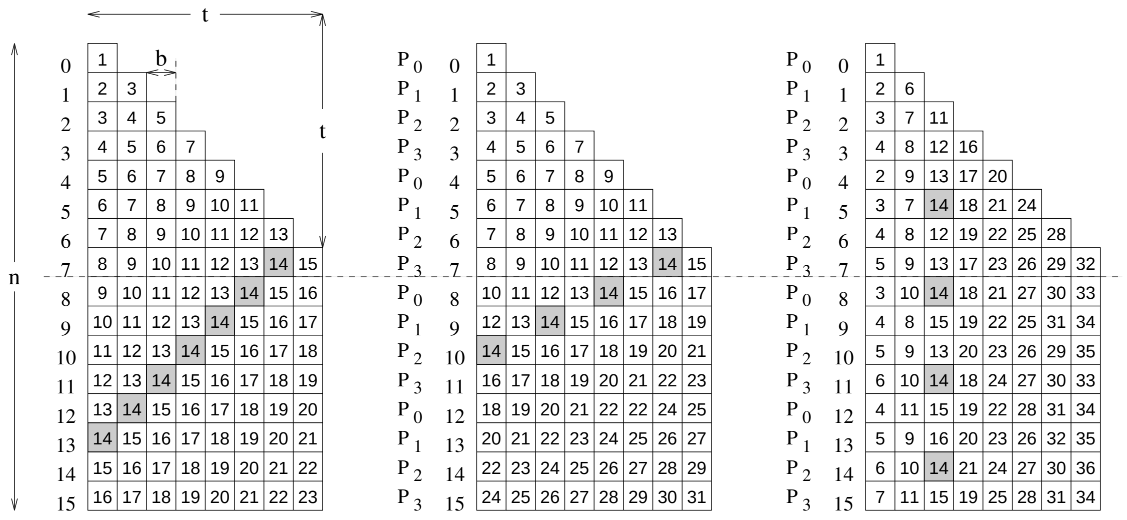
the vector of length t (above the dotted line) with the $(n-t) \times t$ submatrix of L (below the dotted line) from the vector of length $n-t$ (below the dotted line). After these two computation steps, the entries in the lower part of the vector of length $n-t$ are subtracted from the corresponding (i.e., with the same index in the original matrix) entries of the vector accompanying the parent supernode. The computation at any supernode in the tree can commence after the contributions from all its children have been collected. The algorithm terminates after the computation at the triangular supernode at the root of the elimination tree.

In a parallel implementation on p processors, a supernode at level l (see Figure 4.5) from the top is distributed among $p/2^l$ processors. The computation at a level greater than or equal to $\log p$ is performed sequentially on a single processor assigned to that subtree. However, the computation steps mentioned above must be performed in parallel on $p/2^l$ processors for a supernode with $0 \leq l < \log p$.

In [65], Heath and Romine describe efficient pipelined or wavefront algorithms for solving dense triangular systems with block-cyclic row-wise and column-wise partitioning of the triangular matrices. We use variations of the same algorithms on the dense trapezoidal supernodes at each of the parallel levels of the elimination tree. the number of processors among which a supernode is partitioned varies with its level in the tree, but the same basic parallel algorithm is used for each supernode. Figure 4.16(a) shows hypothetical forward elimination on a supernode with an unlimited number of processors on an EREW-PRAM. From this figure, it is clear that, due to data dependencies, at a time only $max(t, n/2)$ processors can remain busy. Since the computation proceeds along a diagonal wave from the upper-left to the lower-right corner of the supernode, at any given time, only one block per row and one element per column is active. From this observation, it can be shown that an efficient parallel algorithm (an algorithm capable of delivering a speedup of $\Theta(p)$ using p processors) for forward elimination must employ a one-dimensional row-wise or column-wise partitioning of the supernode so that all processor can be busy at all times (or most of the time). From a practical perspective, we chose a row-wise block-cyclic partitioning because $n \geq t$ and a more uniform partitioning with reasonable block sizes can be obtained if the rows are partitioned. Figures 4.16(b) and (c) illustrate two variations of the pipelined forward elimination with block-cyclic row-wise partitioning of the supernode. Each box in the figure can be regarded as a $b \times b$ square block of the supernode (note that the diagonal boxes represent lower triangular blocks). In the column-priority algorithm, the computation along a column of the supernode is finished before a new column is started. In the row-priority algorithm, the computation along a row is finished before a new row is started.

4.8.2 Backward Substitution

The algorithm for parallel backward substitution is very similar. Since an upper triangular system is being solved, the supernodes are organized as dense trapezoidal matrices of height t and width n



(a) Pipelined computation on an EREW-PRAM with unlimited number of processors.

(b) Row-priority pipelined computation with cyclic mapping of rows onto four processors.

(c) Column-priority pipelined computation with cyclic mapping of rows onto four processors.

Figure 4.16: Progression of computation consistent with data dependencies in parallel pipelined forward elimination in a hypothetical supernode of the lower-triangular factor matrix L. The number in each box of L represents the time step in which the corresponding element of L is used in the computation. Communication delays are ignored in this figure and the computation time for each box is assumed to be identical. In parts (b) and (c), the supernode is partitioned among the processors using a cyclic mapping. A block-cyclic mapping can be visualized by regarding each box as a $b \times b$ block (the diagonal boxes will represent triangular blocks).

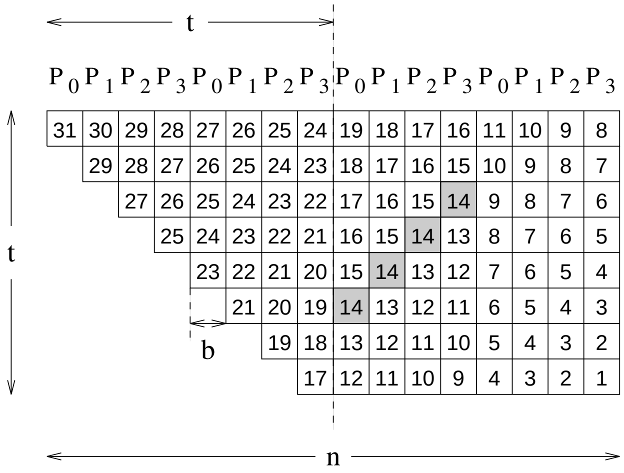


Figure 4.17: Column-priority pipelined backward substitution on a hypothetical supernode distributed among 4 processors using column-wise cyclic mapping.

($n \geq t$) and a column-wise block-cyclic partitioning is used at the top $\log p$ levels of the elimination tree. In backward substitution, the computation starts at the root of the elimination tree and progresses down to the leaves. First, the entries from the right-hand side vector with the same indices as the nodes of a supernode are collected in the first t contiguous locations of a vector of length n. The remaining $n - t$ entries of this vector are copied from the entries with same indices in the vector accompanying the parent supernode. This step is not performed for the root supernode, which does not have a parent and for which $n = t$. The computation at a supernode consists of two steps and can proceed only after the computation at its parent supernode is finished. The first computation step is to subtract the product of the $t \times (n - t)$ rectangular portion of the supernode with the lower part of the vector of size $n - t$ from the upper part of the vector of size t. The second step is to solve the triangular system formed by the $t \times t$ triangle of the trapezoidal supernode and the upper part of the vector of size t. Just like forward elimination, these steps are carried out serially for supernodes at levels greater than or equal to $\log p$ in the elimination tree. For the supernodes at levels 0 through $\log p - 1$, the computation is performed using a pipelined parallel algorithm. Figure 4.17 illustrates the pipelined algorithm on four processors with column-wise cyclic mapping. The algorithm with a block-cyclic mapping can be visualized by regarding each box in Figure 4.17 as a square block (the blocks along the diagonal of the trapezoid are triangular) of size $b \times b$.

In both forward and backward triangular solvers described in this section, if the system needs to be solved with respect to more than one, say m, right-hand sides, then the vectors of length n are replaced by rectangular $n \times m$ matrices. The overall algorithms remain identical except that all vector operations are replaced by the corresponding matrix operations, the size of the matrix being the length of the vector times the number of vectors.

4.8.3 Analysis

In this section we derive expressions for the communication overheads and analyze the scalability of the sparse supernodal multifrontal triangular solvers described in Section 4.8.1. We will present the analysis for the forward elimination phase only; however, the reader can verify that the expressions for the communication overhead are identical for backward substitution.

Communication Overheads

It is difficult to derive analytical expressions for general sparse matrices because the location and amount of fill-in, and hence, the distribution and the number if non-zeros in L, is a function of the the number and position of nonzeros in the original matrix. Therefore, we will focus on the problems in which the original matrix is the adjacency matrix of a two- or three-dimensional graph in which the degree of each node is bounded by a constant. These classes of matrices include the coefficient matrices generated in all two- and three-dimensional finite element and finite difference problems. We also assume as that a nested-dissection based fill-reducing ordering is used, which results in an almost balanced elimination tree. The subtree-to-subcube assignment of the elimination tree to the processors relies heavily on a balanced tree. Although there are bound to be overheads due to unequal distribution of work, it is not possible to model such overheads analytically because the extent of such overheads is data-dependent. From our experience with actual implementations of parallel triangular solvers as well as parallel factorization codes [53], we have observed that such overheads are usually not excessive. Moreover, the overhead due to load imbalance in most practical cases tends to saturate at 32 to 64 processors for most problems and does not continue to increase as the number of processors are increased. In the remainder of this section, we will concentrate on overheads due to inter-processor communication only.

Consider the column-priority pipelined algorithm for forward elimination shown in Figure 4.16(c). Let b be the block size in the block-cyclic mapping. A piece of the vector of size b is transferred from a processor to its neighbor in each step of the algorithm until the computation moves below the upper triangular part of the trapezoidal supernode. If a supernode is distributed among q processors, then during the entire computation at a supernode, $q + t/b - 1$ such communication steps are performed; $q - 1$ steps are required for the computation to reach the last processor in the pipeline and t/b steps to pass the entire data (of length t) through the pipeline. Thus, the total communication time is proportional to $b(q-1)+t$, which is $\Theta(q)+\Theta(t)$, assuming that b is a constant.

Besides the communication involved in the pipelined processing over a supernode, there is some more communication involved in collecting the contributions of the vectors associated with the children of a supernode into the vector associated with the parent supernode. If the two child supernodes are each distributed among q processors, then this communication is equivalent to an all-to-all personalized communication [81] among $2q$ processors with a data size of roughly t/q on each processor. This communication can be accomplished in time proportional to t/q, which is asymptotically smaller than the $\Theta(q) + \Theta(t)$ time spent during the pipelined computation phase at the child supernodes. Therefore, in the remainder of this section, we will ignore the communication required to transfer the contributions of the vector across the supernodes at different levels of the elimination tree.

