Communication-Efficient Bulk Synchronous Parallel Algorithms

by

Chun-Hsi Huang

July 26, 2001

A dissertation submitted to the
Faculty of the Graduate School of
State University of New York at Buffalo
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Acknowledgments

I am deeply grateful to my thesis advisor, Dr. Xin (Roger) He, for the guidance throughout my research. My initial glimpse of the theoretical Computer Science came with Dr. He's seminars and independent studies in 97 and 98. I also appreciate his being always available for discussions and extremely patient giving advice to a novice. These insightful advice and the continued motivation and encouragement have made this dissertation possible.

In addition to the appreciation for serving as my dissertation committee member, I am indebted to Dr. Russ Miller's inspiring CS 531 lectures in Spring 97, which built up my background knowledge about parallel algorithms. Without the inspiration, my current research wouldn't have started. I would also like to thank Dr. Miller's generously providing the access to the SGI Origin 2000[1], enabling the experimental results in this dissertation. These supercomputing experiences turn out to be very helpful in my search for academic jobs.

The arrival of Dr. Ashim Garg in UB further enhances the research program in algorithms of CSE department. I appreciate having Dr. Garg as one of my dissertation

[1] The computational resources and the related technical support used in this dissertation were provided by Center for Computational Research at SUNY Buffalo (UB CCR).

committee members. The success of Dr. Garg's research has motivated my enthusiasm for academic work.

I would also like to thank Dr. Ming-Yang Kao at Yale University for serving as the outside reader of this dissertation.

I thank the Department of Computer Science and Engineering for the teaching assistantship during my study in Buffalo. In particular, I truly appreciate Ms. Helene Kershner and Dr. Raj Acharya for granting me several semesters instructing undergraduate courses. Without such additional financial assistance, I wouldn't have been able to support my family during 98 and 99, let alone the research. Dr. He's support[2] as a research assistant for the past year has made possible my flexible schedule for research and numerous job interviews during the past three months.

My parents, Yih-Hsiung Huang and Ching Cheng, deserve my special gratitude. Without their love and encouragement, my education wouldn't have been possible. I am also grateful to my elder brother Chun-Hsun and little sister Pei-Ching for all the wonderful times I have had when I was home in Taiwan.

Finally, my love and affection go to my wife, Hsiu-Cheng Kao, for her support in 96 when I decided to go overseas for doctoral study and her taking care of our son Jonathan almost alone since his birth in 98.

[2] The research in this dissertation was supported in part by NSF Grant CCR-9912418.

Abstract

Communication has been pointed out to be the major bottleneck for the performance of parallel algorithms. Theoretical parallel models such as PRAM have long been questioned due to the fact that the theoretical algorithmic efficiency does not provide a satisfactory performance prediction when algorithms are implemented on commercially available parallel machines. This is mainly because these models do not provide a reasonable scheme for measuring the communication overhead. Recently several practical parallel models aiming at achieving portability and scalability of parallel algorithms have been widely discussed. Among them, the *Bulk Synchronous Parallel* (BSP) model has received much attention as a bridging model for parallel computation, as it generally better addresses practical concerns like communication and synchronization.

The BSP model has been used in a number of application areas, primarily in scientific computing. Yet, very little work has been done on problems generally considered to be irregularly structured, which usually result in highly data-dependent communication patterns and make it difficult to achieve communication efficiency. Typical examples are fundamental problems in graph theory and computational geometry, which are important as a vast number of interesting problems in many fields are defined in terms of

them. Thus practical and communication-efficient parallel algorithms for solving these problems are important.

In this dissertation, we present scalable parallel algorithms for some fundamental problems in graph theory and computational geometry. In addition to the time complexity analysis, we also present some techniques for worst-case and average-case communication complexity analyses. Experimental studies have been performed on two different architectures in order to demonstrate the practical relevance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A parallel computation model has to achieve several goals to become useful. On one hand, it has to be detailed enough to allow a fairly accurate prediction of the running times of algorithms. On the other hand, it has to be simple enough to make analysis of algorithms possible. A parallel computation model should also reflect the constraints of parallel machines (existing and future ones) but without loss of portability of the algorithms designed for the model. Another goal that should be fulfilled is scalability, *i.e.* the algorithms for the model should work well for a wide range of the ratio $\frac{n}{p}$, where n refers to the problem size and p refers to the number of processors.

Many models for parallel computation have been proposed. The PRAM model gives good guidelines for classification of problems in P in terms of their parallel time complexities on a PRAM. A large number of problems have been shown to lie in NC, *i.e.* they are solvable on a PRAM in poly-logarithmic time using a polynomial number of processors. Some other problems have been shown to be P-complete. This implies that they have no NC algorithms unless $P = NC$, an event considered highly unlikely. The notions of *NC*-class and the *P*-completeness have allowed major advances in our theoretical understanding of shared memory parallel algorithms and their complexities. The PRAM model is also useful for comparing relative performance of two algorithms. If one PRAM algorithm outperforms another PRAM algorithm, the relative performance is not likely to change substantially when both algorithms are adapted to run on a real parallel computer [27].

One fundamental assumption for the PRAM model is that the communication of one message unit between the processors can be done in one time unit, the same as the time needed by a basic computation instruction. For practical models, this assumption does not hold, since the time required to either perform a memory access or routing a message is typically several orders of magnitude larger than the time for an internal CPU operation. To alleviate this problem, some variations of the PRAM model have been proposed, but they still can not fully reflect the communication constraints imposed on practical parallel computers. Recently several practical general purpose parallel computation models have been widely discussed. Among them, the BSP model and its variation, the CGM model, have received much attention as bridging models for parallel computation, as they generally better address practical concerns like communication and synchronization.

Parallel computation has been widely applied to solve highly computation-intensive problems. Typical examples are those *Grand Challenge Applications* posted by U.S. High-Performance Computing and Communication (HPCC) Program, such as magnetic recording technology, rational drug design, high-speed civil transport, catalysis, fuel combustion, ocean modeling, ozone depletion, digital anatomy, air pollution, protein structure design, image understanding and technology linking research to education, etc [56]. A representative example is the *Shake-and-Bake* research project for determining crystal structures [76].

On the other hand, for communication-intensive applications like those problems that are irregularly structured, the communication efficiency, which is vital to getting satisfactory parallel performance, becomes much harder to achieve. In particular, it is generally acknowledged that graph problems have considerably less regular structures than many other problems studied [23, 24]. An irregular structure results in highly data-dependent communication patterns and makes it difficult to achieve communication efficiency. However, a vast number of interesting problems in many fields are defined in terms of graphs. Thus practical and efficient parallel algorithms for communication-intensive problems, like those in graph theory, are quite important.

1.1 The Focus of This Dissertation

In this dissertation, we attempt to design coarse-grained parallel algorithms for problems that are generally considered to be irregularly structured (or communication-intensive), with focus on reducing communication overhead and increasing scalability.

The following sections of this chapter describe in more details the representative programming models for different parallel computing tracks. We also briefly mention the communication libraries (*Message-Passing Interface*, MPI) and hardware platforms we use in the experiments.

Chapter 2 presents an EREW BSP algorithm for solving the *All Nearest Smaller Val-*

ues Problem (ANSVP), a fundamental problem in both graph theory and computational geometry. Our algorithm achieves optimal sequential computation time and uses only three communication supersteps. In the worst case, each communication phase takes no more than an $(\frac{n}{p}+p)$-relation, where n is the problem size and p is the number of the processors. In addition, the average-case analysis shows that, on random inputs, the expected communication requirements for all three steps are bounded above by a p-relation, which is independent of the problem size n. Experiments have been carried out on an SGI Origin 2000 with 32 R10000 processors and a SUN Enterprise 4000 server supporting 8 UltraSPARC processors, using the MPI libraries. We also show that our ANSVP BSP algorithm results in a communication-optimal parallel algorithm for the *parentheses matching* problem. Part of these results were reported in [50, 53].

Chapter 3 presents some applications of the ANSVP, including problems in graph theory and computational geometry, like *monotone polygon triangulation*, *binary tree reconstruction*, *range minimum search* and *finding all nearest neighbors in convex polygons*. In addition, the *string matching* problem, which can also be solved using ANSV algorithm as a subroutine, is described.

Chapter 4 presents scalable coarse-grained parallel algorithms for solving interval graph problems on a variation of EREW BSP model–*Coarse Grained Multicomputers* (CGM), also referred to as the weak-CREW BSP. The problems we consider include: finding *maximum independent set*, *maximum weighted clique*, *minimum coloring* and *cut vertices* and *bridges*. With scalability at $\frac{n}{p} \geq p^{\varepsilon}$, $\forall \varepsilon > 0$ (here n denotes the total input size and p the number of processors), our algorithms for maximum independent set and minimum coloring use optimal computation time and $O(\log p)$ communication rounds, which is independent of the input size and grows slowly only with the number of processors. Equally scalable are our algorithms for finding maximum weighted clique, cut vertices and bridges, which use $O(1)$ communication rounds and optimal local computation time, achieving both communication and computation optimality. These results were reported in [49].

Chapter 5 presents a general methodology for the communication-efficient parallelization of graph algorithms using the divide-and-conquer approach. Specifically, the first practical parallel algorithm, based on the EREW BSP model, for *finding Hamiltonian paths in tournaments* is presented. The algorithm uses only $(3\log p + 1)$ communication supersteps, which is independent of the tournament size, and can reuse the existing linear-time algorithm in the sequential setting. When (1) the ratio of computation and communication throughputs (parameter g) is low, or (2) the local memory size, $O(\frac{n}{p})$, of each individual processor is extremely limited ($\frac{n}{p} \geq p^{\varepsilon}$, $\forall\, \varepsilon > 0$), the revised algorithm requires $O(\log p)$ communication supersteps, while the hidden constant grows with the scalability factor $1/\varepsilon$. These results were reported in [54].

Chapter 6 presents some future work that is related to the current results. The first problem we addressed is the *general parallel prefix computation* (GPC), which has been shown to be the kernel routine of several parallel algorithms in fields such as computer graphics, medical imaging, databases, and computational geometry. Hence, providing a portable parallel algorithm with predictable communication efficiency for the general prefix computation problem becomes an essential step for solving these applications on practical parallel computing platforms. Preliminary survey of this problem was reported in [55]. Also addressed are the fault tolerant issues on BSP and the I/O-efficient algorithms for parallel machines with external memories as major storage and for problems whose input sizes are significantly larger than the main memory size.

1.2 Preliminaries

1.2.1 Idealized Parallel Computing

Various idealized shared-memory models of parallel computation have been used in the study of parallel algorithms and their complexity. Two such models are the PRAM and the circuit. We will describe these models in this section. The circuits described can be translated into PRAM algorithms in a straightforward manner. The class *NC* has provided a simple and robust framework for the classification of problems in *P*. A large number of important problems have been shown to lie in *NC*, *i.e.* to be solvable on a PRAM in poly-logarithmic time using a polynomial number of processors. Other problems have been shown to be *P*-complete, *i.e.* to have no *NC* algorithm unless *P* = *NC*. The class *NC* and the notion of *P*-completeness have allowed major advances to be made in our theoretical understanding of shared memory parallel algorithms and their complexity.

PRAM

A *parallel random access machine* (PRAM) [34, 97, 57] consists of a collection of processors that compute synchronously in parallel and that communicate with a common global random access memory. In one time step, each processor can do (any subset of) the following: read the operand(s) from the common memory, perform a simple arithmetic or logic operation and write a value back to the common memory. There is no explicit communication between processors. Processors can only communicate by writing to, and reading from, the common memory. The processors have no local memory other than a small fixed number of registers that they use to temporarily store the argument and result values. In a *Concurrent Read Concurrent Write* (CRCW) PRAM, any number of processors can read from, or write to, a given memory cell in a single time unit. In a *Concurrent Read Exclusive Write* (CREW) PRAM, at most one processor can write to a given memory cell at any time. In the most restricted model, the *Exclusive Read Exclusive Write* (EREW) PRAM, no concurrency is permitted either in reading or in writing. The CRCW PRAM model has a large number of variants which differ in the convention they adopt for the effect of concurrent writing. Three simple examples of such conventions are: two or more processors can write so long as they write the same value (Common CRCW); one of the processors attempting to write succeeds but the choice of which one succeeds is made nondeterministically (Arbitrary CRCW); and the lowest numbered processor succeeds (assuming some appropriate numbering) (Priority CRCW). In other CRCW models one might have the possibility of concurrent writing in which the value stored is some specified combination of the values written (Combining CRCW).

