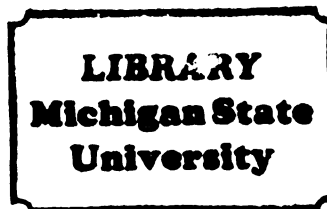


THESIS
c.1



This is to certify that the

dissertation entitled

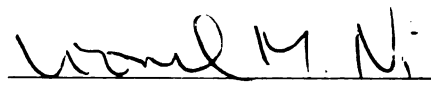
PARALLEL COMPUTATION MODELS:
REPRESENTATION, ANALYSIS AND APPLICATIONS

presented by

Xian-He SUN

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science


Major professor

Date 11/2/90

**PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.**

DATE DUE	DATE DUE	DATE DUE
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

MSU is An Affirmative Action/Equal Opportunity Institution

c:\circ\datedue.pm3-p.1

PARALLEL COMPUTATION MODELS:
REPRESENTATION, ANALYSIS AND APPLICATIONS

By

Xian-He Sun

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1990

615-3495

ABSTRACT

PARALLEL COMPUTATION MODELS: REPRESENTATION, ANALYSIS AND APPLICATIONS

By

Xian-He Sun

Although there are various multiprocessors available, both software support and algorithm development for these machines are far behind their hardware counterpart. This is especially true for multicomputers in which each processor has its own local memory. A new concept, *compute-exchange computation*, is proposed for efficient parallel algorithm design in multicomputers. Based on this new concept, the most frequently used scientific and engineering applications can be described by a simple structured representation. Structured design and rethinking, therefore, become possible. The basic building blocks of these structures, called parallel computation models, are identified and studied. A performance metric of parallel processing, *parallel speedup*, is carefully examined. A new model of speedup is presented which considers the memory capacity as an influential factor and provides the quantum of the tradeoff between time and space.

These research results are useful in many areas of computer science. The dissertation focuses on two areas: efficient algorithm design and performance prediction. Examples chosen from real applications will be used in both areas to illustrate the concept and usefulness of the computation models. The applications used for efficient algorithm design are *solving large scale tridiagonal systems* and *solving electrical power flow problems*. Several novel algorithms are developed for these two applications based on combinations of computation models. Implementation results confirm that these algorithms are competitive with any existing algorithm in the field.

©Copyright By

Xian-He Sun

1990

Dedicated to my parents
Yu Lin and Chang-Xiang Sun
and to my family.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to all of the faculty members who have helped and contributed to make this work a reality. This includes Professor Lionel M. Ni, my thesis advisor, for his support, guidance and inspiration, and Professors Richard Enbody, Moon Jung Chung, and Richard Hill for their insightful comments and serving in my Ph.D. committee, and Professors Nabil Kamel, Abdol Esfahanian and Fathi Salam for their encouragement.

I would like to acknowledge all of my fellow students who have given me help and friendship during my stay at MSU. In particular, I wish to thank Dr. Chung-Ta King, Jayashree Ramanathan, Dongyul Ra, Paul Wolberg, Xian-La Lin, and Honda Shing, for their help and valuable discussions, and David Robinson, Dr. Shixiong Guo and Ten Hwan Tzen for their cooperation and wonderful work.

I am very grateful to my parents and parents-in-law for their years support, concern and understanding. I am indebted to my wife Hong Zhang-Sun for her constant encouragement, assistance and care during the course of my graduate studies.

This research has been supported in part by National Science Foundation under grant ECS-88-14027.

Table of Contents

List of Tables	viii
List of Figures	ix
1 INTRODUCTION	1
1.1 Multicomputers	2
1.2 Parallel Algorithm Design Considerations	3
1.2.1 Sources of Degradation	4
1.2.2 Algorithm Characteristics	6
1.3 Motivation and Problem Statement	8
1.4 Thesis Organization	10
2 PERFORMANCE METRIC OF PARALLEL ALGORITHMS	12
2.1 Preliminary	13
2.2 Models of Speedup	17
2.3 Simplified Models of Speedup	19
2.4 Comparison Study	26
3 PARALLEL COMPUTATION MODELS FOR SCIENTIFIC COMPUT- ING	29
3.1 Structured Representation	30
3.2 Parallel Computation Models	36
3.2.1 Local Computation Model	38
3.2.2 Global-Exchange Computation Model	39
3.2.3 Compute-Aggregate-Broadcast Computation Model	39
3.2.4 Divide-and-Conquer Computation Model	40
3.2.5 Domain Decomposition Model	44
3.2.6 Pipeline (Ring) Computation Model	45

3.2.7	Recursive Doubling Computation Model	46
3.3	Application Considerations	49
4	APPLICATION-DRIVEN ALGORITHM DESIGN	52
4.1	Preliminary	53
4.2	Power Flow Application	54
4.3	Homotopy Method	56
4.4	Implementation Issues and Results	58
5	ARCHITECTURE-DRIVEN ALGORITHM DESIGN	64
5.1	Linear Tridiagonal Solvers	65
5.1.1	The LU Decomposition Method	65
5.1.2	The Parallel Prefix Method	66
5.1.3	A Novel Matrix Partitioning Technique	67
5.2	A Compute-Aggregate-Broadcast Computation Solver	70
5.3	A Global-Exchange Computation Model	73
5.4	A Solver With More Than One Computation Model	75
5.5	Comparison and Experimental Results	77
6	PREDICTION OF PERFORMANCE	81
6.1	Performance Formulations	82
6.2	Structured Prediction	87
6.3	The Influence of Problem Size on Speedup	90
7	CONCLUSION AND FUTURE RESEARCH	97
7.1	Summary and Major Contributions	97
7.2	Future Research Directions	100
	Bibliography	103

List of Tables

5.1	Computation and Communication Counts of Tridiagonal Solvers	77
-----	---	----

List of Figures

1.1	A Generic Multicomputer Architecture	3
2.1	Parallelism Profile of an Application	14
2.2	Shape of the Application	15
2.3	Amdahl's Law	20
2.4	Gustafson's Scaled Speedup	21
2.5	Simplified Memory-Bounded Scaled Speedup	22
2.6	Matrix Multiplication with Local Computation	24
2.7	Matrix Multiplication with Global Computation	24
2.8	Amdahl's law, Gustafson's speedup and SMB speedup	27
3.1	Compute-Exchange Computation	30
3.2	Multicast Data Exchange	32
3.3	Conjunctive Data Exchange	33
3.4	Partitioning of Graphs	34
3.5	FFT (Butterfly) Computation	35
3.6	Odd-Even Cyclic Reduction	37
3.7	Local Computation Model	38
3.8	Global Exchange Computation Model	40
3.9	CAB Computation Model	41
3.10	Divide-and-Conquer Compute-Exchange Computation	42
3.11	Domain Decomposition Computation Model	44
3.12	Pipelined Computation Model	47
3.13	Recursive Doubling Computation Model	50
4.1	The Main Components of Power System	54
4.2	The Homotopy Method	59
4.3	Global-Exchange Model for Merge Checking	60

4.4	Compute-Aggregate-Broadcast Model for Merge Checking	61
4.5	Speedup Versus Different Number of Processors	62
5.1	The Communication Pattern of the PPT Algorithm	72
5.2	Tree Reduction	73
5.3	The Communication Pattern of the G-E Computation	74
5.4	The Communication Pattern of the PPH Algorithm	76
5.5	The Speedup Over The LU Decomposition Method	79
5.6	Efficiency Over The LU Decomposition Method	80
6.1	Parallel Prefix Method	89
6.2	Measured and Estimate Speedup of Parallel Prefix Method	91

Chapter 1

INTRODUCTION

Powerful computer systems able to handle computationally intensive problems are increasingly in demand in many scientific and engineering areas. To cope with these computational requirements, a natural trend is to construct systems consisting of multiple processors. This approach has been shown in recent years to be the most straightforward and cost-effective approach for achieving high performance. One system consisting of many processors communicating through an interconnection mechanism is called a *multiprocessor system* [28]. These processors are usually homogeneous, and the communication latency between processors is relatively small but non-negligible. The communication latency, along with other degradations, makes parallel computation much more complex than sequential computation. Much effort has been exerted and is currently being exerted to obtain highly efficient parallel computation. Although there are various multiprocessors available, software support and efficient algorithm development for these machines are far behind their hardware counterpart.

In scientific computing, there are several issues related to parallelization that do not arise in a serial context. The first issue is *task partitioning*, i.e., the breakdown of the total workload into smaller tasks, some or all of which, can be processed concurrently. This includes the proper sequencing of the tasks when some tasks are interdependent and cannot be executed simultaneously. The second issue is the *synchronization* of concurrent processes. In some methods, processes must wait at predetermined points for the completion of certain computations or for the arrival of certain data. The mechanisms used to enforce such synchronization have an important effect on algorithmic performance. A third issue is the *communication* of interim results between the processes. The objective here is to carry out the communication efficiently, which is problem dependent as well as machine dependent. The fourth issue is to obtain a good match between algorithmic requirements and architectural capabilities, i.e., the pairing of applications and architectures. From a parallel algorithm design point of view, the issue is how to explore all the execution potential of the underlying architecture, which includes the determination of the number of processors needed as well as task and data allocation across the processors. From the architecture point of view, architectures can be tailored for the execution of a fixed set of algorithms if the architectural requirement of the algorithms are understood.

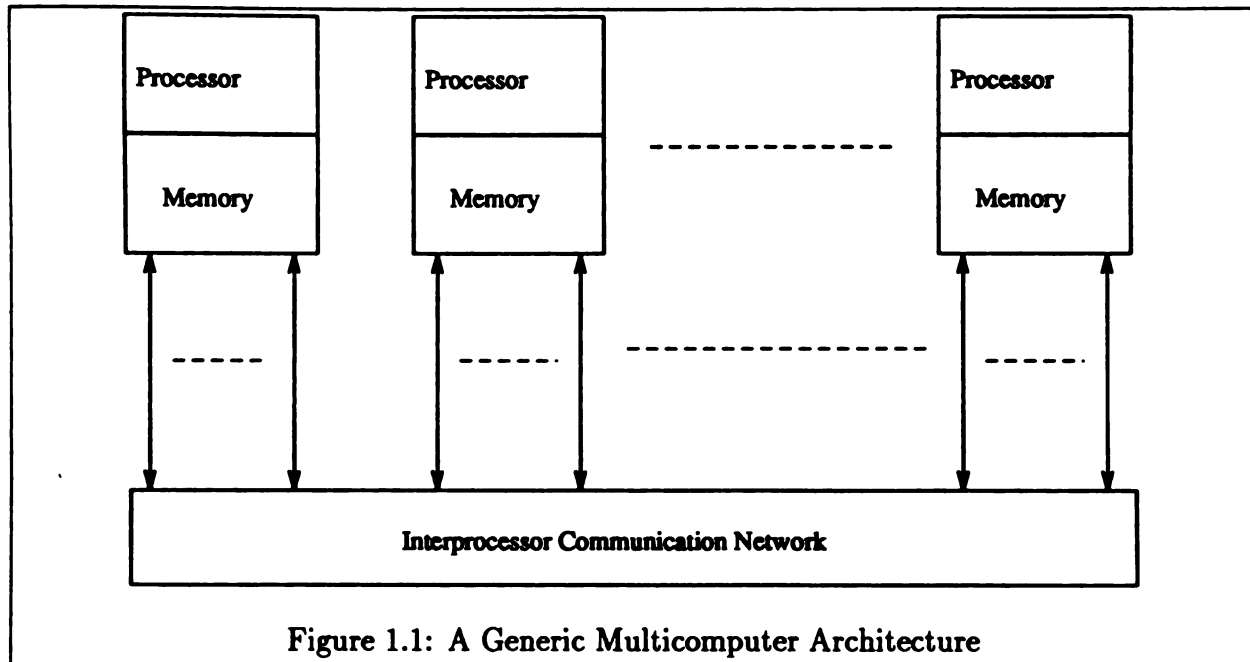
All of the above issues are problem dependent. It is difficult, if not impossible, to derive any acceptable solution for parallel computation as a whole. For this reason the issues have been traditionally studied problem by problem and solved by brute force methods. However, the issues are important in a general context. A more general study is needed that does not reference a special problem. In this dissertation, a new concept, *compute-exchange computation*, is proposed. Based on this new concept, most of the frequently used scientific and engineering applications can be represented by a simple structured representation. Structured design and rethinking, therefore, become possible. The basic building blocks of these structures, called *parallel computation models*, are identified and studied. Issues of performance metrics for parallel computation are also examined. The results of this research are useful in many areas of computer science. The study presented will focus on two areas, efficient algorithm design and performance prediction.

This chapter is organized as follows. Section 1.1 will address the parallel systems considered in this study. Section 1.2 will discuss the parallel programming considerations. The motivation of the research will be given in Section 1.3. Finally, an overview of the dissertation will be found in Section 1.4.

1.1 Multicomputers

The parallel systems considered in this study are *multicomputers*. Multicomputers are message passing distributed-memory multiprocessors [28]. They are organized as an ensemble of small programmable computers, called *nodes*, and communicate through a point-to-point interprocessor communication network. Each memory module is physically associated with each processor. When the number of processors increases, the memory capacity also increases. Multicomputers with hundreds or thousands of processors are available commercially. Examples of first generation multicomputers include JPL's MARK III, Intel's iPSC (up to 128 processors), Ncube's NCUBE (up to 1024 processors), Ametek's S/14 (up to 256 processors) and FPS's T series (up to 2^{14} processors) [55] [23] [21]. Note that the Connection Machine [25], though hypercube interconnected, is a bit-serial SIMD machine. New multicomputers are also emerging. Both Intel's iPSC-2 [46] and Ametek's 2010 [3] are classified as second generation multicomputers. The iWARP [6], being developed by Intel and CMU, is also an example of a high performance multicomputer. All first generation multicomputers adopt the store-and-forward communication mechanism and the hypercube topology. Second generation multicomputers have more advanced communication mechanisms. Although the

processors are physically interconnected as a hypercube or a 2D-mesh, they are logically fully connected [3]. The structured representation proposed in this study aims at these new communication mechanisms. Multicomputers hold a promising potential for providing massive parallelism. In addition to architectural enhancement through an increase in computational capability and a decrease in communication latency, the full potential of a multicomputer cannot be exploited without a careful design of algorithms. The results of this research will benefit this careful design and should be useful in the design of future multicomputers to help accommodate those frequently used parallel computation models. A generic architecture of multicomputers is depicted in Figure 1.1.



1.2 Parallel Algorithm Design Considerations

During the past two decades, the computing community has witnessed a rapid growth of interest in parallel processing. The motivation behind this increasing interest is not that parallel computing is easier to program, but that there are physical and technological limits in uniprocessors. Modern VLSI technologies enable multiprocessors to be built and implemented at a very cost-effective rate for solving computationally intensive problems. However, the expected superior performance can be achieved only when all the advantages of the underlying architecture are exploited. These include partitioning an application in such a way that load balancing can be maximized and allocating the tasks in such a way that the

communication overhead can be minimized. Neither this maximum nor minimum can be achieved easily. In fact, it has been shown that implementing parallel algorithms is much more difficult than most computer scientists had expected [30]. Parallelism degradations exist that could degrade the performance of parallel algorithms. Depending on their characteristics, different algorithms may be influenced by different degradations. In this section we shall address possible degradations and characteristics of parallel algorithms.

1.2.1 Sources of Degradation

Parallel algorithms rarely achieve linear speedup for several reasons [18]. In designing efficient parallel algorithms, we have to know the sources of degradation and how to reduce their influence or remove them completely whenever possible. In this section sources of degradation are discussed. In the next section, parallel algorithms are characterized according to their inherent degradations.

- *Communication delay.* From a computation point of view, in addition to the *transmission* and *propagation delay*, there are two other communication delays [5]:
 - *Communication processing time.* This is the time required to prepare information for transmission.
 - *Queuing time.* Once information is assembled into packets for transmission on some communication link, it must wait in a queue prior to the start of its transmission. Queuing time is generally difficult to quantify precisely, but simplified models are often adequate to obtain valuable qualitative and quantitative conclusions [34].

Communication overhead can be reduced by overlapping communication with computation. The queuing, transmission, and propagation delays can possibly be overlapped with computations, but communication processing cannot.

Communication cost is a determining factor for the complexity of parallel algorithms. Various techniques have been developed to reduce communication overhead. These include designing efficient networks, overlapping communication with computation, and developing algorithms carefully. In second generation multicomputers, the number of hops from a source node to a destination node is not a major factor in determining the message delay times. However, there are three classical ways to reduce communication delay by restructuring algorithms which still hold true [31]: 1) reducing the quantity of information that must be

communicated, 2) reducing the frequency with which messages must be sent, 3) overlapping communication with computation. In practice the above goals may not be mutually compatible.

- *Uneven Allocation.* It is desirable to distribute the computation load evenly so that all processors have an identical amount of work. However, load balancing is very difficult to achieve. High level module load balancing may lead to unbalanced computation due to the underlying data structure. Those processors that finish a step of computation earlier may have to wait until others are complete before continuing. Thus, the processing power of those processors is wasted.

Communication delay and uneven allocation are the main sources of degradation in parallel algorithms. Reducing these two degradations is difficult and may sometimes raise other degradations. Such tradeoffs between degradations can be used as a design technique to improve the overall performance.

- *Redundant Work.* Transferring intermediate results is more costly than computation for certain problems. Instead of transferring the results, we may wish to recompute the results. This kind of redundant work is adopted in solving eigenvalue problems [40] where eigenvalues are transferred and corresponding eigenvectors are recomputed. Another kind of redundant work comes from the load balancing: To reach agreement on shared data, an algorithm can either let one processor compute it while all the other processors wait or let all the processors compute it concurrently. Since the former involves waiting and receiving, the later is more favorable. This kind of redundant work is used in solving linear tridiagonal systems [60]. Redundant work decreases the communication overhead by increasing the computation requirement.
- *Reduced Knowledge.* If performing calculations improves the knowledge base of the algorithm, simultaneous calculations cannot make use of the most recent knowledge but must rely on information that is to some extent out of date. This is often the case in iterative methods where each iteration advances the knowledge base. For example, Jacobi algorithms are widely used in multicomputers, while Gauss-Seidel algorithms, which use the latest available information, usually converge faster than the corresponding Jacobi algorithm. Receiving the newest information requires frequent communication and synchronization, making Gauss-Seidel algorithms difficult to implement efficiently on multicomputers.

- *Non-optimal.* With more than one processor computing concurrently, the computational complexity might be misleading, since an algorithm could do more work and still be fast. The algorithm with the lowest computation count is very often sequential in nature. To achieve a high degree of parallelism, we may have to develop parallel algorithms which require more computation than the optimal sequential algorithms. We reduce the uneven allocation degradation by increasing the computation requirement. This increase in computation requirement will lower the performance gain of parallel processing, at least in the sense of absolute speedup (see Chapter 2).

1.2.2 Algorithm Characteristics

Parallel algorithm characteristics have been studied for mapping parallel algorithms into parallel or distributed architectures [30]. In this section, parallel algorithms are characterized from a different point of view. We study how these characteristics degrade the performance of parallel algorithms.

- *Data Parallelism versus functional parallelism.* Parallelism can be achieved by dividing the data among the processors, or by decomposing the algorithm into segments that can be assigned to different processors. We call the former *data parallelism* and the latter *functional parallelism*. Data parallelism is the parallelism which comes from simultaneous operations across a set of data [26]. Thus, for data parallel algorithms each processor executes the same instructions on a different data set. Task partitioning is more flexible and load balancing is relatively easy to achieve. Algorithms based on functional parallelism typically use a small number of processors.
- *Granularity.* The granularity or grain size deals with average subtask size, which is measured in the number of instructions executed. It has a bearing on data allocation, communication requirements, processing capability, and memory requirements. Fine-grain algorithms often require frequent communication. Large-grain algorithms typically reduce the communication/computation ratio and have less communication penalty. This communication penalty reduction makes some computation models, that may be intolerable for fine-grain algorithms, favorable for large-grain algorithms. For example, with fine-grain parallelism, both pipelined Jacobi iteration and Gauss-Seidel iteration are unacceptable. With large-grain parallelism, these methods can achieve a near linear speedup [49], [5].

- *Degree of Parallelism.* We call the number of processes that can operate independently the degree of parallelism. Load balancing will be much more easily achieved for algorithms with a static degree of parallelism. Unfortunately, parallel algorithms with a dynamic degree of parallelism are not uncommon. An example is parallel Gaussian elimination algorithm. To eliminate the lower triangular elements, $n - 1$ independent processes are available for elimination in the first column, $n - 2$ independent processes are available for elimination in the second column, ..., and finally one process is available for elimination in the $n - 1^{th}$ column. Another example is parallel Given Rotation [47]. For an n dimensional matrix, the degree of parallelism of Given Rotation goes up from 1 to $n/2$ and then goes down from $n/2$ to 1.
- *Concurrency and Pipelining.* From the viewpoint of scheduling, a parallel computation can be characterized as either *concurrent* or *pipelined* [29]. Concurrency exploits spatial parallelism by utilizing several processors executing multiple independent tasks simultaneously. Pipelining exploits temporal parallelism in which each processor (called a *stage*) behaves like a filter or transformer which operates on its input data and passes its output data to the succeeding processor as the later's input. A *systolic array* is a typical example of a pipelined computation in a synchronized environment. Through pipelining, communication occurs only between fixed and neighboring stages. Thus a very high ratio of computation to I/O rate will result. A more detailed study on pipeline computation can be found in [34] [49].
- *Data Dependencies.* Data dependencies have a major impact on the design of parallel algorithms. They influence data granularity, degree of parallelism and synchronization. Data dependency is so important that two nontraditional parallel machines, namely *graph reduction* machines and *dataflow* machines, have been proposed in which the order of instruction executions is implied by data dependencies [61]. By representing the operations as vertices and using directed edges to indicate the dependencies, data dependency can be represented by a directed *dependency graph*. Much information can be obtained by coloring the dependency graph. Operations with different colors cannot execute concurrently. From a design point of view, we would like to distribute operations with the different color to the same processor. The multicoloring method [5], which has been successfully employed in the solution of partial differential equations and in image processing, is a good example of grouping the operations according to data dependency.

- *Methodology.* There are a variety of methods for solving systems of equations, usually classified as *direct* and *iterative* methods. Direct methods follow fixed procedures and find the exact solution with a finite number of operations. Iterative methods do not obtain an exact solution in finite steps, they converge to a solution asymptotically. Nevertheless, iterative methods often yield a solution, within acceptable precision, after a relatively small number of iterations. In this case, they are preferred to direct methods. This is usually the case for linear systems when n is very large and the coefficient matrix is sparse, such as the system $Ax = b$ which arises in the discretization of a linear partial differential equation and in many other applications. Iterative methods may also have smaller storage requirements than direct methods when the matrix A is sparse. The computation count of iterative methods depends on the number of iterations, which is problem dependent. *Curve tracing* is a special method for obtaining solutions, in which we start at one end of the curve and want to get to the other end point. The intermediate results along the curve are unimportant. A curve tracing method for solving power flow problems will be presented in Chapter 4. Several newly developed direct methods for solving tridiagonal linear systems will be given in Chapter 5.

1.3 Motivation and Problem Statement

The complex interaction of the many architectural, hardware, and software features of parallel systems results in a larger space of possible performance behavior and an increase in algorithm design complexity. Designing efficient parallel algorithms requires that users to understand the performance characteristics of parallel machines and to modify their algorithm accordingly. These modifications are problem dependent. Therefore, parallel algorithms have had to be fine-tuned case by case to achieve higher performance. The painful, elusive design process has excluded casual users and restricted parallel computers to a rather small professional community. This situation needs to be changed to make parallel computers usable for other scientists.

We would like to reduce the burden of parallel algorithm design and make the design process more systematic. This raises the obvious questions: What are the techniques for developing efficient parallel algorithms? How could these techniques be used on a given application? To answer the first question, Nelson and Snyder [44] have proposed the concept of parallel paradigm and identified several paradigms. Paradigms are recognized high level

methodologies common to many of the effective parallel algorithms. They are important, because they provide good examples and may help users understand parallel computation. However, these paradigms are described verbally and are isolated from each other. How to connect these paradigms with general applications is unknown.

In this dissertation, I approach these two questions from a different angle. I study parallel algorithm design from a general point of view. First, a representation methodology, *structured representation*, is developed. With this representation, most of the frequently used scientific and engineering applications can be represented by simple formulations. These formulations are combinations of some simple data structures, called *parallel computation models*, and provide adequate information about performance degradations. Parallel computation models are the basic building blocks of structured representations. Since both parallel computation models and parallel paradigms are commonly used data structures, it is not surprising that they share some similarities. Some of them even share the same name. A major advantage of parallel computation models over parallel paradigms is that computation models are based on mathematical formulations, and they are the constructing components of general scientific applications. The structured representation of most frequently used scientific and engineering applications consists of computation models. The design techniques used in computation models are the techniques needed for general scientific algorithm design, and the design techniques are used in a similar way.

A mathematical foundation is necessary in moving toward a systematic design methodology for parallel algorithms. Based on that foundation, different applications can be classified and manipulated. Forming structured representation and acquiring computation models are the first step in developing such a foundation. After computation models have been well identified and studied, the performance of an application can be predicted during its design stage, and systematically mapping algorithms onto architectures becomes possible.

Structured representation and computation models are important. They provide a fast way of estimating and understanding the performance of parallel programs in a parallel system. They lead to efficient parallel algorithm design for scientific and engineering applications. They provide guidelines for programming tools and systematic parallel algorithm design methodology. Also, they suggest how each multicomputer should be used in applications and which multicomputer is the best for a given application.

This study focuses on scientific computations. The models are built for scientific applications and examples are chosen from scientific applications. We are interested in scientific applications since scientific and engineering applications are the major driving force behind

parallel processing and the data structures of scientific applications are relatively simple. With some effort, structured representation and computation models could possibly be extended to general non-numerical applications.

1.4 Thesis Organization

This dissertation is organized as follows. In Chapter 2, we study the performance issues of parallel processing. The study focuses on one performance metric, *parallel speedup*. Three models of speedup are studied. They are *fixed-size speedup*, *fixed-time speedup* and *memory-bounded speedup*. Memory-bounded speedup considers memory capacity as an influential factor and provides the quantum of the tradeoff between time and space. Two sets of speedup formulations are derived for these three models. One set requires more information and provides more accurate estimations. Another set considers a simplified case and provides a clear picture of the possible performance gain of parallel processing.

Structured representation is proposed in Chapter 3. Two different classes of data-exchange are identified. One is *regular data-exchange*. Another is *conjunctive regular data-exchange*. Notation is developed to represent these two classes of data-exchanges. A new concept, *compute-exchange computation*, is introduced. By this concept, every parallel computation can be divided into two parts, compute and data-exchange. Therefore, a parallel algorithm can be written in terms of data exchanges and computations, which is called structured representation. Some basic components of structured representation, called computation models, are also studied in Chapter 3. They are *Local Computation Model*, *Global-Exchange Computation Model*, *Compute-Aggregate-Broadcast Computation Model*, *Divide-and-Conquer Computation Model*, *Domain Decomposition Computation Model*, *Pipeline (Ring) Computation Model*, and *Recursive Doubling Computation Model*.

Parallel algorithms may be modified for different reasons to achieve high performance. The modifications, however, mainly follow two approaches: explore the inherent parallelism of the application to increase the concurrency and take advantage of the underlying architecture to reduce the communication overhead. Two applications are studied to illustrate how structured representation and computation models can be used in efficient algorithm design. The power flow problem is studied in Chapter 4 to demonstrate how structured representation can help in application-driven algorithm design. In Chapter 5, using the design process of linear tridiagonal solvers, we show how the parallel architecture will influence the choice of computation models. The power flow problem determines how electrical power generation

and distribution should be changed with customer demand, so that the power system can be operated safely. We adopt the homotopy mathematical method to solve the power flow problem. Our first approach is an algorithm using the local computation model. Then, we change to a global-exchange computation. After learning more about the homotopy method, we modify our algorithm into a compute-aggregate-broadcast computation. Solving linear tridiagonal systems is a fundamental problem of scientific computation. We have developed tridiagonal solvers on a first generation NCUBE/7 hypercube multicomputer. To take advantage of the hypercube architecture, we change our design from compute-aggregate-broadcast computation to global-exchange computation. Then we change to a hybrid computation, which is a combination of two computation models. The last one gives the best performance. All the algorithms are implemented on an NCUBE multicomputer and the results can be found in the chapters that follow.

A new performance prediction methodology is presented in Chapter 6. This methodology adopts a divide-and-combine strategy. For a given application we write down its structured representation and find out the contained computation models. We then use the well-studied computation models to predict the performance of the given application. Examples are given and the performance measurements of identified computation models are studied. Instead of studying the performance on a given architecture, the performance is studied from a dynamic point of view. We study the influence of problem size on performance. Chapter 7 gives a summary of my major contributions and the direction of future research.

Chapter 2

PERFORMANCE METRIC OF PARALLEL ALGORITHMS

To study efficient parallel algorithm design we first have to understand the performance metrics. For sequential machines, elapsed time and memory space are the metrics by which algorithms are measured. With parallel machines, the performance measurement becomes more difficult. In addition to time and space, the performance parameters for parallel algorithms include the number of processors available, communication overhead and the inherent parallelism of the given application. For this reason, despite the fact that parallel processing has become a common approach for achieving high performance, performance evaluation techniques of parallel processing are weak. There is no well-established metric to measure the performance gain. This weakness is one of the reason which leads to confusion and limits the growth of parallel computation. In this chapter, we shall study issues of performance metrics and seek a better understanding of one metric, *parallel speedup*.

The most commonly used performance metric for parallel processing is *speedup*, which gives the performance gain of parallel processing versus sequential processing. However, with different emphases, speedup has been defined differently. One definition focuses on how much faster a problem can be solved with N processors. Thus, it compares the best sequential algorithm with the parallel algorithm under consideration. This definition is referred to as *absolute speedup*. Absolute speedup has two different definitions, one which considers machine resources and one which does not. In the first definition, speedup is defined as the ratio of elapsed time of the best sequential algorithm on one processor over the elapsed time of the parallel algorithm on N processors. In the second definition, the absolute speedup is defined as the ratio of elapsed time of the best sequential algorithm on the fastest sequential machine over the elapsed time of the parallel algorithm on the parallel machine [47]. Another type of speedup, called *relative speedup*, deals with the inherent parallelism of the parallel algorithm under consideration. It is defined as the ratio of the elapsed time of the parallel algorithm on one processor over the elapsed time of the parallel algorithm on N processors. Relative speedup is used because that the performance of parallel algorithms varies with the number of available processors. Comparing the algorithm itself with different numbers of

processors gives information on the variations and the degradations of parallelism. Absolute speedup and relative speedup are two commonly used speedup measures. Other definitions of speedup also exist. For instance, considering the construction costs of processors, a cost-related speedup is defined in [4].

Among all of the defined speedups, relative speedup is probably the one which has had the most influence on parallel processing. Two well known speedup formulations have been proposed based on relative speedup. One is *Amdahl's law* [2] and another is *Gustafson's scaled speedup* [20]. Both of these two speedup formulations use a single parameter, the sequential portion of a parallel algorithm. They are simple and give much insight into the degradations of parallelism. Amdahl's law has a fixed problem size and is interested in how fast the response time could be. It suggests that massively parallel processing may not gain high speedup. Under the influence of Amdahl's law many parallel computers have been built with a small number of processors. Gustafson [20] approaches the problem from another point of view. He fixes the response time and is interested in how large a problem could be solved within this time. The argument of Gustafson is that the problem size should be increased to meet the available computation power for better results. Experimental results based on his argument show that speedup can increase linearly with the number of processors available [22].

In this chapter we shall provide a careful study of relative speedup and reserve the word speedup for relative speedup, unless explicit state otherwise. We first study three models of speedup, *fixed-size speedup*, *fixed-time speedup* and *memory-bounded speedup*. All three models are based on relative speedup. With both uneven allocation and communication overhead considered, general speedup formulations will be derived for all three models. When communication overhead is not considered and the workload only consists of sequential and perfectly parallel portions, the simplified fixed-size speedup is Amdahl's law; the simplified fixed-time speedup is Gustafson's scaled speedup; and, with one more parameter, the simplified memory-bounded speedup contains both Amdahl's law and Gustafson's speedup as its special cases. Therefore, from different points of view, the three models of speedup are unified.

2.1 Preliminary

The parallelism in an application can be characterized in different ways for different purposes, such as *data dependency graph* [61], *task precedence graph* [11], *Petri Net* [41] and *average*

parallelism [15]. For simplicity, speedup formulations generally use very few parameters and consider very high level characterizations of the parallelism. In our study we consider two main degradations of parallelism, *uneven allocation* and *communication latency*. The former degradation is application dependent. The latter degradation depends on both the application and the parallel computer under consideration. To give an accurate estimate, both of the degradations need to be considered. Uneven allocation is measured by *degree of parallelism*.

Definition 1 *The degree of parallelism of a program is an integer which indicates the number of processors that are busy computing at a particular instant in time, given an unbounded number of available processors.*

The degree of parallelism is a function of time. By drawing the degree of parallelism over the execution time of an application, a graph can be obtained. We refer to this graph as the *parallelism profile*. Software tools are available to determine the parallelism profile of large scientific and engineering applications [36]. Figure 2.1 is the parallelism profile of a hypothetical divide-and-conquer computation (see Section 3.2). By accumulating the time spent at each degree of parallelism, the profile can be rearranged to form the *shape* (see Figure 2.2) of the application [53].