So far we have established that a time proportional to $b(q-1)+t$ (or roughly, $bq+t$) is spent while processing an $n \times t$ trapezoidal supernode on q processors with a block-cyclic mapping that uses blocks of size b. We can now derive an expression for the overall communication time for the entire parallel forward elimination process by substituting for q and t in the expression $bq + t$ for a supernode at level l and summing up the resulting expression over all levels.

Let us first consider a sparse linear system of N equations resulting from a two-dimensional finite element problem being solved on p processors. As a result of using the subtree-to-subcube mapping, q at a level l is $p/2^l$. If a nested-dissection based ordering scheme is used to number the nodes of the graph corresponding to the coefficient matrix, then the number of nodes t in a supernode at level l is $\alpha\sqrt{N/2^l}$, where α is a small constant [91, 90, 45, 53]. Thus the overall communication time is proportional to $\Sigma_{l=0}^{\log p-1}(bp/2^l) + \Sigma_{l=0}^{\log p-1}(\alpha\sqrt{N/2^l})$, which is $\Theta(p) + \Theta(\sqrt{N})$.

The overall computation is proportional to the number of non-zeros in L, which is $\Theta(N \log N)$ for an $N \times N$ sparse coefficient matrix resulting from a two-dimensional finite element problem [45] with a nested-dissection based ordering. Assuming that the computation is divided uniformly among the processors, each processor spends $\Theta((N \log N)/p)$ time in computation. Hence, the parallel runtime for forward elimination algorithm described in Section 4.8.1 is as follows[2]:

$$T_P = \Theta(\frac{N \log N}{p}) + \Theta(\sqrt{N}) + \Theta(p). \tag{4.1}$$

If the underlying graph corresponding to the coefficient matrix is a three-dimensional constant-degree graph (as is the case in three-dimensional finite element and finite difference problems), then the value of t at level l is roughly $\alpha(N/2^l)^{2/3}$, where α is a small constant [45, 53]. The value of q at level l is $p/2^l$. Thus, the total communication time is proportional to $\Sigma_{l=0}^{\log p-1}(bp/2^l)$ + $\Sigma_{l=0}^{\log p-1}(\alpha(N/2^l)^{2/3}$, which is $\Theta(p) + \Theta(N^{2/3})$. Assuming that the overall computation of $\Theta(N^{4/3})$ [45] is uniformly distributed among the processors, the parallel runtime is given by the following equation:

$$T_P = \Theta(\frac{N^{4/3}}{p}) + \Theta(N^{2/3}) + \Theta(p). \tag{4.2}$$

If more than one (say m) right-hand side vectors are present in the system, then each term in Equations 4.1 and 4.2 is multiplied with m.

Scalability Analysis

Recall that for a triangular system resulting from the factorization of an $N \times N$ sparse matrix corresponding to a two-dimensional constant-degree graph,

$$W = \Theta(N \log N). \tag{4.3}$$

If we assume that only the last two terms in Equation 4.1 contribute to the overhead, then from the relation $T_o = pT_P - T_S$, it is easy to see that

$$T_o = \Theta(p^2) + \Theta(p\sqrt{N}). \tag{4.4}$$

[2] Depending on the way the pipelining is implemented for processing a supernode, there may be $\Theta(t)$ and/or $\Theta(q)$ steps of pipeline delay (one step performs $2b^2$ operations) at each supernode. However, the aggregate of such terms does not exceed the asymptotic communication complexity of $\Theta(p) + \Theta(\sqrt{N})$

Balancing W against the first term in the expression for T_o yields the following (see Appendix D for details):

$$W \propto p^2, \tag{4.5}$$

and balancing it against the second term in the expression for T_o yields

$$W \propto \frac{p^2}{\log p}. \tag{4.6}$$

Since p^2 is the dominant term in the two isoefficiency expressions, the overall rate at which the problem size must increase with the number of processors to maintain a fixed efficiency is $\Theta(p^2)$, as given by Equation 4.5.

For a triangular system resulting from the factorization of an $N \times N$ sparse matrix corresponding to a three-dimensional constant-degree graph,

$$W = \Theta(N^{4/3}) \tag{4.7}$$

and

$$T_o = \Theta(p^2) + \Theta(pN^{2/3}). \tag{4.8}$$

Balancing W against each term in the expression for T_o yields the following isoefficiency function (see Chapter 2 for details):

$$W \propto p^2. \tag{4.9}$$

In this section, we have shown that the isoefficiency function for solving sparse triangular systems resulting from the factorization of a wide class of sparse matrices is $\Theta(p^2)$. In [53], we described parallel algorithms for sparse Cholesky factorization of the same class of matrices with an isoefficiency function of $\Theta(p^{1.5})$, which is better than the $\Theta(p^2)$ isoefficiency function of the corresponding triangular solver. However, the amount of computation involved in numerical factorization is much higher than that in a triangular solver. Therefore, as we experimentally demonstrate in Section 4.8.5, despite being less efficient than parallel numerical factorization, triangular solvers can still be speeded up enough n parallel so as to claim only a fraction of the factorization time on the same number of processors.

Comparison with the Scalability of Dense Triangular Solvers

The communication time for solving a dense $N \times N$ triangular system using a row-wise block-cyclic mapping onto p processors with block size b is proportional to $b(p-1)+N$, which is $\Theta(p)+\Theta(N)$. The problem size W is $\Theta(N^2)$ and the total communication overhead T_o is $\Theta(p^2)+\Theta(Np)$ (note that the total communication overhead or the overhead function is the product of p and the communication time). It is easy to see W must grow in proportion to p^2 in order to satisfy the relation $W \propto T_o$ for maintaining a constant efficiency. Thus, the isoefficiency function of a parallel dense triangular solver is $\Theta(p^2)$, indicating that the parallel algorithms described in Section 4.8.1 for sparse forward and backward triangular solvers are asymptotically as scalable as their dense counterparts. From this observation, we can argue that the sparse algorithms, at least in the case of matrices associated with three-dimensional constant-degree graphs are optimal. The topmost supernode in such a matrix is an $N^{2/3} \times N^{2/3}$ dense triangle. Solving a triangular system corresponding to this supernode involves asymptotically a computation of the same complexity as solving the entire sparse triangular system. Thus, the overall scalability cannot be better than that of solving the topmost $N^{2/3} \times N^{2/3}$ dense triangular system in parallel, which is $\Theta(p^2)$.

4.8.4 Data Distribution for Efficient Triangular Solution

In Section 4.8.1 and in [81], we discuss that in order to implement the steps of dense triangular solution efficiently, the matrix must be partitioned among the processors along the rows or along the columns. However, as we have shown in [53], the dense supernodes must be partitioned along both dimensions for the numerical factorization phase to be efficient. The table in Figure 4.18 shows the communication overheads and the isoefficiency functions for parallel dense and sparse factorization and triangular solution using one- and two-dimensional partitioning schemes. The most efficient scheme in each category is denoted by a shaded box in the table. The last column of the table shows the overall isoefficiency function of the combination of factorization and triangular solvers. Note that the triangular solvers are unscalable by themselves if the dense supernodal blocks of the triangular factor are partitioned in two dimensions. However, the asymptotic communication overhead of the unscalable formulation of the triangular solvers does not exceed the communication overhead of the factorization process. As a result, the overall isoefficiency function is dominated

Matrix Type	Partitioning	Factorization, no Pivoting		Forward/Backward Solution		Overall Isoefficiency Function
		Communication Overhead	Isoefficiency Function	Communication Overhead	Isoefficiency Function	
Dense	One-dimensional	$O(N^2 p)$	$O(p^3)$	$O(p^2) + O(Np)$	$O(p^2)$	$O(p^3)$
	Two-dimensional	$O(N^2 p^{1/2})$	$O(p^{3/2})$	$O(N^2 p^{1/2})$	Unscalable	$O(p^{3/2})$
Sparse (resulting from 2-D graphs)	One-dimensional with subtree-subcube	$O(Np)$	$O(p^3)$	$O(p^2) + O(N^{1/2} p)$	$O(p^2)$	$O(p^3)$
	Two-dimensional with subtree-subcube	$O(Np^{1/2})$	$O(p^{3/2})$	$O(Np^{1/2})$	Unscalable	$O(p^{3/2})$
Sparse (resulting from 3-D graphs)	One-dimensional with subtree-subcube	$O(N^{4/3} p)$	$O(p^3)$	$O(p^2) + O(N^{2/3} p)$	$O(p^2)$	$O(p^3)$
	Two-dimensional with subtree-subcube	$O(N^{4/3} p^{1/2})$	$O(p^{3/2})$	$O(N^{4/3} p^{1/2})$	Unscalable	$O(p^{3/2})$

Figure 4.18: A table of communication overheads and isoefficiency functions for sparse factorization and triangular solution with different partitioning schemes.

by that of factorization. Hence, for a solving a system with a single right-hand side vectors (or a small constant number of them), the unscalability of the triangular solvers should not be of much concern. However, if solutions with respect to a number of right-hand side vectors are required, then for both the factorization and triangular solution to be efficient together, each supernode must be redistributed among the processors that share it. This redistribution must convert the original two-dimensional block-cyclic partitioning into a one-dimensional block-cyclic partitioning. In this section we show that the time spent in this redistribution is not asymptotically higher than the parallel run time of the triangular solvers.

Consider an $n \times t$ dense supernode mapped onto a $\sqrt{q} \times \sqrt{q}$ logical grid of processors using a two dimensional partitioning. As shown in Figure 4.19, the redistribution is equivalent to a transposition of each $(n/\sqrt{q}) \times t$ rectangular block of the supernode among the $\sqrt{q}$ processor on which it is horizontally partitioned. This is an all-to-all personalized communication operation [81] among $\sqrt{q}$ processors with each processor holding nt/q words of data. Although Figure 4.19 illustrates redistribution with a plain block partitioning, both the procedure and the cost of redistribution are the same with block-cyclic partitioning as well. The communication time for this all-to-all personalized

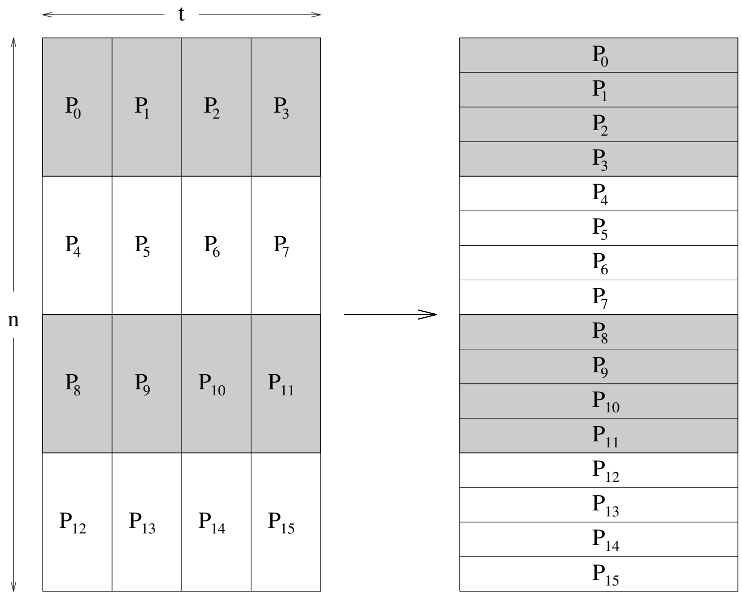


Figure 4.19: Converting the two-dimensional partitioning of a supernode into one-dimensional partitioning.

operation is $\Theta(nt/q)$ [81]. Note that for solving a triangular system with a single right-hand side, each processor performs $\Theta(nt/q)$ computation while processing an $n \times t$ supernode on q processors. Thus, the total overhead for redistribution is of the same order as the time spent by the parallel triangular solver while working on the top $\log p$ levels of the elimination tree, which is less than the total parallel runtime. The actual ratio of the redistribution time and the parallel triangular solution time will depend on the relative communication and computation speeds of the parallel computer being used. In Section 4.8.5, we show that on a Cray T3D, the redistribution time is at most 0.9 times (0.6 times on an average) the parallel triangular solution time with a single right-hand side vector. If more than one right-hand side vectors are used, then the cost of redistribution can be amortized further because the redistribution needs to be done only once.