The complexity of a PRAM algorithm is given in terms of the number of time steps and the maximum number of processors required in any one of those time steps. An important characteristic of the PRAM model is that it is a one-level memory (or shared memory) model, *i.e.* all of the memory locations are uniformly distant from all of the processors, the processors have no local memory and there is no memory hierarchy based on the ideas of network locality. These simplifying properties of the PRAM model have made it extremely attractive as a robust model for theoretical design and analysis of algorithms.

PRAM algorithms must be highly synchronized to work correctly. Usually we assume the PRAM processors are inherently tightly synchronized via a common clock. All processors execute the same statements at the same time step. That is, we do not allow processors to race ahead while others are further back in the code. Hence, there is no need to describe synchronization overhead explicitly.

Circuits

A *circuit* [33] is a directed acyclic graph with n input nodes (in-degree 0) corresponding to the n inputs to the problem, and a number of *gates* (in-degree 2) corresponding to two-argument functions. In a Boolean circuit, the gates are labeled with one of the binary Boolean functions *NAND*, *NOR*, $\wedge$, $\vee$,$\rightarrow$, and $\oplus$, etc. In a typical arithmetic circuit, the input nodes are labeled with some value from Q, the set of rational numbers, and the gates are labeled with some operation from the set $\{+,-,*,/\}$. The size of the circuit is the number of gates. The parallel complexity of a circuit is the depth of the circuit, *i.e.* the maximum number of gates on any directed path.

The algorithms described in circuits can be translated into PRAM algorithms in a straightforward manner.

1.2.2 Special Purpose Parallel Computing

No single model of parallel computation had yet come to dominate developments in parallel computing in the way the von Neumann model has dominated sequential computing [95]. Instead we have a variety of models such as VLSI systems, systolic arrays and distributed memory multicomputers, in which the careful exploitation of network locality is crucial for algorithmic efficiency. For example, in a VLSI system, a design with good network locality will have short wires, and hence will require less area. An efficient systolic algorithm will have a simple, regular structure and use only nearest neighbor communication. An efficient multicomputer algorithm will be one that minimizes the distance that messages have to travel in the network by careful mapping of the virtual process structure onto the physical processor architecture. Of course, an efficient algorithm for, say, a hypercube multicomputer will not necessarily perform well when run on, for example, a 2D mesh multicomputer with the same number of processors. This type of parallel computing is generally referred to as "special purpose" parallel computing [73].

Special purpose parallel systems are particularly appropriate in application areas where the goal of achieving the best possible performance is much more appropriate than that of achieving an architecture-independent design. Examples of such areas include digital signal processing, image processing, computer vision, mobile robot control, particle simulation, dense matrix computations, cryptography, speech recognition, computer graphics, and game playing [73].

The following is a representative sample of special purpose parallel system in use today:

1. VLSI systems (custom VLSI chips, field-programmable gate arrays),

2. Systolic architectures (application specific arrays, programmable systolic architectures),

3. Cellular automata machines, and

4. Multicomputers (2D and 3D meshes, pyramids, fat trees, hypercubes, and butterflies, etc.).

1.2.3 General Purpose Parallel Computing

We have seen that an idealized model of parallel computation such as the PRAM can provide a robust framework within which to develop techniques for the design, analysis and comparison of parallel algorithms. A major issue in theoretical computer science since the late 1970s has been to determine the extent to which the PRAM and related models can be efficiently implemented on physically realistic distributed memory architectures. A number of new routing and memory management techniques have been developed which show that efficient implementation is indeed possible in some cases. However, for most parallel machines, where there exists a tremendous gap between computation and communication throughputs, efficient simulation is not possible. Portable efficient parallel algorithms, therefore, still heavily rely on further research on some general purpose bridging models.

A *general purpose parallel computer* (GPPC) [73] consists of a set of general purpose microprocessors connected by a communications network. The memory is fully distributed, with each processor having its own physically local memory module. The GPPC supports a single address space across all processors by allocating a part of each module to a common global memory system. Each processor thus has access to its own private address space of local variables, and to the global memory system. The purpose of the communications network is simply to support non-local memory accesses in a *uniformly efficient* way through message routing. By uniformly efficient, we mean that the time taken for a processor to read from, or write to, a non-local memory element in another processor-memory pair should be independent of which physical memory module the value is held in. The algorithm designer should not be aware of any hierarchical memory organization based on the particular physical interconnect structure currently used in the communications network. Instead, performance of the communications network should be described only in terms of its global properties, e.g. the maximum time required to perform a non-local memory operation, and the maximum number of such operations which can simultaneously be performed in the network at any time.

A GPPC differs from a PRAM in that it has a two-level memory organization. Each processor has its own physically local memory; all other memory is non-local and accessible at a uniform rate. In contrast, the PRAM has a one-level memory organization; all memory in a PRAM is non-local. The GPPC and PRAM are similar to the extent that they both have no notion of network locality. The GPPC differs from most current distributed memory multicomputers, e.g. hypercubes, in having no exploitable network locality, but is similar in that it is constructed as a network of processor-memory pairs. One formal model that corresponds reasonably closely to the GPPC would be a distributed memory multicomputer with full connectivity, *i.e.* a multicomputer with an interconnection structure corresponding to the complete graph [60]. Another is the *Bulk Synchronous Parallel* (BSP) computer [95] which, along with some of its variations, will be described in the next subsections.

A sharper distinction between special and general purpose parallel computings is expected. Those primarily concerned with achieving the maximum possible performance for a specific application are likely to move more and more towards highly specialized architectures and technologies in the pursuit of performance gains. In contrast, those for whom it is important to achieve architecture-independence and portability in their designs, will increasingly seek a robust and lasting framework within which to develop their designs.

Described below are some popularly used general purpose parallel programming models.

BSP Model

A BSP computer [95, 74, 41], as illustrated in Figure 1.1, consists of the following:

(1) a set of processor-memory pairs,

(2) a communication network that delivers messages in a point-to-point manner, and

(3) a mechanism for the efficient barrier synchronization of all, or a subset, of the processors.

There are no specialized combining, replication or broadcasting facilities in the BSP. If we define a time step to be the time required for a single local operation, *i.e.* a basic

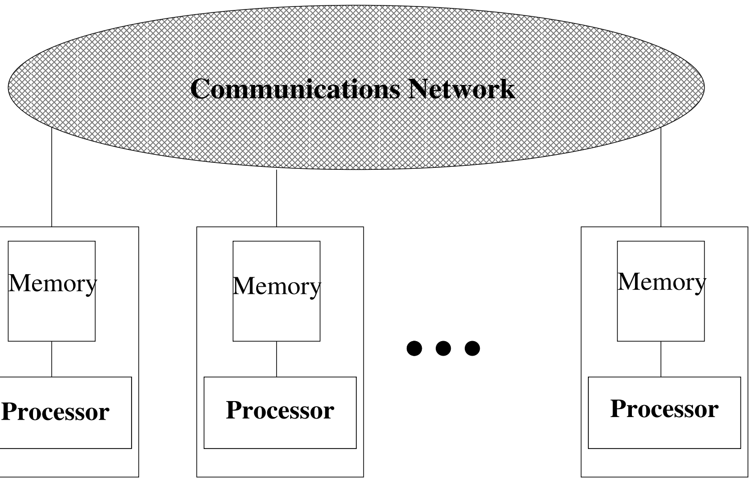

Figure 1.1: Typical BSP architecture

operation on locally held data values, then the performance of any BSP computer can be characterized by the following four parameters:

1. p: number of processors,

2. s: processor speed, *i.e.* number of basic time steps per second,

3. l: synchronization periodicity, i.e. minimal number of time steps between successive synchronization operations, and

4. g: ratio of computation and communication throughputs, *i.e.* (total number of local operations performed by all processors in one second)/(total number of words delivered by the communication network in one second).

The parameter l is related to the network latency, *i.e.* to the time required for a non-local memory access in a situation of continuous message traffic. The parameter g corresponds to the frequency with which non-local memory accesses can be made;

in a machine with a higher value of g, one must make non-local memory accesses less frequently.

We use the term *h-relation* to denote a routing problem where each processor has at most h words of data to send to other processors and each processor is also due to receive at most h words of data from other processors. Specifically, g is related to the time required to realize h-relations in a situation of continuous message traffic; g is the value such that an h-relation can be performed in gh steps.

A BSP algorithm consists of a sequence of parallel *supersteps*, where each superstep is a sequence of steps, followed by a barrier synchronization at which point any memory accesses take effect. During a superstep, each processor has a set of programs or threads which it has to carry out, and it can do the following:

1. perform $\hat{l}$ computation steps, from its set of threads, on values held locally at the start of the superstep;
2. send and receive up to m messages corresponding to non-local read and write requests.

Figure 1.2 shows a BSP superstep.

The complexity of a BSP algorithm is determined as follows. Each superstep is charged $max\{l, \hat{L}, Mg\}$ time steps, where $\hat{L}$ is the maximum number of local computation steps executed by any processor, and M is the maximum number of messages sent by any processor, in that superstep.

The use of parameters l and g to characterize the communications performance of the BSP computer contrasts sharply with the way in which communications performance is

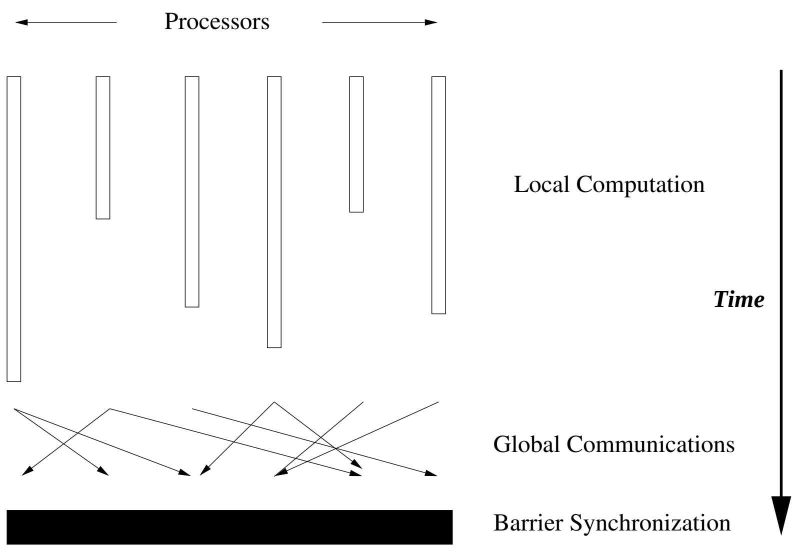

Figure 1.2: A superstep of BSP algorithm

described for most distributed memory architectures on the market, which emphasize the local network properties such as the number of communication channels per-node, the speed of those channels and the graph structure of the network, etc. Emphasizing these local properties reflects the fact that most of those machines are designed to be used in a way where network locality is to be exploited, which is not valid for highly irregularly structured problems.

A major feature of the BSP model is that it lifts considerations of network performance from the local level to the global level. We are thus no longer particularly interested in whether the network is a 2D array, a butterfly or a hypercube, or whether it is implemented in VLSI or in some optical technology. Our interest is in global parameters of the network, such as l and g, which describe its ability to support non-local memory accesses in a uniformly efficient manner.

Valiant's initial intention was to simulate PRAM algorithms on the BSP model [95]. Given a sufficient slackness in the number of processors, with the method proposed in [95], one can simulate PRAM algorithms optimally on distributed memory parallel systems. For some cases and with the help of the communication library, it can be done efficiently [6], but in other cases when g is high, the simulation performs badly [43]. Unfortunately, the latter case is true for most currently available parallel computers. Therefore, algorithms for the BSP model are designed to utilize local computations and to minimize global communications instead of simulating PRAM algorithms.

Over the past ten years, the BSP approach to general purpose parallel computing has been vigorously developed by Professor William F. McColl of Oxford University, in close collaboration with Professor Leslie G. Valiant of Harvard University. Their work on BSP algorithms, architectures and languages has demonstrated convincingly that BSP provides a robust model for parallel computation which offers the prospect of both scalable parallel performance and architecture-independent parallel software.

weak-CREW BSP

In many cases some slight augmentation for the original EREW BSP model facilitates the algorithm description. A slight variation of the original (EREW) BSP model, weak-CREW BSP, was proposed by Goodrich [47]. It is essentially the same as the BSP model with the following exception:

> During a communication superstep, messages can be duplicated by the interconnection media as long as the destinations for any message are a contiguous set of processors $i, i+1, \cdots, j$. Even with message duplication, the

number of messages received by a processor in a superstep is required to be at most $h = O(\frac{n}{p})$.