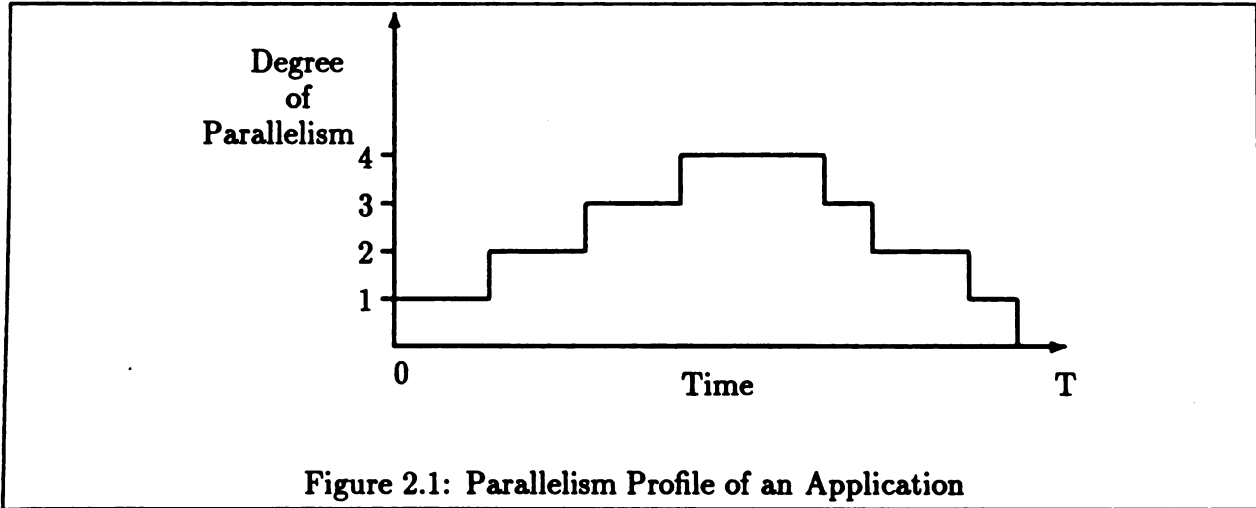


Figure 2.1: Parallelism Profile of an Application

Definition 2 *The average parallelism is the average number of processors that are busy during the execution of the program in question, given an unbounded number of available processors.*

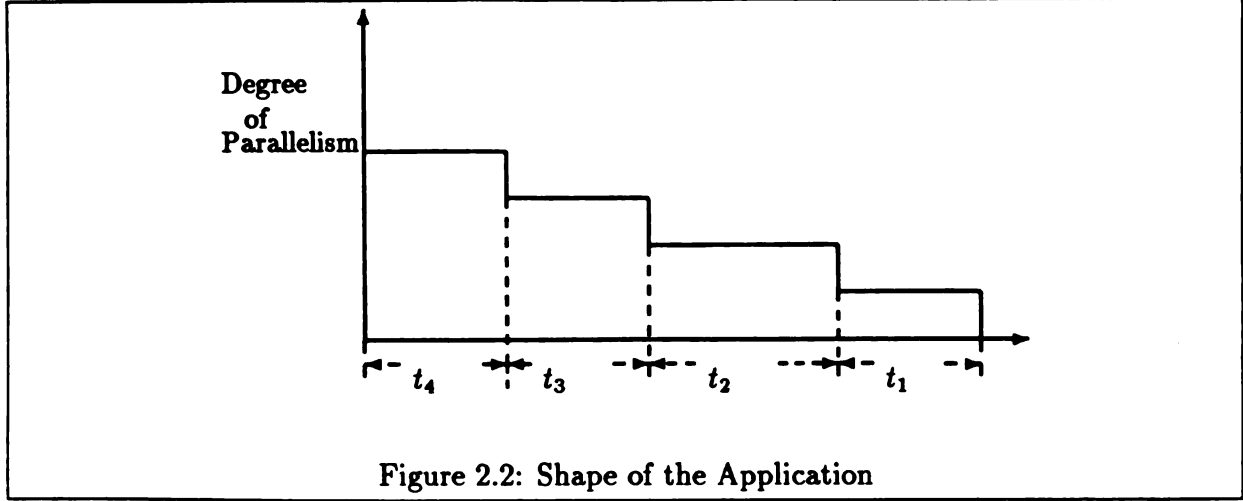


Figure 2.2: Shape of the Application

By definition, average parallelism is the ratio of the total service demand to the execution time with an unbounded number of available processors. This is equal to the speedup, given unbounded number of available processors and without considering communication latency. Therefore, average parallelism can be defined equivalently as follows [15].

Definition 3 *The average parallelism is the speedup, given an unbounded number of available processors and without considering communication latency.*

Let W be the amount of work (computation) of an application. Let W_i be the amount of work executed with degree of parallelism i , and m be the maximum degree of parallelism. Thus, $W = \sum_{i=1}^m W_i$. The execution time for computing W_i with a single processor will be

$$t_i(1) = \frac{W_i}{\Delta}, \quad (2.1)$$

where Δ is the computing capacity of each processor. If there are i processors available, the execution time will be

$$t_i(i) = \frac{W_i}{i\Delta}.$$

With an infinite number of processors available, the execution time will be

$$t_i = t_i(\infty) = \frac{W_i}{i\Delta} \quad \text{for } 1 \leq i \leq m.$$

Therefore, without considering communication latency, the response time on a single processor and on an infinite number of processors will be

$$T(1) = \sum_{i=1}^m \frac{W_i}{\Delta}, \quad (2.2)$$

$$T(\infty) = \sum_{i=1}^m \frac{W_i}{i\Delta}. \quad (2.3)$$

The speedup will be

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^m \frac{W_i}{\Delta}}{\sum_{i=1}^m \frac{W_i}{i\Delta}} = \frac{\sum_{i=1}^m W_i}{\sum_{i=1}^m W_i/i}. \quad (2.4)$$

The average parallelism, A , can be computed in terms of t_i ,

$$A = \sum_{i=1}^m \frac{it_i}{\sum_{i=1}^m t_i} = \frac{\sum_{i=1}^m it_i}{\sum_{i=1}^m t_i}. \quad (2.5)$$

Notice that t_i is the time for executing W_i . When an unbounded number of processors are available, $t_i = \frac{W_i}{i\Delta}$. Substituting $t_i = \frac{W_i}{i\Delta}$ into Eq. (2.5), we have

$$A = \frac{\sum_{i=1}^m it_i}{\sum_{i=1}^m t_i} = \frac{\sum_{i=1}^m W_i}{\sum_{i=1}^m W_i/i} = S_\infty. \quad (2.6)$$

This gives a formal proof for the equivalence of Definition 2 and Definition 3. Average parallelism is a very important factor for speedup and efficiency. It has been carefully examined in [15]. S_∞ gives the best possible speedup based on the inherent parallelism of an algorithm. There are no machine dependent factors considered. With only a limited number of available processors and with communication latency considered, the speedup will be less than the best speedup, S_∞ . If there are N processors available and $N < i$, then some processors have to do $\frac{W_i}{i} \lceil \frac{i}{N} \rceil$ work and the rest of the processors will do $\frac{W_i}{i} \lfloor \frac{i}{N} \rfloor$ work. In this case, assuming W_i and W_j cannot be solved simultaneously for $i \neq j$, the elapsed time will be

$$t_i(N) = \frac{W_i}{i\Delta} \lceil \frac{i}{N} \rceil$$

and

$$T(N) = \sum_{i=1}^m \frac{W_i}{i\Delta} \lceil \frac{i}{N} \rceil. \quad (2.7)$$

The speedup is

$$S_N = \frac{T(1)}{T(N)} = \frac{\sum_{i=1}^m W_i}{\sum_{i=1}^m \frac{W_i}{i} \lceil \frac{i}{N} \rceil}. \quad (2.8)$$

Communication latency is an important factor contributing to the complexity of a parallel algorithm. Unlike degree of parallelism, communication latency is machine dependent. It depends on the communication network, the routing scheme, and the adopted switching

technique. For instance, the switching technique used in first generation multicomputers is *store-and-forward*. Second generation multicomputers adopt *circuit switching* or *wormhole routing* switching techniques. These new switching techniques reduce the communication cost considerably. Let Q_N be the communication overhead when N processors are used in parallel processing; the general speedup becomes

$$S_N = \frac{T(1)}{T(N)} = \frac{\sum_{i=1}^m W_i}{(\sum_{i=1}^m \frac{W_i}{i} \lceil \frac{i}{N} \rceil) + Q_N}. \quad (2.9)$$

2.2 Models of Speedup

In the last section we developed a general speedup formula and showed how the number of processors and degradation parameters will influence the performance. However, speedup is not only dependent on these parameters. It is also dependent on how we view the problem. With different points of view, we will get different models of speedup and will get different speedup formulations.

One viewpoint emphasizes shortening the time it takes to solve a problem by parallel processing. With more and more computation power available, the problem can be solved in less and less time. With more processors available, the system will provide a fast turnaround time and the user will have a shorter waiting time. A speedup formulation based on this philosophy is called a *fixed-size speedup*. In the previous section, we adopted fixed-size speedup implicitly. Equation (2.9) is the general speedup formula for fixed-size speedup. Fixed-size speedup is suitable for many applications.

For some applications we may have a time limitation, but we may not want to obtain the solution in the quickest way. If we have more computation power, we may want to increase the problem size, carry out more operations, get a more accurate solution, and keep the execution time unchanged. This viewpoint leads to a new model of speedup, called *fixed-time speedup*. Many scientific and engineering applications can be represented by some partial differential equations, which can be discretized for different choices of grid spacing. Coarser grids demand less computation, but finer grids give more accurate solutions. If more accurate solutions are desired, this kind of application will fit the fixed-time speedup model. One good example is weather forecasting. With more computation power, we may not want to give the forecast earlier. Rather, we may wish to add more factors into the weather model – increasing the problem size and getting a more accurate solution – thus providing a more precise forecast.

For fixed-time speedup the workload is scaled up with the number of processors available. Let W'_i be the amount of scaled up work executed with degree of parallelism i and m' be the maximum degree of parallelism of the scaled up problem when N processors are available. In order to keep the same turnaround time as the sequential version, we must have

$$\sum_{i=1}^m W_i = \sum_{i=1}^{m'} \frac{W'_i}{i} \lceil \frac{i}{N} \rceil + Q_N$$

Thus, the general speedup formula for fixed-time speedup is

$$S'_N = \frac{\sum_{i=1}^{m'} W'_i}{\sum_{i=1}^{m'} \frac{W'_i}{i} \lceil \frac{i}{N} \rceil + Q_N} = \frac{\sum_{i=1}^{m'} W'_i}{\sum_{i=1}^m W_i}. \quad (2.10)$$

From our experience in using multicomputers, we have found that memory capacity plays a very important role in performance. Existing multicomputers do not support virtual memory, and memory is distributed and associated with each node. The memory associated with each node is relatively small. When solving an application with one processor, the problem size is more often bounded by the memory limitation than by the execution time limitation. With more nodes available, instead of keeping the execution time fixed, we may want to meet the memory capacity and increase the execution time. In general, the question is, if you want to increase the problem size, do you have enough memory for the size increase? If you do have adequate memory space for the size increase, and after the problem size is increased to meet the time limit you still have memory space available, do you want to increase the problem size further by using this unused memory space and to get an even better solution? For memory-bounded speedup the answer is yes. Like fixed-time speedup, memory-bounded speedup is a scaled speedup. The problem size is scaled up with system size. The difference is that in fixed-time speedup the execution time is the dominant factor and in memory-bounded speedup the memory capacity is the dominant factor. Most of the applications which fit fixed-time speedup will fit memory-bounded speedup when more accurate solutions are the goal. A good application for memory-bounded speedup is simulation. If we simulate a nuclear power plant, obtaining an accurate solution will probably be the highest priority.

With memory capacity considered as a factor of performance, the requirement of solving an application contains two parts. One is the computation requirement, which is the workload, and another is the memory requirement. For a given application, these two requirements are related to each other, and the workload can be seen as a function of the memory requirement. Let M represent the memory requirement and let g represent the function, we have $W = g(M)$, or $M = g^{-1}(W)$, where g^{-1} is the inverse function of g .

Under different architectures the memory capacity will change differently with the number of processors available. For multicomputers, the memory capacity increases linearly with the number of nodes available. If $W = \sum_{i=1}^m W_i$ is the workload for sequential execution, $W^* = \sum_{i=1}^{m^*} W_i^*$ is the scaled workload when N processors are available, m^* is the maximum degree of parallelism of the scaled problem, then the memory limitation for multicomputers can be stated as: *the memory requirement for any active node is less than or equal to $g^{-1}(\sum_{i=1}^m W_i)$* . Here the main point is that the memory occupation on each node is fixed. Equation (2.11) is the general speedup formula for memory-bounded speedup.

$$S_N^* = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \lceil \frac{i}{N} \rceil + Q_N} \quad (2.11)$$

2.3 Simplified Models of Speedup

Three general speedup formulations have been proposed for three models of speedup. These formulations contain both uneven allocation and communication latency degradations. They are closer to actual speedup and give better upper bounds on the performance of parallel algorithms. On the other hand, these formulations are problem dependent and difficult to understand. They give more detailed information for each application, but lose the global view of the possible performance gain. In this section, we study a simplified case for speedup, which is the special case studied by Amdahl and Gustafson. We do not consider the communication overhead, so $Q_N = 0$, and we assume that the allocation only contains two parts, a sequential part and a perfectly parallel part. That is $W_i = 0$, for $i \neq 1$ and $i \neq N$. We also assume that the sequential part is independent of the system size, i.e., $W_1 = W'_1 = W_1^*$.

Under this simplified case, the general fixed-size speedup formulation Eq.(2.9) becomes

$$S_N = \frac{W_1 + W_N}{W_1 + \frac{W_N}{N}}, \quad (2.12)$$

which is known as Amdahl's law. From Eq.(2.12) and Fig. 2.3 we can see when the number of processors increases the load on each processor decreases. Eventually, the sequential part will dominate the performance and the speedup is bounded by $\frac{W_1 + W_N}{W_1}$.

Under the simplified conditions, $\sum_{i=1}^m W_i = W_1 + W_N$ and $\sum_{i=1}^{m'} \frac{W'_i}{i} \lceil \frac{i}{N} \rceil + Q_N = W'_1 + \frac{W'_N}{N}$. Therefore, for fixed-time speedup, we have $W_1 + W_N = W'_1 + \frac{W'_N}{N}$. Since $W_1 = W'_1$, we have $W_N = \frac{W'_N}{N}$. That is $W'_N = NW_N$. Equation (2.10) becomes

$$S'_N = \frac{\sum_{i=1}^{m'} W'_i}{\sum_{i=1}^{m'} W_i} = \frac{W'_1 + W'_N}{W_1 + W_N} = \frac{W_1 + NW_N}{W_1 + W_N}. \quad (2.13)$$

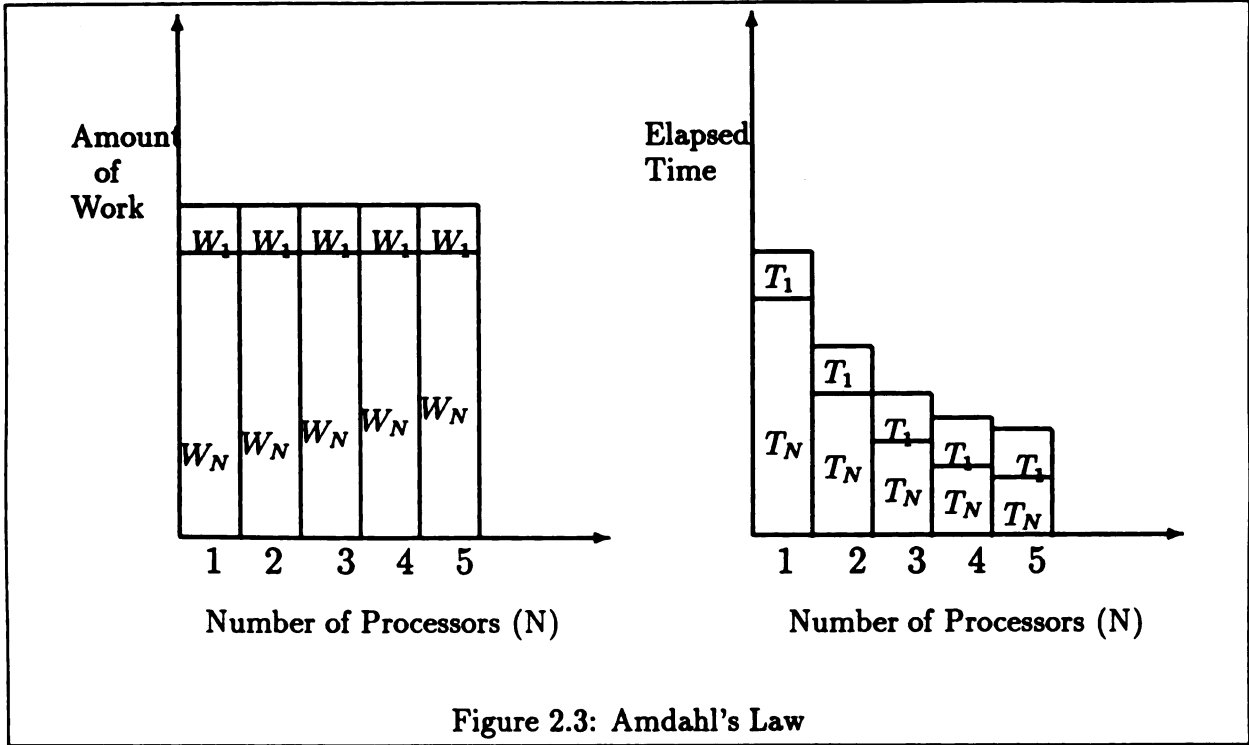


Figure 2.3: Amdahl's Law

The simplified fixed-time speedup formula Eq.(2.13) is Gustafson's scaled speedup, which was proposed by Gustafson in 1988 [20]. From Eq.(2.13) we can see that the parallel portion of the application is scaled up linearly with the system size, and the speedup increases linearly with the system size. The relation of workload and elapsed time for Gustafson's scaled speedup is depicted in Figure (2.4), where T_1 is the execution time for the sequential portion of work. T_N is the execution time for the parallel portion of load.

We need some preparation before deriving the simplified formulation for memory-bounded speedup.

Definition 4 A function g is a semihomomorphism if there exists a function \bar{g} such that for any real number c and any variable x , $g(cx) = \bar{g}(c)g(x)$.

One class of semihomomorphism functions is the power function $g(x) = x^b$, where b is a rational number. In this case, \bar{g} is the same as the function g . Another class of semihomomorphism functions is the single term polynomial $g(x) = ax^b$, where a is a real constant and b is a rational number. For this kind of semihomomorphism functions, $\bar{g}(x) = x^b$, which is not the same as $g(x)$.

The sequential portion of the workload, W_1 , is independent of the system size. If we do not consider the influence of memory on the sequential portion we have the following theorem:

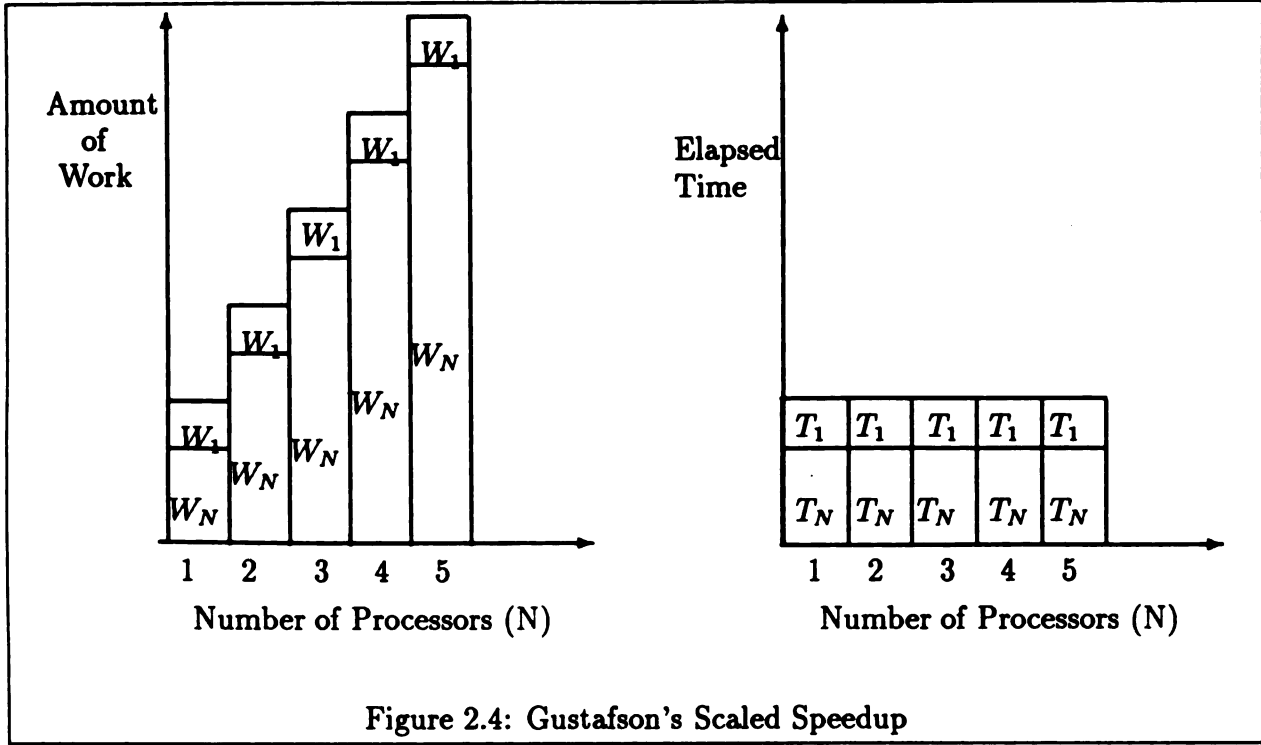


Figure 2.4: Gustafson's Scaled Speedup

Theorem 1 *If $W = g(M)$ for some semihomomorphism function g , $g(cx) = \bar{g}(c)g(x)$, then, with all data being shared by all the available processors and using all the available memory space, the simplified memory-bounded speedup is*

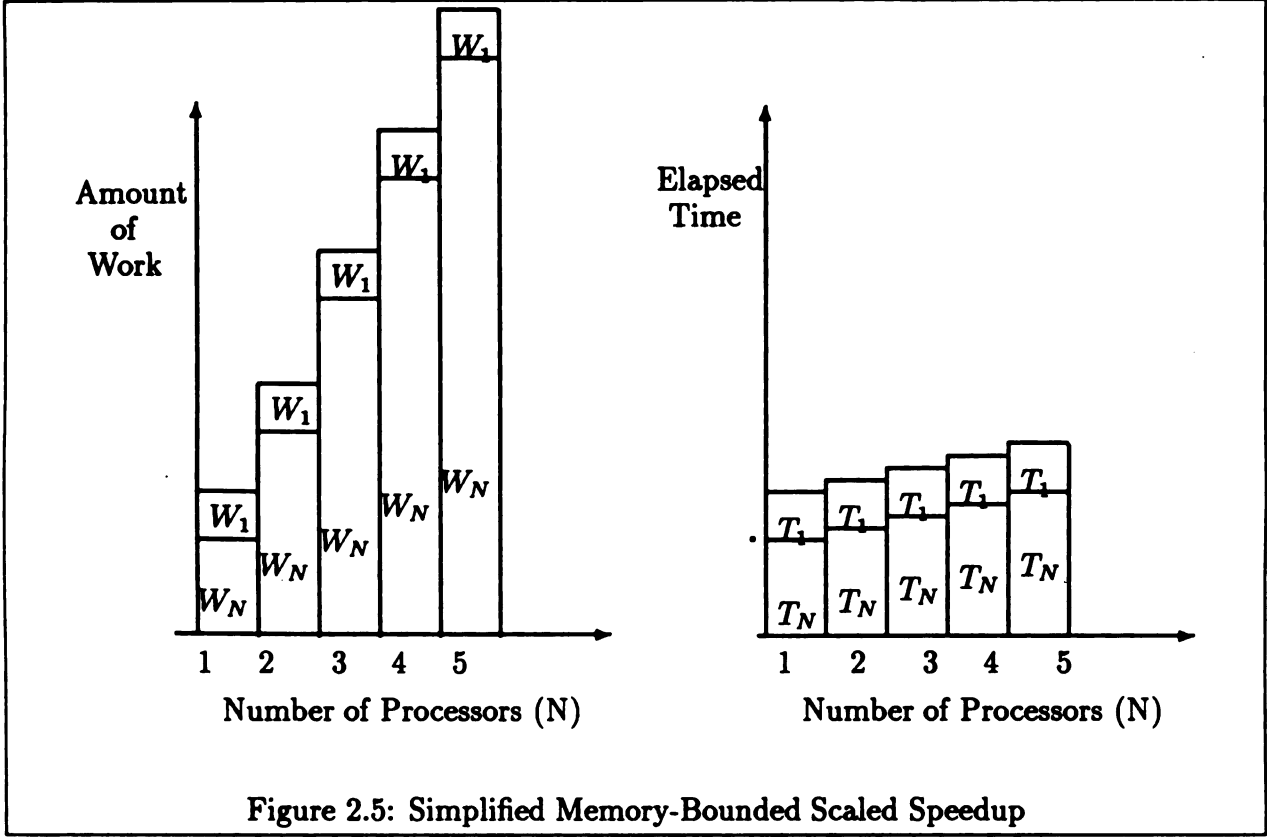
$$S_N^* = \frac{W_1 + \bar{g}(N)W_N}{W_1 + \frac{\bar{g}(N)}{N}W_N} \quad (2.14)$$

Proof: As mentioned before, W_N is the parallel portion of the workload when one processor is used. Let the memory requirement of W_N be M , $W_N = g(M)$. M is the memory requirement when one node is available. With N nodes available, the memory capacity will increase to NM . Using all of the available memory, for the scaled parallel portion W_N^* , $W_N^* = g(NM) = \bar{g}(N)g(M)$. Therefore, $W_N^* = \bar{g}(N)W_N$ and

$$S_N^* = \frac{W_1 + W_N^*}{W_1 + W_N^*/N} = \frac{W_1 + \bar{g}(N)W_N}{W_1 + \frac{\bar{g}(N)}{N}W_N} \quad (2.15)$$

□

In the proof of Theorem (1), we claimed that $W_N^* = g(NM) = \bar{g}g(M)$. This claim is true under two assumptions: 1) the data is shared by all available processors, and 2) all the available memory space is used for better solutions. A computation with the first property is called global computation. Equation (2.14) is the simplified memory-bounded speedup



for global computation when memory is fully used. In general, data may be duplicated on different nodes and the available memory may not be fully used for increased problem size. Replacing the function \bar{g} by a general function G , that is $W_N^* = G(N)W_N$, a more generalized theorem will be

Theorem 2 If $W_N^* = G(N)W_N$ for some function G , then

$$S_N^* = \frac{W_1 + G(N)W_N}{W_1 + \frac{G(N)}{N}W_N}. \quad (2.16)$$

Proof: If $W_N^* = G(N)W_N$ for some integer N , then

$$S_N^* = \frac{W_1^* + W_N^*}{W_1^* + \frac{W_N^*}{N}} = \frac{W_1 + G(N)W_N}{W_1 + \frac{G(N)}{N}W_N}. \quad (2.17)$$

□

Equation (2.16) will be referred to as *simplified memory-bounded (SMB) scaled speedup*. SMB scaled speedup is determined by the function $G(N)$, which represents how the change of memory will influence the change of problem size. When the problem size is independent of

the system size, the problem size is fixed, $G(N) = 1$. In this case, SMB scaled speedup is the same as Amdahl's law, i.e., Eq.(2.16) and Eq.(2.12) are equivalent. The *local computation model* is one computation model studied in Section 3.2. In the local computation model, when more processors are available, work will be replicated on these available processors. Computation is done locally on each node, and communication between nodes is not required. In this case, when memory is increased N times, the workload also increases N times, i.e., $G(N) = N$. In this case, SMB scaled speedup is the same as Gustafson's scaled speedup. SMB scaled speedup contains both Amdahl's law and Gustafson's scaled speedup as its special cases. For most scientific and engineering applications, the computation requirement increases faster than the memory requirement. For these applications, $\bar{g}(N) > N$ and memory-bounded speedup will likely give a higher speedup than fixed-time speedup.

The proposed scaled speedup formulation, Eq.(2.16), may not be easy to fully understand at first glance. Here we use matrix multiplication to illustrate it. A matrix often represents some discretized continuum. Enlarging the matrix size generally will lead to a more accurate solution for the continuum. For matrices with dimension n , the computation requirement of matrix multiplication is $2n^3$ and the memory requirement is $3n^2$ (roughly). Thus,

$$W_N = 2n^3, \quad M = 3n^2.$$

Writing W_N as a function of M , we have

$$W_N = 2 \left(\frac{M}{3} \right)^{\frac{3}{2}}.$$

This means that

$$W_N = g(M) = 2 \left(\frac{M}{3} \right)^{\frac{3}{2}}, \quad \bar{g}(N) = N^{\frac{3}{2}}. \quad (2.18)$$

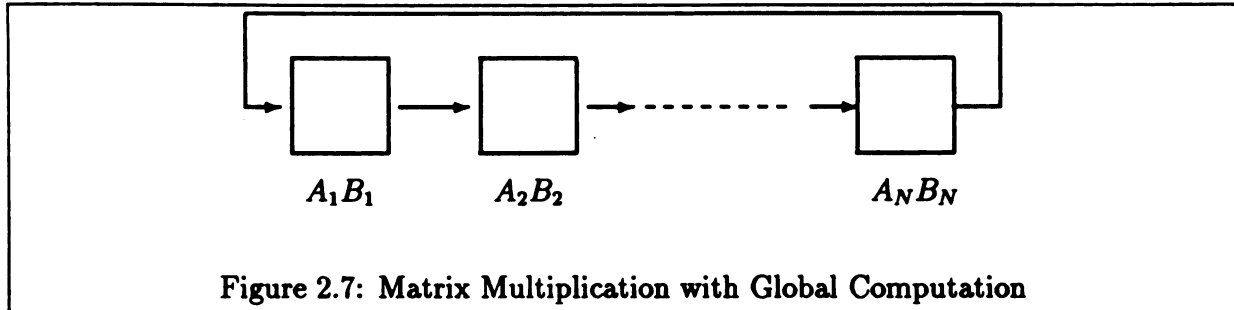
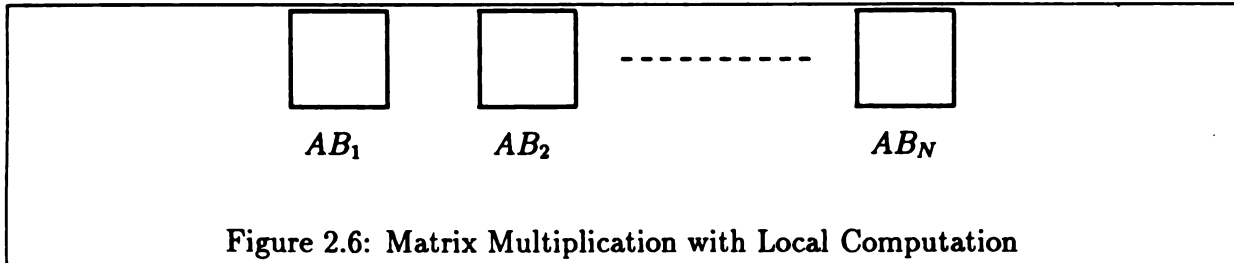
The simplified memory-bounded speedup for global computation will be

$$S_N^* = \frac{W_1 + N^{\frac{3}{2}} W_N}{W_1 + N^{\frac{1}{2}} W_N}. \quad (2.19)$$

Global computation uses distributed local memories as a large shared memory. All the data is distributed and shared. When these local memories are used locally without sharing, the computation is local computation and $W_N^* = Ng(M)$. This means that $\bar{g} = N$. The speedup is

$$S_N^* = \frac{W_1 + NW_N}{W_1 + W_N},$$

which is Gustafson's scaled speedup. For matrix multiplication $C = AB$, let A_i be the i^{th} row of A , $i = 1, \dots, n$, and let B_j be the j^{th} column of B , $j = 1, \dots, n$. The local computation and global computation of the matrix multiplication are shown in Figure (2.6) and (2.7), respectively. In global computation, the rows of matrix A are rotated after each row, column multiply.



We have studied two cases of memory-bounded scaled speedup, global computation and local computation. Most of the applications are some combination of these two computation styles. Data are distributed in one part and duplicated in the other part. The duplication may be required by inherent properties of the given application, or may be added in deliberately to reduce communication. Speedup formulations for these applications depend on the ratio of the global and the local computation. Deriving a speedup formulation for these combined applications is difficult, not only because we are facing a more complicated situation, but also because of the uncertainty of the ratio. The duplicated part might not increase with system size. It might increase, but with a speed which is different from the increasing speed of the global part. Also, an application may start as global computation, but, when the computation power increases, duplication may be added in as a part of the effort for a better solution. In general, $G(N)$ is application dependent. We derive $G(N)$ for a special case as an example. The structure of this derivation can be used as a guideline for general applications.

Lemma 1 *If function g is a semihomomorphism function, $g(cx) = \bar{g}(c)g(x)$; and g^{-1} exists*

and is also a semihomomorphism, $g^{-1}(cx) = h(c)g^{-1}(x)$ for some function h , then \bar{g} has an inverse and $\bar{g}^{-1} = h$. •

Proof: Since

$$cy = g[g^{-1}(cy)] = g[h(c)g^{-1}(y)] = \bar{g}[h(c)]g[g^{-1}(y)] = \bar{g}[h(c)]y,$$

we have

$$\bar{g}[h(c)] = c \quad \text{for any real number } c \quad (2.20)$$

Also, since

$$cy = g^{-1}[g(cy)] = g^{-1}[\bar{g}(c)g(y)] = h[\bar{g}(c)]y,$$

we have

$$h[\bar{g}(c)] = c \quad \text{for any real number } c \quad (2.21)$$

By Eq.(2.20) and Eq.(2.21), the function \bar{g} has an inverse and $\bar{g}^{-1} = h$. □

Theorem 3 Assume $W = g(M)$ for some semihomomorphism function g , where $g(cM) = \bar{g}(c)g(M)$, g inverse exists and is a semihomomorphism. If the workload is scaled up to meet the time limitation with global computation first and the rest of the unused memory space is then used to increase the problem size further with local computation, we have

$$G(N) = (1 + \bar{g}[1 - \frac{\bar{g}^{-1}(N)}{N}])N. \quad (2.22)$$

Proof: By the fixed-time speedup, after the number of nodes changes from 1 to N , the parallel portion of work will increase from W_N to NW_N (see Figure 2.4). The storage requirement is given by the function g^{-1} . For operation requirement NW_N , the memory requirement is $g^{-1}(NW_N) = \bar{g}^{-1}(N)g^{-1}(W_N)$.

Let M represent the size of the memory associated with each node which can be used for parallel processing. Then, when the number of nodes equals 1, the total memory available is M , which is equal to $g^{-1}(W_N)$. When the number of nodes equals N , the total memory available changes to NM . We first fix the execution time and increase the problem size to meet the time limitation. After the fixed-time scale up, the unused memory space is the difference between current available memory and current memory requirement, which equals

$$NM - g^{-1}(NW_N) = NM - \bar{g}^{-1}(N)g^{-1}(W_N) = NM - \bar{g}^{-1}(N)M = (N - \bar{g}^{-1}(N))M.$$

The unused space at each node is

$$\frac{[N - \bar{g}^{-1}(N)]M}{N} = [1 - \frac{\bar{g}^{-1}(N)}{N}]M.$$

The problem size can be further scaled by using this unused memory space. The further scaled computation on each node is given by the function g , and it is equal to

$$g\left(1 - \frac{\bar{g}^{-1}(N)}{N}\right)M = \bar{g}\left(1 - \frac{\bar{g}^{-1}(N)}{N}\right)g(M) = \bar{g}\left(1 - \frac{\bar{g}^{-1}(N)}{N}\right)W_N \quad (2.23)$$

Therefore, the computation on each node becomes

the original operation on each node + the operation increase on each node

$$= W_N + \bar{g}\left(1 - \frac{\bar{g}^{-1}(N)}{N}\right)W_N = \left[1 + \bar{g}\left(1 - \frac{\bar{g}^{-1}(N)}{N}\right)\right]W_N, \quad (2.24)$$

and, for the scaled parallel computation W_N^* ,

$$W_N^* = N \left[1 + \bar{g}\left(1 - \frac{\bar{g}^{-1}(N)}{N}\right)\right]W_N.$$

Thus, we have

$$G(N) = N[1 + \bar{g}(1 - \frac{\bar{g}^{-1}(N)}{N})]. \quad (2.25)$$

□

Figure 2.8 depicts the speedup difference among the fixed-sized model, the fixed-time model and the memory-bounded model. The function, \bar{g} , used in Figure 2.8 is the function for matrix multiplication, $\bar{g}(N) = N^{\frac{3}{2}}$. As most matrix computations have the same function $\bar{g}(N) = N^{\frac{3}{2}}$, the speedup relation depicted by Figure 2.8 is in general true for a large class of applications.