4.8.5 Experimental Results

We implemented the algorithms described in Section 4.8.1 and integrated them with our sparse Cholesky factorization algorithms described in Section 4.4. Table 4.4[3] and Figure 4.20 show the performance of the parallel triangular solvers on a Cray T3D.

In Table 4.4, we show the time in seconds and the performance in MFLOPS on a selected number of processors for five test matrices with the number of right-hand side vectors varying from 1 to 30. To facilitate a comparison of the times for various phases of the solution processes, the table also contains the factorization run time and MFLOPS, as well as the time to redistribute the factor matrix to convert the supernodes from a two-dimensional to a one-dimensional partitioning among the processors. As shown in Table 4.4, for a single right-hand side vector, the highest performance achieved on a 256-processor Cray T3D is approximately 435 MFLOPS, which increases to over 3 GFLOPS if a solution with 30 right-hand side vectors is obtained. Comparing with the single-processor performance for BCSSTK15, this represents roughly 50- and 100-fold enhancement in performance on 256 processors for 1 and 30 right-hand side vectors, respectively. There are two other important observations to be made from the table in Table 4.4. First, despite a highly scalable implementation of sparse Cholesky factorization, parallelization of the relatively less scalable triangular solvers can speed them enough so that their runtime is still a small fraction of the factorization time. Second, although efficient implementations of factorization and triangular solvers use different data partitioning schemes, the redistribution of the data, on an average, takes much less time than the triangular solvers for a single right-hand side vector on the T3D.

Figure 4.20 shows the plots of MFLOPS versus number of processors of the Cray T3D for triangular solutions with different number of right-hand side vectors. The curves for these four test matrices show that both the overall performance and the speedups are much higher if a block of right-hand side vectors is available for solution. The use of multiple right-hand side vectors enhances the single processor performance due to effective use of BLAS-3 routines. It also improves speedups because the cost of certain index computations required in the parallel implementation can be

---

[3] The factorization megaflops, operation count, and number of nonzeros are different for some matrices between Tables 4.3 and 4.4 because Table 4.3 gives the results of an implementation that modifies the subtree-subcube mapping to reduce load imbalance [53]. On the other hand, Table 4.4 [57] implements a strict subtree-subcube mapping and also uses somewhat different parameters in spectral nested dissection to order the matrices.

BCSSTK15: N = 3948; Factorization Opcount = 85.5 Million; Nonzeros in factor = 0.49 Million								
	Factorization time = 2.46 sec.	NRHS	1	2	5	10	20	30
p = 1	Factorization MFLOPS = 34.8	FBsolve time (sec.)	.228	.284	.452	.740	1.33	1.92
	Time to redistribute L = 0.0 sec.	FBsolve MFOLPS	8.6	13.7	21.5	26.5	29.4	30.0
	Factorization time = .107 sec.	NRHS	1	2	5	10	20	30
p = 64	Factorization MFLOPS = 800	FBsolve time (sec.)	.024	.027	.034	.048	.074	.100
	Time to redistribute L = .009 sec.	FBsolve MFOLPS	81.5	145	285	405	527	583
BCSSTK31: N = 35588; Factorization Opcount = 2791 Million; Nonzeros in factor = 6.64 Million								
	Factorization time = 5.59 sec.	NRHS	1	2	5	10	20	30
p = 16	Factorization MFLOPS = 499	FBsolve time (sec.)	.227	.274	.398	.614	1.05	1.51
	Time to redistribute L = .071 sec.	FBsolve MFOLPS	115	194	330	427	498	523
	Factorization time = .721 sec.	NRHS	1	2	5	10	20	30
p = 256	Factorization MFLOPS = 3871	FBsolve time (sec.)	.073	.082	.107	.152	.242	.334
	Time to redistribute L = .035 sec.	FBsolve MFOLPS	363	646	1240	1738	2199	2385
HSCT21954: N = 21954; Factorization Opcount = 2822 Million; Nonzeros in factor = 5.84 Million								
	Factorization time = 2.48 sec.	NRHS	1	2	5	10	20	30
p = 32	Factorization MFLOPS = 1138	FBsolve time (sec.)	.113	.133	.189	.284	.472	.672
	Time to redistribute L = .10 sec.	FBsolve MFOLPS	203	347	609	809	973	1025
	Factorization time = .619 sec.	NRHS	1	2	5	10	20	30
p = 256	Factorization MFLOPS = 4560	FBsolve time (sec.)	.091	.099	.122	.161	.234	.312
	Time to redistribute L = .067 sec.	FBsolve MFOLPS	255	471	953	1452	1991	2244
CUBE35: N = 42875; Factorization Opcount = 7912 Million; Nonzeros in factor = 9.95 Million								
	Factorization time = 7.528 sec.	NRHS	1	2	5	10	20	30
p = 32	Factorization MFLOPS = 1051	FBsolve time (sec.)	.245	.296	.436	.681	1.21	1.72
	Time to redistribute L = .13 sec.	FBsolve MFOLPS	162	269	456	583	660	693
	Factorization time = 1.43 sec.	NRHS	1	2	5	10	20	30
p = 256	Factorization MFLOPS = 5527	FBsolve time (sec.)	.108	.120	.154	.216	.340	.468
	Time to redistribute L = .08 sec.	FBsolve MFOLPS	369	665	1289	1838	2345	2548
COPTER2: N = 55476; Factorization Opcount = 8905 Million; Nonzeros in factor = 12.77 Million								
	Factorization time = 5.764 sec.	NRHS	1	2	5	10	20	30
p = 64	Factorization MFLOPS = 1545	FBsolve time (sec.)	.186	.220	.320	.492	.832	1.10
	Time to redistribute L = .11 sec.	FBsolve MFOLPS	274	463	795	1036	1226	1277
	Factorization time = 1.846 sec.	NRHS	1	2	5	10	20	30
p = 256	Factorization MFLOPS = 4825	FBsolve time (sec.)	.117	.130	.167	.232	.364	.500
	Time to redistribute L = .07 sec.	FBsolve MFOLPS	434	785	1526	2195	2805	3053

Table 4.4: A table of experimental results for sparse forward and backward substitution on a Cray T3D (from [57]). In the above table, “NRHS” denotes the number of right-hand side vectors, “FBsolve time” denotes the total time spent in both the forward and the backward solvers, and “FBsolve MFLOPS” denotes the average performance of the solvers in million floating point operations per second. See footnote in the text.

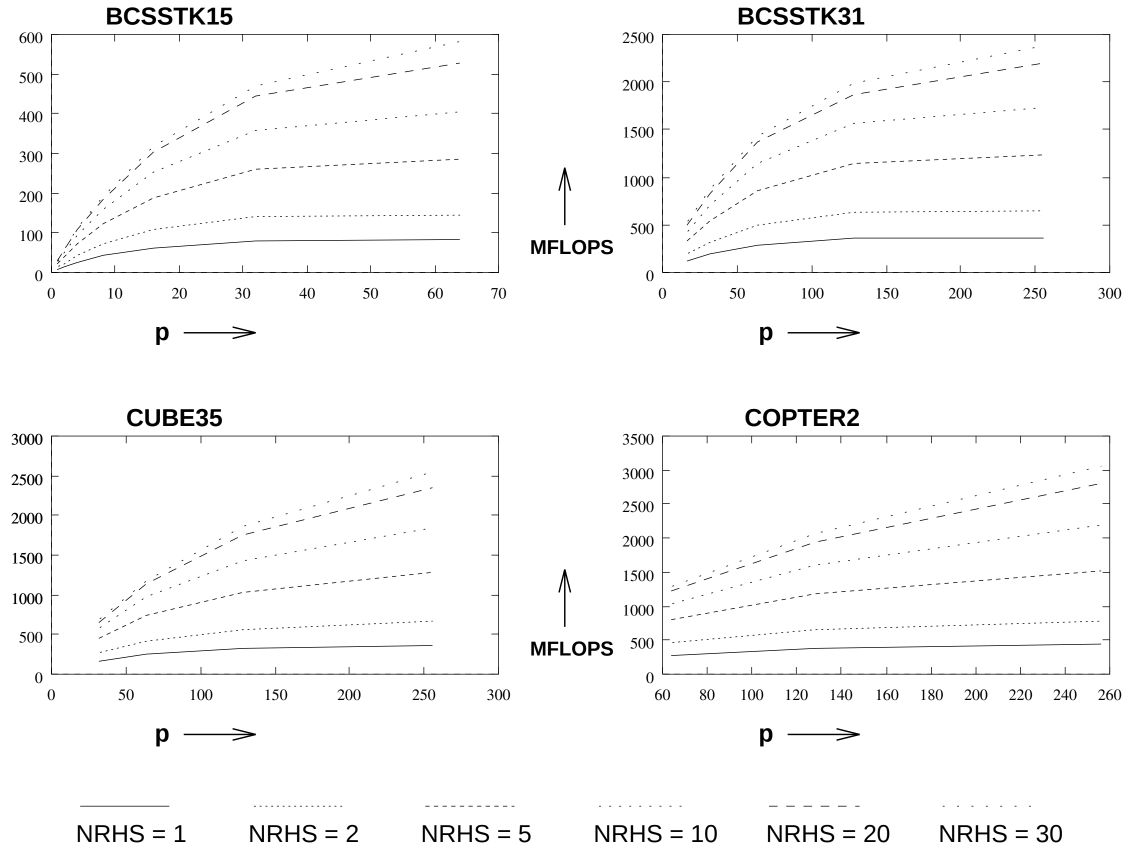


Figure 4.20: Performance versus number of processors on a Cray T3D for parallel sparse triangular solutions with different number of right-hand side vectors (from [57]).

amortized over all the right-hand side vectors.