CGM Model

Another general purpose model, *Coarse Grained Multicomputer* (CGM), proposed by Dehne [30], is essentially similar to the weak-CREW BSP. Work related to the CGM model has been reported in [8, 23, 30, 31, 32, 21]. A CGM computer consists of p processors $P_1, \cdots, P_p$, where each processor has $O(\frac{n}{p})$ local memory. The processors can be connected through any communication medium, *i.e.* any interconnection network or shared memory. Typically, the local memory is considerably larger than $O(1)$. This feature gives the model its name "coarse grained". A CGM algorithm consists of alternating local computation and global communication rounds. In a communication round, a single h-relation (with $h = O(\frac{n}{p})$) is routed, *i.e.* each processor sends $O(\frac{n}{p})$ and receives $O(\frac{n}{p})$ data. The run time of the CGM algorithm is the sum of the run times of the computation and communication rounds. A CGM algorithm with λ rounds and computation cost T_{comp} corresponds to a BSP algorithm with λ supersteps, communication cost $O(g\lambda\frac{n}{p})$, where n is the input size, and the same computation cost T_{comp}.

Compared to the BSP model, a computation/communication round in the CGM model is equivalent to a superstep in the BSP model with $L = \frac{n}{p} \times g$, but includes also the "packing" requirement. It is, therefore, a slightly more powerful model than the BSP model. In general, for those problems whose best possible sequential algorithms take $T_s(n)$ time, ideally algorithm designers would like to design a CGM algorithm using $O(1)$ communication rounds and $O(\frac{T_s(n)}{p})$ total local computation time. But, in many cases, the lower bound of the total communication rounds needed is $O(\log p)$.

EM-BSP*

External memory (EM) algorithms are designed for computational problems in which the size of the internal memory of the computer is only a small fraction of the problem size. Block-wise access to data is a central theme in the design of efficient EM algorithms.

The BSP* model [16, 17, 18] is an extension of BSP designed to encourage block-wise communications between processors. An instance of the p-processor BSP* model is characterized by the parameters g, b, and L. The parameter g is the time (in number of operations) the router needs to deliver a packet of size b (in number of machine words) when in continuous use. b is the minimum packet size that has to be sent in order to achieve the throughput potential of the router. L is the minimum time (in number of operations) between successive synchronization operations, as defined in the original BSP model.

The EM-BSP* model [29] is an extension of BSP* to model secondary memories on each processor (see Figure 1.3). The BSP and CGM models can be extended in a similar way. Some additional parameters are used: M is the local memory size of each processor, D is the number of disk drives of each processor, B is the transfer block size of a local disk drive, and G is the ratio of local computational capacity (number of local computation operations) divided by the local I/O capacity (number of blocks of size B that can be transferred between the local disks and memory) per unit time. (Note: *B is the disk block size, while b is the communication block size. Also, G is the conversion* *factor from I/O operations to CPU instructions, while g is the corresponding factor for communication operations.*)

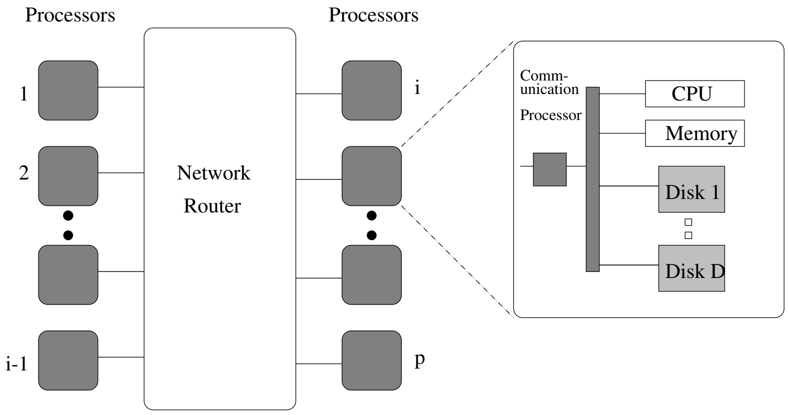

Figure 1.3: A parallel machine with external memory

The need for a model which addresses computation, communication, and I/O costs was identified as an open problem in the Position Statement, ACM Strategic Directions in Computing Research: Working Group on Storage I/O for Large-Scale Computing [26]. The EM-BSP, EM-BSP* and EM-CGM models, introduced in [29], were a step in this direction.

The EM-BSP* model works as follows: Each processor can use all of its D disk drives concurrently, and transfers $D \times B$ items from the local disks to its local memory in a single I/O operation and at cost G. Only one track per disk can be accessed without posing any restriction on which track is accessed on each disk. It takes roughly the same amount of time to access and transfer one block or one word. This reflects the fact that the seek time for a record dominates the time to transmit a record, the transfer delay. A processor can store in its local memory at least one block from each local disk at the same time, *i.e.* $M \geq DB$.

Like a computation on the BSP*, a computation on the EM-BSP* model proceeds in a succession of supersteps. Communication and computation supersteps occur as in the BSP* model and multiple I/O operations are permitted during a single computation superstep. The computation cost and communication cost are the same as for the BSP* model. For each local operation the RAM uniform cost measure is used. For an h-relation, *i.e.* a routing request where each processor sends and receives at most h messages of size b, $gh + L$ time units are charged per communication superstep. The I/O-cost of a computation superstep is $T_{I/O} = max_{j=1}^{p}\{w_{I/O}^{j}\}$ where $w_{I/O}^{j}$ is the I/O-cost incurred by processor j. Each I/O operation costs G time steps. For a computation superstep with at most T_{comp} local operations on each processor, $T_{comp} + T_{I/O} + L$ time units are charged. We assume $G \geq DB$ and $g \geq b$ as reasonable lower bounds for the parameters. As in the BSP* model, it is worthwhile to send messages of size at least b, and the model gives incentives to access all disk drives using block transfers. For instance, a single processor EM-BSP* with D disks is capable of transferring a block of B items to or from each disk in a single I/O operation. An operation involving fewer elements incurs the same cost.

The goodness criterion used by the EM-BSP* model was adapted from the optimality criterion proposed in [42].

Definition 1.2.1 *[29] Let $\mathcal{A}$ be the optimal sequential algorithm on the RAM for the problem under consideration, and let $T(\mathcal{A})$ be its worst case running time. Let c ($c \geq 1$) be a constant. A c-optimal EM algorithm $\mathcal{A}^*$ meets the following criteria:*

- *The ratio* ϕ *between the computation times of* $\mathcal{A}*$ *and* $T(\mathcal{A})\,/p$ *is* $c+o(1)$*;*

- *The ratio* ξ *between the communication time of* $\mathcal{A}*$ *and* $T(\mathcal{A})\,/p$ *is* $o(1)$*; and*

- *The ratio* η *between the I/O-time of* $\mathcal{A}*$ *and the computation time* $T(\mathcal{A})\,/p$ *is* $o(1)$.

All asymptotic bounds refer to the problem size n *as* $n \to \infty$.

We say that an EM algorithm is *one-optimal* if it is *c-optimal* for $c = 1$. The constraint on ϕ is another way of saying that $\mathcal{A}*$ must be *work optimal*. The constraints on ξ and η ensure that the communication and I/O time do not affect the asymptotic running time, even by a constant factor. The terms *communication-efficient* and *I/O-efficient* are used to describe an algorithm for which ξ and η, respectively, are $O(1)$. An algorithm which is work optimal, communication-efficient, and I/O-efficient, therefore, is one whose running time complexity is no worse than the complexity $T(\mathcal{A})\,/p$. Constant factors are ignored. We will call an algorithm *I/O-optimal* if the number of I/O operations matches the lower bound for the number of I/Os required to solve the problem.

The work in this dissertation is mainly on the BSP and CGM models, considering the significant main memory storage amount of currently available parallel machines. We have also mentioned the extension of our current work to the EM-BSP* model in Chapter 6 as the future work.

1.3 Communication Libraries

1.3.1 MPI

The *Message-Passing Interface* (MPI) [78] is a library specification for message passing, proposed as a standard by a broadly based committee of vendors, implementors, and users. The MPI implementations are widely available across all types of parallel environments, including parallel computers, clusters of workstations, and integrated distributed environments (computational grids). Since current systems that conform to the BSP computer model also include networks of workstations, distributed memory processor arrays, and shared memory multiprocessors, programming with the MPI for BSP algorithms makes it relatively easy to take care of portability and performance prediction.

1.3.2 BSPLib

The *BSPLib* [51] is a small communication library for programming in an SPMD (Single Program Multiple Data) manner. The main features of the BSPLib are two modes of communication, one capturing a BSP oriented message passing approach, and the other reflecting a one-sided direct remote memory access (DRMA) paradigm. Compared with the MPI, the BSPLib is a compact library which contains only core primitives. However, these core primitives can be used to realize the various specialized collective communication operations provided in a higher level communication library like the MPI.

1.4 Hardware Platforms

1.4.1 SGI Origin 2000

An SGI Origin 2000[1] is provided by Center for Computational Research (CCR) at UB. The Origin 2000 represents an innovative, new architectural direction for SGI (formerly Silicon Graphics Incorporated). The architecture, known as cache coherent Non-Unified Memory Architecture (ccNUMA), combines the best features of shared memory multiprocessor systems with those of scalable, but non-shared, distributed memory systems. In the ccNUMA architecture, the physical memory is distributed throughout the system (distributed shared memory), one memory bank being shared by two processors. The distribution of memory throughout the system eliminates the memory bus bottlenecks which limit the scalability of bus-based symmetric multiprocessor systems . However, unlike most other distributed memory systems, each processor in the Origin 2000 sees cache-coherent, global memory. This is implemented using a directory-based cache coherency scheme and a a special high-speed, low-latency interconnection network, called CrayLink. The interconnection hardware transparently resolves accesses to remote memory banks, retrieving the contents over the interconnection network, and tracks which processors have which memory cached. Moreover, SGI's Irix operating system will support the automatic migration of remote memory pages to a processor which is making frequent references. This memory architecture, coupled with the memory latency hiding and other advanced features of the R10000 processor, have made the Origin 2000 an extremely powerful parallel machine.

[1]The SGI Origin 2000 hardware description is adapted from CCR's resources webpage.

1.4.2 Sun Enterprise 4000

A Sun Enterprise 4000 server is provided by the department of Computer Science and Engineering. Currently the server is equipped with 8 UltraSparc processors and 4GB main memory. The multiprocessing design of Enterprise 4000 enables the increase of computational power and I/O throughput via low-cost modular upgrades. Furthermore, the efficiency of the multi-threaded Solaris-8 operating environment ensures scalable performance as more CPUs and I/O channels are added to the server. The Enterprise 4000 can be scaled to support extremely I/O-intensive workloads because of the high I/O throughput delivered by the Gigaplane, a 2.6-GB/sec system interconnect. The design of Enterprise 4000 can support up to 14 UltraSparc processors, 14 GB of memory, and 14 I/O channels.

Chapter 2

All Nearest Smaller Values Problem

2.1 Introduction

The all nearest smaller values problem is defined as follows: Let $A = (a_1, a_2, \cdots, a_n)$ be an array of elements from a totally ordered domain. For each a_j, $1 \leq j \leq n$, find the nearest element to the left of a_j and the nearest element to the right of a_j that are less than a_j.

A typical application of the ANSVP is the merging of two sorted lists [20, 69]. Let $A = (a_1, a_2, \cdots, a_n)$ and $B = (b_1, b_2, \cdots, b_n)$ be two increasing arrays to be merged. The reduction to the ANSVP is done by constructing an array $C = (a_1, a_2, \cdots, a_n, b_n, b_{n-1}, \cdots, b_1)$ and then solving the ANSVP with respect to C. If b_y is the right match of a_x, the location of a_x in the merged list is $x + y$. The locations of b_x's can be found similarly.

The ANSVP problem is fundamental in that various applications in graph theory and computational geometry are reducible to it [13]. Some of these applications are described in Chapter 3. The BSP algorithm derived in this chapter can directly be used to design practical communication-efficient parallel algorithms for these problems.