2.4 Comparison Study

It is known that the performance of parallel processing is influenced by the inherent parallelism of the application, by the computation power and by the memory capacity of the parallel computing system. However, how these three factors are related to each other, and how they influence the performance of parallel processing generally is unknown. Discovering

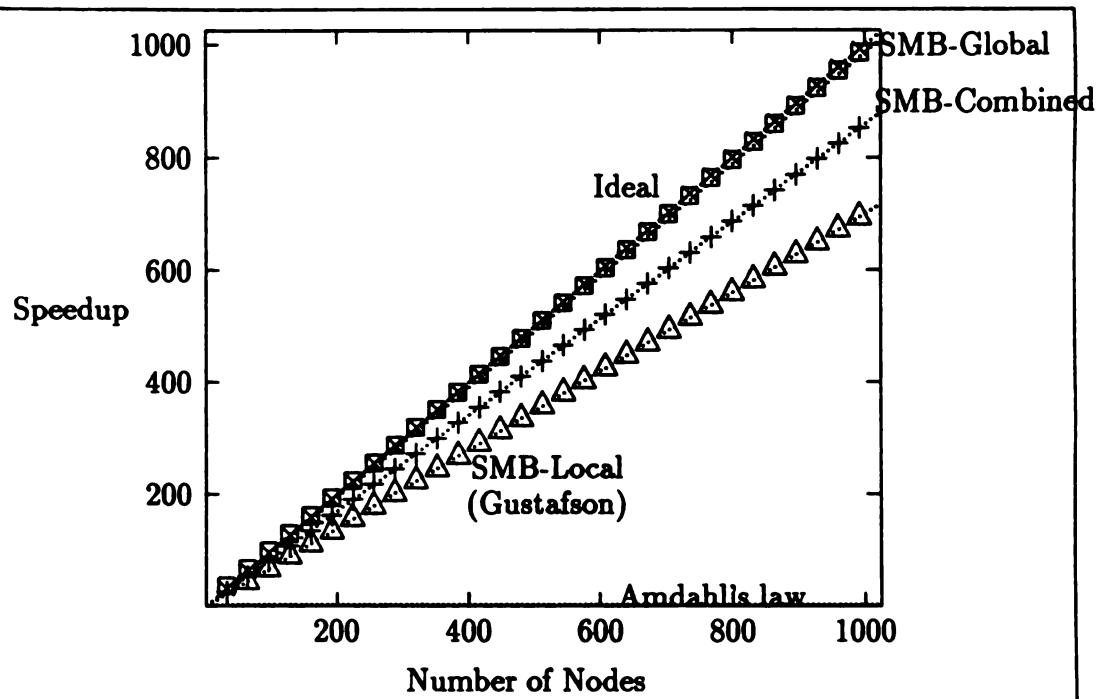


Figure 2.8: Amdahl's law, Gustafson's speedup and SMB speedup

where $W_1 = 0.3$ and $\bar{g}(N) = N^{\frac{3}{2}}$

the answers to these unknowns is very important for designing efficient parallel algorithms and for constructing high performance parallel systems. In this paper one model of speedup, *memory-bounded speedup*, is carefully studied. This model is simple, and it contains all of these three factors as its parameters. It shows the degradations and the possible performance gain of parallel computation.

As part of the study on performance, two other models of speedup have also been studied. They are *fixed-size speedup* and *fixed-time speedup*. Two sets of speedup formulations have been derived for these two models of speedup and memory-bounded speedup. Formulations in the first set are general speedup formulas. These formulas contain more parameters and provide more accurate information. The second set of formulations only considers a special, simplified case. These formulations give the performance in principle and lead to a better understanding of parallel processing. The simplified fixed-size speedup is *Amdahl's law*, the simplified fixed-time speedup is *Gustafson's scaled speedup*, and the simplified memory-bounded speedup contains both Amdahl's law and Gustafson's speedup as special cases. Amdahl's law suggests that the sequential portion of the workload will dominate the performance when the number of processors is large. Gustafson's scaled speedup claims that the influence of the sequential portion is independent of system size. Simplified memory-bounded scaled speedup declares that the sequential fraction will change with the system size. Since the computation requirement increases faster than the memory requirement for most applications, the sequential fraction could be reduced when the number of processors increases.

The three models of speedup, *fixed-size speedup*, *fixed-time speedup* and *memory-bounded speedup*, are based on different viewpoints and are suitable for different classes of applications. Applications exist which do not fit any of the models of speedup, but satisfy some combination of the models.

Chapter 3

PARALLEL COMPUTATION MODELS FOR SCIENTIFIC COMPUTING

With more than one processor operating concurrently, parallel processing enlarges the range of possible performance and makes efficient algorithm design more important. We have studied performance metrics in the previous chapter. In this chapter, we focus on designing parallel algorithms, with emphasis on methodology and analysis.

One of the factors that makes parallel algorithm design difficult is the lack of guidelines. Different applications are rarely related to each other in the design process. Therefore, they have to be handled one by one in an *ad hoc* fashion. The experience of experts hardly benefits general users. In Section 3.1, we propose a representation system, called structured representation, for scientific and engineering applications. With this representation, the similarities between applications become obvious and applications can be compared, classified, and manipulated based on their structures. Developing structured representation is the first step toward the ultimate goal, replacing the *ad hoc* development of parallel applications with a rational methodology based on analysis and measurement.

A parallel algorithm cannot be efficient without considering the architectural aspects of the underlying multiprocessor. This is especially true for multicomputers where communication overhead is a major consideration. Mapping an application onto an architecture is both application dependent and architecture dependent. To lead to a systematic mapping, the basic building blocks of structured representation are identified and studied in Section 3.2. With structured representation, an application can be decomposed into a set of computational models which can be mapped onto the architecture using predefined strategies. We can modify our design by changing certain computation models. In this way, structured design and rethinking become possible. Casual users can develop efficient algorithms based on experts' experience. A general guideline for efficient algorithm design is provided.

3.1 Structured Representation

Parallelism can be achieved by dividing a given application into pieces, called *subtasks*, and solving these pieces concurrently. Ideally, these subtasks can be solved independently, where the exchange of intermediate result is negligible. Some scientific and engineering applications have this nice, *easy parallelism* property. For these applications, it is natural to solve each of the subtasks locally on a different processor. This model of computation is called a *local computation model*.

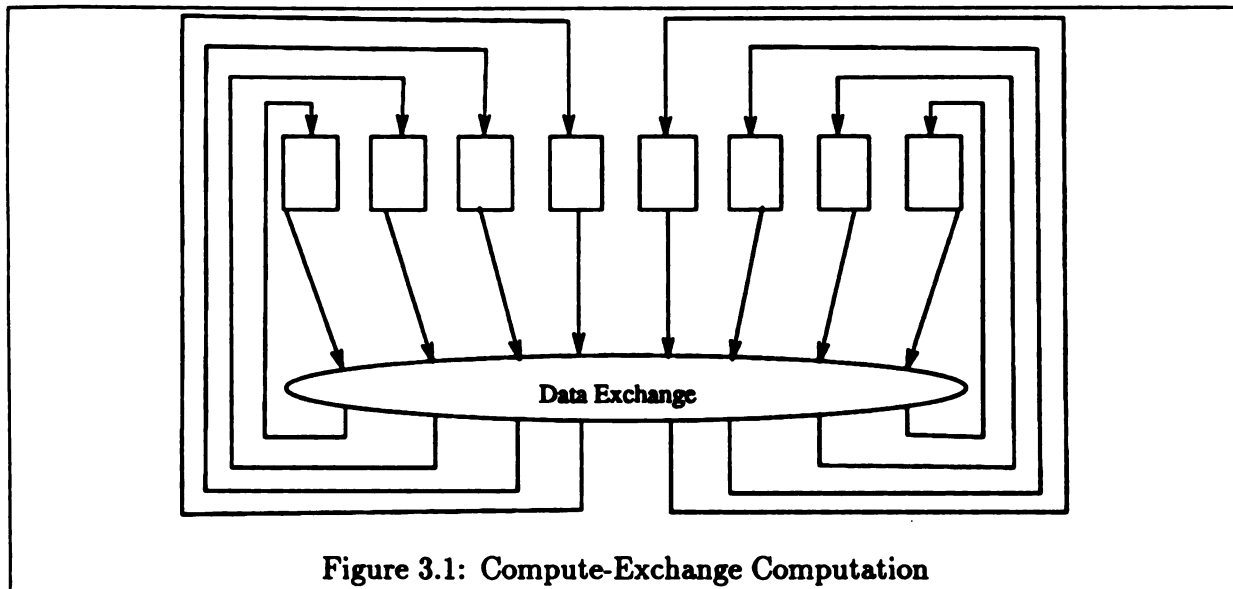


Figure 3.1: Compute-Exchange Computation

The design of local computation algorithms is straightforward. Unfortunately, most applications do not have this easy parallelism property [1]. For most applications communication is necessary for exchanging data and coordinating activities. Although various asynchronous techniques have been designed to reduce the communication overhead, most communication must be achieved in a synchronous fashion, that is the receiving node must receive the communicated message before continuing. This synchronous communication requirement makes efficient algorithm design very difficult. The load needs to be balanced between synchronizations and special care has to be taken to reduce the communication overhead. Figure 3.1 depicts the general parallel computation pattern with synchronous communication considered. It shows that the solving process consists of two phases, compute and data-exchange. These two phases occur alternatively and repetitively, and, therefore, form the *compute-exchange computation*. The data-exchange phase involves communication between compute phases. The communication patterns vary largely from application to application, and may be represented by a notation. To simplify this notation, we restrict ourselves to certain

classes of communication, which are large enough for our purposes – describing the most frequently used scientific and engineering applications.

A processor sending a message in a communication is called a *sender*. A processor receiving message in a communication is called a *receiver*. A processor could be both sender and receiver in a communication. A *graph* $G(V, E)$ is a structure which consists of a set of *vertices* $V = \{v_1, v_2, \dots\}$ and a set of *edges* $E = \{e_1, e_2, \dots\}$ [17]. If we let processors in a communication be vertices in a graph and add directed edge (v, w) from v to w if processor v sends a message to processor w ; a digraph (directed graph) is formed. This digraph is called the *communication digraph*. Following the notations of graph theory [17], the outdegree of a vertex v is the number of edges which have v as their start-vertex. In other words, the outdegree of a vertex v is equal to the number of destinations that v sends its message to. For this reason we also call the outdegree of a vertex the *degree of a sender*. The indegree of a vertex and the degree of a receiver are defined similarly. The *degree of a receiver* is the number of sources from which it receives messages.

Definition 5 *A regular communication is a communication in which all senders have the same degree and all receivers have the same degree.*

For a given undirected graph, if for every two vertices u and v there exists a path whose starting vertex is u and whose ending vertex is v , then the graph is connected. A connected subgraph $G(V', E')$ is a connected component if there is no other connected subgraph containing $G(V', E')$ as its proper subgraph.

The underlying (undirected) graph of a digraph is the graph resulting from the digraph if the direction of the edges is ignored. A connected component of a digraph is the corresponding subdigraph of the connected component of its underlying graph.

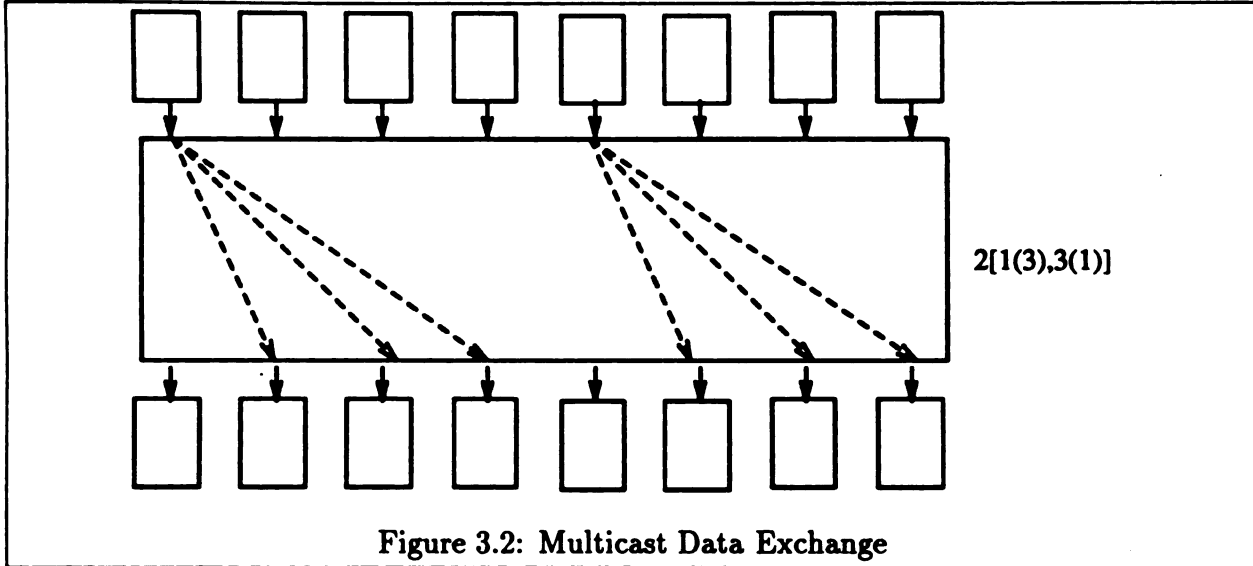
A connected component of the communication digraph is called a pattern of the communication.

Definition 6 *A regular communication is a regular data-exchange if it consists of one or more copies of the same pattern.*

By our definition, the communication requirement of a regular data-exchange is given by the number of patterns it contains. The pattern of a communication is described by the number of senders and the number of receivers in this pattern. The complexity of each sender and of each receiver is given by its degree. Thus, a regular data-exchange can be represented using five parameters as

$$P[S(D), R(d)], \quad (3.1)$$

where P is the number of instance of the pattern, S is the number of senders in each instance of the pattern, and D is the degree of the senders. Similarly, R is the number of receivers in each instance of the pattern, and d is the degree of each receiver. An example of using this notation for presenting communication is given in Figure 3.2. Notation (3.1) describes a communication by five parameters. Since broadcast is not provided in neither first generation nor second generation multicomputers, messages must be sent one at a time. The number of times messages are sent and received is the dominant factor in communication cost. Notation (3.1) indicates the characteristics of a communication. More information may be needed when implementation is considered.



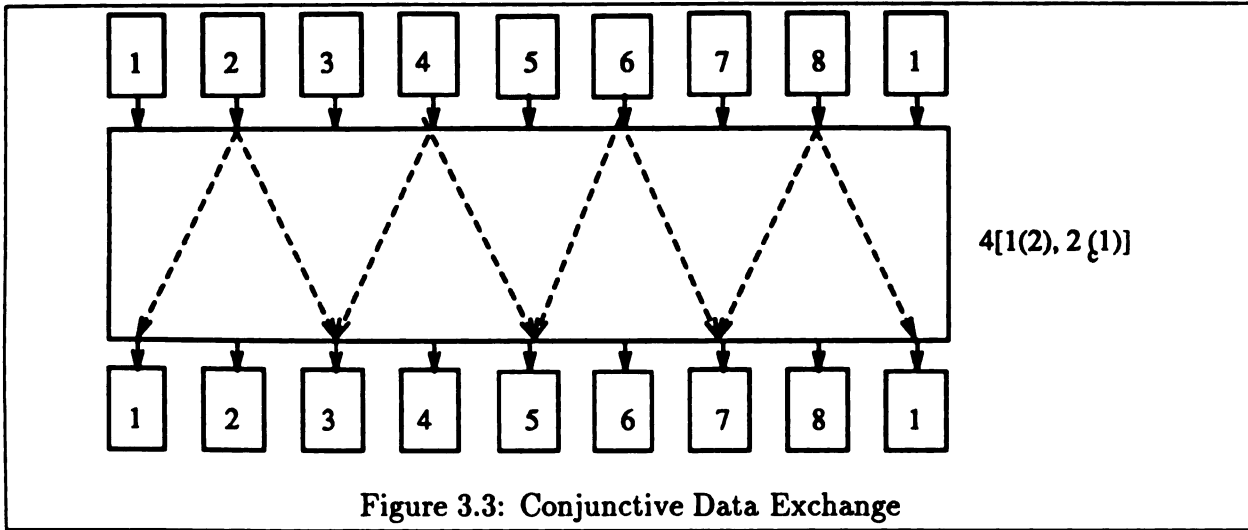
The second class of data-exchange which we want to identify is called *conjunctive regular data-exchange*. We use the same five parameters to identify conjunctive regular data-exchange. The difference between regular data-exchange and conjunctive regular data-exchange is that in conjunctive regular data-exchange the patterns are not disjoint, they conjoin one another. Consider two special cases: conjunction at the sender side only and conjunction at the receiver side only. We have two general notations,

$$P[S(D), R_c(d)], \quad (3.2)$$

and

$$P[S_c(D), R(d)], \quad (3.3)$$

where the subscript, c , points out which side has conjunctions. An example of conjunctive regular data-exchange is given in Figure 3.3, in which the receiver side has conjunction.



A graph $G'(V', E')$ is a partition graph of $G(V, E)$ if $G'(V', E')$ is formed by splitting a subset of vertices $\{v_1, v_2, \dots, v_n\} \in V$ into two subsets of vertices as $\{v'_1, v'_2, \dots, v'_n\}$ and $\{v''_1, v''_2, \dots, v''_n\}$, where v'_i, v''_i are the vertices formed by splitting vertex v_i , and having edges (v'_i, v'_j) and (v''_i, v''_j) when edge (v_i, v_j) exists in graph $G(V, E)$. These divided vertices are called partition vertices. Figure 3.4 shows two partition graphs of the given graphs. With this terminology, conjunctive regular data-exchange can be defined more mathematically as follows.

Definition 7 *A regular communication is a conjunctive regular data-exchange if one of its partition graphs consists of one or more copies of the same pattern. If all partition vertices are senders, the conjunctive regular data-exchange is a sender conjunctive regular data-exchange. If all partition vertices are receivers, the conjunctive regular data-exchange is a receiver conjunctive regular data-exchange.*

Since a regular communication patterns could have more than one partition graph which consists of one or more copies of the same pattern, a conjunctive regular data-exchange could have more than one notation.

Once the data-exchange phase has been identified in an application, we can describe the application in terms of data-exchange. An application might have different data-exchange phases. Writing these data-exchange phases together in order by using the Σ symbol and adding in the compute phases, we have a formula, called a *structured representation*, for each

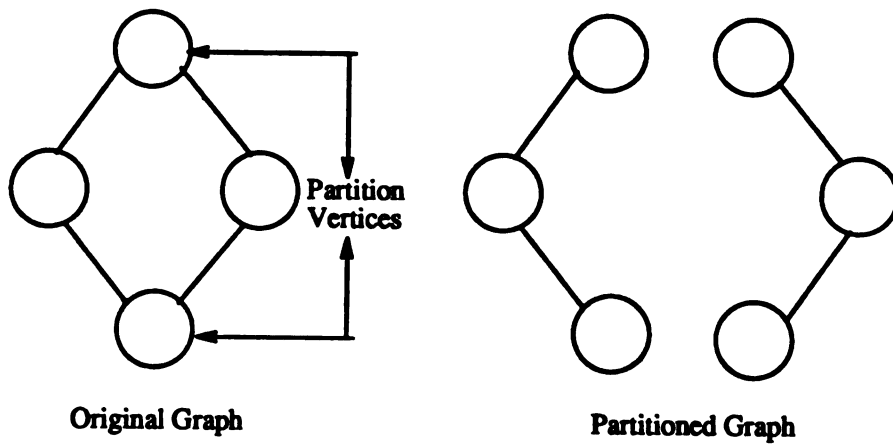
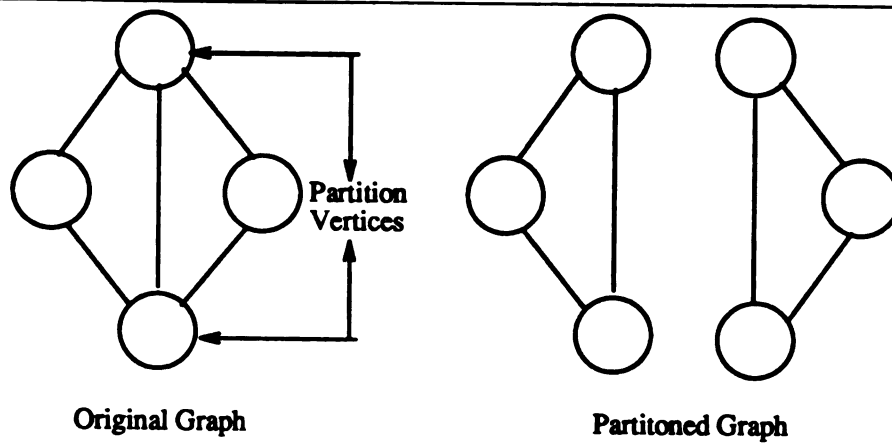
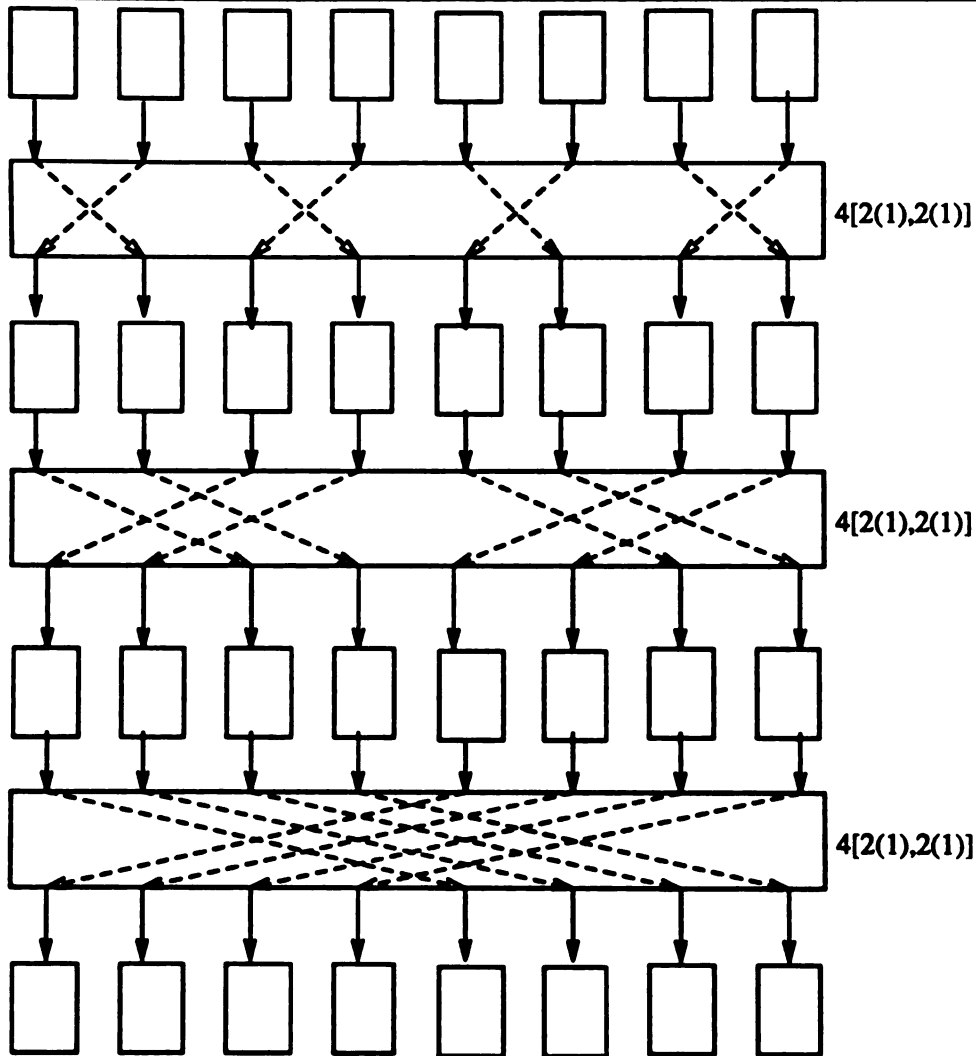


Figure 3.4: Partitioning of Graphs

application. Figure 3.5 shows how to represent the Fast Fourier Transform (FFT) computation in terms of regular data-exchange. The communication divides the computation into layers. The total computation depends on the number of layers as well as the computation requirement at each layer. X_i is the computation work on each processor between data-exchange phase $i - 1$ and i , if we have even allocation. X_i is the computation work of the processor which has the largest workload among all the working processors in the compute phase i , if we have uneven allocation. Notice that X_i is not equal to W_i which is defined in Chapter 2. W_i is the total workload with degree of parallelism i .



$$\sum_{i=1}^k (2^{k-i} [2(1), 2(1)] + X_i) \quad k = \log(N) = 3$$

Figure 3.5: FFT (Butterfly) Computation

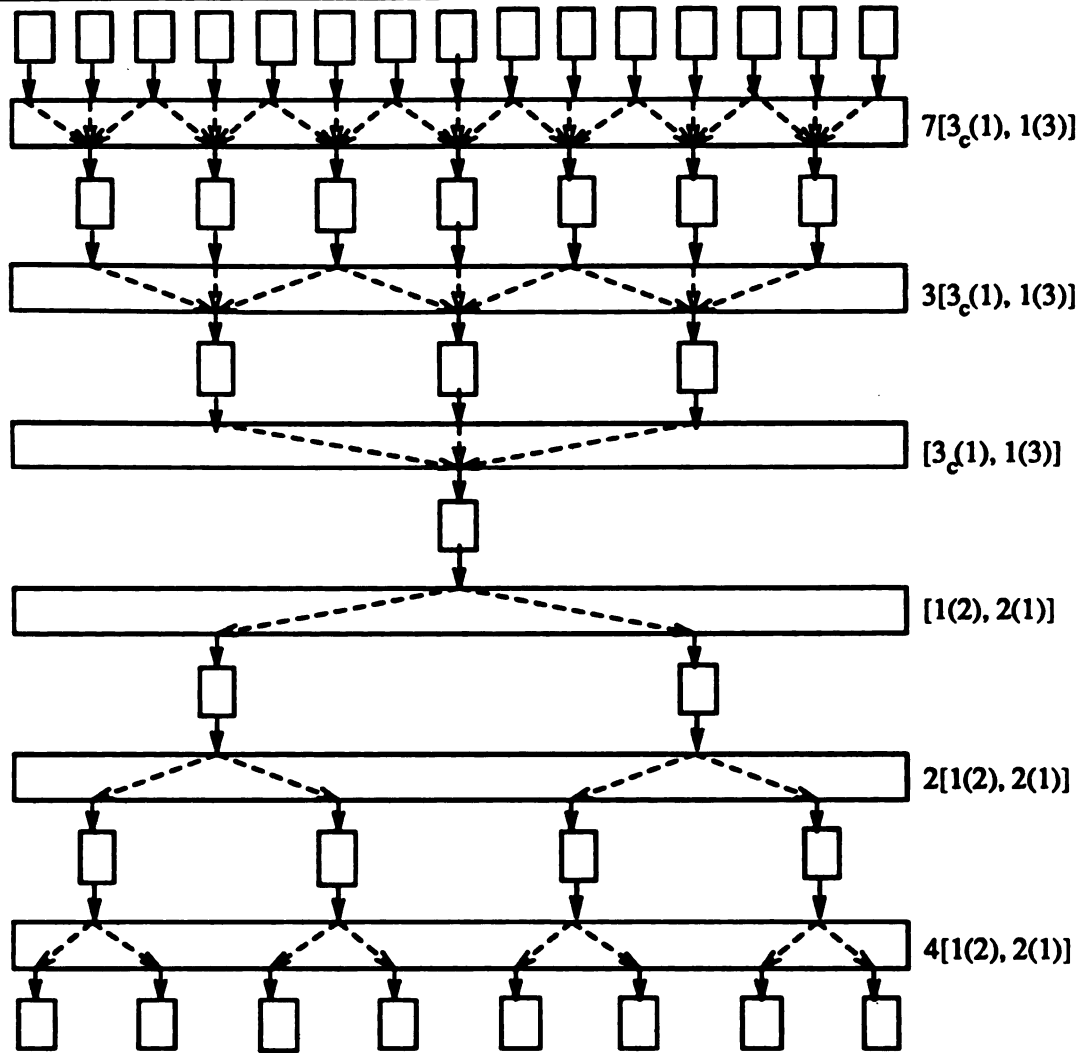
We can see from Figure 3.5 that different communications could have the same data-exchange representation. The reason is that our notation is a high level notation. It provides the communication complexity for the data-exchanges. It does not contain detailed information about how the communication takes place.

We assume that the processors which participate in the data communication also participate in the computation, and the processors which do not participate in the data communication do not participate in the computation. With this assumption, the second application, odd-even cyclic reduction (see Figure 3.6), shows how the structured representation illustrates the uneven allocation degradation. Using the data-exchange information $P[S(D), R(d)]$ at data-exchange phase i , we can see that there are $P \times S$ processors working at compute phase $i - 1$ and there are $P \times R$ processors working at compute phase i . If the data-exchange is a conjunctive regular data-exchange, $P[S_c(D), R(d)]$, the number of working processors at compute phase $i - 1$ will be $P \times (S - 1) + 1$. These can be confirmed by observing Figure 3.6. Similarly, for conjunctive regular data-exchange $P[S(D), R_c(d)]$, the number of working processors at compute phase i will be $P \times (R - 1) + 1$. Structured representation provides information about uneven allocation at an abstract level. It provides the number of processors which participate in the computation at each compute phase. It does not provide the workload information for each of the working processors.

Odd-even cyclic reduction is a commonly used method for scientific applications. A well known parallel algorithm for tridiagonal linear systems is based on odd-even cyclic reduction [33]. From Figure 3.6 we can see that the odd-even cyclic reduction application contains two different structures. The upper half of Figure 3.6 is one structure and the lower half is another structure. This is a common phenomenon of scientific applications. Most of the frequently used scientific applications are combinations of a few simple structures, which we call *computation models*. The information in computation models can be used in general applications. Studying computation models will lead to a general algorithm design guideline for scientific applications. In the next section we will present some of the identified computation models.

3.2 Parallel Computation Models

Structured representation is a precise and perspicuous medium to express computation models. With structured representation, computation models can be defined at different levels for different purposes. In our study, we follow the conventional research of parallel paradigms



$$\sum_{i=1}^{k-1} ((2^{k-i} - 1)[3_c(1), 1(3)] + X_i) + \sum_{i=k}^{2^{(k-1)}} (2^{i-k}[1(2), 2(1)] + X_i)$$

$$k = \lceil \log(N) \rceil = 4$$

Figure 3.6: Odd-Even Cyclic Reduction

and define the computation models from a coarse point of view. We identify the computation models based on design techniques, communication patterns, characteristics, sources of degradation, and define them precisely with structured representations.

3.2.1 Local Computation Model

Parallelism can be achieved by dividing a given problem into pieces and solving these pieces concurrently. In general, these pieces need to communicate in order to exchange data and coordinate their activities. However sometimes the problem can be divided into almost completely independent subtasks where the exchange of the intermediate results is negligible. Some computational problems have this ideal parallel property. The use of the *Local Computation Model* is natural in solving these problems on a multicomputer. The characteristic of the local model is asynchrony.

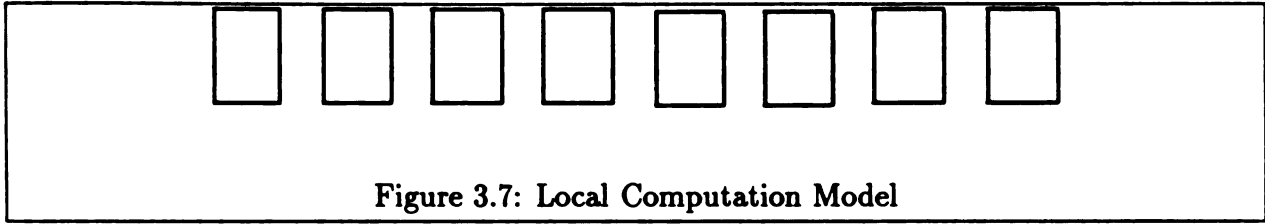


Figure 3.7: Local Computation Model

Local computation for algorithms with data parallelism can be illustrated by matrix multiplication. Given $n \times n$ matrices A and B , we want to compute their product, C . Assuming N is less than n , we partition matrix B by columns evenly into N submatrices B_0, B_1, \dots, B_{N-1} . Each node is loaded with one submatrix B_i ($i = 0, \dots, N-1$) together with the whole matrix A . The resultant products $C_i = A \times B_i$ are then obtained on each processor independently. The common input A shows another characteristic of the local computation model. The processors give different parts of the output, but they may share some input.

The *Homotopy method* [9] (see also Section 4.3) solves systems of equations by curve tracing. To solve a non-trivial equation $P(Z) = 0$, a homotopy function $H(Z, t)$ is defined as

$$H(Z, t) = (1 - t)Q(Z) + tP(Z) \quad t \in [0, 1] \quad (3.4)$$

It can be seen from Eq.(3.5) that when $t = 0$, $H(Z, 0) = Q(Z)$ and when $t = 1$, $H(Z, 1) = P(Z)$. Mathematical results have shown that if $P(Z) \in C^2$, $Q(Z) \in C^2$, then $P(Z) = 0$ can be solved by tracing the solution set of the the following equation

$$H(Z, t) = 0, \quad (3.5)$$

i.e., the equation $P(Z) = 0$ can be solved by

- solving the equation $Q(Z) = 0$,
- following the solution curves of Eq.(3.5) from $t = 0$ to $t = 1$.

Each solution of $Q(Z)$ will lead to a solution curve of $H(Z, t)$. By broadcasting the homotopy function to each of the processors and by starting them from different solutions of $Q(Z)$, the solution curves of Eq. 3.5 can be traced locally on each processor. There is no communication needed for coordination. The curves are traced by following discrete points. The amount of computation on each point and the points themselves is determined at run time.