4.9 Parallel Symbolic Factorization

The symbolic factorization step determines the structure of the lower triangular factor matrix L and sets up the data structures in which to store the original SPD matrix A and the nonzero entries of L to be created during numerical factorization. Symbolic factorization is the least time consuming of all the four steps involved in the direct solution of a sparse linear system. However, it is important to parallelize this step for two important reasons. First, the data (i.e., the original matrix) is already distributed among the processors before the symbolic factorization phase [76] and it would be expensive (very often impossible too due to memory constraints) to gather the data for serial symbolic factorization and then redistribute it. Second, since the other three phases are quite scalable in parallel, symbolic factorization will become a serial bottleneck despite its small run time if it is left unparallelized.

In this section we briefly describe a method for performing symbolic factorization in parallel provided that the original matrix is already distributed among the processors according to the subtree-to-subcube mapping. We then analyze the asymptotic communication overhead involved in the process for sparse matrices arising from two- and three- dimensional constant-degree graphs.

4.9.1 The Serial Algorithm

Figure 4.21 outlines a recursive serial algorithm for symbolic factorization. Just like numerical factorization and triangular solver algorithms, symbolic factorization too is guided by the elimination tree. Note that the algorithm of Figure 4.21 requires the knowledge of the elimination tree. Elimination tree generation has traditionally been coupled with symbolic factorization [93, 63]. In this chapter, we are relying on nested-dissection based ordering strategies that can be computed in parallel and also render the remaining phases of the solution process amenable to parallelization [76]. The elimination tree can be constructed easily (and cheaply) while performing a nested-dissection based ordering. Assume that a bisection algorithm is being used for ordering; i.e., the separator of a subgraph of nested dissection divides the subgraph into two disconnected components. In such an ordering the nodes of a separator are numbered consecutively. These separator nodes are also the consecutive nodes of a relaxed supernode (as defined in Section 4.4) in the elimination tree. The separator node with the smallest index is the one that has two children in the elimination tree (other separator nodes have only one child). These two child nodes are the nodes with the highest index in each of the two disconnected components. This process, when carried out recursively, determines the entire elimination tree.

Assuming that the elimination tree is available, the algorithm in Figure 4.21 determines the structure of each column of L. A call to Symbolic(k) computes $Struct_i$ for all $i \le k$. clearly, if k is the root of the elimination tree, Symbolic(k) performs the entire symbolic factorization. At the end of symbolic factorization, $Struct_k$ is the set of indices $\{j, l_{j,k} \neq 0\}$.

/*

A is the sparse $N \times N$ symmetric matrix to be symbolically factored.

$A = (a_{i,j})$, where $0 \leq i, j < N$.

*/

begin function Symbolic(k)

    $Struct_k := \{j, a_{j,k} \neq 0, j \geq k\}$;

    for all i such that Parent(i) = k in the elimination tree, **do**

    begin

        Symbolic(i);

        $Struct_k := Struct_k \cup Struct_i - \{i\}$;

    end

end function Symbolic.

Figure 4.21: An elimination-tree guided recursive algorithm for symbolic factorization

4.9.2 Parallel Formulation

A parallel formulation of symbolic factorization must generate data structures suitable for parallel numerical factorization. Refer to the parallel multifrontal algorithm described in Section 4.4. The role of symbolic factorization in the context of this algorithm is to generate the structure or the index set associated with each frontal matrix. Since we are dealing with symmetric matrices, simply generating the column (or row) indices is sufficient to carry out sequential numerical factorization. However, the frontal matrices are partitioned among processors in two dimensions in our parallel numerical factorization algorithm, each processor must have a list of both row and column indices of its portion of each frontal matrix. A simple modification of the algorithm shown in Figure 4.21 suffices to achieve this.

Recall from Section 4.4 that the subtrees of the elimination tree rooted at level $\log p$ is processed sequentially on a single processor. The algorithm of Figure 4.21 can be applied to this subtree without modification. If k is the root of this subtree, then $Struct_k$ is copied into two data structures, $Local\_Col\_Struct_k$ and $Local\_Row\_Struct_k$. For processing levels 0 to $\log p - 1$, the flow of the parallel symbolic factorization algorithm is very similar to that of parallel Cholesky factorization. Instead of performing the factorization steps, a processor simply drops those indices from its local sets that correspond to the columns to be eliminated in that step; and, instead of performing the parallel extend-add steps, the row and column indices are exchanged and merged, and indices from A are added whenever required. The criterion for the selection of indices to keep or send to the corresponding processor of the partner subcube is identical to the one used in parallel numerical factorization. The total communication is much less than that in the extend-add steps because only the indices (instead of an entire update matrix) are communicated between pairs of processors.

4.9.3 Overhead and Scalability

From Section 4.5 we know that while processing level l of the elimination tree associated with a two-dimensional N-node constant-degree graph, an $\Theta(\sqrt{N}/2^l \times \sqrt{N}/2^l)$ frontal matrix is distributed on a $\sqrt{p}/2^l \times \sqrt{p}/2^l$ logical mesh of processors. During symbolic factorization, in order to determine the set of indices associated with this frontal matrix, each processor must store $\Theta(\sqrt{N/p})$ indices. Thus the time spent in communicating and merging the row and column index sets by each processors at each of the top l levels of the elimination tree is $\Theta(\sqrt{N/p})$. Since there are $\log p$ such levels, the total time spent in communication by each processor while performing symbolic factorization on the top $\log p$ levels of the elimination tree is $\Theta((\sqrt{N/p}) \log p)$.

The overhead function, T_o (see Section 2.1 for definition), in this case is $\Theta((\sqrt{Np}) \log p)$. The problem size, W, or the serial complexity of the symbolic factorization of sparse matrices associated with an N-node two-dimensional constant-degree graph is of the same order as the total number of nonzeros in the factor matrix L, which is $\Theta(N \log N)$. Using these expressions in Equation 2.4, the isoefficiency function can be derived to be $\Theta(p \log p)$.

In the case of three-dimensional constant-degree graphs, the number of indices per processor at level l are $\Theta(\sqrt{N^{4/3}/2^{4l}}/\sqrt{p/8^l})$, which is $\Theta(N^{2/3}/((\sqrt{2})^l\sqrt{p}))$. The overhead function is $\Theta(N^{2/3}\sqrt{p}\Sigma_{l=0}^{l=\log_8 p-1}(\sqrt{2})^{-l})$, which is $\Theta(N^{2/3}/\sqrt{p})$. When balanced against a problem size of $\Theta(N^{4/3})$, this overhead function yields an isoefficiency function of $\Theta(p)$.

Thus, in this section, we have shown that the already efficient symbolic factorization algorithm can be parallelized with low overheads and the parallel formulation is quite scalable.

4.10 A Complete Scalable Direct Solver for Sparse SPD Systems

Despite more inherent parallelism that dense linear systems, it has been a challenge to develop scalable parallel direct solvers for sparse linear systems. The process of obtaining a direct solution to a sparse system of linear equations usually consists of four phases: ordering, symbolic factorization, numerical factorization, and forward elimination and backward substitution. A scalable parallel solver for sparse linear systems must implement all these phases effectively in parallel. In [76], Karypis and Kumar present an efficient parallel algorithm for a nested-dissection based fill-reducing ordering for such sparse matrices. In Section 4.4, we introduced a highly scalable parallel algorithm for sparse Cholesky factorization, which is the most time consuming phase of solving a sparse linear system with s symmetric positive definite (SPD) matrix of coefficients. In Section 4.8, we have shown that the forward and backward substitution steps can obtain sufficient speedup on hundreds of processors so that numerical factorization still dominates the overall time taken to solve the system in parallel. In addition, we show that, although efficient implementations of factorization and triangular solvers use different data partitioning schemes, the time spent in redistributing the data to change the partitioning schemes is not a bottleneck when compared to the time spent in factorization and triangular solutions. In Section 4.9, we describe an algorithm for computing the symbolic factorization of a symmetric sparse matrix in parallel. We show that the parallel algorithms for all three phases can work in conjunction with each other, and for a wide class of sparse matrices, the combined overall asymptotic scalability (as measured by the isoefficiency metric) of these steps is the same as that of dense matrix factorization.

Figure 4.22 shows another way of appreciating the fact that the work presented in this chapter makes it possible to develop complete balanced parallel sparse direct solvers. This figure shows the parallel complexity, or the asymptotic complexity of the parallel run time of each phase of such a solver provided a sufficient number of processors. For example, in the case of two-dimensional constant node-degree graphs, the serial complexity of factorization is $\Theta(N^{1.5})$ and the isoefficiency function is $\Theta(p^{1.5})$. Thus, $W = \Theta(N^{1.5}) = \Theta(p^{1.5})$; i.e., a fixed efficiency can be maintained if N and p are of the same order. Thus up to $\Theta(N)$ processors can be used to potentially reduce the complexity of sparse Cholesky factorization from $\Theta(N^{1.5})$ to $\Theta(\sqrt{N})$. Similarly, we compute the best parallel complexities of the other phase for both two- and three-dimensional constant

Phase	2-D complexity	3-D complexity
Reordering:	$O(N)$	$O(N)$
Parallel Ordering:	$O(N^{1/2})$	$O(N^{1/2})$
Symbolic Factorization:	$O(N \log N)$	$O(N^{4/3})$
Parallel Symbolic Factorization:	$O(N^{1/2})$	$O(N^{2/3})$
Numerical Factorization:	$O(N^{3/2})$	$O(N^{2})$
Parallel Cholesky Factorization:	$O(N^{1/2})$	$O(N^{2/3})$
Triangular Solutions:	$O(N \log N)$	$O(N^{4/3})$
Parallel Triangular Solvers:	$O((N \log N)^{1/2})$	$O(N^{2/3})$

Figure 4.22: The serial and parallel complexities of the various phases of solving a sparse system of linear equations arising from two- and three-dimensional constant node-degree graphs.

node-degree graphs. For both these types of graphs, the number of processors required to yield the parallel complexity expressions given in Figure 4.22 for any phase does not exceed the number of processors used in the factorization phase. With a slight exception in the case of two-dimensional triangular solvers, Figure 4.22 shows that there will be no bottlenecks in a sparse direct solver that parallelizes all the four phases of the solution process. We therefore hope that the work presented in this chapter will enable efficient parallel solutions of a broad range of scientific computing problems. In Section 4.11, we show how similar algorithms can be developed for some other forms of sparse matrix factorization.

4.11 Application to Gaussian Elimination and QR Factorization

Although we have focussed on sparse Cholesky factorization in this chapter, the serial algorithm of Figure 4.3 can be generalized to Gaussian elimination without pivoting for nearly structurally symmetric sparse matrices [32] and for solving sparse linear least squares problems [99].