A work-optimal CRCW PRAM algorithm for the ANSVP using $O(\frac{n}{\log\log n})$ processors in $O(\log\log n)$ time was proposed in [13]. Kravets and Plaxton [68] presented a parallel algorithm on the hypercube for ANSVP. They proved that any normal hypercube algorithm requires $\Omega(n)$ processors to solve the ANSVP in $O(\log n)$ time and also presented a normal hypercube ANSV algorithm that is optimal for all values of n and p. Katajainen [61] explored the parallel complexity of the ANSVP on a variation of PRAM, DMM (Distributed Memory Machine, also referred to as Local Memory PRAM). He showed that the ANSVP can be solved in $O(\frac{n}{q})$ time and $O(n)$ space with q processors on the DMM model, provided that $q \in \{1, 2, \cdots, O(\frac{n}{\log n})\}$. More precisely, the machine considered consists of q processors and q memory modules connected by a communication network, which allows every processor to access any memory module in constant time if only every memory module is accessed by at most one processor at a time. The result in [61] complements the earlier known results showing that, on a CREW PRAM, $\Omega(\log n)$ is a lower bound for the time complexity of the problem with any number of processors and that, on a hypercube, any $O(\log n)$ algorithm must use $\Omega(n)$ processors [68].

These algorithms have not been implemented [13, 61, 68]. Our main contribution is to provide the first practical and portable general purpose parallel algorithm for solving the ANSVP with provable communication efficiency in three BSP supersteps and optimal sequential computation time. The portability and scalability have been experimentally justified in Section 2.6.

The rest of this chapter is organized as follows: In Section 2.2, we present the BSP algorithm. The algorithm presented in this section assumes that $\frac{n}{p} = \Omega(p)$, a condition true for all commercially available parallel machines and practical problem sizes. The worst-case and average-case BSP cost analyses are given in Section 2.3. Section 2.4 presents the theoretical improvement for BSP computers with local memory size $\frac{n}{p} = o(p)$. Section 2.5 shows that a communication-optimal parallel parenthesis matching algorithm results from the BSP algorithm in Section 2.2. Section 2.6 provides the experimental results on the SGI Origin 2000 and Sun Enterprise 4000 parallel machines. Section 2.8 concludes this chapter.

2.2 The BSP Algorithm

Given a sequence $A = (a_1, a_2, \cdots, a_n)$ and a p-processor BSP computer with any communication media (shared memory or any interconnection network), each P_i $(1 \leq i \leq p)$ stores $(a_{\frac{n}{p}(i-1)+1}, a_{\frac{n}{p}(i-1)+2}, \cdots, a_{\frac{n}{p}i})$. For simplicity, we assume that the elements in the sequence are distinct. We define the nearest smaller value to the right of an element to be its *right match*. The ANSVP can be solved sequentially with linear time using a straightforward stack approach. To find the right matches of the elements, we scan the input, keep the elements for which no right match has been found on a stack, and report the current element as a right match for those elements on the top of the stack that are larger than the current element. The left matches can be found similarly. For brevity and without loss of generality, we will focus on finding the right matches.

Some definitions are given below. Throughout the rest of this chapter, we use i ($1 \leq i \leq p$) for processor related indexing and j ($1 \leq j \leq n$) for array element related indexing.

- For any j:
  - $\mathrm{rm}(j)$ ($\mathrm{lm}(j)$, respectively) $\stackrel{\mathrm{def}}{=}$ the index of the right (left, respectively) match of a_j.
  - $\mathrm{rmp}(j)$ ($\mathrm{lmp}(j)$, respectively) $\stackrel{\mathrm{def}}{=}$ the index of the processor containing $a_{\mathrm{rm}(j)}$ ($a_{\mathrm{lm}(j)}$, respectively).
- For any i:
  - $\min(i) \stackrel{\mathrm{def}}{=}$ the index (in A) of the smallest element in P_i.
  - $\mathrm{rm\_min}(i)$ ($\mathrm{lm\_min}(i)$, respectively) $\stackrel{\mathrm{def}}{=}$ the index of the right (left, respectively) match of $a_{\min(i)}$ with respect to the array $A_{\min} = (a_{\min(1)}, \cdots, a_{\min(p)})$.
  - $\wp_i \stackrel{\mathrm{def}}{=} \{P_x | \ \mathrm{rm\_min}(x) = i\}$; $\varphi_i \stackrel{\mathrm{def}}{=} \{P_x | \ \mathrm{lm\_min}(x) = i\}$.

Based on the above definitions, we observe that $P_{\mathrm{rmp}(\min(i))} = P_{\mathrm{rm\_min}(i)}$ ($P_{\mathrm{lmp}(\min(i))} = P_{\mathrm{lm\_min}(i)}$, respectively). Next we prove a lemma used in our BSP algorithm.

Lemma 2.2.1 *On a p-processor BSP computer, for any i, if* $a_{\mathrm{rm}(\min(i))}$ *exists and* $\mathrm{rmp}(\min(i)) \neq i+1$, *then there exists a unique processor* $P_{k(i)}$, $i < k(i) < \mathrm{rmp}(\min(i))$, *such that* $\mathrm{lmp}(\min(k(i))) = i$ *and* $\mathrm{rmp}(\min(k(i))) = \mathrm{rmp}(\min(i))$. (*Symmetrically, for any i, if* $a_{\mathrm{lm}(\min(i))}$ *exists and* $\mathrm{lmp}(\min(i)) \neq i-1$, *then there exists a unique processor* $P_{k'(i)}$, $\mathrm{lmp}(\min(i)) < k'(i) < i$, *such that* $\mathrm{lmp}(\min(k'(i))) = \mathrm{lmp}(\min(i))$ *and* $\mathrm{rmp}(\min(k'(i))) = i$).

Proof: We show that P_s, where $a_{\min(s)} = \min\{a_{\min(i+1)}, a_{\min(i+2)}, \cdots, a_{\min(\text{rmp}(\min(i))-1)}\}$, is the unique processor described in the lemma. For any s' with $i+1 \leq s' < s$, $\text{rmp}(\min(s'))$ must be $\leq s$. Similarly, for any s' with $s+1 \leq s' < \text{rmp}(\min(i))$, $\text{lmp}(\min(s'))$ must be $\geq s$. This leaves P_s the only candidate processor. We can easily infer that $a_{\min(s)} > a_{\min(i)} > a_{\min(\text{rmp}(\min(i)))}$. Since $a_{\min(s)}$ is the smallest element among those in $P_{i+1}, \cdots, P_{\text{rmp}(\min(i))-1}$, we conclude that P_s is the unique processor $P_{k(i)}$ specified in Lemma 2.2.1. (The symmetric part can be proved similarly.) □

We next outline our algorithm. To begin with, all processors sequentially find the right matches for their local elements, using the stack approach. Those matched elements require no interprocessor communication. We therefore focus on those elements which are not yet matched. The general idea is to find the right matches for those not-yet-matched elements by reducing the original ANSVP to $2p$ smaller "special" ANSVPs, and solve them in parallel.

Next we compute the right and left matches for all $a_{\min(i)}$'s. To do this, we first solve the ANSVP with respect to the array $A_{\min} = (a_{\min(1)}, \cdots, a_{\min(p)})$. Then, for each processor P_i, we define four sequences, Seq1_i, Seq2_i, Seq3_i and Seq4_i. Seq1_i and Seq2_i are defined as follows:

(1) If $a_{\text{rm}(\min(i))}$ does not exist, then Seq1_i and Seq2_i are undefined.

(2) If $a_{\text{rm}(\min(i))}$ exists and $\text{rmp}(\min(i)) = i+1$, then:

$\text{Seq1}_i = (a_{\min(i)}, \cdots, a_{\frac{n}{p}i})$, $\text{Seq2}_i = (a_{\frac{n}{p}i+1}, \cdots, a_{\text{rm}(\min(i))})$.

(3) If $a_{\text{rm}(\min(i))}$ exists, $\text{rmp}(\min(i)) > i+1$, and $P_{k(i)}$ is the unique processor specified in Lemma 2.2.1, then:

$\text{Seq1}_i = (a_{\min(i)}, \cdots, a_{\text{lm}(\min(k(i)))})$, $\text{Seq2}_i = (a_{\text{rm}(\min(k(i)))}, \cdots, a_{\text{rm}(\min(i))})$.

Symmetrically, Seq3_i and Seq4_i are defined as follows:

(1) If $a_{\text{lm}(\min(i))}$ does not exist, then Seq3_i and Seq4_i are undefined.

(2) If $a_{\text{lm}(\min(i))}$ exists and $\text{lmp}(\min(i)) = i-1$, then:

$\text{Seq3}_i = (a_{\frac{n}{p}(i-1)+1}, \cdots, a_{\min(i)})$, $\text{Seq4}_i = (a_{\text{lm}(\min(i))}, \cdots, a_{\frac{n}{p}(i-1)})$.

(3) If $a_{\text{lm}(\min(i))}$ exists, $\text{lmp}(\min(i)) < i-1$, and $P_{k'(i)}$ be the unique processor specified in Lemma 2.2.1. then:

$\text{Seq3}_i = (a_{\text{rm}(\min(k'(i)))}, \cdots, a_{\min(i)})$, $\text{Seq4}_i = (a_{\text{lm}(\min(i))}, \cdots, a_{\text{lm}(\min(k'(i)))})$.

Note that Seq1_i and Seq3_i, if they exist, always reside on P_i, Seq2_i, if it exists, always resides on $P_{\text{rmp}(\min(i))}$, and Seq4_i, if it exists, always resides on $P_{\text{lmp}(\min(i))}$.

The following two lemmas 2.2.2 and 2.2.3 specify how to find the right matches for all unmatched elements. Detailed proofs can be found in [13].

Lemma 2.2.2 *The right matches of all not-yet-matched elements in* Seq1_i *lie in* Seq2_i*. The right matches of all not-yet-matched elements in* Seq4_i*, except its first element, lie in* Seq3_i*.*

Each processor P_i therefore is responsible for identifying right matches for not-yet-matched elements in Seq1_i and Seq4_i. Again, we apply the sequential algorithm at each processor P_i with respect to the two concatenated sequences, $\mathrm{Seq1}_i||\mathrm{Seq2}_i$ and $\mathrm{Seq4}_i||\mathrm{Seq3}_i$.

Lemma 2.2.3 *All elements will be right-matched after the above-mentioned 2p special ANSVPs are solved in parallel.*

We need the following lemma:

Lemma 2.2.4 1. *Suppose that $\wp_i = \{P_{x_1}, P_{x_2}, \cdots, P_{x_t}\}$ where $x_1 < x_2 < \cdots < x_t$. Then:*

$\mathrm{Seq2}_{x_1} = (a_{\mathrm{rm}(\min(x_2))}, \cdots, a_{\mathrm{rm}(\min(x_1))})$, $\mathrm{Seq2}_{x_2} = (a_{\mathrm{rm}(\min(x_3))}, \cdots, a_{\mathrm{rm}(\min(x_2))})$,

$\cdots$, $\mathrm{Seq2}_{x_t} = (a_{\frac{n}{p}(i-1)+1}, \cdots, a_{\mathrm{rm}(\min(x_t))})$.

2. *Suppose that $\varphi_i = \{P_{y_1}, P_{y_2}, \cdots, P_{y_s}\}$ where $y_1 < y_2 < \cdots < y_s$. Then:*

$\mathrm{Seq4}_{y_1} = (a_{\mathrm{lm}(\min(y_1))}, \cdots, a_{\mathrm{lm}(\min(y_2))})$, $\mathrm{Seq4}_{y_2} = (a_{\mathrm{lm}(\min(y_2))}, \cdots, a_{\mathrm{lm}(\min(y_3))})$,

$\cdots$, $\mathrm{Seq4}_{y_s} = (a_{\mathrm{lm}(\min(y_s))}, \cdots, a_{\frac{n}{p}i})$.

Proof: We only prove Statement 1. The proof of Statement 2 is similar. First observe that, for any $P_x, P_y \in \wp_i$, $x < y$ implies $a_{\min(x)} < a_{\min(y)}$ and $k(x) \leq y$. Based on these observations, we have $k(x_l) = x_{l+1}$ for $1 \leq l < t$ and $x_t = i - 1$. The lemma follows from the definition of Seq2. □

The algorithm below finds the right matches and is therefore denoted as Algorithm *$ANSV_r$*. It is described following BSP programming style. In each step, we also mention the standard MPI libraries we use. (MPI is a standard specification for message passing and will be described in more details in Section 2.6.)

Algorithm *$ANSV_r$*:

Input: A partitioned into p subsets of continuous elements. Each processor stores one subset.

(MPI_Scatter)

Output: The right match of each a_i is computed and stored in the processor containing a_i.

1. Each P_i sequentially solves the $ANSV_r$ problem with respect to its local subset.
2. (a) Each P_i computes its local minimum $a_{\min(i)}$.

   (b) All $a_{\min(i)}$'s are globally communicated. (Hence each P_i has the array $A_{\min}$.)