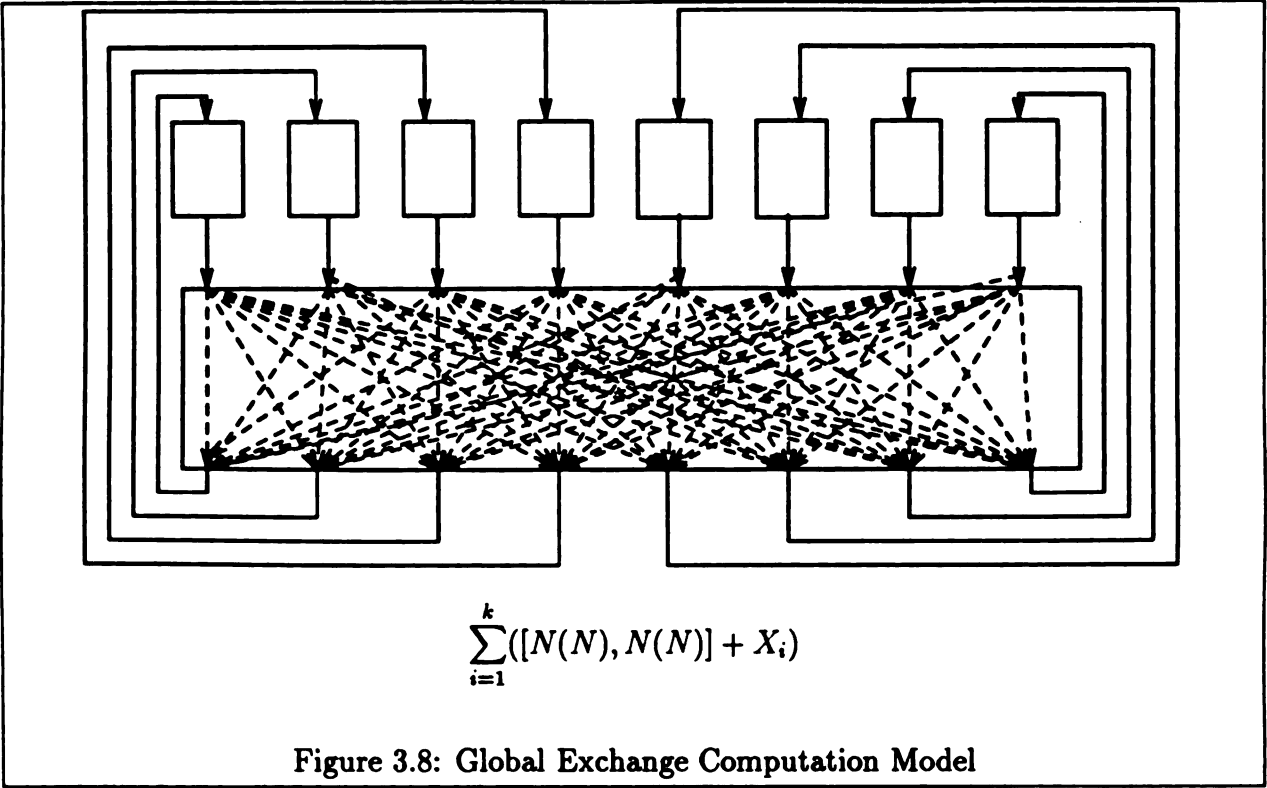
The local computation model is a special case of compute-exchange computation where the data-exchange phase does not exist.

3.2.2 Global-Exchange Computation Model

In the global-exchange computation model each processor executes a computational phase and then performs a total data-exchange. At each data-exchange phase, each processor sends messages to all the other processors and receives messages from all the other processors. The communication requirement at each data-exchange phase is high. The Jacobi iterative method for solving linear systems is an application which is based on global-exchange computation. The characteristic of global-exchange computation is synchronism. If one processor does not finish its computation in the current compute phase, then no other processor can enter the next compute phase. This synchronization requirement and the high communication overhead make designing efficient algorithms for global-exchange computation applications very difficult. Figure 3.8 depicts the global-exchange computation model and gives the structured representation for this model.

3.2.3 Compute-Aggregate-Broadcast Computation Model

Compute-Aggregate-Broadcast Computation Model (CAB) was first proposed in [44]. It is composed of three basic phases bearing those names: a compute phase which performs the computation, an aggregation phase which combines local data into one or a few global values, and a broadcast phase which returns global information back to each processor. This model is characterized by synchronism communication. This synchronism requirement may be caused by data dependency or is demanded by global control for better performance. The compute phase varies widely from application to application. For a multicomputer we would like the

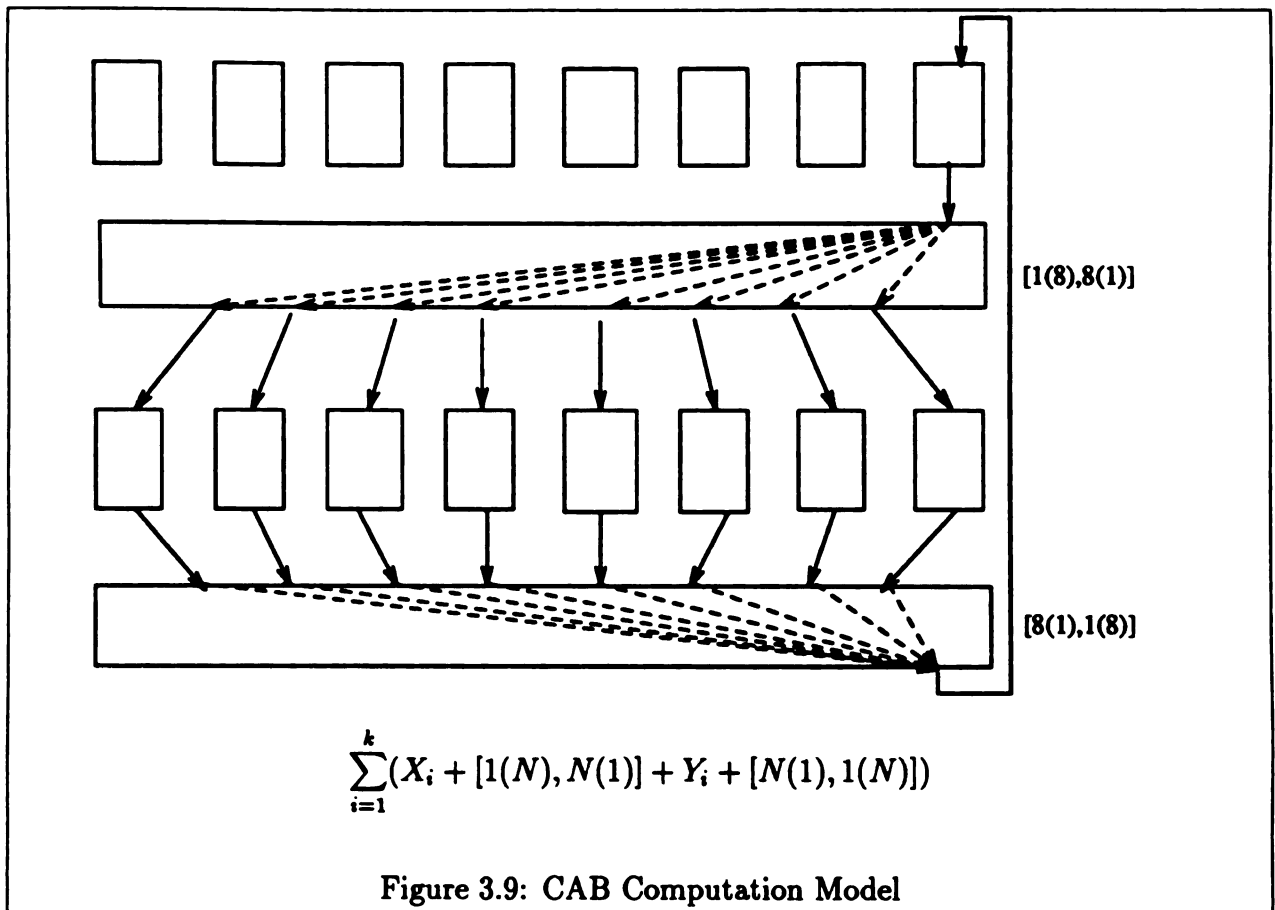


compute phase to be dominant. The aggregation phase can be a tree-based computation that combines data from the processors, then produces a single global information, such as the convergent signal for solving PDE problems, or it can be a data gathering communication pattern, in which all the computed results are sent to a special node that processes them and coordinates the next round of computation.

Both the global exchange computation model and the compute-aggregate-broadcast computation model are commonly used for iterative methods. Comparing these two methods, the CAB computation mode has a lower communication requirement. However, also, it has degradations caused by uneven allocation. At some compute phases only one processor works and all the others are idle.

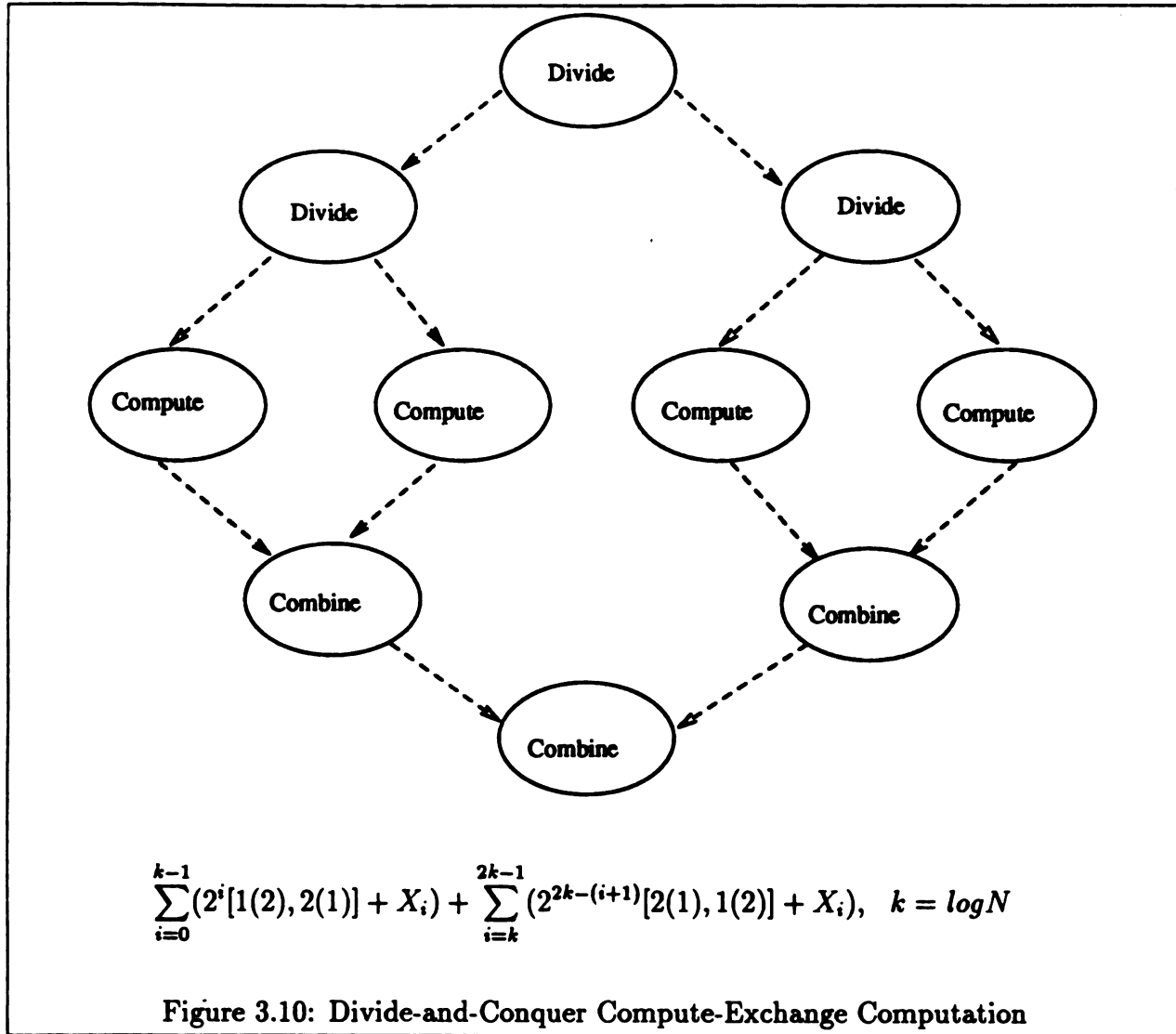
3.2.4 Divide-and-Conquer Computation Model

Divide-and-Conquer is a well known technique in sequential algorithm development [7]. In divide-and-conquer computation, a problem is divided into two or more smaller problems. These smaller problems are solved and their results are combined to give the final solution. The Divide-and-Conquer computation model can be divided into two subclasses, *Recursive Divide-and-Conquer* and *Partitioning*. Recursive divide-and-conquer uses the smaller



problems as a smaller instance of the original problem and does the divide and conquer concurrently. We refer to divide-and-conquer in one stage as partitioning. This model is also the basis for several classes of parallel algorithms, including a number of sorting and searching algorithms [48]. In general the recursive divide-and-conquer computation model has two shortcomings for multicomputers:

1. Synchronization: The algorithm requires synchronization at each merge stage.
2. Degree of Parallelism: The algorithm has a dynamic degree of parallelism.



The following result shows the beauty of recursive divide-and-conquer technique [5].

Proposition 1 *A non-singular triangular linear system $Ax = b$ can be solved in $O(\log^2 n)$ time, using n^3 processors, where n is the dimension of A .*

Proof: It is sufficient to prove that A^{-1} can be found in $O(\log^2 n)$ time with n^3 processors.

We partition the matrix A into blocks:

$$A = \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix}, \quad (3.6)$$

where A_1 is of size $\lceil n/2 \rceil \times \lceil n/2 \rceil$. Since A_1 and A_3 are lower triangular matrices. Moreover, it is easily shown that

$$A^{-1} = \begin{bmatrix} A_1^{-1} & 0 \\ -A_3^{-1}A_2A_1^{-1} & A_3^{-1} \end{bmatrix}, \quad (3.7)$$

Based on the above decomposition, we obtain the following algorithm.

Given an $n \times n$ triangular matrix A :

1. If $n = 1$, then A^{-1} can be found directly.
2. If $n > 1$, then partition A as indicated above and do the following:
 - (a) Find the inverse of A_1 and A_3 concurrently. (Notice that A_1 and A_3 are lower triangular matrices, they can be inverted by using the same algorithm recursively.)
 - (b) Multiply A_2 by A_3^{-1} on the left to obtain $A_3^{-1}A_2$.
 - (c) Right-multiply the result of (b) by A_1^{-1} .

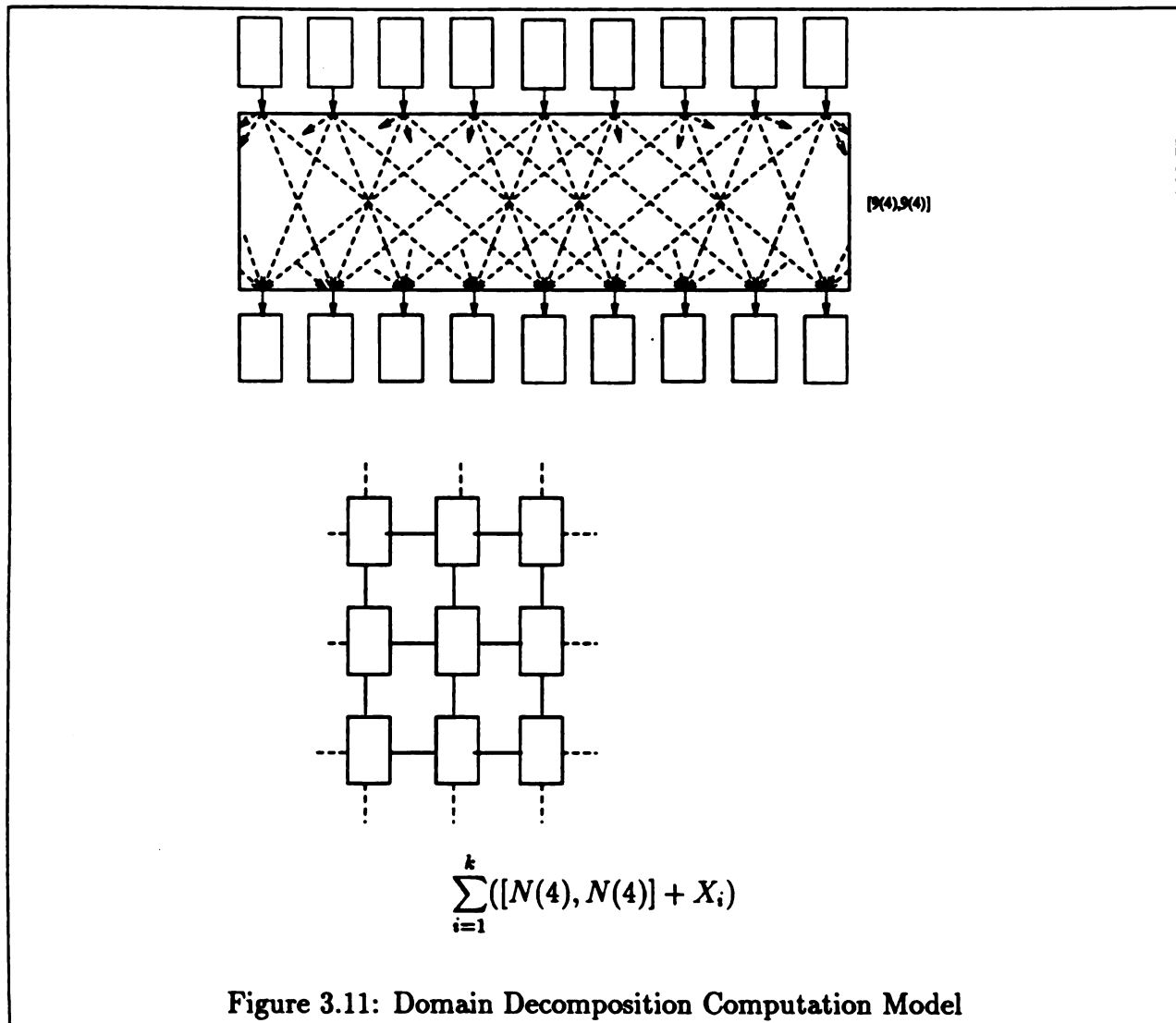
Both steps (b) and (c) take $O(\log n)$ time using n^3 processors. Thus, if $T(n)$ denotes the time required by the algorithm for inverting a matrix of dimensions $n \times n$, we have $T(n) = T(\lceil n/2 \rceil) + O(\log n)$, which yields $T(n) = O(\log^2 n)$ computation using n^3 processors. \square

The method presented here does not make things simpler than the presumably easier task of solving the system $Ax = b$. Furthermore, if communication overhead is properly taken into account, the time requirement can be much higher than $O(\log^2 n)$ for certain computer architectures. For this reason, and in view of its requirement for a large number of processors, the algorithm is theoretically interesting but impractical. A more practical method will be described later in the pipeline model.

Partition models are more practical than recursive divide-and-conquer models, especially for $N \ll n$. Three partition methods for solving tridiagonal linear systems on multicomputers will be given in Chapter 5.

3.2.5 Domain Decomposition Model

Domain decomposition reduces the original large problem into a set of subproblems through decomposition of the problem domain. This technique was originally developed to reduce overall computation time and storage requirements on sequential machines. However, it becomes much more important in parallel algorithm design. The decomposed subproblems may be independent of each other or may be dependent in some way. The goal of the design is to reduce the dependence of the subproblems in order to achieve maximal parallelism. The domain decomposition model is well suited to the finite difference method for solving PDEs where each subproblem is dependent on a fixed number of others [19], [31].



One problem faced by domain decomposition is how to partition the domain. In order

to solve Laplace's equation,

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = 0, \quad (3.8)$$

by the finite difference method, we decompose the xy domain into subregions and assign one region to each processor. The regions can be chosen either as strips or as rectangles. With strip decomposition, each processor communicates with two neighbors. For rectangular regions, each processor has to communicate with three of four neighbors, but has shorter messages. The optimum decomposition depends on the computation/communication ratio of the underlying machine and the number of grid points in each region. Domain decomposition computation is characterized by easily achieved load balance and by neighboring communication. However, if global convergence checking is adopted for the finite difference method, then global communication is needed.

Problems can also be solved in parallel by decomposing the solution domain. The bisection method for solving eigenvalue and polynomial problems belongs to this category. This kind of decomposition has a different communication pattern. We do not consider them as part of the domain decomposition computation model.

3.2.6 Pipeline (Ring) Computation Model

Pipeline (ring) computation increases concurrency by dividing a computation into a number of steps and allowing a number of tasks in various stages to be executed at the same time. Thus the characteristic of this model is that the intermediate result moves in a fixed direction and the final result emerges in the last stage. Communication is automatically overlapped with computation; this is the main advantage of this model [34]. One primary consideration in pipelined computation is to keep the processing speeds of all stages roughly equal. Otherwise, the slowest stage will become the bottleneck of the pipeline. There are two types of pipeline computation models, the *functional pipeline* model and the *data pipeline* model. Traditionally, the term pipelining refers to function pipelining. Different stages in the pipeline perform different functions, and, when data flows through the stages, it is modified along the way. In the data pipelining model, processors in the pipeline perform the same function in a synchronous fashion. *Back substitution* is a well-accepted pipeline paradigm for solving triangular linear systems.

Under the assumption that A is a lower triangular matrix, the i^{th} equation of the system $Ax = b$ is

$$a_{i1}x_1 + a_{i2}x_2 + \dots + a_{ii}x_i = b_i. \quad (3.9)$$

The following parallel version of the back substitution algorithm employs n processors. Suppose that at the beginning of the i^{th} stage, the values of the variables x_1, \dots, x_{i-1} and the expressions $a_{j1}x_1 + \dots + a_{j,i-1}x_{i-1}$ for each $j \geq i$, are available. Then the i^{th} processor evaluates x_i by solving Eq. (3.9):

$$x_i = \frac{1}{a_{ii}}(b_i - a_{i1}x_1 - \dots - a_{i,i-1}x_{i-1}). \quad (3.10)$$

Finally, each processor j , with $j \geq i+1$, evaluates the expression $a_{j1}x_1 + \dots + a_{ji}x_i$ by adding $a_{ji}x_i$ to the previously available expression $a_{j1}x_1 + \dots + a_{j,i-1}x_{i-1}$. The algorithm terminates at the end of the n^{th} stage when all the variables x_1, \dots, x_n have been computed. Clearly, the parallel time required for each stage is constant. Therefore, the total time required by this version of back substitution is $O(n)$ using n processors. Here the communication cost is excluded.

Connecting the output of a later stage to the input of an earlier stage of a pipeline model, we get the *ring computation model*. The ring model is commonly used in iterative methods; wrapped around the pipeline allows successive iteration to continue.

3.2.7 Recursive Doubling Computation Model

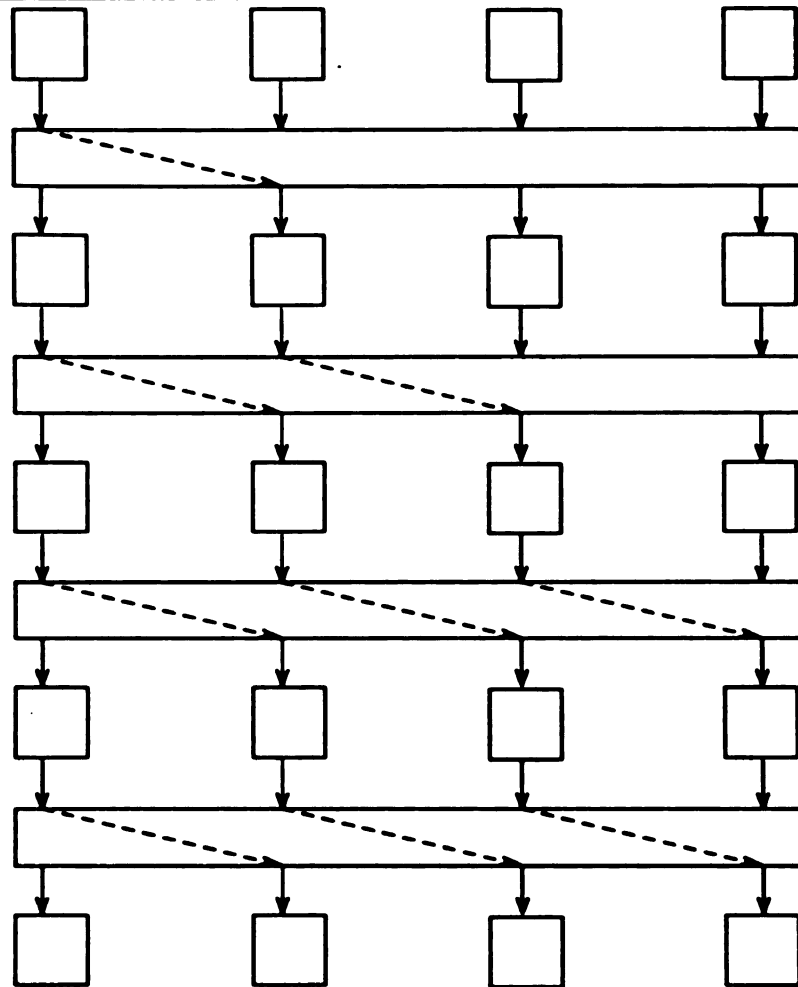
The best known example of parallelism is the computation of the summation $a_0 + a_1 + \dots + a_{n-1}$ in $O(\log n)$ time by $\frac{n}{2}$ processors. It follows the process of *recursive doubling*, which consists of the evaluation of subterms of size 2^i for $i = 0, \dots, \log(\frac{n}{2})$. Computation of the sums and all the partial sums leads to two different communication patterns. Pattern 1 can be seen as the conquer part of the recursive divide-and-conquer model, known as *tree reduction*. Pattern 2 is much different (see Figure 3.13); it is called a *prefix* by some authors. Recursive doubling is applicable to a large number of applications. The following result is due to Kogge and Stone [35].

Proposition 2 *The recursive function*

$$x_1 = b_1, \quad x_i = f_i(x_{i-1}) = f(b_i, g(a_i, x_{i-1})), \quad 2 \leq i \leq n \quad (3.11)$$

can be computed by a recursive doubling parallel algorithm in $O(\log n)$ time, where b_i and a_i are arbitrary constants and f and g are index-independent functions that satisfy the three restrictions:

1. f is associative. $f(x, f(y, z)) = f(f(x, y), z)$.



$$\sum_{i=1}^{N-1} (i[l(1), 1(1)] + Y_i) + \sum_{i=1}^k (N([l(1), 1(1)] + X_i)$$

Figure 3.12: Pipelined Computation Model

2. g distributes over f . $g(x, f(y, z)) = f(g(x, y), g(x, z))$.

3. g is semiassociative, that is, there exists some function h such that $g(x, g(y, z)) = g(h(x, y), z)$.

A large number of linear recursive functions can be reduced to the above form by using a matrix representation. However, this reduction very often increases the sequential computation count. Functions suitable for recursive doubling are given in [57].

Proposition 3 *The LU decomposition of an $n \times n$ tridiagonal linear system can be computed in $O(\log n)$ time, using n processors.*

Proof: Let A be a tridiagonal matrix of order n

$$A = \begin{bmatrix} b_0 & c_0 & & & \\ a_1 & b_1 & c_1 & & \\ & \cdot & \cdot & \cdot & \\ & & a_{n-2} & b_{n-2} & c_{n-2} \\ & & & a_{n-1} & b_{n-1} \end{bmatrix}. \quad (3.12)$$

When $A = LU$ and L is chosen to have its diagonal elements equal to unity, the matrices L and U take the forms

$$L = \begin{bmatrix} 1 & & & & \\ m_1 & 1 & & & \\ & m_2 & 1 & & \\ & & \cdot & \cdot & \\ & & & m_{n-2} & 1 \\ & & & m_{n-1} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} u_0 & c_0 & & & \\ & u_1 & c_1 & & \\ & & u_2 & \cdot & \\ & & & \cdot & \\ & & & & u_{n-2} & c_{n-2} \\ & & & & & u_{n-1} \end{bmatrix}.$$

Note that the superdiagonal elements of U are identical to those of A . Thus we only need to compute m_i and u_i to obtain this factorization. Observing the equation $A = LU$, we can find the recursive relation:

$$u_0 = b_0 \quad u_i = b_i - a_i \frac{c_{i-1}}{u_{i-1}} \quad m_i = \frac{a_i}{u_{i-1}} \quad 1 \leq i \leq n \quad (3.13)$$

To solve for u_i and m_i in parallel, Stone introduced the quantities q_i ($-1 \leq i \leq n$), that satisfy the recurrence equations

$$q_{-1} = 1, \quad q_0 = b_0, \quad q_i = b_i q_{i-1} - a_i c_{i-1} q_{i-2}. \quad (3.14)$$

Let $u_i = \frac{q_i}{q_{i-1}}$ for $i \geq 0$ and write equation (3.14) into matrix form:

$$\begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix} = \begin{bmatrix} b_i & -a_i c_{i-1} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_{i-1} \\ q_{i-2} \end{bmatrix} \quad (3.15)$$

or

$$Q_i = B_i Q_{i-1} \quad 1 \leq i \leq n-1 \quad (3.16)$$

where $q_{-1} = 0$, $q_1 = b_1$. In this form all Q_i ($1 \leq i \leq n-1$) can be expressed in terms of Q_0 :

$$Q_{i+1} = B_i B_{i-1} \dots B_1 B_0 Q_0, \quad i = 1, 2, \dots, n-1. \quad (3.17)$$

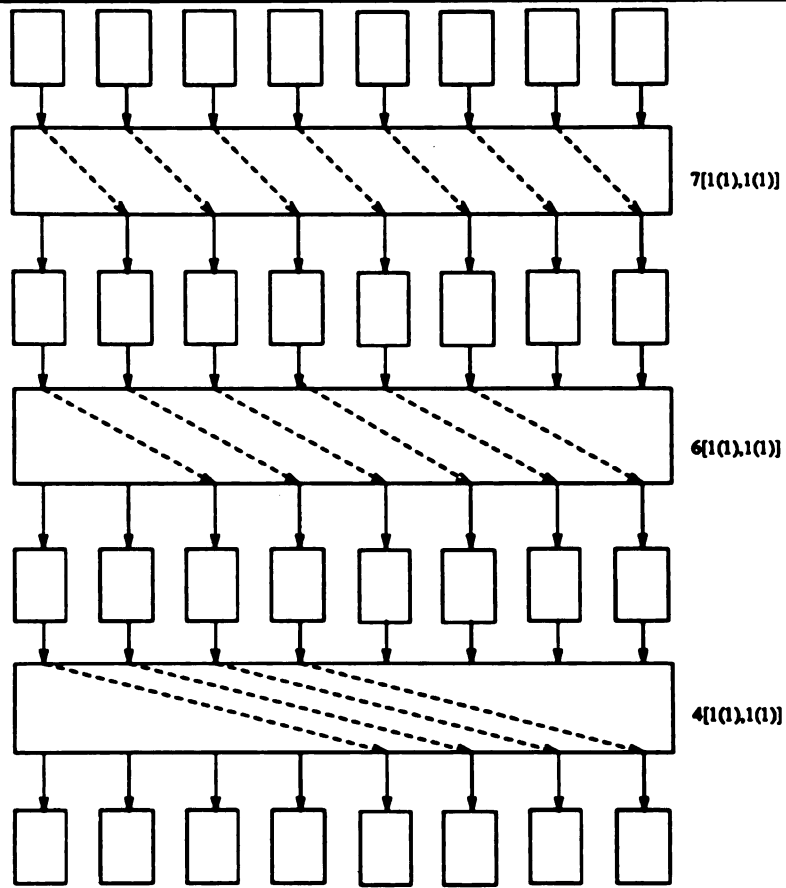
Thus each Q_i can be computed by evaluating the product of 2×2 matrices in parallel, and it is not necessary to compute Q_i only after Q_{i-1} has been computed. Figure 3.13 indicates the computation and communication process for $n=8$. \square

In the hypercube connection topology, the communication pattern shown in Figure 3.13 can be mapped such that the dilation cost is no greater than 2 [33]. This means that the communication is either neighboring communication or passing through one intermediate node.

3.3 Application Considerations

We have identified seven computation models for parallel computing. They are *local computation model*, *global-exchange computation model*, *compute-aggregate-broadcast computation model*, *divide-and-conquer computation model*, *domain decomposition computation model*, *pipelined computation model* and *recursive doubling computation model*. We have found that various scientific applications are combinations of these models. Most of the examples given in this chapter are simple applications. Complicated applications may involve more than one computation model. For designing efficient parallel algorithms we have to understand the problem under consideration, which includes understanding the given application, the applied mathematical method and understanding the underlying computer architecture. The computation model concept provides a way to gain an insightful view and understand the relation between applications and architectures. Based on the performance information of computation models, parallel algorithms may be deliberately switched from one model to another to achieve better performance.

The performance of computation models is problem dependent as well as machine dependent. It depends on the topology, the switching technique and the routing scheme of



$$\sum_{i=0}^{k-1} ((2^k - 2^i)[1(1), 1(1)] + X_i) \quad k = \log N$$

Figure 3.13: Recursive Doubling Computation Model

the underlying multicomputer. Topology determines which nodes are directly connected by physic links. Hypercube is the popular topology for first generation multicomputers. 2-D mesh topology has become a strong alternative to hypercube in second generation multicomputers. The switching techniques determine how two non-adjacent nodes communicate. First generation multicomputers adopted the *store-and-forward* switching technique, while second generation multicomputers use the *circuit-switching* or the *wormhole routing* switching technique. The routing scheme decides the sequence of channels (called paths) used for communication between any pair of nodes. In first generation multicomputers, the routing scheme is the shortest path. In second generation multicomputers, the fixed path routing scheme is used. Different multicomputers may adopt different topologies, switching techniques and routing schemes. Performance considerations for different multicomputers may be different. Computation models should be studied on different multicomputers. Most of my experience is on a first generation NCUBE multicomputer. It is a hypercube multicomputer with store-and-forward switching and shortest path routing.

Chapter 4

APPLICATION-DRIVEN ALGORITHM DESIGN

Designing efficient parallel algorithms is difficult. The structured representation and computation model concept provide a guideline for efficient algorithm design. With structured representation, structured rethinking becomes possible. The bottlenecks of an algorithm will be easier to find and high parallelism will be easier to explore. The performance information of the computation models will suggest which structure is better for a given multicomputer. Therefore, structured design becomes possible. We have used structured representation and computation models on several applications. These applications include solving *electrical power flow problem*, solving *tridiagonal linear systems* and solving *dense linear systems*. The first application is one of the most important problems in the electrical power field. The second application is a fundamental problem of scientific computing.