Gaussian elimination without pivoting is numerically stable for diagonally dominant matrices; i.e., the matrices in which the sum of the absolute values of all the non-diagonal elements of a row or a column is less than the absolute value of the diagonal element of the same row or column. If the matrix A to be factored is not perfectly symmetric in structure, it can be treated as a symmetric structure matrix by explicitly storing a zero element $a_{j,i}$ if $a_{i,j}$ is nonzero. The multifrontal factorization can then be guided by an elimination tree constructed from this symmetric pattern matrix. For such matrices, the dense Cholesky factorization steps of F^k (lines 8–12) are replaced by steps of dense Gaussian elimination. The parallel algorithm also works similarly, except that the frontal and update matrices are now full square matrices rather than triangular matrices as in the case of Cholesky factorization.

The least square problem (LSP) $min_x ||Ax - b||_2$ is commonly solved [50] by factoring the $m \times n$ matrix A ($m \geq n$) into the product QR, where Q is an $m \times n$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. Matstoms [99, 100] has recently developed a multifrontal algorithm for QR factorization for sparse A. Matstoms' approach avoids storing Q explicitly and is based on the observation that the matrix R is a Cholesky factor of the $n \times n$ symmetric positive definite matrix $A^T A$. The LSP is solved from the semi-normal equation $R^T Rx = A^T b$ with a few steps of iterative refinement. The elimination tree and symbolic factorization of $A^T A$ are used to guide the multifrontal QR factorization. The frontal matrices corresponding to the leaves of the elimination tree are derived from the original matrix as in the algorithm of Figure 4.3. The steps of dense QR factorization are then performed on these frontal matrices to obtain the corresponding update matrices. The process of forming the frontal matrix corresponding to a node higher up in the tree involves assembling the contributions from the update matrices of the node's children in the tree and nonzeros from the row and column corresponding to the node in the matrix A via extend-add operations. The difference between the multifrontal QR factorization and the algorithm of Figure 4.3 is that the frontal matrices can be square or rectangular and steps of dense QR factorization are performed in lines 8–12. Some parallel formulations of sparse QR factorization have been proposed in the literature [117, 128, 129]. These algorithms are based on a one-dimensional partitioning and their isoefficiency function has a lower bound of $\Omega(p^3)$. The parallel multifrontal algorithm described in this chapter can be modified to along the lines of [99, 100] to develop a more scalable parallel formulation of sparse QR factorization.

Chapter 5

CONCLUDING REMARKS AND FUTURE WORK

In this dissertation, we have presented the results of our research on scalability analysis of parallel algorithms for a variety of numeric computations and on the design of some new parallel algorithms for sparse matrix computations that are more scalable than the previously known algorithms for solving the same problems.

We have surveyed a number of techniques and formalisms that have been developed for studying the scalability issues, and discuss their interrelationships. It is clear that a single metric is not sufficient to analyze parallel systems. Different metrics are useful depending on whether the number of processors, the run time, the problem size, or the efficiency is kept constant. However, we show some interesting relationships between the technique of isoefficiency analysis and many other methods for scalability analysis. For example, we show that instances of a problem with increasing size can be solved in a constant parallel run time by employing an increasing number of processors if and only if the isoefficiency function of the parallel system is $\Theta(p)$. We show that for a wide class of parallel systems, the relationship between the problem size and the number of processors that minimize the run time for that problem size is given by an isoefficiency curve.

The analytical power of isoefficiency analysis is demonstrated in Chapter 3. We have used this technique to analyze a variety of algorithms that have applications in scientific computing, and have often derived interesting conclusions. In Chapter 4, we have demonstrated that the this method of analysis can also guide the development of better parallel algorithms. An important feature of isoefficiency analysis is that in a single expression, it succinctly captures the effects of characteristics of a parallel algorithm as well as the parallel architecture on which it is implemented. In can help identify the scalability bottlenecks in a parallel system and thus aid in eliminating or reducing the impact of these bottlenecks. By analyzing the known parallel formulations of sparse and dense matrix factorization, we learned that a two dimensional partitioning of the matrix reduces the isoefficiency function by a factor of $p^{1.5}$. Also, a smart assignment of processors to process different parts of the elimination tree improves the isoefficiency function by a factor of $(\log p)^3$ over a simple assignment. These observations lead us to attempt to devise a parallel formulation of sparse matrix formulation that includes the benefits of both kinds of optimizations. Our analysis had shown that achieving this would make parallel factorization of a wide class of sparse matrices optimally scalable.

Having developed an optimally scalable parallel algorithm for the most time consuming step of numerical factorization in solving a sparse system of linear equations, we looked at symbolic factorization and solution of sparse triangular systems resulting from the factorization of the coefficient matrix. In Section 4.8, we have shown that although individually less scalable than numerical factorization, the symbolic factorization and forward and backward substitution steps do not effect the overall asymptotic scalability of a parallel sparse direct solver. In addition, we show that, although efficient implementations of factorization and triangular solvers use different data partitioning schemes, the time spent in redistributing the data to change the partitioning schemes is not a bottleneck when compared to the time spent in factorization and triangular solutions. Along with some recently developed parallel ordering algorithms, the algorithms presented in this thesis make it possible to develop complete scalable parallel direct solvers for sparse linear systems.

Our implementations of sparse matrix factorization described in Chapter 4, in their current form apply to symmetric positive definite matrices. However, the algorithm is applicable to a somewhat broader class of problems. With minor modifications, it can be used to factor those symmetric pattern sparse matrices for which the ordering can be computed prior to factorization. This can be done for matrices whose numerically stable factorization does not require pivoting (row and/or column interchanges to ensure that the diagonal element of a pivot column is not much smaller than the elements that it divides). SPD matrices constitute a subclass of this class of matrices. Another subclass contains matrices that are diagonally dominant but are not positive definite. An $N \times N$ sparse matrix A is diagonally dominant if and only if $|A[i, i]| > 2\Sigma_{j=0}^{N-1}|A[i, j]|$ and $|A[i, i]| > 2\Sigma_{j=0}^{N-1}|A[j, i]|$ for $0 \leq i < N$. An LU decomposition of diagonally dominant matrices using Gaussian elimination is numerically stable in the absence of pivoting. It is quite straightforward to adapt the algorithm of Section 4.4 for such matrices. The only modification required in the algorithm of Section 4.4 is to perform steps of Gaussian elimination instead of

Cholesky factorization on the frontal matrices.

There is a class of sparse matrices that require pivoting in order to ensure numerical stability. It would be interesting to investigate if the algorithms discussed in this thesis can be extended to support limited pivoting that would provide numerical stability in such problems without incurring excessive overheads. Consider the class of symmetric matrices for which stable factorization requires pivoting. In the multifrontal algorithm, processing an m-node supernode of the elimination tree is equivalent to a rank-m update. If all the m pivots can be found by permuting these m rows and columns among themselves, then incorporating pivoting in the algorithm of Section 4.4 is not too difficult. As described in Section 4.4, a level-l supernode is processed by a group of $p/2^l$ processors. The information regarding any change in the indices due to pivoting at the level-l supernode needs to be shared among only $p/2^l$ processors to affect the exchange and to update the indices in the already factored part of the matrix. If renumbering nodes within a supernode is not sufficient for numerical stability, then an interchange may be required between a level-l supernode and another node from its parent supernode at level-$(l-1)$. This interchange requires communication among the $p/2^{l-1}$ processors that share the level-$(l-1)$ supernode. In particular, it introduces extra synchronization points, as the computation at the remainder of the children supernodes has to finish (and the update matrices have to be added into the frontal matrix of the parent node) before pivot exchange can be performed. An additional complication accompanied with renumbering the nodes during numerical factorization is that the exact location and amount of fill-in cannot be determined apriori.

An alternative to Gaussian elimination with pivoting is QR factorization. QR factorization is numerically stable without pivoting; however, it requires more computation than Gaussian elimination or Cholesky factorization. As discussed in Section 4.11, our parallel sparse Cholesky algorithm can be modified to devise a scalable parallel formulation of QR factorization. It would be worthwhile to compare Gaussian elimination with pivoting (whose scalability will be limited by the amount of pivoting required) or QR factorization (which is costlier, but more scalable) for sparse matrix factorization on large parallel computers.

Some possible interesting applications of scalable parallel direct solvers could be in developing parallel hybrid and multigrid solvers and in preconditioning parallel iterative solvers. For example, in a finite-element application, a direct solution over a coarse mesh can often be effective in finding a fast iterative solution over a finer mesh. Besides the applications requiring the solution of a linear system of equations, there are numerous other interesting scientific computing problems that can benefit significantly from scalable parallel algorithms; some examples are, N-body simulations, solving integral equations, singular value decomposition, etc. It is important to analyze these problems, determine lower-bounds on the scalability of solving these problems on parallel computers, and to attempt to develop parallel algorithms matching or close to the lower bounds.

BIBLIOGRAPHY

[1] S. Abraham and K. Padmanabhan. Performance of multicomputer networks under pin-out constraints. *Journal of Parallel and Distributed Computing*, pages 237–248, July 1991.

[2] Anant Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2:398–412, October 1991.

[3] Alok Aggarwal, Ashok K. Chandra, and Mark Snir. Communication complexity of PRAMs. Technical Report RC 14998 (No. 64644), IBM T. J. Watson Research Center, Yorktown Heights, NY, Yorktown Heights, NY, 1989.

[4] A. V. Aho, John E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[5] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[6] Edward Anderson. Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations. Technical Report 805, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1988.

[7] Cleve Ashcraft. The domain/segment partition for the factorization of sparse symmetric positive definite matrices. Technical Report ECA-TR-148, Boeing Computer Services, Seattle, WA, 1990.

[8] Cleve Ashcraft. The fan-both family of column-based distributed cholesky factorization algorithms. In A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Graph Theory and Sparse Matrix Computations*. Springer-Verlag, New York, NY, 1993.

[9] Cleve Ashcraft, S. C. Eisenstat, and J. W.-H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal on Scientific and Statistical Computing*, 11:593–599, 1990.

[10] Cleve Ashcraft, S. C. Eisenstat, J. W.-H. Liu, and A. H. Sherman. A comparison of three column based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Yale University, New Haven, CT, 1990. Also appears in *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1991.

[11] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan. A parallel FFT on an MIMD machine. *Parallel Computing*, 15:61–74, 1990.

[12] Cevdet Aykanat, Fusun Ozguner, Fikret Ercal, and Ponnuswamy Sadayappan. Iterative algorithms for solution of large sparse systems of linear equations on hypercubes. *IEEE Transactions on Computers*, 37(12):1554–1567, 1988.

[13] David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4:23–35, 1990.

[14] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NASA Ames Research Center, Moffett Field, CA, 1992.