   (MPI_Allgather, MPI_Barrier)
3. Each P_i solves the $ANSV_r$ and the $ANSV_l$ problems with respect to $A_{\min}$; and identify the sets $\wp_i$ and φ_i.
4. Each P_i computes $a_{\mathrm{rm}(\min(x))}$ for every $P_x \in \wp_i$ and $a_{\mathrm{lm}(\min(y))}$ for every $P_y \in \varphi_i$.
5. Each P_i determines $\mathrm{Seq1}_i$, $\mathrm{Seq3}_i$ and receives $\mathrm{Seq2}_i$, $\mathrm{Seq4}_i$ as follows:

   (a) Each P_i computes the unique $k(i)$ and $k'(i)$ (as in Lemma 2.2.1), if existing, and determines $\mathrm{Seq1}_i$ and $\mathrm{Seq3}_i$.

   (b) Each P_i determines $\mathrm{Seq2}_x$ for every $P_x \in \wp_i$, and $\mathrm{Seq4}_y$ for every $P_y \in \varphi_i$ (as in Lemma 2.2.4).

   (c) Each P_i sends $\mathrm{Seq2}_x$, for every $P_x \in \wp_i$, to P_x and $\mathrm{Seq4}_y$, for every $P_y \in \varphi_i$, to P_y.

   (MPI_Send, MPI_Recv, MPI_Barrier)

6. (a) Each P_i finds the right matches for the unmatched elements in $\text{Seq}1_i$ and $\text{Seq}4_i$

   (b) Each P_i collects the matched $\text{Seq}4_y$'s from all P_y's$\in \varphi_i$.

   (MPI_Send, MPI_Recv, MPI_Barrier)

2.3 Complexity Analysis

2.3.1 Computation and Communication Complexities

Recall from Chapter 1 that the term *h-relation* denotes a routing problem where each processor has at most h words of data to send to other processors and each processor is also due to receive at most h words of data from other processors. In each BSP superstep, if at most w arithmetic operations are performed by each processor and the data communicated forms an *h-relation*, then the cost of this superstep is $w+h*g+L$. The cost of a BSP algorithm using S supersteps is simply the sum of the costs of all S supersteps:

$$BSP\ cost = comp.\ cost + comm.\ cost + \ synch.\ cost = W + H*g + L*S,$$

where H is the sum of the maxima of the *h-relation*s in each superstep and W is the sum of the maxima of the local computations in each superstep.

Note that the definition of an h-relation distinguishes the cost of a balanced communication pattern from one that is skewed. A communication pattern in which each processor sends a unit-size message to some other (distinct) processor counts as a 1-relation. However, a communication pattern that transfers the same number of mes-

Table 2.1: BSP cost breakdown of *Algorithm* $ANSV_r$

	Cost		
Step	comp.	comm.	synch.
1	$T_s(\frac{n}{p})$		
2(a)	$T_\theta(\frac{n}{p})$		
2(b)		pg	L
3	$2T_s(p)$		
4	$\max_i\{T_\theta(\frac{n}{p}+\lvert\varphi_i\rvert)+T_\theta(\frac{n}{p}+\lvert\wp_i\rvert)\}$		
5(a)	$\max_i\{T_\theta(\text{rm\_min}(i) - \text{lm\_min}(i))\}$		
5(b)	$O(\frac{n}{p})$		
5(c)		$\max_i\{(\Sigma_{P_x\in\wp_i}\lvert\text{Seq2}_x\rvert + \Sigma_{P_y\in\varphi_i}\lvert\text{Seq4}_y\rvert)g\}$	L
6(a)	$\max_i\{T_s(\lvert\text{Seq1}_i\rvert + \lvert\text{Seq2}_i\rvert) + T_s(\lvert\text{Seq4}_i\rvert+\lvert\text{Seq3}_i\rvert)\}$		
6(b)		$\max_i\{\Sigma_{P_x\in\varphi_i}\lvert\text{Seq4}_x\rvert g\}$	L

sages, but in the form of a broadcast from one processor to all the others, counts as a p-relation. Hence, unbalanced communication, which is the most likely to cause congestion, is charged a higher cost. Thus the cost model does take into account congestion phenomena arising from the limits on each processor's capacity to send and receive data, and from the extra traffic that might occur on the communication links near a busy processor.

Here we assume the sequential computation time for the $ANSV_r$ problem of input size n is $T_s(n)$, and the sequential time for finding the minimum of n elements is $T_\theta(n)$. Then the BSP cost breakdown of Algorithm $ANSV_r$ can be derived as Table 5.1.

Since $\frac{n}{p} = \Omega(p)$ and $T_s(n) = T_\theta(n) = O(n)$, the computation time in each step is obviously linear in the local data size, namely $O(\frac{n}{p})$. Steps 2 (b), 5 (c) and 6 (b) involve communication. Thus the algorithm takes three supersteps. Based on Lemma 2.2.4

and the fact that $|\varphi_i| + |\wp_i| \leq p$, the communication steps 5 (c) and 6 (b) can each be implemented by an $(\frac{n}{p} + p)$-relation. Therefore we have:

Theorem 2.3.1 *The ANSVP with input size n can be solved on a p-processor BSP machine in three supersteps using linear local computation time and at most an $(\frac{n}{p} + p)$-relation in each communication phase, provided $p \leq n/p$.*

2.3.2 Average-Case Communication Complexity

The communication complexity analysis given in the last subsection is for the worst case. In our experiments, we have observed that, for fixed p, the communication costs do not seem to depend on the input size n. Since applications using the ANSVP tend to have randomly generated inputs, finding the average-case communication complexity actually gives more precise communication time estimates. Although the communication cost of step 2 (b) of Algorithm $ANSV_r$ is always $\Theta(pg)$, we will show in this section that, on random inputs, the expected communication costs of steps 5 (c) and 6 (b) are actually both $O(g)$, much smaller than the communication cost $O(\frac{n}{p}g)$ as indicated in the worst case analysis. Thus the total average communication cost of Algorithm $ANSV_r$ is $\Theta(pg)$. Since p is normally far smaller than n/p, the average communication cost of our algorithm is much better than that indicated in the worst-case analysis.

We focus on step 5 (c). The analysis of step 6 (b) is similar. In step 5 (c), processor P_i sends out Seq2_x for each $P_x \in \wp_i$. We will show that the expected size of Seq2_x and the expected size of $\wp_i$ are both $O(1)$. This will establish our claim.

Consider the $ANSV_r$ problem on the array $A_{\min} = (a_{\min(1)}, \cdots, a_{\min(p)})$. For convenience, we define rm_min$(i) = p+1$ if $a_{\min(i)}$ has no right match. Note that, for $i+1 \leq u \leq p$, rm_min$(i) = u$ if and only if $a_{\min(i+1)}, \cdots, a_{\min(u-1)}$ are all greater than $a_{\min(i)}$, and $a_{\min(u)} < a_{\min(i)}$. Since the input is random, we have:

$$P_r[\text{rm\_min}(i) = u] = \frac{1}{2^{u-i}}, \text{ for } i+1 \leq u \leq p \,; \tag{2.1}$$

$$P_r[\text{rm\_min}(i) = p+1] = \frac{1}{2^{p-i}}. \tag{2.2}$$

Suppose that rm_min$(i) = u$ for some $i < u \leq p$. Then $a_{\min(k(i))}$ is the minimum value among $a_{\min(i+1)}, \cdots, a_{\min(u-1)}$. Since the input is random, the probability of all possibilities are equal. Thus, for $i < v < u \leq p$, we have:

$$P_r[k(i) = v \mid \text{rm\_min}(i) = u] = \frac{1}{u-i-1} \,.$$

The expected value of $|\text{Seq2}_i|$ can be described as:

$$E(|\text{Seq2}_i|) = \sum_{u=i+1}^{p+1} P_r[\text{rm\_min}(i) = u] \cdot E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u). \tag{2.3}$$

Assume that rm_min$(i) = u$ and $k(i) = v$. Suppose that the right match of $a_{\min(u)}$ is the t-th ($1 \leq t \leq n/p$) element in the sub-array contained in P_u, and the right match of $a_{\min(v)}$ is the x-th ($1 \leq x \leq t$) element in the sub-array contained in P_u. Then $|\text{Seq2}_i| = t - x + 1$. Thus:

$$
\begin{aligned}
E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u \mid k(i) = v) &= \sum_{t=1}^{\frac{n}{p}} \frac{1}{2^t}\Big(\sum_{x=1}^{t-1} \frac{1}{2^x}(t-x+1) + \frac{1}{2^{t-1}} \cdot 1\Big) \\
&= \sum_{t=1}^{\frac{n}{p}} \frac{1}{2^t}\Big(t-1+\frac{1}{2^{t-1}}\Big) \\
&= \frac{5}{3} - \frac{\frac{n}{p}+1}{2^{\frac{n}{p}}} - \frac{1}{3}\frac{1}{2^{\frac{2n}{p}-1}} < \frac{5}{3}.
\end{aligned}
$$

This gives:

1. $E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = p+1) = 0$,

2. $E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = i+1)$

   $= \sum_{s=1}^{\frac{n}{p}} s \cdot P_r[$ the right match of $a_{\min(i)}$ is the sth element in the sub-array of $P_{i+1}]$

   $= \sum_{s=1}^{\frac{n}{p}} s \cdot \frac{1}{2^s} = 2(1 - \frac{1}{2^{\frac{n}{p}}} - \frac{\frac{n}{p}}{2^{\frac{n}{p}+1}}) < 2$, and

3. $E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u)$, $i+2 \le u \le p$, can be derived as follows:

$$
\begin{aligned}
E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u) &= \sum_{s=i+1}^{u-1} \frac{1}{u-i-1} \cdot E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u \mid k(i) = v) \\
&\le \sum_{s=i+1}^{u-1} \frac{1}{u-i-1} \cdot \frac{5}{3} = \frac{5}{3}.
\end{aligned}
$$

Therefore,

(1) $E(\left|\text{Seq2}_p\right|) = 0$;

(2) $E(\left|\text{Seq2}_{p-1}\right|) \le \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 2 = 1$; and

(3) $E(|\text{Seq2}_i|)$, $1 \leq i \leq p-2$, can be derived as follows:

$$\begin{aligned} E(|\text{Seq2}_i|) &= \sum_{u=i+1}^{p+1} P_r[\text{rm\_min}(i) = u] \cdot E(|\text{Seq2}_i| \mid \text{rm\_min}(i) = u) \\ &\leq \frac{1}{2^{p-i}} \cdot 0 + \frac{1}{2} \cdot 2 + \sum_{u=i+2}^{p} \frac{1}{2^{u-i}} \cdot \frac{5}{3} \\ &\leq 1 + (\frac{1}{2} - \frac{1}{2^{p-i}}) \cdot \frac{5}{3} \leq 1 + \frac{1}{2} \cdot \frac{5}{3} = \frac{11}{6}. \end{aligned}$$

Next, we compute the expected size of $\wp_i$. Note that, for $1 \leq x < i$, processor $P_x \in \wp_i$ if and only if $a_{\min(x)} > a_{\min(i)}$ and $a_{\min(l)} > a_{\min(x)}$ for all $x+1 \leq l \leq i-1$. Thus:

$$E(|\wp_i|) = \sum_{x=1}^{i-1} P_r[P_x \in \wp_i] = \sum_{x=1}^{i-1} \frac{1}{2^{i-x}} = 1 - \frac{1}{2^i} < 1.$$

Hence, the expected communication requirements for steps 5 (c) and 6 (b) are both bounded above by $\frac{11}{6} \times 1 = \frac{11}{6}$. Step 2 (b) therefore becomes the dominant step in communication requirement, which takes simply a p-relation.

This concludes the following theorem:

Theorem 2.3.2 *The ANSVP with input size n can be solved on a p-processor BSP machine in three supersteps using linear local computation time, with the average-case communication requirement bounded above by a p-relation for each communication phase.*

2.4 Theoretical Improvement

Finding right and left matching processors in step 2 (b) can be finished in a single superstep, provided $\frac{n}{p} = \Omega(p)$, a condition true for all commercially available parallel machines. For theoretical completeness, we will also consider the situation when $\frac{n}{p} = o(p)$, an extremely fine-grained case. In this section, we theoretically improve the scalability of Algorithm $ANSV_r$ to $\frac{n}{p} = \Omega(p^{\varepsilon})$, for any $\varepsilon > 0$.

We conceptually imagine the p-processor BSP model as a complete α-*ary* tree machine, with α^k leaf processors and $\frac{\alpha^{k+1}-1}{\alpha-1}$ processors in total. (Here α is a parameter used for grouping processors which will be determined later.) Each non-leaf processor and its rightmost child are actually the same processor. An example using 16 processors with $\alpha = 4$ is shown in Figure 2.1.