The structured representation and computation model concept helps us in changing our design from one structure to another. For solving the power flow problem, we started with a local computation design, and changed to a global-exchange computation design. After studying the structure of the global-exchange algorithm, we adopted a compute-aggregate-broadcast design which gives the best performance [59]. The algorithm design for the power flow application is mainly influenced by the chosen mathematical method, the *homotopy method*. This kind of algorithm design is called application-driven algorithm design. We also developed three algorithms for solving tridiagonal systems. They are all based on the same matrix modification formula. The algorithm design for solving tridiagonal systems is mainly influenced by the topology of the underlying multicomputer [60]. This kind of algorithm design is called architecture-driven algorithm design. Some of the design changes which we have made are only suitable for the hypercube topology. More information concerning the power flow application can be found in the rest of this chapter and a detailed study of solving tridiagonal systems can be found in the next chapter.

4.1 Preliminary

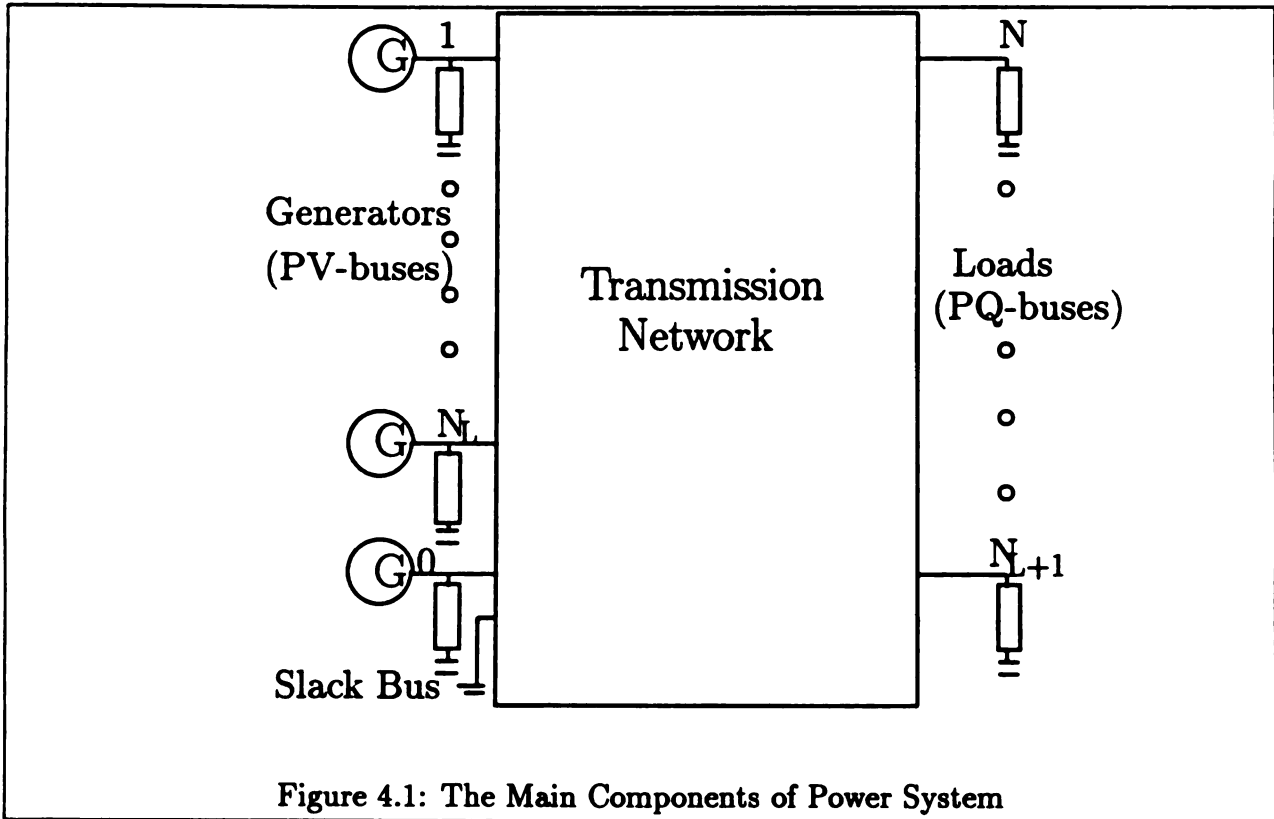
Determining steady state solutions of large-scale interconnected power systems is by far one of the most important problems facing theoretical as well as applied researchers in the field. It has been formulated mathematically [64] and commonly referred to as the power flow, or load flow, problem. The most frequent and acceptable numerical procedures for finding solutions to the load flow problem are variants of the Newton method. It is well known, however, that the Newton method does not always converge to a solution, and if it does, there is no guarantee that it would find all possible solutions to the load flow problem. The Newton method is a local algorithm because it depends on the initial guess and, therefore, it is dependent on the type of nonlinearity of the governing equations, i.e., the load flow equations.

Our work aims at solving the load flow problem using the *globally convergent probability-one homotopy method* originally proposed in [9]. The homotopy method has been shown to have the following distinct features [52], [45], [51], [10], [37], [38]. First, numerical computation of the homotopy method can be systematically implemented. Second, the homotopy method is globally convergent, i.e., one may choose any initial guesses and the homotopy method is guaranteed to converge to all solutions with probability one. However, for solving large-scale load flow problems, the computation requirements of the homotopy method increases exponentially. The exponential increasing of computation requirements is the case for most scientific and engineering applications. Large-scale scientific computing is one of the major driving forces behind parallel computation, but the move into parallel territory requires new conceptual strategies for formulating a problem, and new algorithms to shape the problem for parallel computers. This implies that a brute-force approach to solving a large-scale problem, such as the load flow of power systems, on parallel computers does not render the problem tractable.

The power flow problem is a good example for illustrating the design process of large scale scientific computing and how the structured representation and computation model concept can be used to help the design. The design process of the power flow application consists of four parts – finding an appropriate mathematical model for the application, choosing a suitable numerical method to solve this model, modifying the mathematical method to reduce the computation, and changing the algorithm design to achieve high performance.

4.2 Power Flow Application

The power flow, or load flow, problem has been modeled on the sinusoidal steady state nodal analysis of circuit theory. Figure 4.1 depicts the general model of a power system. It consists of three main components: generators (suppliers), loads (consumers), and a transmission network that connects generators and loads. Consumers demand electrical power to be supplied. Suppliers generate the power and transmit the power to the consumers. The demand changes with time. The suppliers, therefore, have to vary their production to fit the demand. Changing supply according to demand is done not only for economic reasons, but, more importantly, for safety considerations. In order to operate a power system safely, power generation must satisfy a set of inequality constraints. A solution which satisfies this set of constraints is called a steady state solution. The power flow problem is a matter of determining the steady state solutions.



Let the power system have $N + 1$ buses or nodes with a voltage reference (or datum). Let the load buses be subscripted from 1 to N_L , and let the generator buses be subscripted from N_{L+1} to N . We choose to subscript the slack bus by 0 and denote the $N \times N$ node or bus complex admittance matrix by $[Y]$, where its ki^{th} component is $y_{ki} = G_{ki} + jB_{ki}$ ($j = \sqrt{-1}$). Let E_k denote the complex voltage of bus k . In polar coordinates it is

$$E_k = V_k e^{j\Theta_k}, \quad (4.1)$$

where V_k represents its voltage amplitude and Θ_k represents its phase angle. In rectangular coordinates, it becomes

$$E_k = V_k \cos \Theta_k + j V_k \sin \Theta_k = X_k + j Y_k, \quad (4.2)$$

where X_k represents the real part of the complex voltage E_k and Y_k represents its imaginary part. Let $S_k = p_k + j q_k$ be the complex injected power at node k , then the injected complex power balance equation may be written as

$$[E^*] [Y] E - S^* = 0 \quad (4.3)$$

or

$$E_k \sum_{i=0}^N y_{ki}^* E_i^* + E_k^* \sum_{i=0}^N y_{ki} E_i - 2P_k = 0, \quad k = 1, 2, \dots, N \quad (4.4)$$

$$E_k \sum_{i=0}^N y_{ki}^* E_i^* - E_k^* \sum_{i=0}^N y_{ki} E_i - 2jQ_k = 0, \quad k = 1, 2, \dots, N_L \quad (4.5)$$

$$E_k E_k^* - V_k^2 = 0, \quad k = N_{L+1}, \dots, N, \quad (4.6)$$

where the superscript * denotes the complex conjugate.

The above system of equations is a polynomial system. They are the *full load equations in complex form*. The full load equations can be represented in other forms. They can be written in *rectangular coordinates* as,

$$\sum_{i=0}^N [G_{ki}(X_k X_i + Y_k Y_i) + B_{ki}(X_i Y_k - X_k Y_i)] - P_k = 0, \quad k = 1, 2, \dots, N \quad (4.7)$$

$$X_k^2 + Y_k^2 - V_k^2 = 0, \quad k = 1, 2, \dots, N_L \quad (4.8)$$

$$\sum_{i=0}^N [G_{ki}(X_i Y_k - X_k Y_i) - B_{ki}(X_i X_k + Y_k Y_i)] - Q_k = 0, \quad k = N_{L+1}, \dots, N. \quad (4.9)$$

Or, it can be written in equivalent rectangular coordinate equations as

$$X_k \sum_{i=0}^N (G_{ki} X_k - B_{ki} Y_i) + Y_k \sum_{i=0}^N (B_{ki} X_i - G_{ki} Y_i) - P_k = 0, \quad k = 1, 2, \dots, N \quad (4.10)$$

$$X_k^2 + Y_k^2 - V_k^2 = 0, \quad k = 1, 2, \dots, N_L \quad (4.11)$$

$$Y_k \sum_{i=0}^N (G_{ki} X_i - B_{ki} Y_i) - X_k \sum_{i=0}^N (B_{ki} X_i + G_{ki} Y_i) - Q_k = 0, \quad k = N_{L+1}, \dots, N, \quad (4.12)$$

where the additional equations (4.8) and (4.11) represent the constraints on the amplitude of the voltage of the PV-buses.

The full load equations also can be represented in terms of trigonometric functions, called standard power flow equations, as

$$\sum_{i=0}^N V_k V_i (G_{ki} \cos \Theta_{ki} + B_{ki} \sin \Theta_{ki}) - P_k = 0, \quad k = 1, 2, \dots, N \quad (4.13)$$

$$\sum_{i=0}^N V_k V_i (G_{ki} \sin \Theta_{ki} - B_{ki} \cos \Theta_{ki}) - Q_k = 0, \quad k = N_{L+1}, \dots, N. \quad (4.14)$$

There are three types of buses in a power system network:

1. PQ bus: a bus where the real and reactive powers are specified.
2. PV bus: a bus where the real power and the voltage amplitude are specified.
3. A Slack bus: a fictitious concept whereby one of the generator buses has only its complex voltage specified. One purpose of this bus is to guarantee that the total power injection into the network equals the total power consumed.

It is conventional to model loads as PQ-buses, one generator as a slack bus, and the rest of the generators as PV buses, as depicted in Figure 4.1. With some assumptions, two simplified models are also proposed for power flow problems [51], which can be used to obtain approximate solutions with reduced computation.

4.3 Homotopy Method

The homotopy method [9] (see Section 3.2.1) is able to find all the complex roots of a system of polynomials with probability one. Let $P(x) = 0$ denote a system of n equations in n unknowns.

$$P(Z) = \begin{bmatrix} P_1(z_1, z_2, \dots, z_n) \\ P_2(z_1, z_2, \dots, z_n) \\ \dots \\ P_n(z_1, z_2, \dots, z_n) \end{bmatrix} = 0, \quad (4.15)$$

where Z is an n -dimensional complex vector (z_1, \dots, z_n) . The homotopy method solves this system by solving a trivial system

$$Q(Z) = 0, \quad (4.16)$$

and then follows the solution curves of

$$H(Z, t) = (1 - t)Q(Z) + tP(Z) = 0, \quad (4.17)$$

from $t = 0$ to $t = 1$. If $Q(Z) = 0$ is chosen correctly, the following properties hold [39]:

- *Triviality*

The solutions of $Q(Z) = 0$ are known.

- *Smoothness*

The solution set of $H(Z, t) = 0$ for $0 \leq t < 1$ consists of a finite number of smooth paths, each parameterized by t in $[0, 1)$.

- *Accessibility*

Every isolated solution of $H(Z, 1) = P(Z) = 0$ is reached by some path originating at $t = 0$. It follows that this path starts at a solution of $H(Z, 0) = Q(Z) = 0$.

Let the degree of P_i be d_i for $1 \leq i \leq n$. In general, Eq.(4.15) has at most $d = d_1 \times d_2 \times \dots \times d_n$ isolated roots. It has been proven [9] with random complex numbers b_1, b_2, \dots, b_n that

$$Q(Z) = \begin{bmatrix} z_1^{d_1} - b_1 \\ \dots \\ z_n^{d_n} - b_n \end{bmatrix} = 0 \quad (4.18)$$

can be used to find all the solutions of (4.15) by tracing the solution curves of equation (4.17) from $t = 0$ to $t = 1$ with probability 1. The robustness, stability, and accuracy properties of the homotopy method make it the best known choice for power flow problems.

The number $d = 2 \times 2 \times \dots \times 2$ gives the upper bound of the possible solutions of the power load system (4.4)–(4.6). In practice, power flow systems have many fewer less solutions than this upper bound. Many of the homotopy curves cannot reach $t=1$, because they go to infinity when they approach $t=1$ (see Figure 4.2). These divergent curves generally require more computation than convergent curves and consume most of the computation time. Not only does the system of equations (4.4)–(4.6) have many fewer solutions than the upper bound, but also, in practical consequences, the majority of polynomial systems have the same property. These kinds of systems are called deficient by Li et al. [37] and called m -homogeneous by Morgan and Sommese [43].

Based on [43], we conclude that equation (4.17) is 2-homogeneous and, instead of 2^{2N} solutions, it has at most $[N : 2N]$ (i.e., N out of $2N$) solutions. The following modified initial function is proposed to trace the $[N : 2N]$ possible solutions, and the correctness of the function is followed by the theoretical results of the general deficient and m-homogeneous systems [37] [51]. For the general complex form (4.4) – (4.6), the initial polynomial system is chosen as

$$Q(Z) = \begin{bmatrix} (E_k + \alpha_k)(\sum_{i=1}^N y_{ki}^* E_i^* + \beta_k), & k = 1, 2, \dots, N, \\ (\sum_{i=1}^N y_{ki} E_{ki} + \alpha_{N+k})(E_k^* + \beta_{N+k}), & k = 1, 2, \dots, N_L, \\ (E_k + \alpha_{N+N_L+k})(E_k^* + \beta_{N+N_L+k}), & k = N_{L+1}, \dots, N \end{bmatrix} \quad (4.19)$$

where $\alpha_i, \beta_i, i = 1, \dots, 2N$ are random complex scalars. For “almost all” $(\alpha_1, \dots, \alpha_{2N}) \in \mathbb{C}^{2N}$, $(\beta_1, \dots, \beta_{2N}) \in \mathbb{C}^{2N}$, Eq. (4.19) has $[N : 2N]$ solutions. The equations of the polynomial system $Q(Z)$ are products of linear equations. $Q(Z)$ is easy to solve. It has $[N : 2N]$ distinct complex roots and all the solutions of equations (4.4) – (4.6) can be obtained by tracing the solutions curves of equation (4.17).

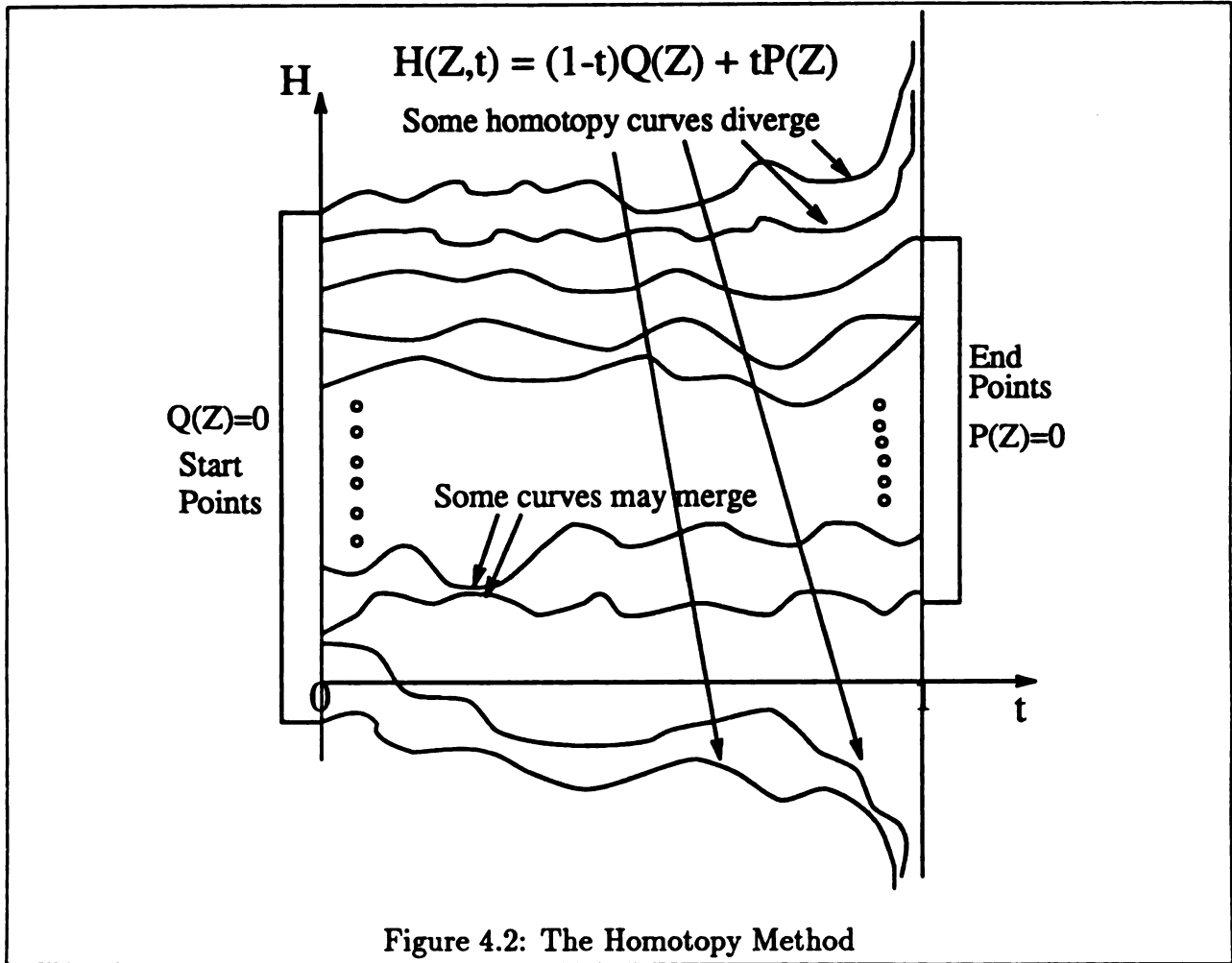
With the specified initial system of equations (4.19), the computation time has been reduced considerably. That is one reason for choosing the complex full flowed model for our study.

4.4 Implementation Issues and Results

With the homotopy method, the solution process for system (4.4) – (4.6) is the process of curve tracing. We need to trace the solution curves, which are also called *homotopy curves*, of the homotopy function (4.17), starting from $t = 0$ and gradually increasing t until it becomes 1. Let t_0, t_1, \dots, t_k be the set of *sampling instants*, where $t_0 = 0$ and $t_k = 1$. Define $s_i = t_i - t_{i-1}$ as the *stepping distance* from t_{i-1} to the next sampling instant t_i for $1 \leq i \leq k$. Given $Z_j(t_{i-1})$, we have to solve $Z_j(t_i)$ for $t_i = t_{i-1} + s_i$. Several different techniques, such as Newton’s iterative method, the ODE-based method, and normal flow method [63] may be used for the curve tracing. Theoretically, all the homotopy curves can be traced concurrently and independently. Therefore, the power flow problem is readily solved by the local computation model.

Though homotopy curves have been proven disjoint, it is possible that one curve merges to another curve [8] in the numerical implementation of curve tracing. This *curve merging* will lead to the occurrence of curve missing, and consequently, solutions being missed. The stepping distances have to be very small to reduce the possibility of curve merging. Small

stepping distances increase the computation time significantly. The performance of the local computation approach is not very encouraging.



To reduce the computation cost of curve tracing, we adopt a dynamic stepping strategy [10]. Run time information is used to adjust the stepping distance to make it relatively large. A predictor-corrector method is adopted for curve tracing. This curve tracing method contains two steps, first predict the value $Z(t_1)$ based on the value and derivative at t_{i-1} , then use Newton's iterative method to correct the predicted value until it satisfies Eq. (4.17). If after a certain number (threshold) of iterations, the predicted value does not converge to a solution of equation (4.17), we say the tracing at t_{i-1} has failed. When this occurs, the stepping distance $s_i = t_i - t_{i-1}$ is reduced and the procedure is repeated. If the predicted value converges to a solution within the threshold, the tracing is successful. The stepping distance is increased for the next step. The iteration threshold is an important parameter for the dynamic curve tracing method. Curve merging checks are also performed. We introduce a series of c merge checking instants, $0 = t_0 < t_1^* < t_2^* < \dots < t_c^* < 1$, where these c merge

checking instants are a subset of the sampling instants. Note that tracing different homotopy curves dynamically may lead to different sets of sampling instants and a different number of sampling instants, k . However, for the purpose of merge checking, these c merge checking instants must be included in any set of sampling instants.

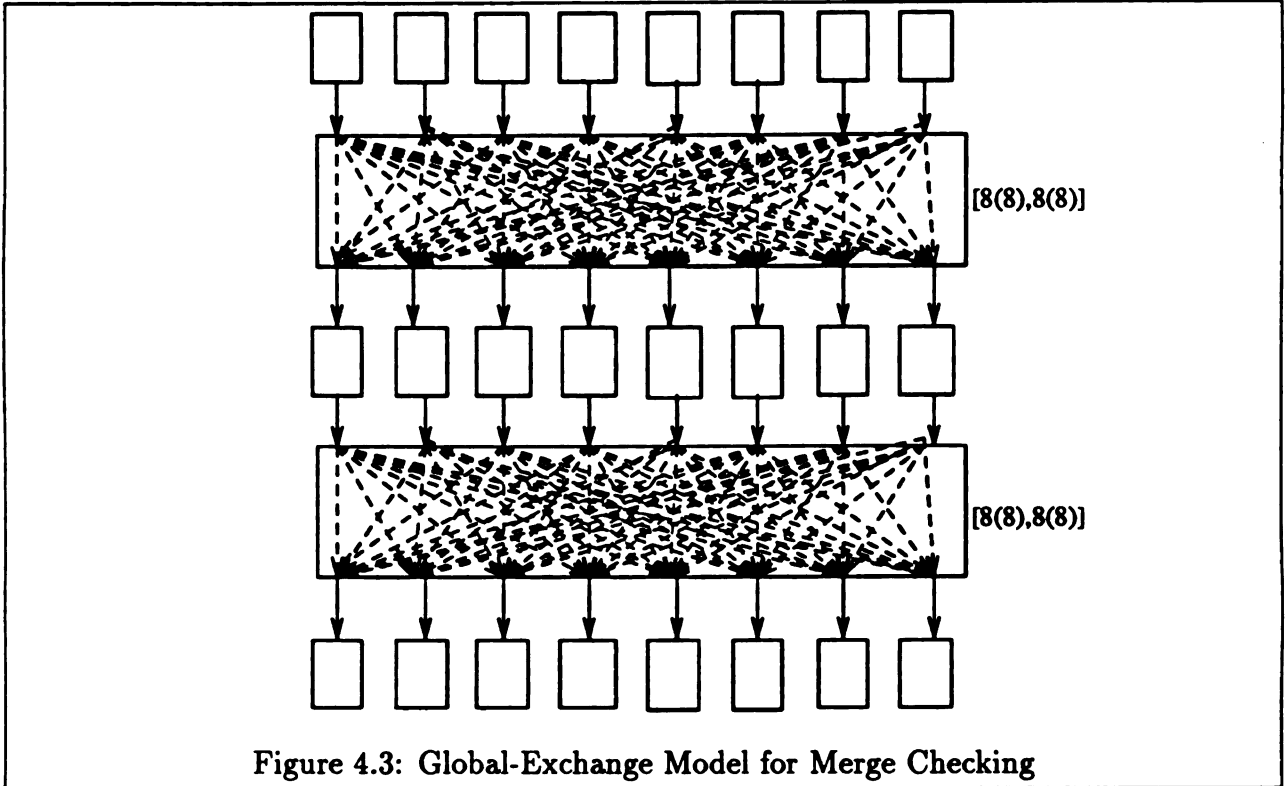


Figure 4.3: Global-Exchange Model for Merge Checking

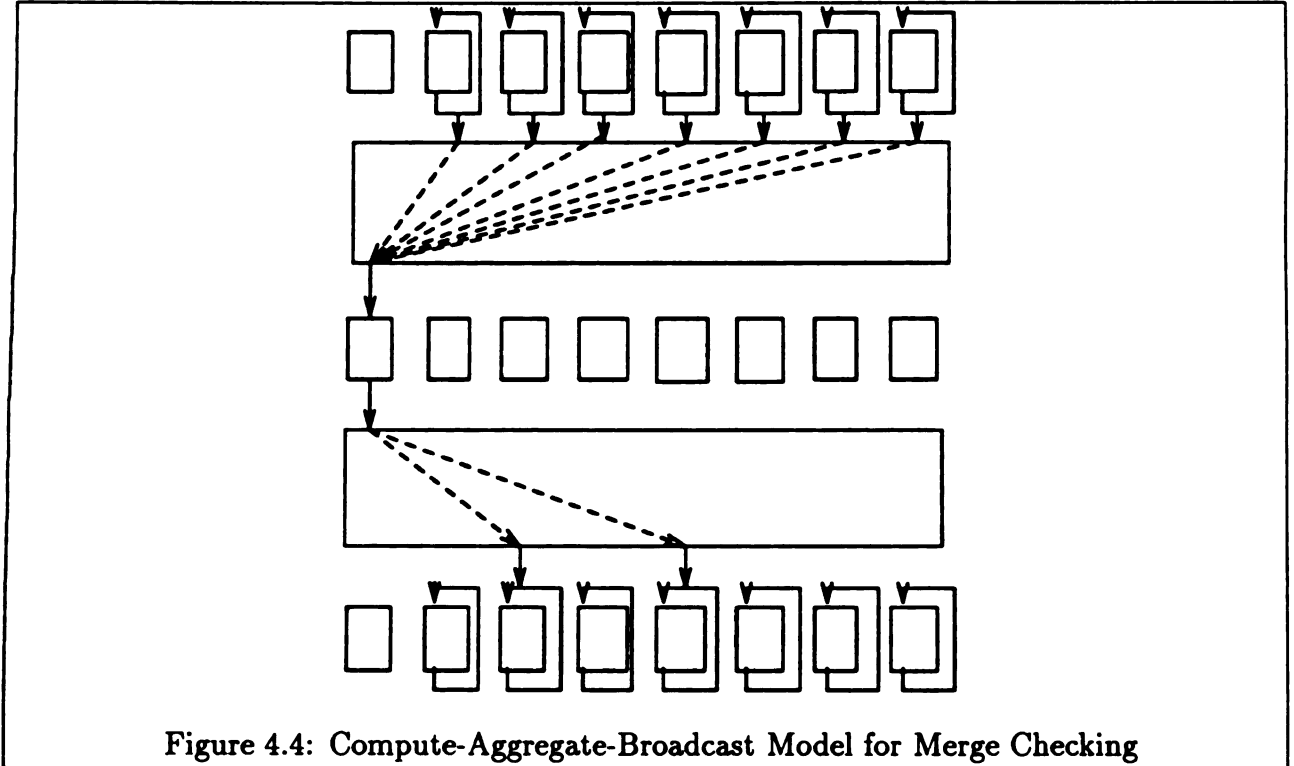
Merge checking requires global information. Communication becomes unavoidable at merge checking instants. This communication requirement changes our computation from a local computation to a compute-exchange computation. Initially, this compute-exchange computation is implemented in a synchronous fashion, which corresponds to the global-exchange computation model (see Figure 4.3). All the nodes synchronize at merge checking instants, and merge sort is used at these points to sort the intermediate values $Z(t_i^*)$. If more than one curve has the same value, curve merging has occurred. The merged curves will then be retraced from t_{i-1}^* with a more conservative iteration threshold. With the merge checking strategy, the stepping distance can be relatively large. If only a few retracings have been done, performance should be improved. This is true when the synchronous merge checking algorithm is implemented with a single processor. However, with more processors, the merge checking algorithm does not show any superiority over conservative tracing without merge checking. To analyze degradations of parallelism we use the notation introduced in Chapter 3 to write down the synchronous merge checking algorithm. With eight processors, it is

$$X_{c+1} + \sum_{i=1}^c ([8(8), 8(8)] + X_i), \quad (4.20)$$

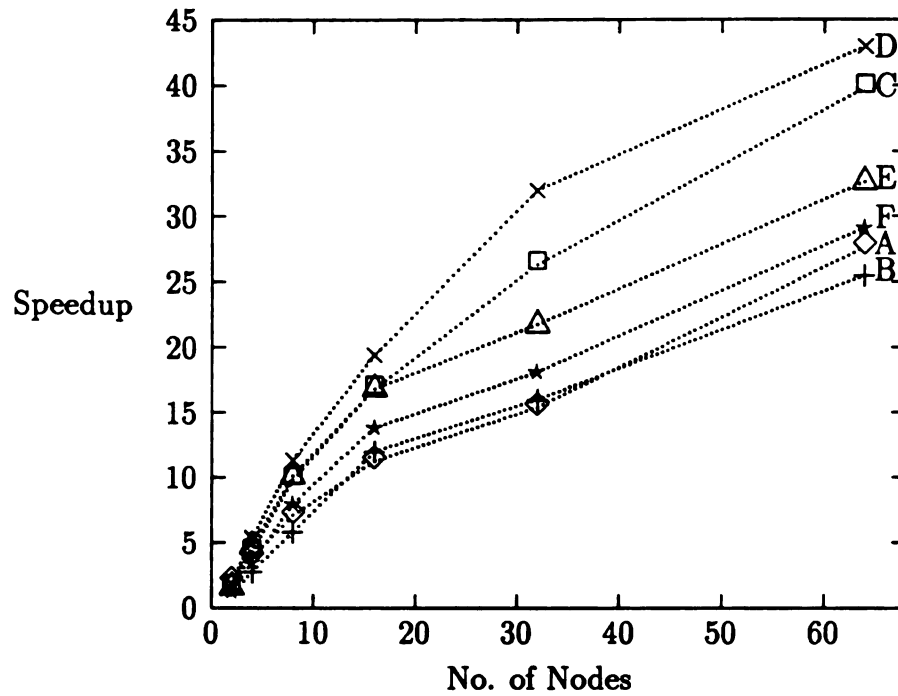
when no curve merging happens. When curve merging does occur at j of the c merge checking points, it becomes

$$X_{c+1} + \sum_{i=1}^{c+j} ([8(8), 8(8)] + X_i), \quad (4.21)$$

where X_i is the workload of the node with the heaviest load during the two contiguous synchronization. Note that we use dynamic stepping. Each curve, depending on its shape, may require a different number of sampling instances from t_{i-1}^* to t_i^* . X_i is the workload of the curve with the biggest number of sampling instances between t_{i-1}^* and t_i^* . Equations (4.20) and (4.21) also show that when curve merging occurs, some nodes are busy doing retracing while all the others are waiting. This reveals that the load is not balanced between synchronization, and that curve retracing makes the load imbalance even worse.



To overcome load imbalancing, the algorithm is redesigned to be asynchronous (see Figure 4.4). We choose one node as the checking node. All the other nodes work as the computing nodes. Computing nodes do the curve tracing and send the computed value at t_i^* to the checking node. The checking node does the merge checking. When curve merging



Method A: 1 iteration and without merge checking
 Method B: 1 iteration and merge checking
 Method C: 2 iteration and merge checking
 Method D: 3 iteration and merge checking
 Method E: 4 iteration and merge checking
 Method F: 5 iteration and merge checking

Figure 4.5: Speedup Versus Different Number of Processors

is found, it sends a message to the corresponding nodes with the needed information. These corresponding nodes will do retracing, while the others continue. Since the load is imbalance between each pair of merge checking instants, each node arrives at the merge checking point at a different time and the contention for the checking node is negligible (if the number of available processors is not very large). The checking node collects and broadcasts data. The computation becomes a compute-aggregate-broadcast computation. The asynchronous compute-aggregate-broadcast algorithm was implemented on a 64-node NCUBE multicomputer with different iteration thresholds. The results of the implementation are shown in Figure 4.5. Method *A* is based on the conservative local computation approach. Method *B* to method *F* are based on the compute-aggregate-broadcast approach with different iteration thresholds. It is very interesting to find out that the iteration threshold used for convergence check has a significant impact on the system performance. When the iteration threshold is very small, a divergence may occur to imply the need of a smaller stepping interval. When the number of iterations is large, a convergence may be resulted from the merging to another curve. In our experiments, method *D* caused three retracing computations; whereas, method *E* and method *F* caused 20 and 40 retracing computations, respectively.

In our implementation, two merge-checking points, 0.1 and 0.96 were selected. For three and four merge-checking points, we obtained very similar results as in Figure 4.5. The optimal iteration threshold is apparently problem-dependent. Choosing an appropriate iteration threshold is important. We can see from Figure 4.5 that method *D*, where the iteration threshold is equal to 3, achieves a better performance than other methods. The speedup used in Figure 4.5 is the relative speedup.

Chapter 5

ARCHITECTURE-DRIVEN ALGORITHM DESIGN

In this chapter we study another application, solving tridiagonal linear systems, and show how machine architecture will influence the algorithm design. We are interested in solving the following tridiagonal linear system of equations

$$Ax = d, \quad (5.1)$$

where A is a tridiagonal matrix of order n ,

$$A = \begin{bmatrix} b_0 & c_0 & & & \\ a_1 & b_1 & c_1 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-2} & b_{n-2} & c_{n-2} \\ & & & a_{n-1} & b_{n-1} \end{bmatrix}. \quad (5.2)$$

$x = (x_0, \dots, x_{n-1})^T$ and $d = (d_0, \dots, d_{n-1})^T$. We assume that A , x , and d have real coefficients. Extension to the complex case is straightforward.