[15] M. L. Barton and G. R. Withers. Computing performance as a function of the speed, quantity, and the cost of processors. In *Supercomputing '89 Proceedings*, pages 759–764, 1989.

[16] Jarle Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12:335–342, 1989.

[17] S. Bershader, T. Kraay, and J. Holland. The giant-Fourier-transform. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications: Volume I*, pages 387–389, 1989.

[18] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[19] Edward C. Bronson, Thomas L. Casavant, and L. H. Jamieson. Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):195–205, 1990.

[20] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. Massively parallel methods for engineering and science problems. *Communications of the ACM*, 37(4):31–41, April 1994.

[21] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozman, MT, 1969.

[22] S. Chandran and Larry S. Davis. An approach to parallel vision algorithms. In R. Porth, editor, *Parallel Processing*. SIAM, Philadelphia, PA, 1987.

[23] V. Cherkassky and R. Smith. Efficient mapping and implementations of matrix algorithms on a hypercube. *The Journal of Supercomputing*, 2:7–27, 1988.

[24] N. P. Chrisopchoides, M. Aboelaze, E. N. Houstis, and C. E. Houstis. The parallelization of some level 2 and 3 BLAS operations on distributed-memory machines. In *Proceedings of the First International Conference of the Austrian Center of Parallel Computation*. Springer-Verlag Series Lecture Notes in Computer Science, 1991.

[25] Z. Cvetanovic. Performance analysis of the FFT algorithm on a shared-memory parallel architecture. *IBM Journal of Research and Development*, 31(4):435–451, 1987.

[26] William J. Dally. Wire-efficienct VLSI multiprocessor communication network. In *Stanford Conference on Advanced Research in VLSI Networks*, pages 391–415, 1987.

[27] Eric F. Van de Velde. Multicomputer matrix computations: Theory and practice. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 1303–1308, 1989.

[28] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10:657–673, 1981.

[29] Laurent Desbat and Denis Trystram. Implementing the discrete Fourier transform on a hypercube vector-parallel computer. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications: Volume I*, pages 407–410, 1989.

[30] Iain S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report TR/PA/92/86, Research and Technology Division, Boeing Computer Services, Seattle, WA, 1992.

[31] Iain S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[32] Iain S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, 5(3):633–641, 1984.

[33] Shantanu Dutt and Nam Trinh. Analysis of k-ary n-cubes for the class of parallel divide-and-conquer algorithms. Technical report, Department of Electrical Engineering, University of Minnesota, Minneapolis, MN, 1995.

[34] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[35] Horace P. Flatt. Further applications of the overhead model for parallel systems. Technical Report G320-3540, IBM Corporation, Palo Alto Scientific Center, Palo Alto, CA, 1990.

[36] Horace P. Flatt and Ken Kennedy. Performance of parallel processors. *Parallel Computing*, 12:1–20, 1989.

[37] G. C. Fox, M. Johnson, G. Lyzenga, S. W. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors: Volume 1*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[38] G. C. Fox, M. Johnson, G. Lyzenga, S. W. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors: Volume 1*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[39] G. C. Fox, S. W. Otto, and A. J. G. Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[40] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[41] G. A. Geist and E. G.-Y. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[42] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal on Scientific and Statistical Computing*, 9(4):639–649, 1988. Also available as Technical Report ORNL/TM-10383, Oak Ridge National Laboratory, Oak Ridge, TN, 1987.

[43] A. George. Nested dissection of a regular finite-element mesh. *SIAM Journal on Numerical Ananlysis*, 10:345–363, 1973.

[44] A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

[45] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[46] A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communication reduction in parallel sparse Cholesky factorization on a hypercube. In M. T. Heath, editor, *Hypercube Multiprocessors 1987*, pages 576–586. SIAM, Philadelphia, PA, 1987.

[47] A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10(3):287–298, May 1989.

[48] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Transactions on Mathematical Software*, 2:322–330, 1976.

[49] John R. Gilbert and Robert Schreiber. Highly parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13:1151–1172, 1992.

[50] Gene H. Golub and Charles Van Loan. *Matrix Computations: Second Edition*. The Johns Hopkins University Press, Baltimore, MD, 1989.

[51] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August, 1993. Also available as Technical Report TR 93-24, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[52] Ananth Grama, Vipin Kumar, and V. Nageshwara Rao. Experimental evaluation of load balancing techniques for the hypercube. In *Proceedings of the Parallel Computing '91 Conference*, pages 497–514, 1991.

[53] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. Submitted for publication in *IEEE Transactions on Parallel and Distributed Computing*. Postscript file available in *users/kumar* at anonymous FTP site *ftp.cs.umn.edu*.

[54] Anshul Gupta and Vipin Kumar. The scalability of matrix multiplication algorithms on parallel computers. Technical Report TR 91-54, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1991. A short version appears in *Proceedings of 1993 International Conference on Parallel Processing*, pages III-115–III-119, 1993.

[55] Anshul Gupta and Vipin Kumar. Performance properties of large scale parallel systems. *Journal of Parallel and Distributed Computing*, 19:234–244, 1993. Also available as Technical Report TR 92-32, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[56] Anshul Gupta and Vipin Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993. A detailed version available as Technical Report TR 90-53, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[57] Anshul Gupta and Vipin Kumar. Parallel algorithms for forward and back substitution in direct solution of sparse linear systems. In *Proceedings of* Supercomputing '95, December 1995.

[58] Anshul Gupta, Vipin Kumar, and A. H. Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):455–469, 1995. Also available as Technical Report TR 92-64, Department of Computer Science, University of Minnesota, Minneapolis, MN. A short version appears in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 664–674, 1993.

[59] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[60] John L. Gustafson. The consequences of fixed time performance measurement. In *Proceedings of the 25th Hawaii International Conference on System Sciences: Volume III*, pages 113–124, 1992.

[61] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[62] S. W. Hammond and Robert Schreiber. Efficient ICCG on a shared-memory multiprocessor. *International Journal of High Speed Computing*, 4(1):1–22, March 1992.

[63] M. T. Heath, E. G.-Y. Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[64] M. T. Heath and Padma Raghavan. Distributed solution of sparse linear systems. Technical Report 93-1793, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[65] M. T. Heath and C. H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9(3):558–588, 1988.

[66] Mark D. Hill. What is scalability? *Computer Architecture News*, 18(4), 1990.

[67] Paul G. Hipes. Matrix multiplication on the JPL/Caltech Mark IIIfp hypercube. Technical Report C3P 746, Concurrent Computation Program, California Institute of Technology, Pasadena, CA, 1989.

[68] C.-T. Ho, S. L. Johnsson, and Alan Edelman. Matrix multiplication on hypercubes using full bandwidth and constant storage. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 447–451, 1991.

[69] Laurie Hulbert and Earl Zmijewski. Limiting communication in parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1184–1197, September 1991.

[70] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, New York, NY, 1993.

[71] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, September 1989.

[72] S. L. Johnsson, R. Krawitz, R. Frye, and D. McDonald. A radix-2 FFT on the connection machine. Technical report, Thinking Machines Corporation, Cambridge, MA, 1989.

[73] C. Kamath and A. H. Sameh. The preconditioned conjugate gradient algorithm on a multiprocessor. In R. Vichnevetsky and R. S. Stepleman, editors, *Advances in Computer Methods for Partial Differential Equations*. IMACS, 1984.

[74] Ray A. Kamin and George B. Adams. Fast Fourier transform algorithm design and tradeoffs. Technical Report RIACS TR 88.18, NASA Ames Research Center, Moffet Field, CA, 1988.

[75] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

[76] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. Technical Report TR 95-036, Department of Computer Science, University of Minnesota, 1995.

[77] George Karypis, Anshul Gupta, and Vipin Kumar. Parallel formulation of interior point algorithms. Technical Report 94-20, Department of Computer Science, University of Minnesota, Minneapolis, MN, April 1994. A short version appears in *Supercomputing '94 Proceedings*.

[78] George Karypis and Vipin Kumar. A high performance sparse Cholesky factorization algorithm for scalable parallel computers. Technical Report TR 94-41, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. Submitted to the *Eighth Symposium on the Frontiers of Massively Parallel Computation*, 1995.

[79] S. K. Kim and A. T. Chronopoulos. A class of Lanczos-like algorithms implemented on parallel computers. *Parallel Computing*, 17:763–777, 1991.

[80] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. Technical Report RC13572, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1988.

[81] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.

[82] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Solutions Manual for Introduction to Parallel Computing*. Benjamin/Cummings, Redwood City, CA, 1994.

[83] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. Technical Report 91-55, Computer Science Department, University of Minnesota, 1991. To appear in *Journal of Distributed and Parallel Computing*, 1994.

[84] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, 1994. Also available as Technical Report TR 91-18, Department of Computer Science Department, University of Minnesota, Minneapolis, MN.

[85] Vipin Kumar and V. N. Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[86] Vipin Kumar and V. N. Rao. Load balancing on the hypercube architecture. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 603–608, 1989.

[87] Vipin Kumar and V. N. Rao. Scalable parallel formulations of depth-first search. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, NY, 1990.

[88] Vipin Kumar and Vineet Singh. Scalability of parallel algorithms for the all-pairs shortest path problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, October 1991. A short version appears in the *Proceedings of the International Conference on Parallel Processing*, 1990.

[89] C. E. Leiserson. Fat-trees: Universal networks for hardware efficient supercomputing. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 393–402, 1985.

[90] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

[91] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:177–189, 1979.

[92] J. W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. Technical Report CS-90-04, York University, Ontario, Canada, 1990. Also appears in *SIAM Review*, 34:82–109, 1992.

[93] J. W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[94] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, PA, 1992.

[95] Robert F. Lucas. *Solving planar systems of equations on distributed-memory multiprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, 1987.

[96] Robert F. Lucas, Tom Blank, and Jerome J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer Aided Design*, CAD-6(6):981–991, November 1987.

[97] Y. W. E. Ma and Denis G. Shea. Downward scalability of parallel architectures. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 109–120, 1988.

[98] Dan C. Marinescu and John R. Rice. On high level characterization of parallelism. Technical Report CSD-TR-1011, CAPO Report CER-90-32, Computer Science Department, Purdue University, West Lafayette, IN, Revised June 1991. To appear in *Journal of Parallel and Distributed Computing*, 1993.

[99] Pontus Matstoms. *The multifrontal solution of sparse linear least squares problems*. PhD thesis, Department of Mathematics, Linkoping University, S-581 83 Linkoping, Sweden, March 1992.

[100] Pontus Matstoms. Sparse QR factorization in MATLAB. Technical Report LiTH-MAT-R-1992-05, Department of Mathematics, Linkoping University, S-581 83 Linkoping, Sweden, March 1993.