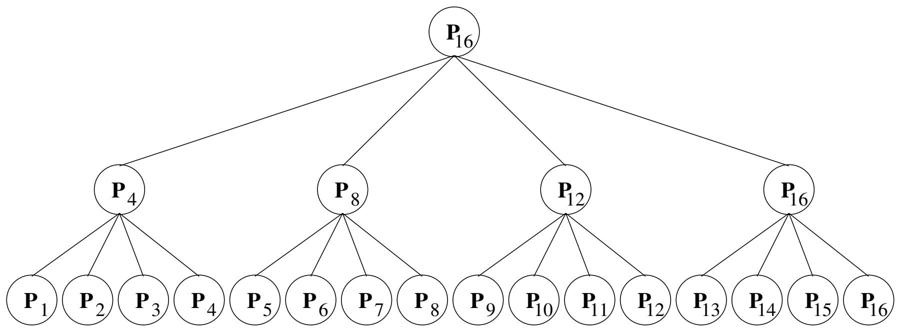

Figure 2.1: A conceptual 4-ary tree with respect to a 16-processor BSP model

Now we need to find right and left matches for $(a_{\min(1)}, a_{\min(2)}, \cdots, a_{\min(p)})$, and each $a_{\min(i)}$ is currently stored at processor P_i. For simplicity, we assume $p = \alpha^k$. The algorithm can easily be extended for BSP machines with any p.

We perform the following steps:

1. Each internal tree node finds the minimum value of its descendants.

2. Each processor P_i finds the right (left) match of $a_{\min(i)}$ by climbing up the tree until it reaches a node such that the value of its immediate right (left) sibling is smaller than $a_{\min(i)}$.

3. Then it proceeds down the tree aiming at the leftmost (rightmost) leaf whose value is smaller than $a_{\min(i)}$.

We let $\alpha = \min\{\lceil \frac{n}{p} \rceil, p\}$. The required number of supersteps is therefore $O(\log_\alpha p)$. Thus, with scalability measure $\frac{n}{p} = \Omega(p^\varepsilon)$, $\log_\alpha p \leq \frac{\log p}{\varepsilon \log p} = \frac{1}{\varepsilon} = O(1)$. Each superstep requires an α-relation and the local computation time is obviously $O(\frac{n}{p})$.

2.5 Parenthesis Matching – A Typical ANSVP Application

A typical and direct application of ANSVP is the parallel *parenthesis matching* problem. A sequence of parentheses is *balanced* if every left (right, respectively) parenthesis has a matching right (left, respectively) parenthesis. Let a balanced sequence of parentheses be represented by $(a_1, a_2, \cdots, a_n)$, where a_k represents the k-th parenthesis. It is required to determine the matching parenthesis of each parenthesis.

The parallel parenthesis matching algorithm will employ a fundamental operation, *prefix sum*, which is defined in terms of a binary, associative operator $\otimes$. The computation takes as input a sequence $< b_1, b_2, \cdots, b_n >$ and produces as output a sequence $< c_1, c_2, \cdots, c_n >$ such that $c_1 = b_1$ and $c_k = c_{k-1} \otimes b_k$ for $k = 2, 3, \cdots, n$.

We will describe the parenthesis matching algorithm first. The details regarding how prefix sum operation can be implemented on BSP will be described next.

Algorithm Parenthesis Matching :

Input: A legal parenthesis list, partitioned into p subsets of continuous elements. Each processor stores one subset.

Output: The mate location of each parenthesis is stored in the processor containing that parenthesis.

1. Each processor assigns 1 for each left parenthesis and -1 for each right parenthesis.
2. Perform BSP prefix sum operation on this array and derive the resulting array A.
3. Perform Algorithm $ANSV_r$ on A.

Note that the BSP prefix sum operation used in step 2 can be implemented in 2 BSP supersteps, provided that $\frac{n}{p} = \Omega(p)$, as below.

Assume that the initial input is $b_1, b_2, \cdots, b_n$ and the processors are labeled $P_1, P_2, \cdots, P_p$. Processor P_i initially contains $b_{\frac{n}{p}(i-1)+1}, b_{\frac{n}{p}(i-1)+2}, \cdots, b_{\frac{n}{p}i}$.

2.1 Each processor computes all prefix sums of $b_{\frac{n}{p}(i-1)+1}, b_{\frac{n}{p}(i-1)+2}, \cdots, b_{\frac{n}{p}i}$ and stores the results in local memory locations $s_{\frac{n}{p}(i-1)+1}, s_{\frac{n}{p}(i-1)+2}, \cdots, s_{\frac{n}{p}i}$.

2.2 Each P_i sends $s_{\frac{n}{p}i}$ to processor P_p.

2.3 Processor P_p computes prefix sums of the p values received in step 2.2 and store these values at $t_1, t_1, \cdots, t_p$.

2.4 Processor P_p then sends t_i to $P_{i+1}, 1 \leq i \leq p-1$.

2.5 Each processor adds the received value to $s_{\frac{n}{p}(i-1)+1}, s_{\frac{n}{p}(i-1)+2}, \cdots, s_{\frac{n}{p}i}$.

Steps 2.2 and 2.4 are involved with communication and we can easily arrange the prefix sum algorithm in 2 BSP supersteps, each taking a p-relation in the communication phase.

Note that Chapter 4 presents a coarse-grained parallel prefix sum algorithm that relieves the condition $\frac{n}{p} = \Omega(p)$. This prefix sum algorithm can also be applied here to improve the scalability.

2.5.1 The Average-Case Communication Complexity

Since step 2 of Algorithm Parenthesis Matching takes only a p-relation. Step 3 therefore becomes dominant in communication requirement. Although the reduction from parenthesis matching to ANSVP is obvious, their average-case communication complexity analyses are very different since these two problems have significantly different problem domains.

In this section we first prove the average-case lower bound of the communication requirement of parallel parenthesis matching and show that our BSP ANSV algorithm actually results in an asymptotically communication-optimal parallel algorithm for this problem.

Average-Case Lower Bound

Let S_n be the set of all possible legal parenthesis of length n, where $n = 2m$. We have $|S_n| = C_m = \frac{1}{m+1}\begin{pmatrix} 2m \\ m \end{pmatrix}$, the m-th Catalan number [27]. Some definitions in Section 2.2 are inherited here. In addition, given a sequence $(a_1, a_2, \cdots, a_n) \in S_n$, we define

(1) rm(j) (lm(j), respectively) to be the index of the right (left, respectively) matching parenthesis if a_j is a left (right, respectively) parenthesis;

(2) $M_s(i)$ and $M_r(i)$ to be the total number of right and left parentheses in P_i that are not locally matched, while $(a_1, a_2, \cdots, a_n)$ is evenly partitioned and distributed among the p processors. Note that $M_s(i)$ ($M_r(i)$, respectively) is the size of the message sent (received, respectively) by processor P_i in step 3 of the Algorithm Parenthesis Matching.

Moreover, for a randomly selected sequence from S_n, we denote the event that the j-th element is a right (left, respectively) parenthesis as $a_j = r$ ($a_j = l$, respectively). The following lemma can be derived.

Lemma 2.5.1 $P_r[a_j = r] = (\sum_{k=0}^{\lfloor \frac{j}{2} \rfloor - 1} C_k C_{m-1-k})/C_m$.

Proof: We count the number of legal sequences where a_j is a right parenthesis. The subsequence between a_j and its left match must be a legal sequence and hence consisting of an even number of parentheses. Thus, the left match of a_j can only occur at locations $j-1, j-3, \cdots,$ and $j - (2\lfloor \frac{j}{2} \rfloor - 1)$. In addition, the total number of sequences with $a_j = r$ and the left match of a_j being at location $a_{j-(2k-1)}$ is $C_{k-1}C_{m-k}$. (This is because the sequence $a_{j-2k+2}\dots a_{j-1}$ must be a legal sequence consisting of $(k-1)$ pairs of parentheses and the sequence $a_1\dots a_{j-2k}a_{j+1}\dots a_n$ must be a legal sequence consisting of $(m-k)$ pairs of parentheses). Therefore, the total number of sequences with $a_j = r$ is $\sum_{k=0}^{\lfloor \frac{j}{2}\rfloor -1} C_k C_{m-1-k}$, which completes the proof. □

We also observe that:

(1) $P_r[a_j = l] = 1 - P_r[a_j = r]$,

(2) $P_r[a_j = r] = P_r[a_{n-j+1} = l]$, and

(3) $E(M_s(i)) = E(M_r(p-i+1))$.

Observations (2) and (3) can be seen by reversing the sequence and switching the roles of right and left parentheses.

Lemma 2.5.2 *Let a_k be a parenthesis in processor P_i. The probability that a_k is a right parenthesis and is not locally matched in P_i is: $P_r[a_k = r] - \sum_{t=0}^{\lfloor \frac{k-\frac{n}{p}(i-1)}{2}\rfloor -1} C_t C_{m-1-t}/C_m$.*

Proof: The probability that a_k is a right parenthesis is $P_r[a_k = r]$. a_k can only be locally matched by a left parenthesis a_{k-2t-1} for $t = 0, 1, \dots \lfloor \frac{k-\frac{n}{p}(i-1)}{2} \rfloor - 1$. By the argument in the proof of Lemma 2.5.1, for each fixed t, the probability that a_{k-2t-1} is the matching left parenthesis of a_k is $C_t C_{m-1-t}/C_m$. This proves the lemma. □

Lemma 2.5.3 *The expected maximum size of messages (sent and received) by any processor is given by $E(M_s(p))$.*

Proof: By Lemma 2.5.2, the size of the message sent by processor P_i $(1 \le i \le p)$ is: $E(M_s(i)) = \sum_{k=\frac{n}{p}(i-1)+1}^{\frac{n}{p}i} T_k(i)$, where $T_k(i) = P_r[a_k = r] - \sum_{t=0}^{\lfloor \frac{k-\frac{n}{p}(i-1)}{2}\rfloor -1} C_t C_{m-1-t}/C_m$) is the probability that a_k is an unmatched right parenthesis within P_i.

For any fixed $i_1 < i_2$ and k, the second term ($\sum_{t=0}^{\lfloor \frac{k-\frac{n}{p}(i-1)}{2} \rfloor -1} C_t C_{m-1-t}/C_m$) in $T_k(i_1)$ and $T_k(i_2)$ are identical. The first term ($P_r[a_k = r]$) in $T_k(i_1)$ is less than the first term ($P_r[a_k = r]$) in $T_k(i_2)$. Thus $E(M_s(p)) = max_{0 \le i \le p-1}\{E(M_s(i))\}$. Thus, the maximum size of the message sent by any processor is $E(M_s(p))$.

Similarly, the maximum size of messages received by any processor is $E(M_r(1))$ which, according to Observation (3), equals to $E(M_s(p))$. □

Thus, to lower bound the expected maximum message size, we need to lower bound $E(M_s(p))$. In the following lemmas, we will show that, for a random legal sequence, most right parentheses in processor P_p are not locally matched.

Lemma 2.5.4 $\frac{C_x C_{m-x-1}}{C_m} \ge \frac{m^{\frac{3}{2}}}{8\sqrt{\pi}(m-x-1)^{\frac{3}{2}} x^{\frac{3}{2}}}$.