Solving tridiagonal linear systems is one of the most important problems in scientific computing. It is involved in the solution of differential equations and in various other areas of science and engineering. Sequential algorithms for solving Eq. (5.1) have been well studied [13]. A variety of methods have been developed for solving Eq. (5.1) on parallel computers. A good survey of these methods can be found in [24], [33], [32], [47]. Among them, the *recursive doubling method* (RCD) developed by Stone [56] and the *cyclic reduction* or *odd-even reduction method* (OER) developed by Hockney [27] are able to solve Eq. (5.1) in $O(\log n)$ time using n processors. However, both methods will fail if pivoting is required. In a realistic parallel processing environment, the number of processors, N , is less, or much less, than the dimension of the matrix, n . *Wang's partition* algorithm [62] is a relatively new method which is designed for a practical parallel environment, where N is less than n . However, it has some inherent sequential properties to remove the "fill-ins". More recently, a *parallel prefix* (PP) method developed by Egecioglu, Koc and Laub [16] modifies the RCD method to be applied to a limited number of processors, i.e., $N < n$. While the OER

algorithm is a popular choice for vector machines, the RCD algorithm is a good candidate for multicomputers.

Three parallel algorithms are presented in this thesis. All these algorithms are developed based on a novel matrix partitioning technique which partition the $n \times n$ tridiagonal matrix A into N tridiagonal submatrices. This matrix partitioning technique requires solving a small $2(N - 1) \times 2(N - 1)$ tridiagonal system. The *parallel partition LU* (PPT) algorithm solves this small tridiagonal system in a single processor. If N is small, the *parallel partition global-exchange* (PPG) algorithm solves this small tridiagonal system in every processor redundantly to reduce the communication overhead. If N is large, the solving of this small tridiagonal system can be performed in parallel. This algorithm is called the *parallel partition hybrid* (PPH) algorithm. In many applications, the tridiagonal matrix A is evenly diagonal dominant. In this case half of the off-diagonal elements of the small $2(N - 1)$ dimensional matrix tend to zero exponentially. A fast and highly parallel algorithm, namely the *parallel diagonal dominant* (PDD) algorithm, is proposed in [60]. We proved that the PDD algorithm provides an approximate solution that is equal to the exact solution within the machine accuracy when $n/N \gg 1$. The PDD algorithm is a fast algorithm for a special case. We will not study the PDD algorithm in this chapter.

Communication mechanisms have a great impact on the performance of multicomputers. Thus, the communication pattern of parallel algorithms should be carefully designed to reduce the communication complexity. The communication consideration may lead to design modification. In this study, the algorithms are evaluated based on both computation and communication complexities.

5.1 Linear Tridiagonal Solvers

Two known methods will be used in our algorithms. For completeness, they are briefly presented in this section.

5.1.1 The LU Decomposition Method

The LU decomposition method is also called the Gaussian elimination method without pivoting. The LUP decomposition method is the Gaussian elimination method with pivoting. They are the most commonly accepted sequential algorithms and are used in the linear system package, LINPACK [13]. The pivoting referred to in this paper is partial pivoting. The LINPACK routine SGTSL solves Eq. (5.1) using the LUP decomposition method.

The algorithm first factors the matrix A into a product form

$$A = LU \quad (\text{or } PA = LU \text{ when pivoting is involved}), \quad (5.3)$$

where L and U are lower and upper triangular matrices and P is the product of all of the row permutations. Then $Ax = d$ (or $PAx = Pd$) can be solved by solving both

$$Ly = d \quad (\text{or } Ly = Pd) \quad \text{and} \quad Ux = y. \quad (5.4)$$

It can be easily verified that the LU and LUP algorithms can solve Eq. (5.1) in $8n - 7$ and $11n - 12$ arithmetic operations for the non-pivoting and pivoting cases, respectively.

5.1.2 The Parallel Prefix Method

The RCD method uses $O(\log n)$ parallel computation time to solve Eq. (5.1) on a parallel computer with n processors (see Section 3.2). The PP method [16] modifies the RCD method for N processors, where $N < n$.

Equation (5.1) can be written as

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \quad 0 \leq i \leq n-1, \quad (5.5)$$

where $x_{-1} = x_n = 0$. Solving for x_{i+1} , we have

$$x_{i+1} = \left(-\frac{b_i}{c_i}\right)x_i + \left(-\frac{a_i}{c_i}\right)x_{i-1} + \left(\frac{d_i}{c_i}\right) = \alpha_i x_i + \beta_i x_{i-1} + \gamma_i. \quad (5.6)$$

Here $c_i \neq 0$ is assumed. Equation (5.6) can be written in matrix form as

$$\begin{bmatrix} x_{i+1} \\ x_i \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix} \quad (5.7)$$

Define

$$X_{i+1} = \begin{bmatrix} x_{i+1} \\ x_i \\ 1 \end{bmatrix} \quad \text{with } x_{-1} = x_n = 0, \quad (5.8)$$

$$B_i = \begin{bmatrix} \alpha_i & \beta_i & \gamma_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (5.9)$$

Equation (5.6) becomes

$$X_{i+1} = B_i X_i, \quad 0 \leq i \leq n-1, \quad (5.10)$$

and X_i ($1 \leq i \leq n$) can be expressed in terms of X_0 as

$$X_i = B_{i-1} B_{i-2} \dots B_1 B_0 X_0, \quad 1 \leq i \leq n. \quad (5.11)$$

Now solving Eq. (5.1) reduces to finding the partial products of matrices B_i for $0 \leq i \leq n-1$. If $N < n$, evenly distribute matrices B_i to N processors, perform sequential matrix multiplication on each processor, then use the prefix method on N processors. There are $(\log N) + 1$ parallel communication steps for implementing the prefix method with N processors.

Let $C_i^j = B_i B_{i+1} \dots B_j$. Then C_i^j is a 3×3 matrix. Since the last row of C_i^j always equals $[0, 0, 1]$, only six entries of C_i^j need to be transferred at each parallel communication. Parallel *recursive doubling computation* can be used to obtain all the partial matrix products. The actual communication complexity of the PP method depends on the mapping of computation units to processors, the interconnection topology of the multicomputer, and the underlying communication mechanism. In the case of the hypercube topology, the communication pattern shown in Figure (3.13) can be mapped such that the dilation cost is no greater than 2 [33].

5.1.3 A Novel Matrix Partitioning Technique

Our parallel algorithms are based on the divide and conquer model of parallel computation. In the divide part, matrix A is partitioned into submatrices. For convenience we assume that $n = Nm$. Matrix A in Eq. (5.1) can be written as

$$A = \tilde{A} + \Delta A, \quad (5.12)$$

where the submatrices, A_i ($i = 0, \dots, N-1$), are $m \times m$ tridiagonal matrices. Let e_i be a column vector with its i^{th} ($0 \leq i \leq n-1$) element being 1 and all the other entries being zero. We have

$$\tilde{A} = \begin{bmatrix} A_0 & & \\ & A_1 & \\ & & \ddots \\ & & & A_{N-1} \end{bmatrix}$$

$$\Delta A = \begin{bmatrix} & & & & \\ & c_{m-1} & & & \\ a_{1m} & & & & \\ & c_{2m-1} & & & \\ & a_{2m} & & & \\ & & \ddots & & \\ & & & c_{m(N-1)-1} & \\ & & & a_{m(N-1)} & \end{bmatrix}$$

$$\Delta A = \begin{bmatrix} a_m e_m, & c_{m-1} e_{m-1}, & a_{2m} e_{2m}, & c_{2m-1} e_{2m-1}, & \dots, & c_{(N-1)m-1} e_{(N-1)m-1} \end{bmatrix} \begin{bmatrix} e_{m-1}^T \\ e_m^T \\ \vdots \\ \vdots \\ e_{(N-1)m-1}^T \\ e_{(N-1)m}^T \end{bmatrix} = V E^T,$$

where both V and E are $n \times 2(N-1)$ matrices. Thus we have

$$A = \tilde{A} + V E^T. \quad (5.13)$$

Based on the matrix modification formula originally defined by Sherman and Morrison [54] for rank-one changes and generalized by Woodbury [14], Eq. (5.1) can be solved by

$$\begin{aligned} x &= A^{-1}d = (\tilde{A} + V E^T)^{-1}d \\ &= \tilde{A}^{-1}d - \tilde{A}^{-1}V(I + E^T \tilde{A}^{-1}V)^{-1}E^T \tilde{A}^{-1}d. \end{aligned} \quad (5.14)$$

Note that I is a $2(N-1) \times 2(N-1)$ identity matrix and $I + E^T \tilde{A}^{-1}V$ is a $2(N-1) \times 2(N-1)$ band matrix with bandwidth 5. We introduce a $2(N-1) \times 2(N-1)$ elementary transformation matrix P such that

$$P z = (z_1, z_0, z_3, z_2, \dots, z_{2N-3}, z_{2(N-2)})^T \quad \text{for all } z \in R^{2(N-1)}. \quad (5.15)$$

Based on the property that $P^{-1} = P$, Eq. (5.14) becomes

$$x = \tilde{A}^{-1}d - \tilde{A}^{-1}VP(P + E^T \tilde{A}^{-1}VP)^{-1}E^T \tilde{A}^{-1}d. \quad (5.16)$$

Note that $Z = (P + E^T \tilde{A}^{-1}VP)$ is a $2(N-1) \times 2(N-1)$ tridiagonal matrix. Let

$$\tilde{A}\tilde{x} = d, \quad (5.17)$$

$$\tilde{A}Y = VP, \quad (5.18)$$

$$h = E^T \tilde{x}, \quad (5.19)$$

$$Z = P + E^T Y, \quad (5.20)$$

$$Zy = h, \quad (5.21)$$

$$\Delta x = Yy. \quad (5.22)$$

From Eqs. (5.17)-(5.22), (5.16) becomes

$$x = \tilde{x} - \Delta x, \quad (5.23)$$

where \tilde{A} is an $n \times n$ matrix, Y , V , and E are $n \times 2(N-1)$ matrices, Z and P are $2(N-1) \times 2(N-1)$ matrices, h and y are $2(N-1) \times 1$ vectors, and Δx , \tilde{x} , and d are $n \times 1$ vectors. Based on Eqs. (5.17)-(5.23), the computation required to solve Eq. (5.1) in a sequential computer is described below.

In Eqs. (5.17) and (5.18), \tilde{x} and Y are solved by the LU decomposition method. By the structure of \tilde{A} and VP , this is equivalent to solving

$$A_i [\tilde{x}^{(i)}, v^{(i)}, w^{(i)}] = [d^{(i)}, a_{im}e_0, c_{(i+1)m-1}e_{m-1}], \quad i = 0, \dots, N-1. \quad (5.24)$$

Here we have $a_0 = c_{n-1} = 0$, $e_0, e_{m-1} \in R^m$, $\tilde{x}^{(i)}$ and $d^{(i)}$ are the i^{th} block of \tilde{x} and d , respectively, and $v^{(i)}, w^{(i)}$ are possible non-zero column vectors of the i^{th} row block of Y . Equation (5.24) implies that we only need to solve three linear systems of order m with the same LU decomposition for each i ($i = 0, \dots, N-1$). In addition, we can skip the first $m-1$ forward substitutions for the third system since the first $m-1$ components of the vector at the right hand side are all zeros.

Equation (5.19) only picks $2(N-1)$ specified components from the vector \tilde{x} . There is no computation or communication involved. The evaluation of Eq. (5.20) uses those possible non-zero entries of specified $2(N-1)$ rows of Y together with P to form matrix Z . Again there is no computation or communication involved.

Equation (5.21) solves a $2(N - 1) \times 2(N - 1)$ tridiagonal system, which is the major computation involved in the conquer part of our algorithms. Since Y has at most two non-zero entries at each row, the evaluation of Eq. (5.22) takes four arithmetic operations per row.

Based on the above observations, together with a careful scaling process, the computational complexity required to solve Eq. (5.1) in a sequential processor is stated in the following theorem.

Theorem 4 *Equation (5.1) can be solved using Eqs. (5.17)-(5.23) with $17n - 6N - 23$ operations without pivoting and $24n - 13N - 34$ operations with pivoting.*

5.2 A Compute-Aggregate-Broadcast Computation Solver

Based on the matrix partitioning technique described previously, a parallel algorithm can be easily developed with a compute-aggregate-broadcast computation. This algorithm is called the *Parallel Partition LU* (PPT) algorithm and is described in this section.

Using N processors, the PPT algorithm to solve Eq. (5.1) consists of the following steps:

- Step 0. Allocate A_i , $d^{(i)}$ and elements a_{im} and $c_{(i+1)m-1}$ to the i^{th} node, where $0 \leq i \leq N-1$.
- Step 1. Use the LU decomposition method described in Section 5.1 to solve Eq. (5.24). All N computations can be executed in parallel and independently on N processors.
- Step 2. Send $\tilde{x}_0^{(i)}$, $\tilde{x}_{m-1}^{(i)}$, $v_0^{(i)}$, $v_{m-1}^{(i)}$, $w_0^{(i)}$, $w_{m-1}^{(i)}$ ($0 \leq i \leq n-1$) to a special node to form matrix Z and vector h (Eqs. (5.19) and (5.20)). Hereafter the subindex indicates the component of the vector.
- Step 3. Use the LU decomposition method to solve $Zy = h$ (Eq. (5.21)) on that special node. Note that Z is a $2(N-1)$ dimensional tridiagonal matrix.
- Step 4. Send y_{2i-1} and y_{2i} to the i^{th} node, ($i = 0, \dots, N-1$). Here we set $y_{-1} = y_{2(N-1)} = 0$.
- Step 5. Compute Eqs. (5.22) and (5.23). We have

$$\begin{aligned} \Delta x^{(i)} &= \begin{bmatrix} v^{(i)}, w^{(i)} \end{bmatrix} \begin{bmatrix} y_{2i-1} \\ y_{2i} \end{bmatrix}, \\ x^{(i)} &= \tilde{x}^{(i)} - \Delta x^{(i)}. \end{aligned} \tag{5.25}$$

This step is executed in parallel on N processors.

As mentioned in Section 1.2, the underlying communication mechanism of multicomputers has a great impact on the performance of parallel algorithms. Before we address the communication complexity issue, let's take a close look at different communication mechanisms. In the *store-and-forward* mechanism, which is used in all first generation hypercube multicomputers, the message transfer time between two adjacent processors can be expressed as $\alpha + \beta S$, where α is the communication latency, β is the transmission time per byte, and S is the number of bytes in the message. If a message is delivered to a node h hops away, the message transfer time can be roughly estimated to be $h(\alpha + \beta S)$. In second generation multicomputers, advanced communication mechanisms are adopted, such as the *circuit switching* used in iPSC-2 [46] and the *wormhole routing* used in Ametek 2010 [3]. In these new communication paradigms, the message transfer time is almost independent of the distance (number of hops) between processors. Thus, in second generation multicomputers, the transfer time of a message with S bytes can always be expressed as $\alpha + \beta S$ regardless of the distance the message has to traverse.

The PPT algorithm has two communication patterns. The *data gathering* communication required in Step 2 has to transfer 6 data elements per processor. The *data scattering* communication required in Step 4 has 2 data elements per processor. Figure 5.1 shows these two communication patterns for the case when $N = 8$ and processor 0 is the special node for solving Eq. (5.21).

The best way to handle the data gathering and scattering communications is to use the *tree reduction* technique, as shown in Figure 5.2 for the case of data scattering when $N = 8$. The data gathering communication is simply an inverse of the data scattering. It has been shown that the spanning tree can be perfectly embedded in a hypercube [50]. Thus the communication time required to scatter messages is $\log N$ units for N processors. Note that in the first data transfer, the message has to contain the data for the rest of the processors to be visited. Each processor, upon receiving a message, has to strip its own data and forward the rest of the data to the following processors. As a result, the communication time required in data scattering (and also in data gathering) can be estimated to be

$$\alpha \log N + (N - 1)\beta S. \quad (5.26)$$

Equation (5.26) can be applied to all second generation multicomputers and to first generation hypercube multicomputers using the embedding scheme described in [50]. Now we are ready to state the computation and communication complexities of the PPT algorithm.

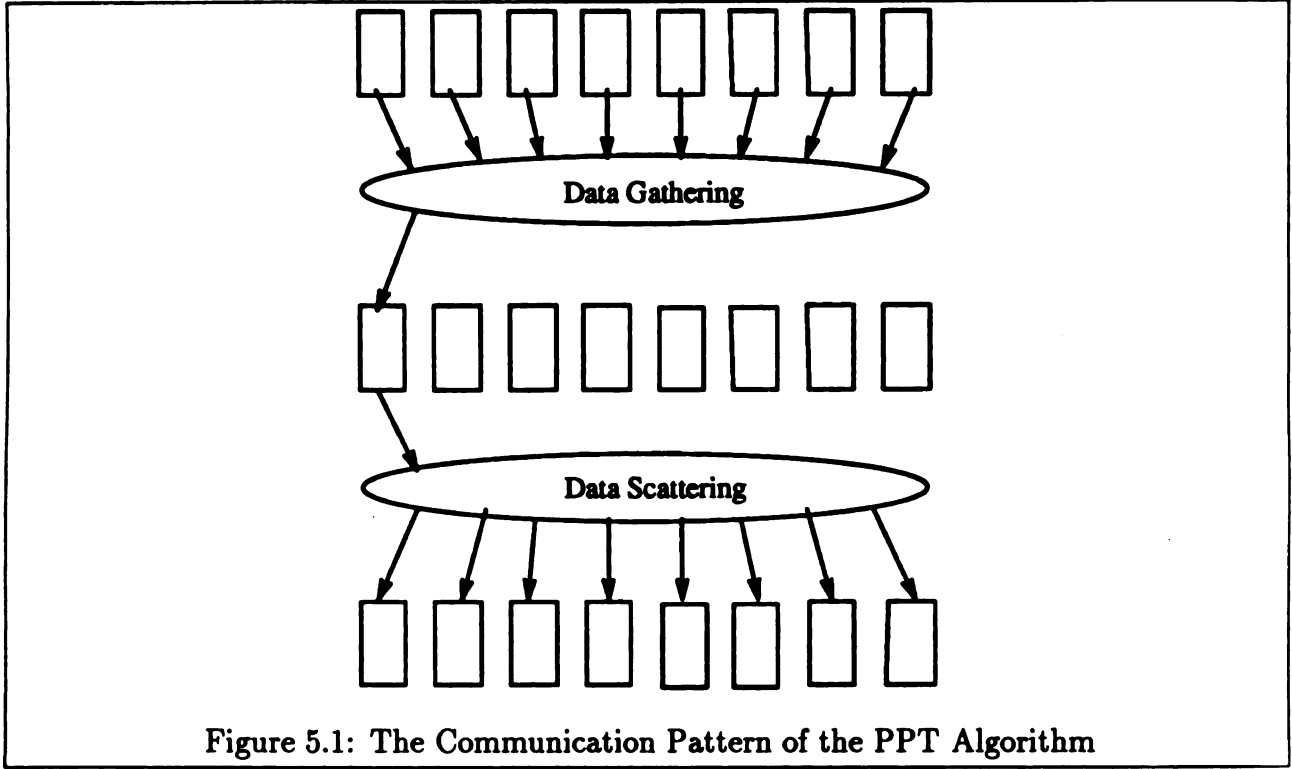


Figure 5.1: The Communication Pattern of the PPT Algorithm

Theorem 5 *The PPT algorithm solves Eq. (5.1) in $17(n/N) + 16N - 45$ and $24(n/N) + 22N - 69$ parallel arithmetic operations for non-pivoting and pivoting, respectively. With 4 bytes per data element, it requires $2(\alpha \log N + 16(N - 1)\beta)$ communication time units, where N is the number of processors and $n = mN$.*

Let T_{LU} , T_{SPT} , and T_{PPT} be the time required to solve Eq. (5.1) using the sequential LU decomposition algorithm, the sequential partitioning algorithm (Eqs. (5.17)-(5.23)), and the PPT algorithm, respectively. Let τ_{comp} represent the unit of a computation operation normalized to the communication time. From those theorems shown previously, we have

$$T_{LU} = (8n - 7)\tau_{comp} \quad \text{without pivoting} \quad (5.27)$$

$$T_{LUP} = (11n - 12)\tau_{comp} \quad \text{with pivoting} \quad (5.28)$$

$$T_{SPT} = (17n - 6N - 23)\tau_{comp} \quad \text{without pivoting} \quad (5.29)$$

$$T_{SPT} = (24n - 13N - 34)\tau_{comp} \quad \text{with pivoting} \quad (5.30)$$

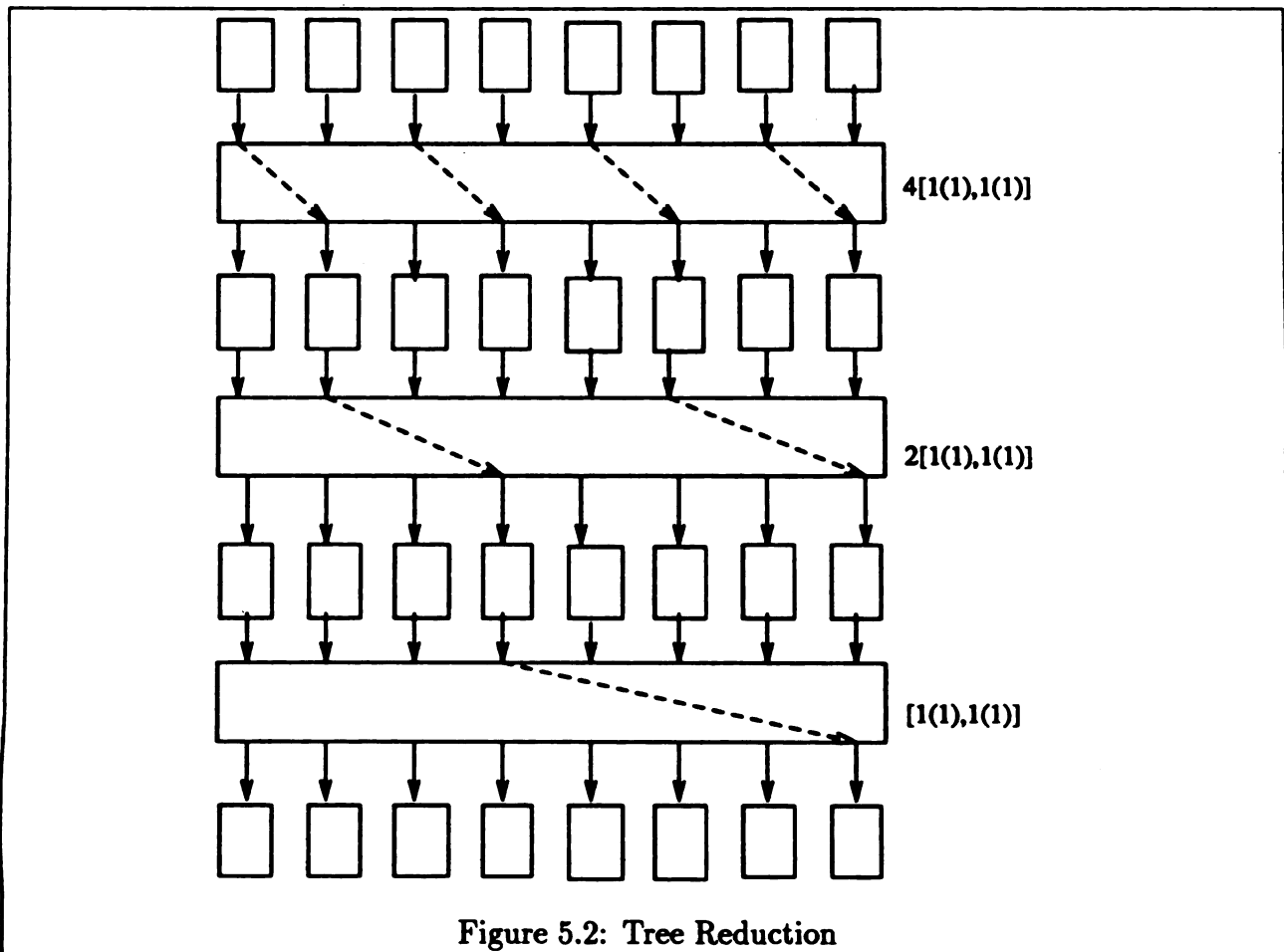
$$T_{PPT} = \left(17\frac{n}{N} + 16N - 45\right)\tau_{comp} + 2(\alpha \log N + 16(N - 1)\beta) \quad \text{without pivoting} \quad (5.31)$$

$$T_{PPT} = \left(24\frac{n}{N} + 22N - 69\right)\tau_{comp} + 2(\alpha \log N + 16(N - 1)\beta) \quad \text{with pivoting} \quad (5.32)$$

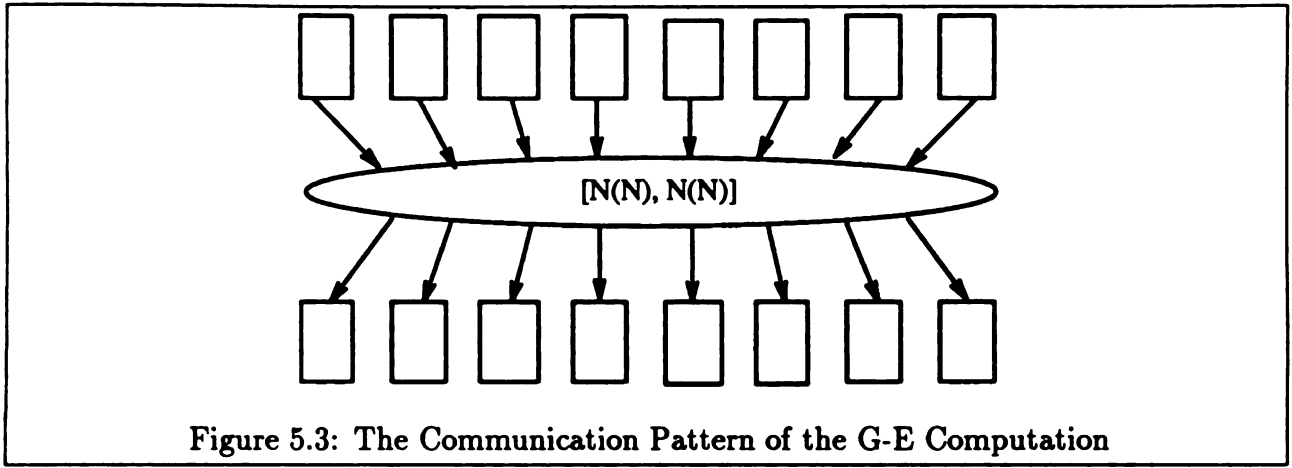
Dividing the time required for a sequential algorithm by the time required to execute the PPT algorithm, an estimated speedup of the PPT algorithm over other sequential algorithms can be obtained. In Section 5.5, the theoretical results obtained here will be compared with the measured results obtained from experiments on a 64-node NCUBE.

5.3 A Global-Exchange Computation Model

From Figure 5.1 we can see that the PPT method is a compute-aggregate-broadcast computation. Step 1 of PPT is the first parallel computation phase. Step 2 is the data aggregation phase. Step 3 is the single node computation phase and step 4 is the broadcast phase. Step 5 is the second parallel computation phase. On hypercube multicomputers, the data aggregation (gathering) and data broadcast (scattering) are achieved by *tree reduction* and *inverse tree reduction*. Both tree reduction and inverse tree reduction take $O(\log N)$ steps, where N is the number of nodes involved in the computation, on hypercube architectures (see Figure 5.2).



Using the communication pattern shown in Figure 3.5, the total data-exchange also can be achieved in $O(\log N)$ time on hypercube architectures. The $O(\log N)$ time communication suggests that we can add redundant computation, reduce the communication overhead and, therefore, improve the overall performance. Step 2 of the PPT method can be changed to total data exchange and step 4 of the PPT method can be removed. Step 3 will be redundant on every working node. This redundant computation combined with step 5 of the PPT method forms a computation phase. With this total data-exchange strategy, we achieve a global-exchange computation method which is called the *Parallel Partition Global-exchange* (PPG) method (see Figure 5.3).



Using P processors, the PPG algorithm for solving Eq. (5.1) consists of the following steps:

Step 0. Allocate A_i , $d^{(i)}$ and elements a_{im} , $c_{(i+1)m-1}$ to the i^{th} node, where $0 \leq i \leq N-1$.

Step 1. Use the LU decomposition method described in Section 5.1 to solve Eq. (5.24). All the N computations can be executed in parallel and independently on N processors.

Step 2. Send $\tilde{x}_0^{(i)}$, $\tilde{x}_{m-1}^{(i)}$, $v_0^{(i)}$, $v_{m-1}^{(i)}$, $w_0^{(i)}$, $w_{m-1}^{(i)}$ ($0 \leq i \leq N-1$) to every other node to form matrix Z and vector h (Eqs. (5.19) and (5.20)).

Step 3. Use the LU decomposition method to solve $Zy = h$ (Eq. (5.21)) on each node and compute Eqs. (5.22) and (5.23) in parallel on N processors.

The computation requirement of the PPG algorithm is the same as the PPT method while the communication requirement is reduced from $2(\alpha \log N + 16(N-1)\beta)$ to $(\alpha \log N + 16(N-1)\beta)$. This reduction is not significant for our machine where $N = 64$, but it may have a larger influence when more processors become available.

5.4 A Solver With More Than One Computation Model

The PPT and PPG algorithms discussed above provide better performance than most of the existing parallel algorithms for solving Eq. (5.1) when $N \ll n$. However, the efficiency of the PPT algorithm decreases as the number of processors, N , increases, because the major computation in the conquer part (Step 3 in Section 5.2) of the PPT algorithm is sequential. For this reason, we use the PP method (see Section 5.1), the limited processor version of the RCD method, to solve Eq. (5.21). In order to apply the PP method, all the superdiagonal elements of the coefficient matrix must be non-zero. The following theorem is needed to apply the PP method to solve Eq. (5.21).

Theorem 6 *If all the superdiagonal elements of the matrix A are non-zero, the tridiagonal matrix $Z = P + E^T \tilde{A}^{-1} V P$ has non-zero superdiagonal elements.*

Proof: The superdiagonal elements of Z are either equal to 1 or the first components of the solutions

$$A_i w^{(i)} = c_{(i+1)m-1} e_{m-1}, \quad i = 1, \dots, N-2, \quad (5.33)$$

where $w^{(i)}, e_{m-1} \in R^m$. Suppose $w_0^{(i)} = 0$, then $w_j^{(i)} = 0$ for $j = 1, \dots, m-1$, since the superdiagonal elements of A_i are non-zero. Therefore, we have $A_i w^{(i)} = 0$, which is a contradiction to Eq. (5.33). \square

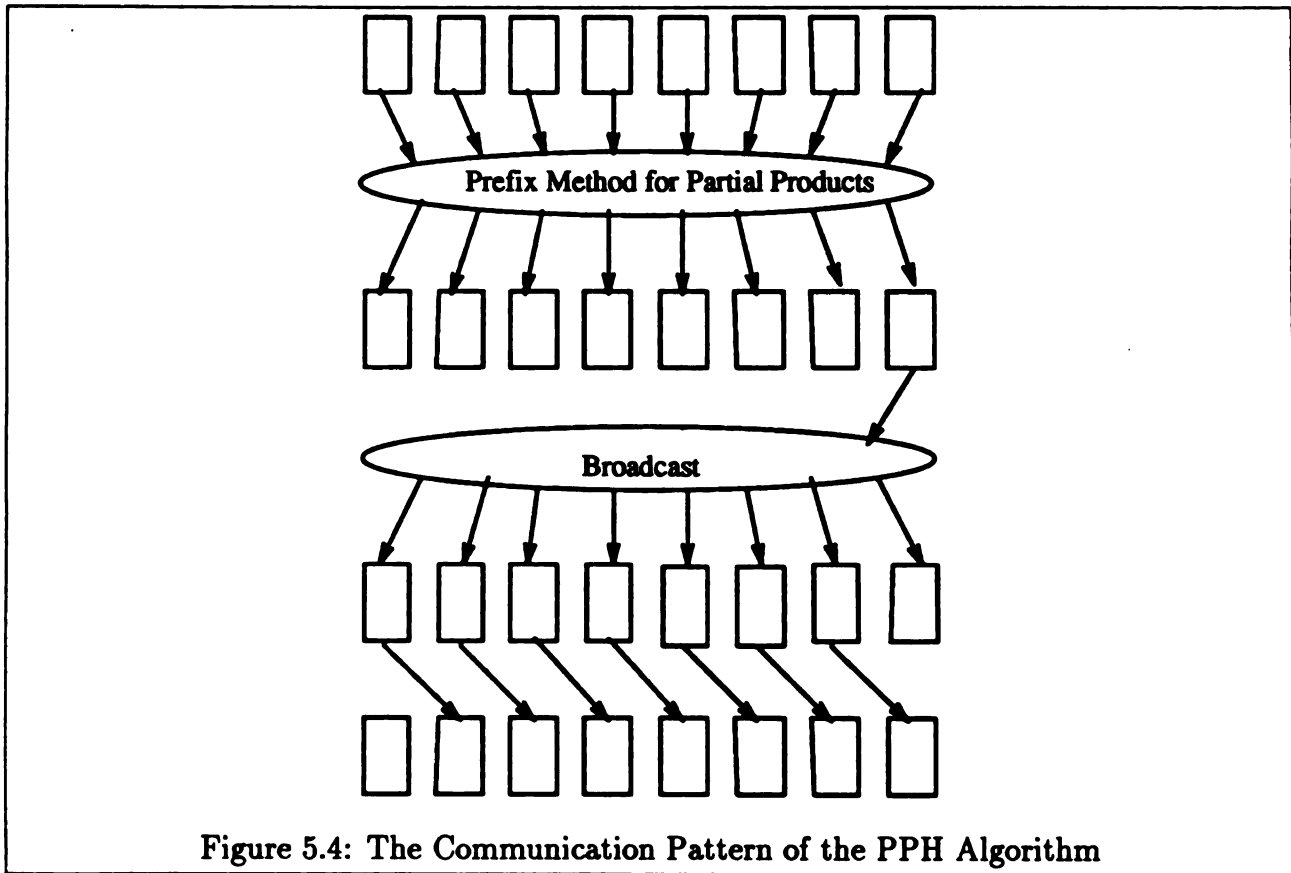
The new algorithm, namely the *parallel partition hybrid* (PPH) algorithm, consists of two computation models, the recursive doubling computation model and the compute-aggregate-broadcast computation model. The PPH algorithm is similar to the PPT algorithm except for the following changes. After Step 1 of the PPT algorithm, the $(2i-1)^{th}$ and $2i^{th}$, for $i = 0, \dots, N-1$, rows of the tridiagonal matrix Z are stored in the i^{th} node (here the $(-1)^{th}$ and $2(N-1)^{th}$ rows of Z are assumed to be zero). Step 2 in the PPT algorithm which performs data gathering is eliminated. Step 3 of the PPT algorithm is then replaced by the PP method as described in Section 2. Step 4 performs the following data transfer.

Step 4. Send y_{2i+1} from the i^{th} node to the $(i+1)^{th}$ node for $i = 0, \dots, N-2$.