[101] Rami Melhem. Toward efficient implementation of preconditioned conjugate gradient methods on vector supercomputers. *International Journal of Supercomputer Applications*, I(1):70–97, 1987.

[102] Cleve Moler. Another look at Amdahl's law. Technical Report TN-02-0587-0288, Intel Scientific Computers, 1987.

[103] Mo Mu and John R. Rice. A grid-based subtree-subcube assignment strategy for solving partial differential equations on hypercubes. *SIAM Journal on Scientific and Statistical Computing*, 13(3):826–839, May 1992.

[104] Vijay K. Naik and M. Patrick. Data traffic reduction schemes Cholesky factorization on aynchronous multiprocessor systems. In *Supercomputing '89 Proceedings*, 1989. Also available as Technical Report RC 14500, IBM T. J. Watson Research Center, Yorktown Heights, NY.

[105] David M. Nicol and Frank H. Willard. Problem size, parallel architecture, and optimal speedup. *Journal of Parallel and Distributed Computing*, 5:404–420, 1988.

[106] A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared memory architectures. *IEEE Transactions on Computers*, C-36(5):581–591, 1987.

[107] S. F. Nugent. The iPSC/2 direct-connect communications technology. In *Proceedings of the Third Conference on Hypercubes, Concurrent Computers, and Applications*, pages 51–60, 1988.

[108] Daniel Nussbaum and Anant Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):57–61, 1991.

[109] Dianne P. O'Leary and G. W. Stewart. Assignment and scheduling in parallel matrix factorization. *Linear Algebra and its Applications*, 77:275–299, 1986.

[110] V. Pan and J. H. Reif. Efficient parallel solution of linear systems. In *17th Annual ACM Symposium on Theory of Computing*, pages 143–152, 1985.

[111] Alex Pothen, H. D. Simon, and K.-P. Liou. Partioning sparce matrices with eigenvectors of graphs. *SIAM Journal of Mathematical Analysis and Applications*, 11(3):430–452, 1990.

[112] Alex Pothen, H. D. Simon, and Lie Wang. Spectral nested dissection. Technical Report 92-01, Computer Science Department, Pennsylvania State University, University Park, PA, 1992.

[113] Alex Pothen, H. D. Simon, Lie Wang, and Stephen T. Bernard. Towards a fast implementation of spectral nested dissection. In *Supercomputing '92 Proceedings*, pages 42–51, 1992.

[114] Alex Pothen and Chunguang Sun. Distributed multifrontal factorization using clique trees. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 34–40, 1991.

[115] Roland Pozo and Sharon L. Smith. Performance evaluation of the parallel multifrontal method in a distributed-memory environment. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 453–456, 1993.

[116] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, New York, NY, 1987.

[117] Padma Raghavan. *Distributed sparse matrix factorization: QR and Cholesky factorizations*. PhD thesis, Pennsylvania State University, University Park, PA, 1991.

[118] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, New York, NY, 1990.

[119] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

[120] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing '92 Proceedings*, 1992.

[121] Youcef Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[122] Youcef Saad and M. H. Schultz. Parallel implementations of preconditioned conjugate gradient methods. Technical Report YALEU/DCS/RR-425, Yale University, Department of Computer Science, New Haven, CT, 1985.

[123] Robert Schreiber. Scalability of sparse direct solvers. Technical Report RIACS TR 92.13, NASA Ames Research Center, Moffet Field, CA, May 1992. Also appears in A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms* (An IMA Workshop Volume). Springer-Verlag, New York, NY, 1993.

[124] S. L. Scott and J. R. Goodman. The impact of pipelined channels on k-ary n-cube networks. *IEEE Transactions on Parallel and Distributed Systems*, pages 2–16, January 1994.

[125] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. Scalability of parallel sorting on mesh multicomputers. *International Journal of Parallel Programming*, 20(2), 1991.

[126] B. Speelpening. The generalized element method. Technical Report UIUCDCS-R-78-946, Department of Computer Science, University of Illinois, Urbana, IL, November 1978.

[127] Chunguang Sun. Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors. Technical Report CTC92TR102, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, August 1992.

[128] Chunguang Sun. Parallel sparse orthogonal factorization on distributed-memory multiprocessors. Technical Report CTC93TR162, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, December 1993.

[129] Chunguang Sun. Parallel multifrontal solution of sparse linear least squares problems on distributed-memory multiprocessors. Technical Report CTC94TR185, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, July 1994.

[130] Xian-He Sun and John L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17:1093–1109, December 1991. Also available as Technical Report IS-5053, UC-32, Ames Laboratory, Iowa State University, Ames, IA.

[131] Xian-He Sun and L. M. Ni. Another view of parallel speedup. In *Supercomputing '90 Proceedings*, pages 324–333, 1990.

[132] Xian-He Sun and Diane Thiede Rover. Scalability of parallel algorithm-machine combinations. Technical Report IS-5057, Ames Laboratory, Iowa State University, Ames, IA, 1991. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[133] P. N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

[134] Zhimin Tang and Guo-Jie Li. Optimal granularity of grid iteration problems. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I111–I118, 1990.

[135] Clark D. Thompson. Fourier transforms in VLSI. *IBM Journal of Research and Development*, C-32(11):1047–1057, 1983.

[136] Walter F. Tichy. Parallel matrix multiplication on the connection machine. Technical Report RIACS TR 88.41, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1988.

[137] Fredric A. Van-Catledge. Towards a general model for evaluating the relative performance of computer systems. *International Journal of Supercomputer Applications*, 3(2):100–108, 1989.

[138] Henk A. van der Vorst. A vectorizable variant of some ICCG methods. *SIAM Journal on Scientific and Statistical Computing*, III(3):350–356, 1982.

[139] Henk A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing*, 5:45–54, 1987.

[140] Sesh Venugopal and Vijay K. Naik. Effects of partitioning and scheduling sparse matrix factorization on communication and load balance. In *Supercomputing '91 Proceedings*, pages 866–875, 1991.

[141] Sesh Venugopal and Vijay K. Naik. SHAPE: A parallelization tool for sparse matrix computations. Technical Report DCS-TR-290, Department of Computer Science, Rutgers University, New Brunswick, NJ, June 1992.

[142] Jeffrey Scott Vitter and Roger A. Simons. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Transactions on Computers*, May 1986.

[143] Jinwoon Woo and Sartaj Sahni. Hypercube computing: Connected components. *Journal of Supercomputing*, 1991. Also available as TR 88-50 from the Department of Computer Science, University of Minnesota, Minneapolis, MN.

[144] Jinwoon Woo and Sartaj Sahni. Computing biconnected components on a hypercube. *Journal of Supercomputing*, June 1991. Also available as Technical Report TR 89-7 from the Department of Computer Science, University of Minnesota, Minneapolis, MN.

[145] Patrick H. Worley. *Information Requirements and the Implications for Parallel Computation*. PhD thesis, Stanford University, Department of Computer Science, Palo Alto, CA, 1988.

[146] Patrick H. Worley. The effect of time constraints on scaled speedup. *SIAM Journal on Scientific and Statistical Computing*, 11(5):838–858, 1990.

[147] Patrick H. Worley. Limits on parallelism in the numerical solution of linear PDEs. *SIAM Journal on Scientific and Statistical Computing*, 12:1–35, January 1991.

[148] Xiaofeng Zhou. Bridging the gap between Amdahl's law and Sandia laboratory's result. *Communications of the ACM*, 32(8):1014–5, 1989.

[149] J. R. Zorbas, D. J. Reble, and R. E. VanKooten. Measuring the scalability of parallel computer systems. In *Supercomputing '89 Proceedings*, pages 832–841, 1989.

Appendix A

COMPLEXITY OF FUNCTIONS AND ORDER ANALYSIS

Order analysis and the asymptotic complexity of functions are used extensively in this thesis to analyze the performance of algorithms.

A.1 Complexity of Functions

When analyzing parallel algorithms in this thesis, we use the following three types of functions:

1. **Exponential functions:** A function f from reals to reals is called an **exponential** function in x if it can be expressed in the form $f(x) = a^x$ for $x, a \in \Re$ (the set of real numbers) and $a > 1$. Examples of exponential functions are 2^x, 1.5^{x+2}, and $3^{1.5x}$.

2. **Polynomial functions:** A function f from reals to reals is called a **polynomial** function of **degree** b in x if it can be expressed in the form $f(x) = x^b$ for $x, b \in \Re$ and $b > 0$. A **linear** function is a polynomial function of degree one and a **quadratic** function is a polynomial function of degree two. Examples of polynomial functions are 2, $5x$, and $5.5x^{2.3}$.

   A function f that is a sum of two polynomial functions g and h is also a polynomial function whose degree is equal to the maximum of the degrees of g and h. For example, $2x + x^2$ is a polynomial function of degree two.

3. **Logarithmic functions:** A function f from reals to reals that can be expressed in the form $f(x) = \log_b x$ for $b \in \Re$ and $b > 1$ is **logarithmic** in x. In this expression, b is called the **base** of the logarithm. Examples of logarithmic functions are $\log_{1.5} x$ and $\log_2 x$. Unless stated otherwise, all logarithms in this thesis are of base two. We use $\log x$ to denote $\log_2 x$, and $\log^2 x$ to denote $(\log_2 x)^2$.

Most functions in this thesis can be expressed as sums of two or more functions. A function f is said to **dominate** a function g if $f(x)$ grows at a faster rate than $g(x)$. Thus, function f dominates function g if and only if $f(x)/g(x)$ is a monotonically increasing function in x. In other words, f dominates g if and only if for any constant $c > 0$, there exists a value x_0 such that $f(x) > cg(x)$ for $x > x_0$. An exponential function dominates a polynomial function and a polynomial function dominates a logarithmic function. The relation *dominates* is transitive. If function f dominates function g, and function g dominates function h, then function f also dominates function h. Thus, an exponential function also dominates a logarithmic function.

A.2 Order Analysis of Functions

In the analysis of algorithms, it is often cumbersome or impossible to derive exact expressions for parameters such as run time, speedup, and efficiency. In many cases, an approximation of the exact expression is adequate. The approximation may indeed be more illustrative of the behavior of the function because it focuses on the critical factors influencing the parameter.

Consider three cars A, B, and C. Assume that we start monitoring the cars at time $t = 0$. At $t = 0$, car A is moving at a velocity of 1000 feet per second and maintains a constant velocity. At $t = 0$, car B's velocity is 100 feet per second and it is accelerating at a rate of 20 feet per second per second. Car C starts from a standstill at $t = 0$ and accelerates at a rate of 25 feet per second per second. Let $D_A(t)$, $D_B(t)$, and $D_C(t)$ represent the distances traveled in t seconds by cars A, B, and C. From elementary physics, we know that

$$\begin{aligned} D_A(t) &= 1000t, \\ D_B(t) &= 100t + 20t^2, \\ D_C(t) &= 25t^2. \end{aligned}$$

Now, we compare the cars according to the distance they travel in a given time. For $t > 45$ seconds, car B outperforms car A. Similarly, for $t > 20$ seconds, car C outperforms car B, and for $t > 40$ seconds, car C outperforms car A. Furthermore, $D_C(t) < 1.25 D_B(t)$ and $D_B(t) < D_C(t)$ for $t > 20$, which implies that after a certain time, the difference in the performance of cars B and C is bounded by the other scaled by a constant multiplicative factor. All these facts can be captured by the order analysis of the expressions.