Proof: According to Stirling's approximation ($\sqrt{2\pi n}(\frac{n}{e})^n \le n! \le \sqrt{2\pi n}(\frac{n}{e})^{n+\frac{1}{12n}}$), we have

$$\begin{aligned}
C_x &= \frac{1}{x+1}\frac{(2x)!}{x!x!} \\
&\ge \frac{1}{x+1}\frac{\sqrt{(2\pi)(2x)}(\frac{2x}{e})^{2x}}{[\sqrt{2\pi x}(\frac{x}{e})^{x+\frac{1}{12x}}][\sqrt{2\pi x}(\frac{x}{e})^{x+\frac{1}{12x}}]} \\
&\ge \frac{2^{2x}e^{\frac{1}{6x}}}{(x+1)\sqrt{\pi x}(x^{\frac{1}{6x}})}.
\end{aligned}$$

Analogously,

$$\begin{aligned}
C_{m-x-1} &\ge \frac{2^{2(m-x-1)}e^{\frac{1}{6(m-x-1)}}}{(m-x)\sqrt{\pi(m-x-1)}(m-x-1)^{\frac{1}{6(m-x-1)}}} \quad \text{and} \\
C_m &\le \frac{1}{m+1}\frac{\sqrt{(2\pi)(2m)}(\frac{2m}{e})^{2m+\frac{1}{24m}}}{\sqrt{2\pi m}(\frac{m}{e})^m\sqrt{2\pi m}(\frac{m}{e})^m}
\end{aligned}$$

$$\leq \quad \frac{1}{m+1}\frac{1}{\sqrt{\pi m}}\frac{2^{2m+\frac{1}{24m}}m^{\frac{1}{24m}}}{e^{\frac{1}{24m}}}.$$

Therefore,

$$\begin{aligned}
\frac{C_x C_{m-x-1}}{C_m} &\geq \frac{\frac{2^{2x}e^{\frac{1}{6x}}}{(x+1)\sqrt{\pi x}(x^{\frac{1}{6x}})}\frac{2^{2(m-x-1)}e^{\frac{1}{6(m-x-1)}}}{(m-x)\sqrt{\pi(m-x-1)}(m-x-1)^{\frac{1}{6(m-x-1)}}}}{\frac{1}{m+1}\frac{1}{\sqrt{\pi m}}\frac{2^{2m+\frac{1}{24m}}m^{\frac{1}{24m}}}{e^{\frac{1}{24m}}}} \\
&\geq \frac{m^{\frac{3}{2}}}{4\sqrt{\pi}x^{\frac{3}{2}}(m-x-1)^{\frac{3}{2}}}\left(\frac{x}{x+1}\right)\left(\frac{m-x-1}{m-x}\right)\left(\frac{m+1}{m}\right)\times \\
&\quad \frac{e^{\frac{1}{6x}}}{x^{\frac{1}{6x}}}\left(\frac{e^{\frac{1}{6(m-x-1)}}}{(m-x-1)^{\frac{1}{6(m-x-1)}}}\right)\left(\frac{e^{\frac{1}{24m}}}{m^{\frac{1}{24m}}}\right)\left(\frac{1}{2^{\frac{1}{24m}}}\right) \\
&\geq \frac{m^{\frac{3}{2}}}{4\sqrt{\pi}x^{\frac{3}{2}}(m-x-1)^{\frac{3}{2}}}\left(\frac{1}{2}\right).
\end{aligned}$$

□

Lemma 2.5.5 $\frac{p-3}{p}\cdot\frac{n}{p} \leq E(M_s(p)) < \frac{n}{p}$.

Proof: By Lemma 2.5.1, we have

$$\begin{aligned}
E(M_s(p)) &= \textstyle\sum_{k=\frac{n}{p}(p-1)+1}^{n}\left(P_r[a_k = r] - \sum_{t=0}^{\lfloor\frac{k-\frac{n}{p}(p-1)}{2}\rfloor-1} C_t C_{m-1-t}/C_m \right) \\
&= (C_0C_{m-1}+C_1C_{m-2}+\cdots+C_{\lfloor\frac{\frac{n}{p}(p-1)+1}{2}\rfloor-1}C_{m-\lfloor\frac{\frac{n}{p}(p-1)+1}{2}\rfloor})/C_m + \\
&\quad (C_1C_{m-2}+C_2C_{m-3}+\cdots+C_{\lfloor\frac{\frac{n}{p}(p-1)+2}{2}\rfloor-1}C_{m-\lfloor\frac{\frac{n}{p}(p-1)+2}{2}\rfloor})/C_m + \\
&\quad \cdots + \\
&\quad (C_{\lfloor\frac{n}{p}\rfloor-1}C_{m-\lfloor\frac{n}{p}\rfloor}+C_{\frac{n}{p}}C_{m-\frac{n}{p}-1}+\cdots+C_{m-1}C_0)/C_m \\
&\geq \frac{n}{p}\left[\ (C_{\lfloor\frac{n}{p}\rfloor-1}C_{m-\lfloor\frac{n}{p}\rfloor}+\cdots+C_{\lfloor\frac{\frac{n}{p}(p-1)+1}{2}\rfloor-1}C_{m-\lfloor\frac{\frac{n}{p}(p-1)+1}{2}\rfloor})/C_m\ \right].
\end{aligned}$$

(The last inequality is true because there are n/p lines on the left hand side and each line contains the common expression in the bracket.)

Fix m, and let functions $F(x) = \frac{C_x C_{m-x-1}}{C_m}$ and $G(x) = \frac{m^{\frac{3}{2}}}{8\sqrt{\pi}(m-x-1)^{\frac{3}{2}} x^{\frac{3}{2}}}$. Lemma 2.5.4 supports the sketches of F and G as shown in Figure 2.2. Based on Figure 2.2 and the fact that $C_m = \sum_{k=0}^{m-1} C_k C_{m-1-k}$, we conclude the following:

$$
\begin{aligned}
E(M_s(p)) &\geq \frac{n}{p} [\ (C_{\lfloor \frac{n}{p} \rfloor -1} C_{m-\lfloor \frac{\frac{n}{p}}{2} \rfloor} + \cdots + C_{\lfloor \frac{\frac{n}{p}(p-1)+1}{2} \rfloor -1} C_{m-\lfloor \frac{\frac{n}{p}(p-1)+1}{2} \rfloor})/C_m\ ] \\
&\geq \frac{n}{p}(\frac{1}{m}(\lfloor \frac{\frac{n}{p}(p-1)+1}{2} \rfloor - \lfloor \frac{n}{p} \rfloor + 1)) \\
&\geq \frac{n}{p}\ (\ \frac{p-3}{p}\ ).
\end{aligned}
$$

(The second inequality results from the fact that the terms in the right hand side of the first inequality are the larger $(\lfloor \frac{\frac{n}{p}(p-1)+1}{2} \rfloor - \lfloor \frac{n}{p} \rfloor + 1)$ terms of $\sum_{k=0}^{m-1} \frac{C_k C_{m-1-k}}{C_m} = 1$.)

In addition, we can easily infer $E(M_s(p)) < \frac{n}{p}$, since $E(M_s(p)) = \sum_{k=\frac{n}{p}(i-1)+1}^{\frac{n}{p}i} (\ P_r[a_k = r] - \sum_{t=0}^{\lfloor \frac{k-\frac{n}{p}(i-1)}{2} \rfloor -1} C_t C_{m-1-t}/C_m\ ) < \frac{n}{p}$. This completes the proof. □

The Communication Optimality

We have proved that $E(M_s(p)) \geq \frac{p-3}{p} \cdot \frac{n}{p}$. Since our BSP algorithm for parenthesis matching employs prefix sum operation, which introduces an additional p-relation, and the BSP ANSV algorithm, which introduces at most a p-relation (refer to Lemma 2.2.4), in addition to the messages required to be sent/received, we conclude that the average-case message volume of our BSP parenthesis matching algorithm is bounded above by $(E(M_s(p)) + \theta(p))$, which is asymptotically communication-optimal, provided that

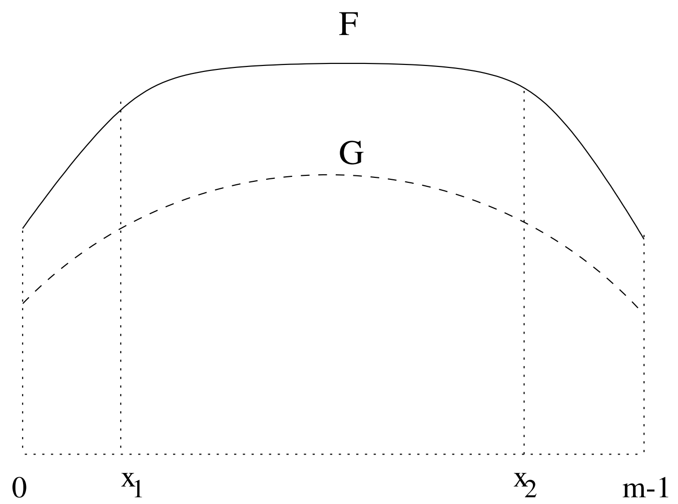

Figure 2.2: The sketches of F and G, where $x_1 = \lfloor \frac{n}{p} \rfloor - 1$ and $x_2 = \lfloor \frac{\frac{n}{p}(p-1)+1}{2} \rfloor - 1$

$\frac{n}{p} = \Omega(p)$.

2.6 Experimental Results

To demonstrate the practical relevance, we have implemented Algorithm $ANSV_r$ on a Sun Enterprise 4000 server supporting 8 UltraSPARC processors and on an SGI Origin 2000 with 32 R10000 processors. Job submission on the Enterprise is interactive, while, on the SGI Origin 2000, it runs the *Portable Batch System* (PBS) for batch job processing.

The MPI Implementations

Our program is about 350 lines, written in C, using the MPI (Message Passing Interface) library for interprocessor communication. MPI is a standard specification for message passing libraries. MPT (on SGI Origin 2000) and MPICH (on Sun Enterprise 4000) are both portable implementations of the full MPI specification for a wide variety of parallel and distributed environments, including parallel computers, clusters of workstations, integrated distributed environments (computational grids) and shared-memory symmetric multiprocessors. Yet both MPT and MPICH provide unified system calls for communication between processes via the various communication media. Programming with MPI libraries makes it possible for developing portable and efficient parallel programs. In our experiment, the codes on SGI Origin 2000 and Sun Enterprise 4000 are exactly the same. Since MPI interface provides wide support for most commercially available parallel machines, and current systems that conform to the BSP computer model include networks of workstations, distributed memory processor arrays, and shared memory multiprocessors, Algorithm $ANSV_r$ can easily be ported to any commercially available machines.

MPT, MPICH and MPICH-GM (used on CCR's Sun Myrinet Cluster) are all compatible with MPICH-G2, another portable implementation of the full MPI specification, supported by Globus [35]. (The Globus project [35] is developing fundamental technologies needed to build computational grids.) To facilitate future expansion, our code is thus based on the MPI. Another portable communication system, BSPLib [51], has been popularly used for BSP programming model [90]. Algorithm $ANSV_r$ can be implemented with BSPLib in a similar manner.

Message Packing and Communication Postponing

The advantage of postponing communication until the end of local computation creates major performance-enhancement opportunities [51]. Combining all of the messages between each processor pair means that transmission startup costs are paid only once per superstep, instead of once per message (although it does require more memory for buffering). The freedom to reorder transmissions to different processors means that patterns guaranteed to avoid congestion can be set up in software, rather than requiring expensive hardware solutions that work during the data transfers.

In our experiment, the MPI-based BSP program follows the BSP programming style, that is, communication in each superstep starts only after the local computation finishes.

The MPI Primitives

We use MPI_Bcast for broadcasting. Some collective communication routines, such as MPI_Allreduce, MPI_Allgather, are also used for global communication. MPI_Scatter and MPI_Gather are used to distribute input data and collect results. Due to the property of Algorithm $ANSV_r$ that data items routed in each superstep are always in contiguous locations, only MPI_Send and MPI_Recv are used for message passing. MPI provides MPI_Pack and MPI_Unpack for packing elements, but they are not used in our implementation. Timings were done by invoking MPI_WTime routine, which is intended to be a high-resolution, elapsed (or wall) clock.

Throughout Sections 2.6.1 and 2.6.2, we use $ANSV(n, p)$ to denote the parallel running time of Algorithm $ANSV_r$, where n and p stand for the input size and the number of processors, respectively.

2.6.1 Running Times on SGI Origin 2000

We investigated the algorithm's behavior on UB CCR's SGI Origin 2000 machine, using up to 32 processors. The experiment was done on input sizes from 0.1 to 8 millions. Each data point presented was obtained as the average of 5 test runs, each on a different randomly generated array.

Figures 2.3 and 2.4 depict the parallel running times when input sizes range from 0.1 to 8 millions.