In Step 3 the PP method has two communication patterns. The first communication pattern computes all partial matrix products (see Figure 3.13). The second communication pattern is a *broadcast* which broadcasts the computed x_0 (or X_0 with another two known components, 0,1) to all other nodes. Broadcast has a communication pattern similar to data

scattering as shown in Figure 5.2. However, in broadcast all nodes receive the same data. Here we assume each floating point number has 4 bytes. Considering second generation multicomputers, based on the communication model discussed in Section 5.2, the communication time required to obtain all partial products of B_i (Figure 3.13) is $(\alpha + 24\beta)(1 + \log N)$ time. Here we have $S = 24$ because each message transfer includes 6 data elements. The broadcast communication takes $\log N$ steps and the message is one data element, which takes $(\alpha + 4\beta)\log N$ time. The shift communication required in Step 4 takes $\alpha + 4\beta$ time. The communication pattern of the PPH algorithm is shown in Figure 5.4 for $N = 8$. Counting the arithmetic operations and communication times, we have the following theorem.

Theorem 7 *With $n = Nm$, the PPH algorithm solves Eq. (5.1) in $17(n/N) + 20\log N + 17$ and $24(n/N) + 20\log N + 4$ parallel arithmetic operations for non-pivoting and pivoting, respectively. It requires $((\alpha + 4\beta) + (\alpha + 24\beta))(1 + \log N)$ communication steps.*



Let T_{SPH} be the time required to execute the PPH algorithm in a sequential processor and T_{PPH} be the time required to execute the PPH algorithm on N processors. We then have

Algorithm	Computation	Communication
Wang's Partition	$21\frac{n}{N} - 12 - 8\frac{n}{N^2}$ *	$2(N-1)(\alpha + 10\beta) + (\alpha + 16\beta)$
Recursive Doubling(PP)	$35\frac{n}{N} + 20\log N - 29$	$(1 + \log N)(\alpha + 24\beta) + \log N(\alpha + 4\beta)$
Cyclic Reduction	$20\frac{n}{N}$ *	$O(\log n)$ **
PPT(non-pivoting)	$17\frac{n}{N} + 16N - 45$	$2[\alpha \log N + 16(N-1)\beta]$
PPT(pivoting)	$24\frac{n}{N} + 22N - 69$	$2[\alpha \log N + 16(N-1)\beta]$
PPH(non-pivoting)	$17\frac{n}{N} + 20\log N + 17$	$[(\alpha + 4\beta) + (\alpha + 24\beta)](1 + \log N)$
PPH(pivoting)	$24\frac{n}{N} + 20\log N + 4$	$[(\alpha + 4\beta) + (\alpha + 24\beta)](1 + \log N)$

*scalar count of Table V [62] divided by N

**the number of communication steps for the cyclic reduction method is not known. However it is easy to find that it has at least the complexity of $O(\log n)$.

Table 5.1: Computation and Communication Counts of Tridiagonal Solvers

$$T_{SPH} = (17n + 8N - 41) \tau_{comp} \quad \text{without pivoting}$$

$$T_{SPH} = (24n - 5N - 41) \tau_{comp} \quad \text{with pivoting}$$

$$T_{PPH} = \left(17\frac{n}{N} + 20\log N + 17\right) \tau_{comp} + ((\alpha + 4\beta) + (\alpha + 24\beta))(1 + \log N) \quad \text{without pivoting}$$

$$T_{PPH} = \left(24\frac{n}{N} + 20\log N + 4\right) \tau_{comp} + ((\alpha + 4\beta) + (\alpha + 24\beta))(1 + \log N) \quad \text{with pivoting}$$

It is interesting to note that the PPH algorithm can reach a speedup of 2 over the PP algorithm for $1 < N \ll n$. When $N = n$, no matrix partitioning is needed and the PPH algorithm is virtually the RCD algorithm. When $N=1$, there is no conquer part and as the LU decomposition method is used in the dividing part, the algorithm becomes the LU decomposition algorithm.

5.5 Comparison and Experimental Results

In this section we compare our methods with some existing methods for solving Eq. (5.1). The arithmetic operation counts and communication steps of those methods are summarized in Table 5.1.

Wang's partition method and cyclic reduction method are well known. However, it is difficult to compare these two methods with our methods on multicomputers. Although we believe that the limited processor version of these two methods will increase the computation complexity, we list their computation counts in Table 5.1 assuming they can be perfectly partitioned, i.e., the computation count is scaled down by a factor of N . The communication

cost of cyclic reduction is in the order of $O(\log n)$ steps and each step takes at least $\alpha + S\beta$ time. Michielse and Vorst modified Wang's algorithm for local memory systems. The communication count is based on their result [42]. However, the computation count in their modification is slightly greater than Wang's algorithm and thus is not listed in the table. As shown in Table 5.1 the computation counts of all our methods are better than other methods. In terms of communication counts, the PPH method has the same order as the PP method. This is reasonable because they all use the communication pattern shown in Figure 3.13. The cyclic reduction method has a high communication count, and thus is popular for vector machines. For our other methods, the communication complexity is less than Wang's method. In the following discussion, we shall compare our methods with respect to the number of processors.

Figure 5.5 shows the estimated and measured speedup of the PP, PPT, PPH algorithms with respect to the SGTSL routine (see Section 5.1). These algorithms are implemented on a 64-node NCUBE multicomputer. NCUBE is a first generation multicomputer adopting the store-and-forward communication mechanism. As indicated in Section 5.2, the communication pattern for the parallel prefix method shown in Figure 3.13 cannot be perfectly embedded in hypercube, and the worst dilation cost is 2. Thus, in the communication counts of the PP and PPH algorithms, the factor $(\alpha + 24\beta)$ is doubled. In our NCUBE machine, the following system parameters are measured: $\alpha = 5.0$, $\beta = 0.013$, and $\tau_{comp} = 0.08$ (without normalization). The dimension of matrix A is chosen as $n = 6400$. This value is limited by the memory constraint of individual processors. As N increases, the PPH algorithm will outperform the PPT algorithm because the dimension of the matrix Z (Eq. (5.21)) increases as N increases, which favors the parallel approach used in the PPH algorithm. Again, due to the limited number of processors available in our NCUBE, the maximum number of processors used is $N = 64$. The performance of the PPH algorithm seems to be underestimated compared with the measured results. This is mainly caused by assuming a dilation of 2 for all communications occurring in Figure 3.13. However, in actual implementation, some communications have a dilation of 1.

As n goes to infinity, the asymptotic speedup, compared with the LU decomposition method, of all our methods is $0.471N$. The asymptotic speedup for the PP method is $0.229N$. Dividing the speedup by the number of processors, N , Figure 5.6 shows the *efficiency* of our methods. For all methods, the efficiency decreases as the number of processors N increases. This is mainly the result of the increasing ratio of communication to computation.

As mentioned in Chapter 2, there are two commonly accepted measures for the speedup

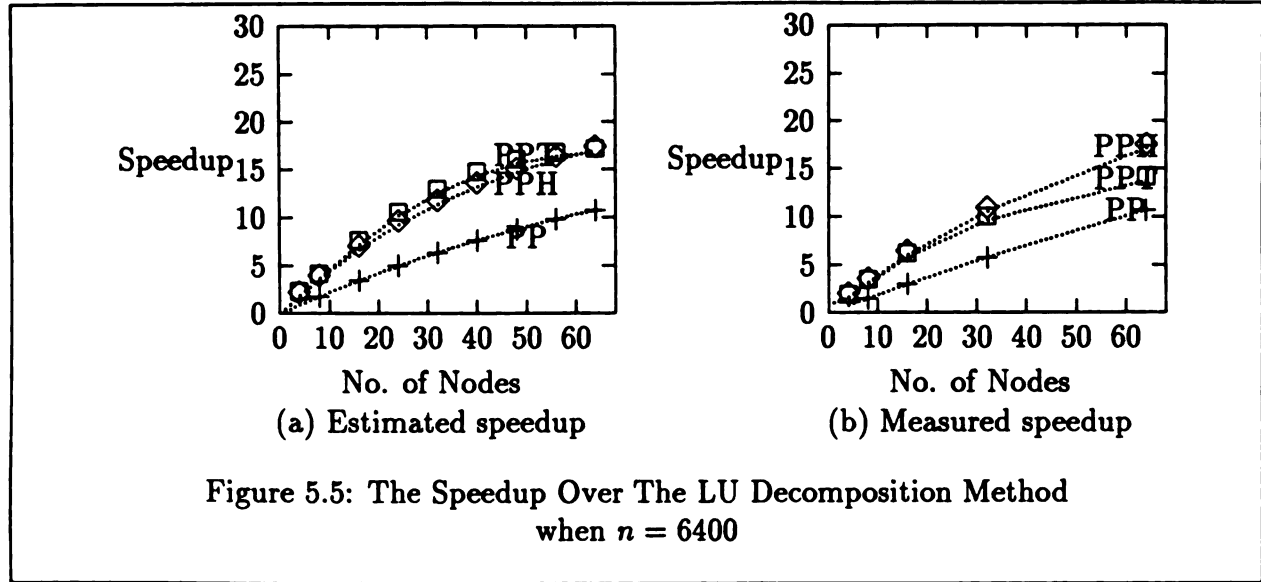


Figure 5.5: The Speedup Over The LU Decomposition Method when $n = 6400$

and the efficiency of parallel algorithms [47]. One focuses on how much faster a problem can be solved by N processors. Thus, it compares the best serial algorithm with the parallel algorithm under consideration, which is defined as

$$S_p = \frac{\text{execution time using the fastest sequential algorithm on one processor}}{\text{execution time using the parallel algorithm on } N \text{ processors}}. \quad (5.34)$$

Both Figures 5.5 and 5.6 are based on the above measure and the best sequential algorithm chosen is the LINPACK subroutine, SGTSL. Another measure is interested in the parallelism inherent in an algorithm and is defined as

$$S'_p = \frac{\text{execution time using one processor}}{\text{execution time using } N \text{ processors}}. \quad (5.35)$$

The latter is the *relative speedup*. Since each node only has 512K memory capacity for NCUBE multicomputers, the single node execution for solving a 6400 dimension tridiagonal system is impossible. The estimated self speedup and efficiency of our methods can be found in [60].

One of the advantages of our approach is its diversity. In the divide part, pivoting may or may not be used. There has been a tacit assumption for the first three methods listed in Table 5.1 that no pivoting is required. In fact, it does not appear to be feasible to incorporate a pivoting strategy into those algorithms (Wang's algorithm allows very limited pivoting). However, pivoting can be used by the PPT, PPG and PPH algorithms with slightly higher operation counts. Thus, the PPT, PPG and PPH algorithms are more stable than others in cases where pivoting is required. In the conquer part, the major computation requirement is

U
c
v
f
s
u
a
s
t
t
g
p
a
o
D

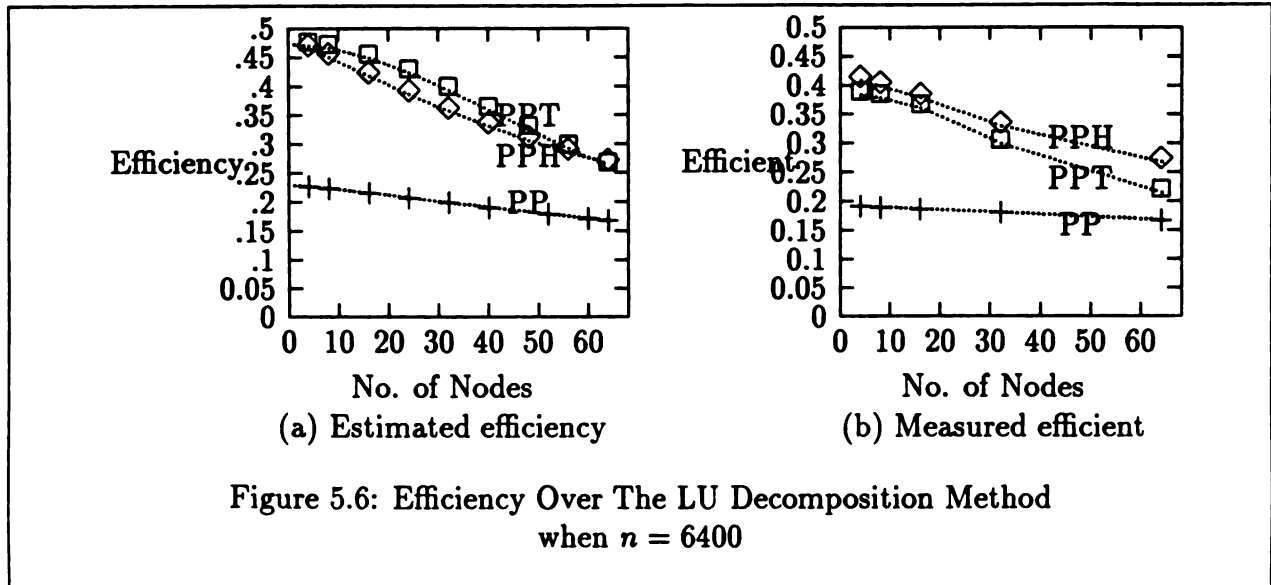


Figure 5.6: Efficiency Over The LU Decomposition Method
when $n = 6400$

devoted to solving Eq. (5.21). The methods used in this part are irrelevant to the methods used in the divide part. Other strategies may also be used to solve Eq. (5.21).

Unless the matrix A is positive definite or diagonal dominant, there is no general rule to guarantee that all the submatrices A_i ($i = 0, \dots, N - 1$) of \tilde{A} (see Section 5.1) are non-singular. The small matrix Z in the conquer part may be singular too. The latter case is unlikely to happen. However, for a certain class of matrices at hand, very often we can find a criterion to avoid those singularities and make the algorithms work.

In this chapter, using the solution of tridiagonal linear system as an example, we have shown how to change the algorithm design to fit the architecture and to reduce communication overhead. We have changed our design from compute-aggregate-broadcast computation to global exchange computation and to hybrid computation for better performance. In general, we may have different approaches for a given application. The performance of computation models will suggest which approach is best for a given architecture, and we can achieve a more efficient algorithm design. Another new parallel algorithm, *Parallel Diagonal Dominant* (PDD) is also proposed for solving diagonal dominant tridiagonal systems. Detailed information about the PDD algorithm can be found in [60].

Chapter 6

PREDICTION OF PERFORMANCE

Performance measurement techniques can be divided into three categories: *pre-run*, *run-time* and *post-run* [12]. Pre-run performance measurement is also called performance prediction. It identifies the computationally intensive parts of an algorithm and predicts the performance of an algorithm. Run-time performance measurement collects execution data that can be analyzed on line. The post-run performance environment provides the measured implementation results. Comparing with run-time and post-run performance measurements, performance prediction is more economical for algorithm evaluation, and, without actual implementation, it is not restricted by physical limitations which may be imposed by the number of available processors and/or the maximum memory capacity. The predicted results not only can be used for evaluating algorithms, they also can be compared with the implementation results. The prediction and implementation results comparison often leads to a better understanding of the algorithm and to a possible performance enhancement. Further, performance prediction is very important for real time systems. In real time systems, the machines have to respond within a given deadline. Within the time limitation, we would like to obtain the most accurate solution possible. Knowing the turnaround time before execution is the key to achieving this goal. However, providing an accurate prediction of the performance of parallel algorithms is difficult. As we have learned from previous chapters, the performance of a parallel algorithm is dependent on the application, the number of processors available and the problem size.

Execution time can be predicted by using a parallelism profile (see Figure 2.1). A parallelism profile provides information about uneven allocation degradation. The amount of work executed with degree of parallelism i , W_i , can be obtained from this information and a lower bound of elapsed time can be derived based on W_i . Equation 2.7

$$T(N) = \sum_{i=1}^m \frac{W_i}{i\Delta} \lceil \frac{i}{N} \rceil \quad (6.1)$$

is the predicted elapsed time based on the parallelism profile. $T(N)$ is a lower bound of the

execution time, but it certainly is not a greatest lower bound. Communication latency is a very important factor influencing the performance. By considering communication overhead, a better lower bound for execution time is

$$T(N) = \frac{\sum_{i=1}^m \frac{w_i}{i} \lceil \frac{i}{N} \rceil + Q_N}{\Delta}, \quad (6.2)$$

where Q_N is the communication overhead when N processors are used (see Eq.2.9).

Structured representation provides adequate information for both uneven allocation and communication latency degradations. With some measured data, the performance of an application can be predicted based on its structured representation and Eq.(6.2). This performance prediction can be obtained by collecting information from each compute and data-exchange phase. In this approach, the parallel computation requirement of each compute phase should be known and the performance of each communication primitive should be premeasured. This performance prediction can also be obtained through a modular approach by using the performance information of computation models. This modular approach has many advantages over the detailed approach, which we will discuss in the next section.

This chapter is organized as follows. In Section 6.1, we will show the performance formulations of two computation models, the domain decomposition computation model and the recursive doubling computation model. An example is presented in Section 6.2 to illustrate how the computation models and the memory-bounded speedup model can be used in performance prediction. The influence of problem size on the speedup of computation models is analyzed in Section 6.3.

6.1 Performance Formulations

The most commonly used performance metrics for parallel algorithms are elapsed time and speedup. In this section, the performance formulations of two parallel computation models are derived based on their structured representations, in terms of elapsed time and in terms of speedup. Performance formulations for other computation models can be derived by following similar arguments. These performance formulations of computation models can be used to predict the performance of a large class of scientific applications.

Using the performance of computation models to predict the performance of a given application has advantages over predicting the performance of the given application directly through its structured representation. With the computation model approach, an application is divided into parts. The performance of these parts, or computation models, will suggest

which section of your algorithm is the bottleneck. Second, the performance of an application is not only dependent on the application and the machine architecture, but also dependent on how the application is mapped onto the architecture. The performance information of computation models contains the mapping information. It provides a more accurate estimate. Also, the computation models are naturally associated with commonly used mathematical methods. With the modular approach, when a mathematical method is chosen, the structured representation of a given application will be known, and the performance prediction of the given application is available. The computation model based modular approach is easy to understand, has less error accumulation, requires less communication and less architectural knowledge.

Figure 3.11 in Section 3.2 shows a structured representation for a domain decomposition application. In general, the structured representation of the domain decomposition computation model is

$$\sum_{i=1}^k ([N(d), N(d)] + X_i),$$

where N is the number of processors available, k is an integer, which sometime depends on the convergence check, and d is the degree of each sender and each receiver. Studying the domain decomposition computation more carefully, we find that it has the following characteristics.

- *Communication overhead is independent of system size*

The degrees of senders and receivers are fixed for domain decomposition applications. It is independent of problem size and system size. The communication pattern is also independent of problem size and system size. For second generation multicomputers, the communication latency of the domain decomposition computation model is fixed. In fact, even for first generation multicomputers we generally can manage to map a domain decomposition onto an architecture such that the communication delay is independent of the system size. We use Q to denote the common communication latency.

- *Perfect degree of parallelism*

The domain decomposition applications do not have uneven allocation degradation. All the processors do useful work at each compute phase and the loads are evenly distributed. Domain decomposition computations are good candidates for parallel processing.

- With a fixed number of iterations at each compute phase, the computation requirement at each compute phase is the same, $X = X_i$ for $i = 1, \dots, k$. This is the case for most domain decomposition applications.

The execution time of domain decomposition applications can be derived from the structured representation directly. The sequential execution time is

$$T(1) = \frac{N(\sum_{i=1}^k X_i)}{\Delta},$$

where Δ is the computing capacity of each processor (see Section 2.1). Notice that, as defined in Section 2.1, $W_N = N(\sum_{i=1}^k X_i)$ for domain decomposition computations. The parallel execution time for domain decomposition computations will be

$$T(N) = \frac{(\sum_{i=1}^k X_i) + Q}{\Delta}.$$

Three different speedup formulations can be derived for the domain decomposition model by following the three different models of speedup given in Section 2.2. For the fixed-size speedup model the speedup is

$$S_N = \frac{W_N}{Q + \frac{W_N}{N}}. \quad (6.3)$$

For the fixed-time speedup model, we have

$$\begin{aligned} W_N &= \frac{W'_N}{N} + Q \\ W'_N &= N(W_N - Q) \end{aligned}$$

where W'_N is the scaled workload for fixed-time speedup (see Section 2.1). Thus, the fixed-time speedup is

$$S'_N = \frac{W'_N}{\frac{W'_N}{N} + Q} = \frac{N(W_N - Q)}{W_N}, \quad (6.4)$$

For the memory-bounded speedup model, if we let the integer k be fixed, we have

$$W_N^* = NW_N,$$

where W_N^* is the scaled workload (see Section 2.1). The memory-bounded speedup is

$$S_N^* = \frac{NW_N}{Q + W_N}. \quad (6.5)$$

The recursive doubling computation model is another computation model presented in Section 3.2 (see Figure 3.13). The structured representation of the recursive doubling computation model is

$$\sum_{i=0}^{k-1} ((2^k - 2^i)[1(1), 1(1)] + X_i), \quad k = \log N. \quad (6.6)$$

The recursive doubling computation model has interesting properties. It has a dynamic degree of parallelism and has a fixed computation requirement at each compute phase while the number of compute phases increases with the system size. More precisely, this model has the following characteristics.

- The computation requirements at each compute phase are the same, $X = X_i$ for $i = 1, \dots, k$.
- Communication at data-exchange phases is regular data-exchange. The communication patterns at each data-exchange phase are the same. Although the number of disjoint patterns varies, communication latency at each data-exchange phase is the same.
- The number of compute and data-exchange phases will increase as a function of $\log(\text{system size})$.
- In both fixed-size and fixed-time speedup models, if Q is fixed, X will decrease when N increases.
- The degree of parallelism varies from phase to phase. $P \times R$ gives the degree of parallelism at each compute phase, which is equal to $2^k - 2^i$ at the i^{th} compute phase. This means that $W_{2^k - 2^i} = (2^k - 2^i)X = (2^k - 2^i)W_1$, and $W_j = 0$ for $j \neq 2^k - 2^i$, where $i \in [0, 1, \dots, k - 1]$.

Based on the structured representation (6.6), the sequential execution time of the recursive doubling computation is

$$T(1) = \frac{\sum_{i=0}^{k-1} (2^k - 2^i)X}{\Delta}.$$

The parallel execution time is

$$T(N) = \frac{k(X + Q)}{\Delta}.$$

Just as with the domain decomposition computation model, three speedup formulations can be obtained corresponding to the three models of speedup previously discussed. Following the notation defined in Section 2.1, we let $W = \sum_{i=0}^{k-1} W_i$ be the work load when the application is running on a single processor, and define W'_i and W_i^* correspondingly. For the fixed-size speedup model, we have

$$\begin{aligned} S_N &= \frac{W}{\sum_{i=0}^{k-1} [\frac{W_i}{j} + Q_j]}, \quad j = 2^k - 2^i \\ &= \frac{W}{\sum_{i=0}^{k-1} [W_1 + Q]} = \frac{W}{k[W_1 + Q]} \\ &= \frac{W}{k[\frac{W}{\sum_{i=0}^{k-1} (2^k - 2^i)} + Q]}. \end{aligned}$$

Thus, notice here that $\sum_{i=0}^{k-1} (2^k - 2^i) = kN - (N - 1)$,

$$S_N = (N - \frac{N-1}{k}) \frac{W}{W + Q[\sum_{i=0}^{k-1} (2^k - 2^i)]} \quad (6.7)$$

$$= (N - \frac{N-1}{k}) \frac{W}{W + [(k-1)N+1]Q}. \quad (6.8)$$

By definition, the speedup formulation of the recursive doubling model for the fixed-time speedup is

$$\begin{aligned} S'_N &= \frac{\sum_{i=0}^{k-1} W'_i}{\sum_{i=0}^{k-1} [\frac{W'_i}{j} + Q_j]}, \quad j = 2^k - 2^i \\ &= \frac{\sum_{i=0}^{k-1} (2^k - 2^i) W'_1}{\sum_{i=0}^{k-1} [W'_1 + Q]} = (N - \frac{N-1}{k}) \frac{W'_1}{W'_1 + Q}. \end{aligned}$$

Notice that

$$W = \sum_{i=0}^{k-1} [\frac{W'_i}{j} + Q] = k(W'_1 + Q),$$

and

$$W'_1 = \frac{W - kQ}{k},$$

we have

$$S'_N = (N - \frac{N-1}{k}) \frac{W - kQ}{W}. \quad (6.9)$$

Similarly, for memory-bounded speedup we also have

$$S_N^* = (N - \frac{N-1}{k}) \frac{W_1^*}{W_1^* + Q}.$$

The workload on each processor will not change with the number of processors available in memory-bounded speedup. Thus, the memory-bounded speedup is

$$S_N^* = (N - \frac{N-1}{k}) \frac{W}{W+Q}. \quad (6.10)$$

From Eq. (6.8) we can see that with the degree of parallelism considered, the speedup formulation can be completely different from the simplified speedup formulas presented in Section 2.3.

The structured representation of a given application contains adequate information about degradations, and provides more information than a parallelism profile. By using the domain decomposition and recursive doubling computation models, we have shown how elapsed time and speedup can be predicted based on structured representations. We also have shown how to derive speedup formulations based on structured representations for different models of speedup. Performance formulations of other computation models can be derived similarly. The performance formulations of computation models can be used to predict the performance of general scientific applications.

6.2 Structured Prediction

We have identified computation models and have shown performance formulations for some of the identified computation models. We would like to predict the performance of general scientific applications by using the performance information of parallel computation models. The idea is that a given application can be seen as a combination of computation models. These computation models can be found by studying the structured representation of the application. The performance information of these computation models can then be combined to provide a performance prediction of this application. If the measured performance data of computation models are applied to a performance prediction, performance data can be estimated for this application. If the performance formulations of computation models are applied, a performance formula can be obtained for the application which shows how different parts of the application will vary with system size and problem size. Therefore, different parts can be modified for different situations to achieve the best performance. We call this kind of analysis *structured analysis*.

The combination of elapsed times is straightforward, but the combination of speedups is somewhat more complex. The combination formulas for the elapsed times and speedups are shown in Eq.(6.11) and Eq.(6.12) respectively for the combination of two models. The

combination formulas in the case of n models can be easily obtained by using Eq. (6.11) and Eq.(6.12) recursively.

$$\text{Execution time} = \text{Execution time of part one} + \text{Execution time of part two} . \quad (6.11)$$

$$\begin{aligned} \text{Speedup} = & \text{Speedup of part one} \times \text{Ratio of part one} \\ & + \text{Speedup of part two} \times \text{Ratio of part two} , \end{aligned} \quad (6.12)$$

where the ratio is the ratio over total execution time. Speedup can be derived from the execution time. We would like to have speedup as a function of the speedups of the computation models, since the variations of each model are easier to observe. Also, determining speedup from Eq. (6.12) is simpler than determining speedup from Eq. (6.11), when the ratio is independent of problem size and system size.

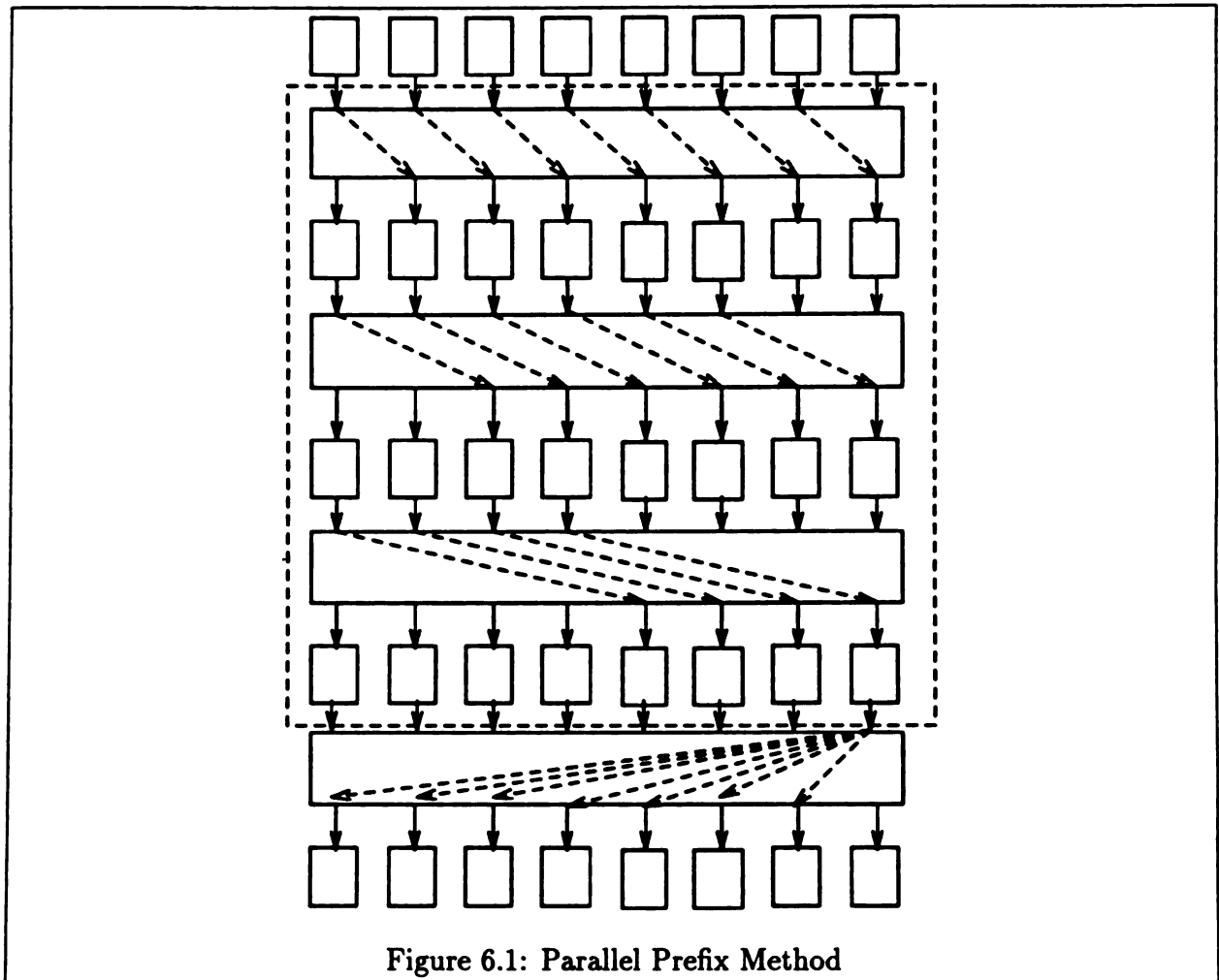
We use one simple example, the parallel prefix method for solving tridiagonal systems (see Figure 6.1), to illustrate this performance prediction approach. The parallel prefix method has the following structured representation

$$X_0 + \underbrace{\sum_{i=0}^{k-1} (2^k - 2^i)([1(1), 1(1)] + X_{i+1}) + [1(N), N(1)]}_{\text{Recursive Doubling}} + X_{k+1}, \quad (6.13)$$

where the compute phases are counted from 0 to $k + 1$. From this representation we can see that the parallel prefix method consists of three parts, the preprocessing part, the recursive doubling part and the post processing part. The preprocessing part and the post processing part are local computations in which the workload is evenly distributed among the available processors. The middle part is a recursive doubling computation.

The degree of parallelism of the postprocessing part is equal to $P \times S$, where P is the number of patterns in the broadcast data-exchange phase, and S is the number of senders in each of the patterns. Since $P = 1$ and $S = N$ in this case, $P \times S = N$. The degree of parallelism of the preprocessing part is also equal to N . Since recursive doubling computation models have equal load at each compute phase, $X_1 = X_i, i = 1, \dots, k - 1$. Also, since recursive doubling computations do not achieve perfect parallelism at any compute phase, $W_N = N(X_0 + X_{k+1})$. Using the recursive part as a basic structure, the sequential execution time of the prefix method is

$$\frac{[NX_0 + \sum_{i=0}^{k-1} (2^k - 2^i)(X_1 + Q) + Q_N + NX_{k+1}]}{\Delta}, \quad (6.14)$$



and the predicted parallel execution time of the prefix method is

$$\frac{[X_0 + k(X_1 + Q) + Q_N + X_{k+1}]}{\Delta}, \quad (6.15)$$

where Δ is the computing capacity of each processor, Q is the communication overhead of data-exchange $[1(1), 1(1)]$ and Q_N is the communication overhead of data-exchange $[1(N), N(1)]$. Q is independent of N , the number of processors available, and Q_N is a function of N . After removing the computing capacity Δ , Eq.(6.15) becomes

$$[X_0 + k(X_1 + Q) + Q_N + X_{k+1}]. \quad (6.16)$$

Using Eq.(6.12) the speedup of the prefix method is

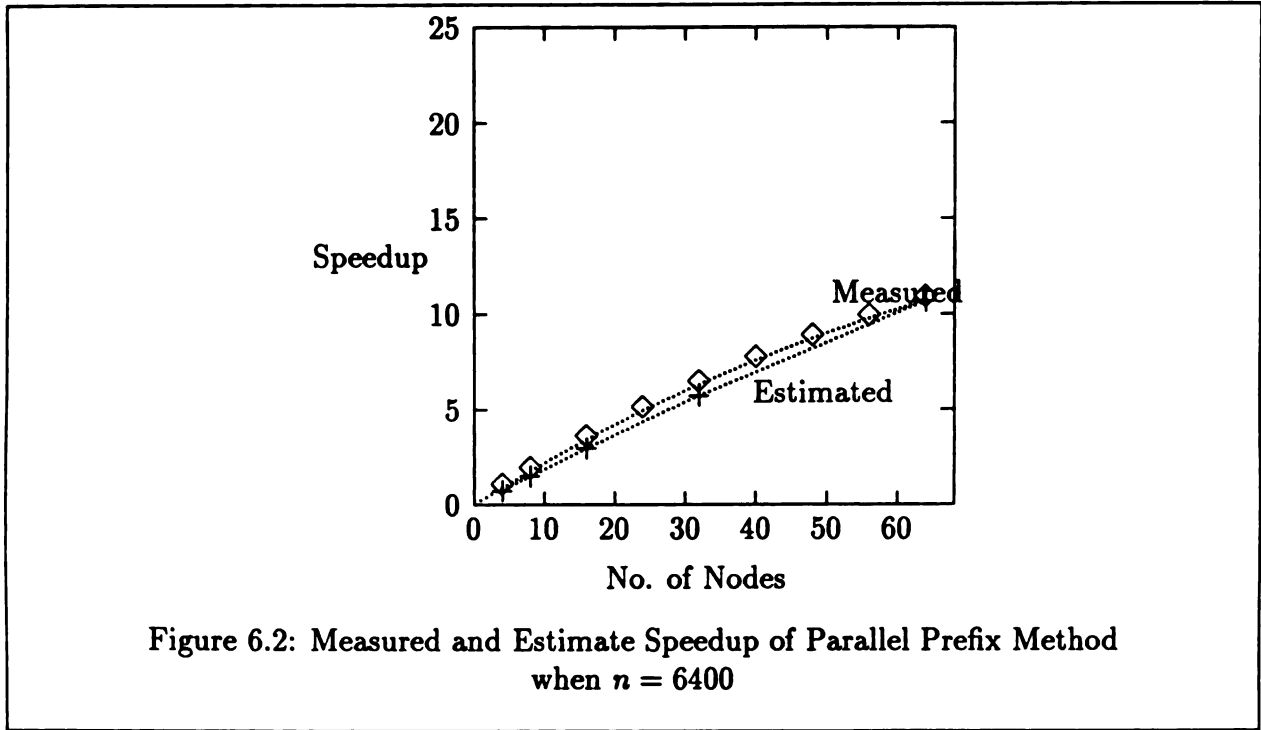
$$\begin{aligned} & \frac{NX_0}{X_0} \frac{X_0}{[Eq.(6.16)]} + (N - \frac{N-1}{k}) \frac{X_1}{X_1+Q} \frac{k(X_1+Q)}{[Eq.(6.16)]} + \frac{NX_{k-1}}{X_{k-1}} \frac{X_{k-1}}{[Eq.(6.16)]} \\ = & N \frac{X_0+X_{k-1}}{[Eq.(6.16)]} + (N - \frac{N-1}{k}) \frac{X_1}{X_1+Q} \frac{k(X_1+Q)}{[Eq.(6.16)]} \end{aligned} \quad (6.17)$$

where X_1 is fixed. It is independent of system size and problem size. Equation (6.18) shows that the prefix method contains two parts, the local computation part and the recursive doubling computation part. The local computation part has speedup N and the recursive doubling part has speedup $(N - \frac{N-1}{k}) \frac{X_1}{X_1+Q}$. The local computation part has a higher speedup. We would like the local computation part to dominate the total computation. From Eq.(6.18) we can see that when the problem size increases so does the ratio of local computation, and when the system size increases the ratio of local computation will decrease. Figure 6.2 shows the estimated and measured speedup of the prefix method. We can see that they are very close to each other.