The Θ Notation: From the above example, $D_C(t) < 1.25D_B(t)$ and $D_B(t) < D_C(t)$ for $t > 20$; that is, the difference in the performance of cars B and C after $t = 0$ is bounded by the other scaled by a constant multiplicative factor. Such an equivalence in performance is often significant when analyzing performance. The Θ notation captures the relationship between these two functions. The functions $D_C(t)$ and $D_B(t)$ can be expressed by using the Θ notation as $D_C(t) = \Theta(D_B(t))$ and $D_B(t) = \Theta(D_C(t))$. Furthermore, both functions are equal to $\Theta(t^2)$.

Formally, the Θ notation is defined as follows: given a function $g(x)$, $f(x) = \Theta(g(x))$ if and only if for any constants $c_1, c_2 > 0$, there exists an $x_0 \geq 0$, such that $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for all $x \geq x_0$.

The O Notation: Often, we would like to bound the growth of a particular parameter by a simpler functions. From the example given earlier in this appendix, we have seen that for $t > 45$, $D_B(t)$ is always greater than $D_A(t)$. This relation between $D_A(t)$ and $D_B(t)$ is expressed using the O (big-oh) notation as $D_A(t) = O(D_B(t))$.

Formally, the O notation is defined as follows: given a function $g(x)$, $f(x) = O(g(x))$ if and only if for any constant $c > 0$, their exists an $x_0 \geq 0$, such that $f(x) \leq cg(x)$ for all $x \geq x_0$. From this definition we deduce that $D_A(t) = O(t^2)$ and $D_B(t) = O(t^2)$. Furthermore, $D_A(t) = O(t)$ also satisfies the conditions of the O notation.

The Ω Notation: The O notation sets an upper bound on the rate of growth of a function. The Ω notation is the converse of O notation; that is, it sets a lower bound on the rate of growth of a function. From the example given earlier in this appendix, $D_A(t) < D_C(t)$ for $t > 40$. This relationship can be expressed using the Ω notation as $D_C(t) = \Omega(D_A(t))$.

Formally, given a function $g(x)$, $f(x) = \Omega(g(x))$ if and only if for any constant $c > 0$, there exists an $x_0 \geq 0$, such that $f(x) \geq cg(x)$ for all $x \geq x_0$.

Properties of Functions Expressed in Order Notation

The order notations for expressions have a number of properties that are useful when analyzing the performance of algorithms. Some of the important properties are as follows:

1. $x^a = O(x^b)$ if and only if $a \leq b$.

2. $\log_a(x) = \Theta(\log_b(x))$ for all a and b.

3. $a^x = O(b^x)$ if and only if $a \leq b$.

4. For any constant c, $c = O(1)$.

5. If $f = O(g)$ then $f + g = O(g)$.

6. If $f = \Theta(g)$ then $f + g = \Theta(g) = \Theta(f)$.

7. $f = O(g)$ if and only if $g = \Omega(f)$.

8. $f = \Theta(g)$ if and only if $f = \Omega(g)$ and $f = O(g)$.

Appendix B

PROOF OF CASE I IN SECTION 2.3.1

Here we give a brief proof of $E = 1 - 1/x_j$ for a more general form of T_o than the one given in Section 2.3.1. Here x_j is the exponent of p in the dominant term of T_o.

Let $T_o(W, p) = \Sigma_{i=1}^{i=n} c_i W^{y_i} (\log W)^{u_i} p^{x_i} (\log p)^{z_i}$, where c_i's are constants and $x_i \geq 0$ and $y_i \geq 0$ for $1 \leq i \leq n$, and u_i's and z_i's are 0's or 1's. Now let us compute $\frac{d}{dp} T_o(W, p)$.

$$
\begin{aligned}
T_o(W, p) &= \Sigma_{i=1}^{i=n} c_i W^{y_i} (\log W)^{u_i} p^{x_i} (\log p)^{z_i}, \\
\frac{d}{dp} T_o(W, p) &= \Sigma_{i=1}^{i=n} c_i W^{y_i} (\log W)^{u_i} (x_i p^{x_i - 1} (\log p)^{z_i} + z_i p^{x_i - 1} (\log p)^{z_i - 1}).
\end{aligned}
$$

If all z_i's are either 0 or 1, then the above equations can be rewritten as follows:

$$
\begin{aligned}
\frac{d}{dp} T_o(W, p) &= \Sigma_{i=1}^{i=n} c_i W^{y_i} (\log W)^{u_i} (x_i p^{x_i - 1} (x_i \log p)^{z_i} + z_i), \\
\frac{d}{dp} T_o(W, p) &\approx \Sigma_{i=1}^{i=n} c_i x_i W^{y_i} (\log W)^{u_i} x_i p^{x_i - 1}.
\end{aligned}
$$

Equating $\frac{d}{dp} T_o(W, p)$ to T_P according to Equation 2.10, we get

$$
\begin{aligned}
\Sigma_{i=1}^{i=n} c_i x_i W^{y_i} (\log W)^{u_i} x_i p^{x_i - 1} &= \frac{W + \Sigma_{i=1}^{i=n} c_i W^{y_i} (\log W)^{u_i} p^{x_i} (\log p)^{z_i}}{p}, \\
W &= \Sigma_{i=1}^{i=n} c_i (x_i - 1) W^{y_i} (\log W)^{u_i} p^{x_i} (\log p)^{z_i}. \qquad \text{(B.1)}
\end{aligned}
$$

The above equation determines the relation between W and p for which the parallel execution time is minimized. The equation determining the isoefficiency function for the parallel system with the overhead function under consideration will be as follows (see discussion in Section 2.3.2):

$$
W = \frac{E}{1 - E} \Sigma_{i=1}^{i=n} c_i W^{y_i} (\log W)^{u_i} p^{x_i} (\log p)^{z_i}. \qquad \text{(B.2)}
$$

Comparing Equations B.1 and B.2, if the jth term in T_o is the dominant term and $x_j > 1$, then the efficiency at the point of minimum parallel execution time will be given by $E_0 \approx 1 - 1/x_j$.

Appendix C

PROOF OF CASE II IN SECTION 2.3.1

Here we show that the speedup S is maximum for $p_o = \Theta(W^{(1-y_j)/x_j})$, where x_j is the exponent of p in the dominant term of T_o.

From Equation 2.11, the relation between W and p_0 is given by the solution for p from the following equation:

$$W = \Sigma_{i=1}^{i=n} c_i(x_i - 1)W^{y_i}p^{x_i}.$$

If the jth term on R.H.S. of the above is the dominant term according to the condition described in Section 2.3.1, then we take $p_0 \approx (W^{1-y_j}/(c_j(x_j - 1)))^{1/x_j}$ as the approximate solution. Now we show that the speedup is indeed (asymptotically) maximum for this value of p_0. We know that

$$S = \frac{Wp}{W + T_o(W, p)}.$$

Since the maximum speedup condition is true in asymptotics, we will drop the constants and write order expressions only on the R.H.S..

$$\begin{aligned} S &= \Theta(\frac{W \times W^{\frac{1-y_j}{x_j}}}{W + \Sigma_{i=1}^{i=n}(W^{y_i} \times W^{\frac{1-y_j}{x_j}x_i})}), \\ S &= \Theta(\frac{W^{1+\frac{1-y_j}{x_j}}}{W + \Sigma_{i=1}^{i=n} W^{y_i+\frac{1-y_j}{x_j}x_i}}). \end{aligned}$$

The summation $\Sigma_{i=1}^{i=n} W^{y_i+(1-y_j)x_i/x_j}$ in the denominator on the R.H.S. is at least $\Omega(W)$, because for $i = j$, $W^{y_i+(1-y_j)x_i/x_j} = W$. So we can ignore the first W in the denominator. Rewriting the expression for speedup, we get

$$\begin{aligned} S &= \Theta(\frac{W^{1+\frac{1-y_j}{x_j}}}{\Sigma_{i=1}^{i=n} W^{y_i+\frac{1-y_j}{x_j}x_i}}), \\ S &= \Theta(\frac{1}{\Sigma_{i=1}^{i=n} W^{y_i-1+(\frac{1-y_j}{x_j}-1)x_i}}). \end{aligned}$$

Clearly, the above expression will be maximum when the denominator is minimum, which will happen for the minimum possible value of $(1 - y_j)/x_j$.

Appendix D

DERIVATION OF THE ISOEFFICIENCY FUNCTION FOR PARALLEL TRIANGULAR SOLVERS

Consider solving the triangular systems resulting from the factorization of a sparse matrix associated with a two-dimensional neighborhood graph. From Equation 4.3, $W = O(N \log N$ and from Equation 4.4, $T_o = O(p^2) + O(p\sqrt{N})$. In order to maintain a fixed efficiency, $W \propto T_o$; i.e., the following two conditions must be satisfied:

$$W \propto p^2 \tag{D.1}$$

and

$$W \propto p\sqrt{N}. \tag{D.2}$$

Equation D.1 clearly suggests an isoefficiency function of $O(p^2)$. From Equation D.2,

$$N \log N \propto p\sqrt{N},$$

$$\sqrt{N} \log N \propto p, \tag{D.3}$$

$$\log N + \log\log N \propto \log p. \tag{D.4}$$

Discarding the lower order term $\log\log N$ from Equation D.4, we get

$$\log N \propto \log p. \tag{D.5}$$

From Equations D.3 and D.5,

$$\sqrt{N} \propto \frac{p}{\log p},$$

$$N \propto (\frac{p}{\log p})^2,$$

$$N \log N \propto \frac{p^2}{\log p},$$

$$W \propto \frac{p^2}{\log p}. \tag{D.6}$$

Thus, we have derived Equation 4.6. Similarly, we can derive the isoefficiency function for the triangular systems resulting from the factorization os sparse matrices associated with three-dimensional neighborhood graphs. Recall from Chapter 4 that for such systems, $W = O(N^{4/3})$ and $T_o = O(p^2) + O(pN^{2/3})$. If $W \propto T_o$, then $W \propto p^2$ (the first term of T_o) and $W \propto pN^{2/3}$. The second condition yields

$$N^{4/3} \propto pN^{2/3},$$

$$N^{2/3} \propto p,$$

$$N^{4/3} \propto p^2,$$

$$W \propto p^2, \tag{D.7}$$

which is same as Equation 4.9.