By the properties of fixed computation at each compute phase, the speedup formulation used for the recursive doubling part is the memory-bounded speedup. Thus, using one simple example, we have shown how the computation models and memory-bounded speedup can be used in performance predictions.

6.3 The Influence of Problem Size on Speedup

In the last two sections we showed how the performance of general applications can be predicted using the performance information of computation models. We also derived performance formulations for some computation models. In this section we will study the



performance of computation models in more depth. We will study how problem size influences performance. The study starts with general formulas and then special cases will be noted. These special cases are those that are suitable for some computation models.

As shown in Chapter 2, speedup is a function of both problem size and the number of processors available. The study is three fold. First, we only consider the simplified condition discussed in Section 2.3. Then, communication overhead is considered. Finally, uneven allocation degradation is added in to give a more accurate analysis.

According to Eq. (2.13), the simplified fixed-size speedup is

$$S_N = \frac{W_1 + W_N}{W_1 + \frac{W_N}{N}}. \quad (6.19)$$

Notice that by Eq. (2.1)

$$t_i(1) = \frac{W_i}{\Delta}.$$

We can rewrite Eq. (6.19) as

$$S_N = \frac{t_1(1) + t_N(1)}{t_1(1) + \frac{t_N(1)}{N}}. \quad (6.20)$$

In general, we have

$$t_i = t_i(1) = \frac{W_i}{\Delta} \quad \text{for } 1 \leq i \leq m,$$

where m is the maximum degree of parallelism (see Section 2.1). When $t_i = 0$ for $i \neq 1$ and $i \neq N$, the simplified speedup Eq. (6.19) can be rewritten as a function of workload W ,

$$S_N(W) = \frac{T_1(W)}{T_N(W)} = \frac{t_1(W) + t_N(W)}{t_1(W) + t_N(W)/N}. \quad (6.21)$$

Equation (6.21) is equivalent to Eq. (6.19), but execution time has been written as a function of problem size. Therefore, Eq. (6.21) is easier to use to analyze the influence of problem size. In this section, instead of writing speedup in terms of workload, we write speedup formulations in terms of execution time $t_i(1)$, where the subscript i indicates the execution of workload W_i and 1 indicates that only one processor is used in the execution.

If the sequential execution time t_1 and the perfectly parallel portion of the execution time t_N increase with the problem size W at the same rate, then when the problem size increases c times

$$S_N(cW) = \frac{T_1(cW)}{T_N(cW)} = \frac{c_1 t_1(W) + c_1 t_N(W)}{c_1 t_1(W) + c_1 t_N(W)/N} = S_N(W). \quad (6.22)$$

Thus, in this case, speedup Eq. (6.21) is independent of problem size.

The problem becomes more complicated when communication overhead is considered and the communication overhead is a function of problem size. In this case we could let the sequential execution time, t_1 , be independent of the problem size or we could let it vary with the problem size. In the former case, we have

$$S_N(W) = \frac{t_1 + t_N(W)}{t_1 + t_N(W)/N + Q(W)}. \quad (6.23)$$

Assume that t_N is proportional to W with coefficient c_1 and Q is proportional to W with coefficient c_3 , then after the problem size is changed from W to cW , we have

$$S_N(cW) = \frac{t_1 + ct_N(W)}{t_1 + ct_N(W)/N + cQ(W)}. \quad (6.24)$$

The variation of speedup is difficult to understand through Eq. (6.24). We need to use a new approach, the derivative of S_N , to study the variation. By Eq. (6.23), the derivative of S_N on W will be

$$S_N^d(W) = \frac{c_1(t_1 + Q + t_N/N) - (c_3 + c_1/N)(t_1 + t_N)}{(t_1 + t_N/N + Q)^2}. \quad (6.25)$$

The sign of S_N^d is determined by its numerator. Notice that under our assumption, we have $c_1 Q = c_3 t_N$. Thus the numerator

$$\begin{aligned} & c_1(t_1 + Q + t_N/N) - (c_3 + c_1/N)(t_1 + t_N) \\ &= c_1(t_1(1 - 1/N) + Q) - c_3(t_1 + t_N) \\ &= t_1[c_1(1 - 1/N) - c_3]. \end{aligned}$$

When $c_1 > \frac{N}{N-1}c_3$, $S_N^d > 0$. Therefore, S_N will increase with the problem size. If $c_1 < \frac{N}{N-1}c_3$, $S_N^d < 0$, speedup will decrease as the problem size increases. If $c_1 = \frac{N}{N-1}c_3$, $S_N^d = 0$, speedup is independent of the problem size. This gives the following results.

Theorem 8 *If the sequential execution time is independent of the problem size, and the parallel execution time and the communication overhead are proportional to the problem size with coefficient c_1, c_3 respectively, then, the speedup Eq. (6.23) will increase with problem size when $c_1 > \frac{N}{N-1}c_3$. It will decrease with an increase in problem size when $c_1 < \frac{N}{N-1}c_3$ and will be independent of problem size when $c_1 = \frac{N}{N-1}c_3$.*

The following theorem gives an upper bound of the influence of problem size.

Theorem 9 *Under the same assumptions of Theorem 8, the speedup is bounded by $S_N(W) + t_N/Q$*

Proof:

$$\begin{aligned} S_N(cW) &= \frac{t_1 + t_N(cW)}{t_1 + Q(cW) + t_N(cW)/N} = \frac{t_1 + ct_N}{t_1 + cQ + ct_N/N} \\ &= \frac{t_1}{t_1 + cQ + ct_N/N} + \frac{ct_N}{t_1 + cQ + ct_N/N} < \frac{t_1 + t_N}{t_1 + cQ + ct_N/N} + \frac{ct_N}{cQ} \\ &< S_N(W) + t_N/Q \quad \text{with the assumption } c \geq 1 \end{aligned} \quad (6.26)$$

It is easy to see that if the sequential execution time is also proportional to the problem size then the speedup (6.23) is independent of the problem size.

The performance model (6.23) fits the domain decomposition computation model (see Section 3.2) well. In the domain decomposition model, there is no uneven allocation degradation. The problem can be perfectly partitioned and the degree of parallelism will be N with N processors available. Each node has a fixed number of communication requirements. The communication latency is independent of system size and is a function of problem size. The sequential workload, W_1 , may or may not change with the problem size depending on the loading of the initial data. However, the communication overhead is proportional to the problem size. Commonly, it is a rational power function of the problem size. For instance,

using the finite difference method to solve Laplace's equation, Eq. (3.8), each node will solve a sublinear system $A_i x_i = b_i$ and send the result to its four neighbors. If A_i is an n by n matrix, then the length of the result is n , and, with the Gaussian elimination method, the solving process has a complexity $O(n^3)$. Theorem 10 gives a prediction of the speedup when communication overhead is a power function of problem size.

Theorem 10 *If the sequential execution time, t_1 , is independent of the problem size, the parallel operation is proportional with the problem size, $t_N = c_1 W$, and the communication overhead is a power function of the problem size, $Q = aW^{\alpha}$, then when $c_3 < \frac{[t_1(1-1/N)+Q]t_N}{(t_N+t_1)Q}$, the speedup (6.29) will increase with the problem size; when $c_3 > \frac{[t_1(1-1/N)+Q]t_N}{(t_N+t_1)Q}$, the speedup will decrease with an increase in problem size; and the speedup is independent of the problem size when $c_3 = \frac{[t_1(1-1/N)+Q]t_N}{(t_N+t_1)Q}$.*

Proof: We prove the theorem by observing the sign of the derivative of $S_N(W)$ on the variable W . The sign is determined by the numerator of $S_N^d(W)$, which is equal to

$$\begin{aligned} & c_1(t_1 + t_N/N + Q) - (c_1/N + c_3aW^{\alpha-1})(t_1 + t_N) \\ &= \frac{1}{W} (c_1W(t_1 + t_N/N + Q) - (c_1W/N + c_3aW^{\alpha-1}W)(t_1 + t_N)) \\ &= \frac{1}{W} (t_N(t_1 + t_N/N + Q) - (t_N/N + c_3Q)(t_1 + t_N)) \\ &= \frac{1}{W} (t_1t_N(1 - 1/N) + t_NQ - c_2Q(t_1 + t_N)). \end{aligned}$$

Therefore, $S_N^d(W)$ will be less than zero when $c_3 > \frac{[t_1(1-1/N)+Q]t_N}{(t_N+t_1)Q}$. $S_N^d(W)$ will be greater than zero when $c_3 < \frac{[t_1(1-1/N)+Q]t_N}{(t_N+t_1)Q}$ and $S_N^d(W)$ will be equal to zero if $c_3 = \frac{[t_1(1-1/N)+Q]t_N}{(t_N+t_1)Q}$. \square

Notice here that when the sequential workload is equal to zero, the sequential execution time t_1 is also equal to zero. In this case,

$$\frac{[t_1(1-1/N)+Q]t_N}{(t_N+t_1)Q} = 1,$$

and Theorem 10 becomes easier to understand. Theorem 11 gives the upper bound of the influence of problem size when the communication latency Q is a rational power function of the problem size.

Theorem 11 *Under the assumptions of Theorem 10, for any real number $c \geq 1$, we have $S_N(cW) \leq S_N(W) + c^{1-\alpha} \frac{t_N(W)}{Q(W)}$*

Proof:

$$\begin{aligned}
 S_N(cW) &= \frac{t_1 + t_N(cW)}{t_1 + Q(cW) + t_N(cW)/N} \\
 &= \frac{t_1}{t_1 + c^{c_3} Q(W) + c t_N(W)/N} + \frac{c t_N(W)}{t_1 + c^{c_3} Q(W) + c t_N(W)/N} \\
 &\leq S_N(W) + \frac{c t_N(W)}{c^{c_3} Q(W)} = S_N(W) + c^{1-c_3} \frac{t_N(W)}{Q(W)}.
 \end{aligned}$$

□

If we let the sequential execution time, t_1 , vary with the problem size, by similar arguments we have the following corresponding theorems.

Theorem 12 *Under the assumptions in Theorem 10, if we also let the sequential execution time be proportional to the problem size, with coefficient c_2 , then when $c_3 < 1$, the speedup (6.23) will increase with the problem size; when $c_3 > 1$, the speedup will decrease with an increase in problem size; and when $c_3 = 1$ the speedup is independent of the problem size.*

Theorem 13 *Under the assumptions of Theorem 12, for any real number $c \geq 1$, when $c_3 \geq 1$, $S_N(cW) \leq S_N(W)$; and when $c_3 < 1$, $S_N(cW) \leq c^{1-c_3} S_N(W)$.*

Proof: By Theorem 12, we only need to prove the $c_3 < 1$ case. When $c_3 < 1$,

$$\begin{aligned}
 S_N(cW) &= \frac{c t_1 + c t_N}{c t_1 + c^{c_3} Q + c t_N/N} \\
 &= \frac{t_1 + t_N}{c^{c_3-1} (c^{1-c_3} t_1 + Q + c^{1-c_3} t_N/N)} \\
 &\leq c^{1-c_3} S_N(W).
 \end{aligned}$$

□

All of the above six theorems, Theorem 8 to Theorem 13, are based on speedup (6.23). With the consideration of the sequential execution and communication latency degradations, they show how the speedup is influenced by workload and what is the limitation of this influence. They are useful prediction formulations for domain decomposition computations and for other scientific and engineering applications. In the remaining portion of this subsection, we study the influence of problem size with degree of parallelism considered. Recall that t_i is the execution time for workload W_i when one processor is available. Assuming that each t_i , $i = 1, \dots, m$ is proportional to the problem size, then the speedup will be independent of the problem size from the degree of parallelism point of view. This is confirmed by the following equation,

$$S_N(cW) = \frac{\sum_{i=1}^m t_i(cW)}{\sum_{i=1}^m t_i(cW)/i} = \frac{c \sum_{i=1}^m t_i(W)}{c \sum_{i=1}^m t_i(W)/i} = S_N(W). \quad (6.27)$$

Most of the frequently used applications do not distribute load evenly. For achieving high performance, we would like workload with a high degree of parallelism to increase with the problem size, and workload with lower degrees of parallelism to be fixed, independent of problem size. In general, we can divide the integer set $1, 2, \dots, m$ into two disjoint subsets $i_1, i_2, \dots, i_{m_1}, j_1, j_2, \dots, j_{m_2}$ such that $m_1 + m_2 = m$ and that workload in degree of parallelism $i_k, k = 1, \dots, m_1$ is independent of the problem size, and the workload in degree of parallelism $j_k, k = 1, \dots, m_2$ increases with the problem size. By this partition, we have

$$S_N(W) = \frac{\sum_{i=1}^m t_i(W)}{\sum_{i=1}^m t_i(W)/i} = \frac{\sum_{k=1}^{m_1} t_{i_k} + \sum_{k=1}^{m_2} t_{j_k}(W)}{\sum_{k=1}^{m_1} t_{i_k}/i_k + \sum_{k=1}^{m_2} t_{j_k}(W)/j_k}.$$

Extending the average parallelism concept introduced in Section 2.1, we have

$$S_N(W) = \frac{t_1^* + t_N^*(W)}{t_1^* + t_N^*(W)/A^*} \leq A^*, \quad (6.28)$$

where $t_1^* = \sum_{k=1}^{m_1} t_{i_k}/i_k$, $t_N^*(W) = \sum_{k=1}^{m_2} t_{j_k}(W)$, and $A^* = \sum_{k=1}^{m_2} t_{j_k}/[\sum_{k=1}^{m_2} t_{j_k}/j_k]$ is the extended average parallelism. Thus, when the number of processors is fixed, we return to Eq. (6.21) and the results for Eq. (6.21) can be directly used for Eq. (6.28). Both the super sequential execution time t_1^* and the extended average parallelism A^* of Eq. (6.28) are independent of problem size. However, both of them are functions of system size.

The prediction formulation Eq. (6.28) is useful for divide-and-conquer computations. For certain divide-and-conquer computations the divide-and-conquer parts have a fixed number of operations which are not influenced by the problem size. Only the computation part scales up with the problem size. Since the computation part has N as the degree of parallelism, by Eq. (6.28), near perfect speedup can be achieved when the problem size is large enough.

Chapter 7

CONCLUSION AND FUTURE RESEARCH

The design, representation and performance considerations of parallel algorithms on multicomputers have been presented in this thesis. In this chapter, the work reported in this thesis is summarized and the major contributions of this dissertation are highlighted. Next, improvements to the current work are suggested, and future plans and research directions will also be discussed.

7.1 Summary and Major Contributions

The central issues in the development of efficient parallel algorithms involve exploring the inherent parallelism in applications and the matching of applications to architectures. The present study is an attempt to better understand of these issues. The contributions of this study include developing a notation for describing the structures of the most frequently used scientific and engineering applications, identifying the basic building blocks of these structures, providing a guideline for the design of efficient parallel algorithms, developing and implementing efficient parallel algorithms for several applications, proposing a new approach for predicting the performance of parallel algorithms, and presenting a more general speedup model, namely, memory-bounded speedup.

Traditionally, parallel algorithms have been designed by brute force methods and fine tuned on each architecture to achieve high performance. Rather than studying the development and matching case by case, a systematic approach was proposed in my research by studying the basic building blocks, called *computation models*, of frequently used applications. A notation was first developed in Chapter 3. Using this notation, most of the frequently used scientific and engineering applications can be represented by simple formulas. These formulas constitute the *structured representation* of the corresponding applications. The structured representations are simple, adequate and easy to understand. They also contain sufficient information about uneven allocation and communication latency degradations. With the structured representations, applications can be compared, classified and

partitioned. There are innumerable applications. However, most of the applications are combinations of some computation models. Structured representations relate general applications to computation models. Studying computation models leads to a guideline for efficient parallel algorithm design for general applications.

Seven models have been identified and presented in Chapter 3. They are the *Local Computation Model*, the *Global-Exchange Computation Model*, the *Compute-Aggregate-Broadcast Computation Model*, the *Divide-and-Conquer Computation Model*, the *Domain Decomposition Computation Model*, the *Pipelined Computation model* and the *Recursive Doubling Computation Model*. The structured representation of each of the computation models was presented, and the computation and communication patterns of the computation models were studied. With the computation model concept, the matching of application to architectures becomes transparent. The performance information of computation models suggests which structure is best suited for a specific multicomputer architecture, and which part of an algorithm is the performance bottleneck. Therefore, structured design and rethinking become possible.

Structured representation and computation models have been used on the development of parallel algorithms for several scientific and engineering applications. These applications include solving *electrical power flow problem* [52], solving *tridiagonal linear systems* [60], solving *dense linear systems* [49] and solving *tridiagonal symmetric algebraic eigenvalue problems*. (The work on the last two applications, which are published in [49] and [40], are not included in this dissertation). We used different computation models for different applications. The algorithm for solving dense linear systems is based on the pipelined computation model. The tridiagonal symmetric algebraic eigenvalue problems were solved with local computation and divide-and-conquer computation approaches. The newly proposed divide-and-conquer algorithm for solving eigenvalue problems is very efficient. It competes well with any existing parallel algorithm for solving eigenvalue problems. Solving tridiagonal linear systems is a fundamental problem of scientific computing, and solving power flow problems is, by far, one of the most important problems facing researchers in the electrical power field. Structured representation and computation model concepts help us in changing from one design to another. Four new algorithms were developed for solving tridiagonal linear systems. Three of the four algorithms are presented in this thesis. They are the partition LU, the partition global-exchange and the partition hybrid algorithms. The partition LU algorithm is based on compute-aggregate-broadcast computation. The partition global-exchange algorithm uses global-exchange computation. The partition hybrid algorithm is a combination

of compute-aggregate-broadcast computation and recursive doubling computation. All of the developed parallel algorithms have been implemented on an NCUBE/7 multicomputer. Our algorithms yield a higher speedup and efficiency compared to existing algorithms. Using structured representation, We modified our design for solving the power flow problem from the local computation model to the global-exchange model, and then to the compute-aggregate-broadcast computation model. The last one gave the best performance. Based on the compute-aggregate-broadcast algorithm, we have developed a package, called *PowerCube* [45], which adapts the most rigorous mathematical techniques and is able to obtain all the steady state solutions of power systems. The package currently runs on NCUBE and BBN GP-1000 multicomputers. The related details of solving electrical power flow problems and tridiagonal linear systems can be found in Chapter 4 and Chapter 5 respectively.

In association with the study of efficient algorithm design, the performance issues of parallel algorithms were also studied [58]. Three models of speedup were discussed and analyzed in Chapter 2. They are *fixed-size speedup*, *fixed-time speedup* and *memory-bounded speedup*. Two sets of speedup formulations were derived for these three models. One set requires more information and gives more accurate estimates. Another set considers a simplified case and provides a clearer picture of the possible performance gain with parallel processing. The simplified fixed-size speedup constitutes *Amdahl's law*, whereas the simplified fixed-time speedup is *Gustafson's scaled speedup*. The simplified memory-bounded speedup contains both Amdahl's law and Gustafson's scaled speedup as its special cases. This study proposes a new metric for performance evaluation and leads to a better understanding of parallel processing.

Performance prediction techniques have been developed based on structured representation and based on some precollected performance data. The performance predictions consider both communication overhead and uneven allocation as degradations of parallelism, and give a good bound on performance. Performance prediction is very important in real time systems where we want to get the most accurate solution within a time limitation. The predicted performances are also very helpful for efficient algorithm design. Using structured representation, performance can be predicted for different levels. The lower level accumulates the predicted performance of each compute and data-exchange phase. The higher level uses computation models as the basic modules and provides a more accurate and automatic approach. The performance formulations of two of the computation models were derived in Chapter 6. The performance of a large class of applications could be predicted by combining these formulations and by using the higher level approach. The analysis process also showed

how we could derive the performance formulations of a given application through the lower level approach. An example was given to show the performance prediction techniques. Results obtained from our implementation of this example matched the predicted performance well.

7.2 Future Research Directions

In this research, the structured representation method was developed to describe the most frequently used scientific and engineering applications. The basic data structures of these applications, called computation models, were identified and studied and a new metric for the performance of parallel algorithms was proposed. Applying the structured representation, the identified computation models and the new metric of performance, the issues in efficient algorithm design and in performance prediction were studied. All these topics, namely, the representation, the basic data structure and the metric of performance, are fundamental problems in computer science. Because of their fundamental properties, the results presented in this dissertation can be applied to other areas of computer science. Further, these problems are difficult to solve perfectly due to their fundamental nature. Future research areas can be identified along three directions – improving the current results of these fundamental problems, continuing the research of applying these fundamental results to efficient algorithm design and performance prediction, extending the application of these fundamental results to other areas of computer science.

Several questions remain open along the first direction of research. For structured representation, we have only developed notation for two classes of data-exchange, i.e., regular data-exchange and conjunctive regular data-exchange. Communication phases do exist that do not belong to either of these two kinds of data-exchanges. Can we extend our notation to cover more complicated communications? Structured representation provides the information about how many processors participate in each data-exchange phase. If we assume that a processor computes if and only if it participates in data-exchange, then we know how many processors do useful work at each compute phase and, therefore, have some knowledge of uneven allocation degradation. However, the workload of each processor is unknown. Processors may compute at the same compute phase but with different workload. Can we extend or modify our current notation to contain the workload information? Furthermore, we assume that the communication is achieved in a synchronous fashion, can we extend our current notation to contain asynchronous communication? Could we use graph

theoretical results and terminologies to make structured representation more general and more fruitful? For instance, the results of bipartite graphs could be used to further divide a communication pattern into smaller pieces. Also, further study on computation models is needed. This includes studying more applications and identifying more models, studying each model on different architectures, and deciding how to identify a computation model. With structured representation, a computation model can be identified at different levels – finer or coarser. Which level is better is uncertain at this time. For example, divide-and-conquer is a commonly used technique, which we identified in the conventional way (see Section 3.2). However, it can be identified as two tree structures, which may be easier to measure and understand. Three models of speedup, namely, *fixed-size speedup*, *fixed-time speedup* and *memory-bounded speedup*, are studied for the performance of different classes of applications. Applications exist which do not fit any of these models of speedup, but satisfy some combination of the models.

The study of efficient algorithm design and performance prediction is still in a preliminary stage. This is especially true for the research in performance prediction. Only four applications have been studied on a single architecture – the hypercube architecture. We would like to study more applications on different architectures and explore the dynamic load balancing issues. We need more experimental data to support the theoretical results. Our representation is based on advanced technology, so it does not consider the distance between two processors as an important factor. The algorithms which have been implemented on the first generation NCUBE multicomputer should be tested further on second generation multicomputers. More algorithms should be implemented on second generation multicomputers to determine the accuracy of our predictions and to improve our prediction schemes. Structured representation contains adequate information about an application, but finding the structured representation for an application is difficult. It requires a deep understanding of the application itself. How to detect the structured representation of a given application automatically and systematically is an interesting research issue. Some tools exist for detecting the degree of parallelism of a given application [36]. These tools provide starting points for developing tools that can detect the structures and computation models contained within applications automatically. The proposed performance prediction method is based on structured representation and computation models. Performance tools can be developed for performance estimation and prediction by applying the method. Further study of the performance prediction methodology is needed before a tool can be developed, since there are numerous implementation issues involved. This is another interesting research issue which

can be studied based on the results of this dissertation.

During my dissertation, I have conducted research on several fundamental problems and applied the research results to different areas of computer science. Further research can be continued along these directions. This research can also be extended to new areas to attack untouched problems. For instance, structured representation could be modified for use with shared-memory multiprocessors; the computation model concept may be extended for non-numerical computations; kernel programs for machine benchmarks can be developed based on the identified computation models. I plan to continue my current research and am always looking forward to meeting new challenges.

Bibliography

Bibliography

- [1] G. Almasi and A. Gottlieb. *Highly parallel computing*. The Benjamin/Cummings Publishing Company, Inc., 1989.
- [2] G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Conf.*, pages 483–485, 1967.
- [3] W. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *Computer*, pages 9–25, Aug. 1988.
- [4] M. Barton and G. Withers. Computing performance as a function of the speed, quantity, and cost of the processors. In *Proc. Supercomputing'89*, pages 759–764, 1989.
- [5] D. Bertsekas and J. Tsitsiklis. *Parallel And Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., 1989.
- [6] S. Borkar and et al. Iwarp: An integrated solution to high-speed parallel computing. In *Proc. of Supercomputing'88*, pages 330–339, 1988.
- [7] R. Brassard and Bratley. *Algorithmics Theory and Practice*. Prentice-Hall, Inc., 1987.
- [8] S. Chang, X. Sun, and L. Ni. A projection tool for visualizing multidimensional curves. Michigan State University, 1989. Technical Report, MSU-CPS-ACS-16.
- [9] S. Chow, J. Mallet-Paret, and J. Yorke. Finding zeroes of maps: homotopy methods that are constructive with probability one. *Math. Comp.*, 32:887–899, 1978.
- [10] S. Chow, L. Ni, and Y. Shen. A parallel homotopy method for solving a system of polynomial equations. In *Proc. of the 3rd SIAM Int'l Conf. on Parallel Processing for Scientific Computing*, pages 121–125, Chicago, 1987.
- [11] E. Coffman. Introduction to deterministic scheduling theory. In J. W. E.G. Coffman and Sons, editors, *Computer and Job/Shop Scheduling Theory*, pages 1–50. 1976.

- [12] F. Darema. Parallel applications performance methodology. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 49–58. The ACM press, 1989.
- [13] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *Lapack Users' Guide*. SIAM, Philadelphia, 1979.
- [14] I. Duff, A. Erisman, and J. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.
- [15] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus efficiency in parallel system. *IEEE Transactions on Computers*, pages 403–423, March 1989.
- [16] O. Egecioglu, D. Koc, and A. Laub. A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors. Technical Report No. TRC88-1, 1988. Dept. of Computer Science, Univ. of California, Santa Barbara.
- [17] S. Even. *Graph Algorithms*. Computer Science Press, 1979.
- [18] R. Finkel. Large-grain parallelism - three case studies. In L. Jamieson, D. Gannon, and R. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 21–63. The MIT press, 1987.
- [19] W. Gropp and d.E. Keyes. Complexity of parallel implementation of domain decomposition techniques for elliptic partial differential equations. *SIAM J. on SSTC*, 9(2), March 1988.
- [20] J. Gustafson. Reevaluating amdahl's law. *CACM*, 31:532–533, May 1988.
- [21] J. Gustafson, S. Hawkinson, and K. Scott. The architecture of a homogeneous vector supercomputer. In *Proc. of 1986 Int'l Conf. on Parallel Processing*, pages 649–652, Chicago, 1986.
- [22] J. Gustafson, G. Montry, and R. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. on SSTC*, 9(4), July 1988.
- [23] J. Hayes, T. Mudge, Q. Stout, S. Collet, and J. Palmer. Architecture of a hypercube supercomputer. In *Proc. of 1986 Int'l Conf. on Parallel Processing*, pages 653–660, 1986.

- [24] D. Heller. A survey of parallel algorithms in numerical algebra. *SIAM Review*, 20:740–777, Oct. 1978.
- [25] W. Hill. *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.
- [26] W. Hills and G. Steele. Data parallel algorithms. *Communications of the ACM*, 29, Dec. 1986.
- [27] R. Hockney. A fast direct solution of poisson's equation using fourier analysis. *J. ACM*, 12:95–113, 1965.
- [28] K. Hwang. Advanced parallel processing with supercomputer architectures. *Proc. of the IEEE*, pages 33–47, Oct. 1987.
- [29] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processin*. McGraw-Hill Book Co., 1984.
- [30] L. Jamieson. Characterizing parallel algorithms. In L. Jamieson, D. Gannon, and R. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 65–100. The MIT press, 1987.
- [31] V. N. J.H. Saltz and D. Nicol. Reduction of the effects of the communication delays in scientific algorithms on message passing multiprocessor architectures. *SIAM J. SCI. STAT. COMPUT.*, 8(1), Jan. 1987.
- [32] S. Johnsson. Communication efficient basic linear computations on hypercube multiprocessors. *J. of Parallel and Distributed Computing*, (4):133–172, 1987.
- [33] S. Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. on SSTC*, 8(3):354–392, May 1987.
- [34] C. King, W. Chow, and L. Ni. Pipelined data parallel algorithm – concept and modeling. In *ACM Int'l. Conf. on Supercomputing*, 1988.
- [35] P. Kogge and H. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE-TC*, c-22:786–793, 1973.
- [36] M. Kumar. Measuring parallelism in computation intensive scientific/engineering applications. *IEEE-TC*, 37(9):1088–1098, Sep. 1988.

- [37] T. Li, T. Sauer, and J. Yorke. Numerical solution of a class of deficient polynomial systems. *SIAM J. Numer. Anal.*, 24(2), Apr. 1987.
- [38] T. Li, T. Sauer, and J. Yorke. The random product homotopy and deficient polynomial system. *Numer. Math.*, 51(481), 1987.
- [39] T. Li and X. Wang. A more efficient homotopy for solving deficient polynomial systems. Submitted for publication.
- [40] T. Li, H. Zhang, and X. Sun. Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problem. accepted to appear in *SIAM J. of Scientific and Statistical Computing*.
- [41] M. A. Marsan, G. Bailbo, and G. Conte. A class of generalized stochastic petri nets for the performance analysis of multiprocessor systems. *ACM TOCS*, pages 93–122, May 1984.
- [42] P. Michielse and H. Vorst. Data transport in wang's partition method. *Parallel Computing*, 7:87–95, 1988.
- [43] A. Morgan and A. Sommese. A homotopy for solving general polynomial systems that respects m-homogeneous structures. *App. Mat. and Comp.*, 24:101–113, 1987.
- [44] P. Nelson and L. Snyder. Programming paradigms for nonshared memory parallel computers. In L. Jamieson, D. Gannon, and R. Douglass, editors, *The Characteristics of Parallel Algorithms*. The MIT press, 1987.
- [45] L. Ni, F. Salam, T. Tzen, X. Sun, and S. Guo. Powercube: A software package for solving load flow problems. In *Proc. of the 32nd Midwest Symposium on Circuits and System*, Illinois, Aug. 1989.
- [46] S. Nugent. The ipsc-2 direct-connect communication technology. In *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, Jan. 1988.
- [47] J. Ortega and R. Voigt. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, June 1985.
- [48] M. Quinn. *Designing Efficient Algorithms For Parallel Computers*. McGraw-Hill, 1987.

- [49] D. Robinson, X. Sun, and R. Enbody. A pipelined parallel approach to solving dense linear systems. In *Proc. of the Fourth Conf. on Hypercube concurrent Computers and Applications*, March 1989.
- [50] Y. Saad and M. Schultz. Data communication in hypercube. Research Report, YALEU/DCS/RR-42, Computer Science Department, Yale University, Oct. 1985.
- [51] F. Salam, L. Ni, S. Guo, and X. Sun. Parallel processing for the load flow of large-scale power system: The approach and application. In *Proc. of the 28th IEEE Conf. on Decision and Control*, Tampa, Florida, Dec 1989.
- [52] F. Salam, L. Ni, X. Sun, and S. Guo. Parallel processing for the steady state solutions of large-scale nonlinear models of power systems. In *Proc. of the 1989 IEEE Int'l Symposium on Circuits and Systems (ISCAS)*, Portland, Oregon, May 1989.
- [53] K. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proc. of ACM SIGMETRICS and Performance'89*, May 1989.
- [54] J. Sherman and W. Morrison. Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix. *Ann. Math. Stat.*, 20(621), 1949.
- [55] Y. Shih and J. Fier. Hypercube systems and key application. In K. Hwang and D. DeGroot, editors, *Parallel Processing for Supercomputing and Artificial Intelligence*. McGraw-Hill, 1988.
- [56] H. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. of ACM*, 20(1):27-38, Jan. 1973.
- [57] H. Stone. Parallel computers. In *Introduction to computer architecture*. SRA Inc., Chicago, III, 1980 (2nd ed.).
- [58] X. Sun and L. Ni. Another view on parallel speedup. In *Proc. of Supercomputing'90*, NY, NY, Nov. 1990.
- [59] X. Sun, L. Ni, F. Salam, and S. Guo. Compute-exchange computation for solving power flow problems: The model and application. In *Proc. of the Fourth SIAM Conf. on Parallel Processing for Scientific Computing*, Dec. 1989.

- [60] X. Sun, H. Zhang, and L. Ni. Parallel algorithms for solution of tridiagonal systems on multicomputers. In *Proc. of the 1989 ACM Int'l Conf. on Supercomputing*, Crete, Greece, June 1989.
- [61] A. Veen. Dataflow machine architectures. *ACM Computing Surveys*, 18(4):365–396, Dec. 1986.
- [62] H. Wang. A parallel method for tridiagonal equations. *ACM Trans. Math. Software*, 7:170–183, June 1981.
- [63] L. Wason. Numerical linear algebra aspects of globally convergent homotopy methods. *SIAM Review*, pages 529–545, Dec. 1986.
- [64] F. Wu. Stability, and security, and reliability of interconnected power systems. *Large Scale Systems: Theory and Applications*, 7:99–113, 1984.

MICHIGAN STATE UNIV. LIBRARIES



31293007914785