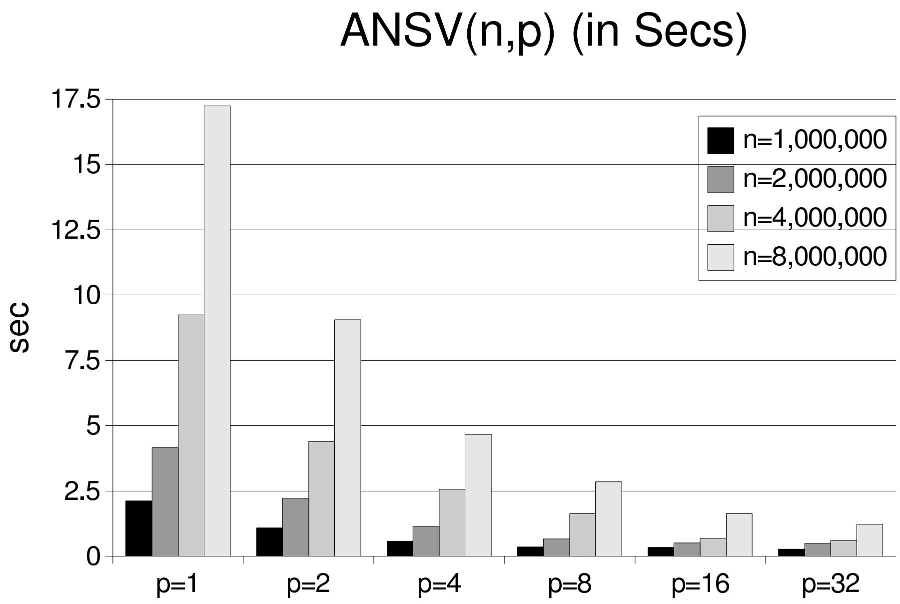

Figure 2.3: SGI Origin 2000 running times of Algorithm $ANSV_r$ (1)

The speed-ups achieved when the input size is above 1 million are almost linear as shown in Fig 2.3. When input size is between 0.1 and 0.8 millions, synchronization overhead starts to take up a larger portion of the total running time. As the number of processors increases to 32 and the input size is 0.1 million, almost 37.5% (0.036 secs/0.096 secs) of the parallel running time is contributed by barrier synchronization.

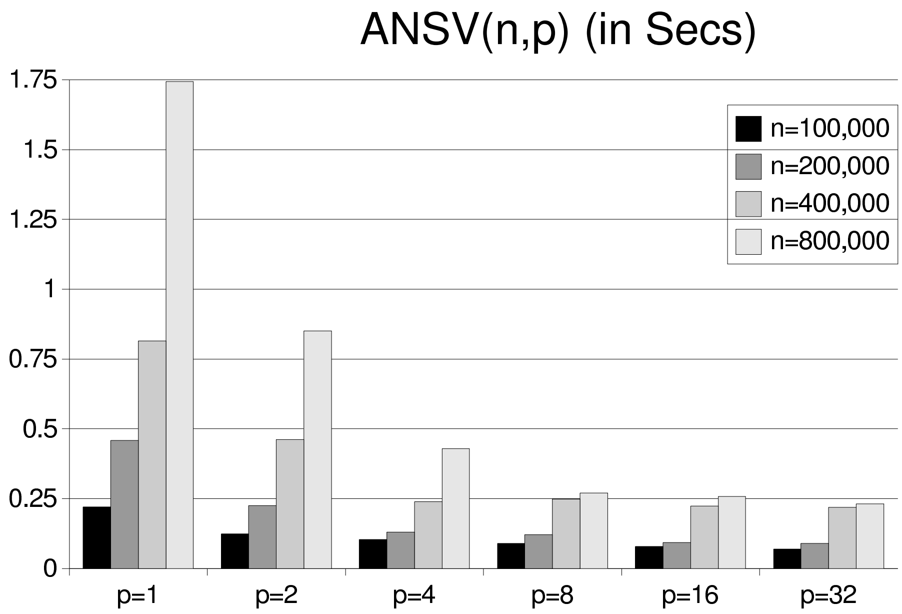

Figure 2.4: SGI Origin 2000 running times of Algorithm $ANSV_r$ (2)

This is partly because of the high cost of barrier synchronization [52], and partly because of the extremely small, actually < 2, hidden constant of the linear-time sequential stack algorithm. When 16 processors are used on 0.1 million data items, barrier synchronization uses about 27.2% (0.018 secs/0.066 secs) of the total execution time. However, when only 4 processors are used, this ratio significantly reduces to about 4.6% (0.00468 secs/0.09988 secs).

2.6.2 Running Times on Sun Enterprise 4000

The timings on Sun Enterprise 4000 multiprocessing server were made under time sharing. However, all experiments were carried out when system CPU idle time was at least 80%. Figure 2.5 shows the average running times for input sizes of 0.84 and 4.2 millions on up to 7 processors. When two processors are used, the speed-up is about 1.6, approximately 80% of the optimal speed-up. When the number of processors scales to 7 (during our experiment, the 8th processor had been constantly occupied by some CPU-intensive jobs, thus was not included in our experiment), the speed-up is about 4, which is about 55% of the optimal speed-up. (Here the p-processor speed-up is measured based on ANSV$(n,1)$/ANSV(n,p).)

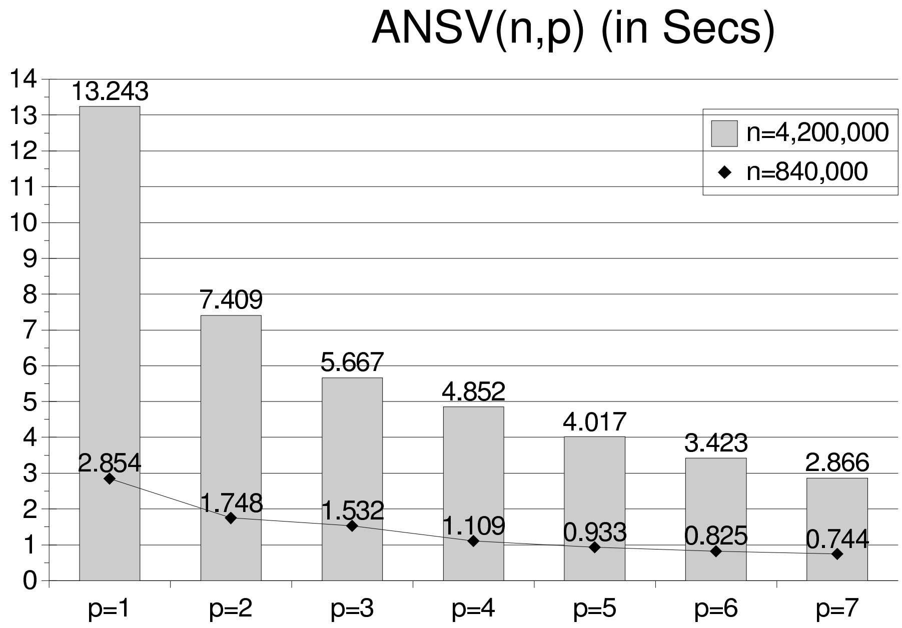

Figure 2.5: Sun Enterprise 4000 running times of Algorithm $ANSV_r$

2.7 The MPI Blocking Communication Primitives

Most communication libraries support *nonblocking* communication. A nonblocking send start call initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication, *i.e.* to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

The *direct remote-memory access* (DRMA) and *bulk synchronous remote-memory access* (BSMP) communication modes of BSPLib are both *nonblocking*. MPI also provides nonblocking communication primitives for interprocessor communication. In fact, we could use some carefully chosen combination of the various nonblocking communication primitives available in MPI to produce an MPI-based BSP programming model. However, the nonblocking primitives do not put a clear delimiter between communication and synchronization. In order to correctly measure the synchronization overhead separately, we use the blocking MPI_Send/Recv primitives in the implementation.

In the blocking mode, the send call does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. On the other hand, message buffering can be expensive, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering. MPI offers the choice of several communication modes that allow one to control the choice of the communication protocol.

The send call used in our implementation uses the *standard* communication mode. In this mode, it is up to the MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Thus, a send in standard mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. The standard mode send is non-local: successful completion of the send operation may depend on the occurrence of a matching receive.

There are three additional communication modes and three additional send functions are provided for the these modes. The communication mode is indicated by a one-letter prefix: *B* for *buffered*, *S* for *synchronous*, and *R* for *ready*.

Buffered Mode (MPI_Bsend/recv)

A buffered mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is *local*, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is insufficient buffer space. The amount of available buffer space is controlled by the user.

Synchronous Mode (MPI_Ssend/recv)

A send that uses the synchronous mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is *non-local*.

Ready Mode (MPI_Rsend/recv)

A send that uses the ready communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Nonblocking send start calls can use the same four modes as blocking sends: standard, buffered, synchronous and ready. These carry the same meaning. Sends of all modes, ready excepted, can be started whether a matching receive has been posted or not; a nonblocking ready send can be started only if a matching receive is posted. In all cases, the send start call is local: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode. The same naming conventions as for blocking communication are used: a prefix of *B*, *S*, or *R* is used for buffered, synchronous or ready mode. In addition, a prefix of *I* (for *immediate*) indicates that the call is nonblocking.

Synchronous Mode (MPI_ISsend/recv)

If the send mode is synchronous, then the send can complete only if a matching receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the send-complete call is non-local. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender "knows" the transfer will complete, but before the receiver "knows" the transfer will complete.)

Buffered Mode (MPI_IBsend/recv)

If the send mode is buffered then the message must be buffered if there is no pending receive. In this case, the send-complete call is local, and must succeed irrespective of the status of a matching receive.

Buffered Mode (MPI_Isend/recv)

If the send mode is standard then the send-complete call may return before a matching receive occurred, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive occurred, and the message was copied into the receive buffer.

2.8 Concluding Remarks

Long communication latency and the overhead of synchronization tend to be the two critical factors that greatly affect the speed-up of a parallel algorithm on present multiprocessor architectures. The $ANSV_r$ algorithm is intended to minimize both by reducing the number of supersteps and the total messages in each superstep. The parallel algorithm design is based on BSP programming model, which conforms to most of the commercially available parallel machines. BSP model provides a practical platform for coarse-grained parallel algorithm design and analysis, which gives satisfactory performance prediction. We conclude that, while taking the optimal local computation time, the communication time required by our $ANSV_r$ algorithm is independent of the input size, and grows only linearly with the number of processors. Since ANSVP serves as a kernel routine for a large number of applications in computational geometry and graph theory as will be described in more details in the next chapter, we conclude that communication-efficient practical parallel algorithms for these problems are indeed achievable. Note that these applications are generally considered to be communication-intensive, and no prior attempt has been made to design practical parallel algorithms for them. Our conclusion also implies that, by identifying better internal structures, we should be able to reduce the communication requirement for those irregularly structured problems.

Experiments on the SGI Origin 2000 and Sun Enterprise 4000 demonstrate good speed-ups. We use two portable MPI implementations, MPT and MPICH, to implement the parallel code. In order to further separate the communication time from synchronization time, we use the blocking send/receive MPI primitives for more precise time measuring. Unlike the two nonblocking communication modes, DRMA and BSMP, provided in BSPLib, the various MPI blocking primitives enable the separation of synchronization overhead from the communication time in each superstep on different parallel computing machines.

The highly compatible MPI implementations make the portability of BSP-based parallel programs relatively easy. The Globus project for computational grids also supports one MPI implementation, MPICH-G2, which is compatible with most of the current MPI implementations. Our BSP algorithm can easily be implemented on any commercially available parallel machines with predictable parallel performance.

Chapter 3

Applications of ANSVP

The ANSVP is fundamental in that the following applications have been shown to be reducible to it: (1) *monotone polygon triangulation*, (2) *binary tree reconstruction*, (3) *string matching*, (4) *finding range minimum* and (5) *computing all nearest neighbors in convex polygons* [13, 22]. Triangulating monotone polygons serves as the main subroutine for simple polygon triangulation, which has wide applications in computational geometry and computer graphics [3]. Range minimum search has been used to solve the *scaled string matching* [7] and *parallel triconnectivity* [82] problems. Many important computational problems can be formulated as a string matching problem. Specific application areas include text editing, computer vision, speech recognition, and molecular biology [57]. Finding all nearest neighbors in convex polygons and binary tree reconstruction are considered basic problems in computational geometry and graph theory, respectively, and has a number of applications [79, 19].

In addition, parallel ANSV algorithm plays an important role in solving the following problems: *forest matching* [62], and *computing the convex hull of a sorted point set* [13].

The following sections define some of the above applications and describe how ANSV algorithm can be applied.

3.1 Monotone Polygon Triangulation

A polygon P is considered *monotone* with respect to a line L if it can be split into two polygonal chains A and B, both of which are monotone with respect to L, meaning that all straight lines orthogonal to L intersect with A and B only at a single point. We are asked to triangulate such polygons. Figure 3.1 shows a triangulated monotone polygon.

Without loss of generality, a polygonal chain A is defined to be monotone if the vertices are in increasing order by the x-coordinate. A monotone polygon thus is composed of two such monotone polygonal chains: the *upper* and *lower* chains. Also without loss of generality, the upper chain is assumed to go from the vertex with minimum x-coordinate to the vertex with maximum x-coordinate. A *one-sided monotone polygon* (OSMP) is a monotone polygon whose upper (or lower) chain is a straight line, which is called the *distinguished edge*.

Monotone polygon triangulation consists of two phases: decomposition into one-sided monotone polygons [3, 46], and triangulating the one-sided monotone polygons [40, 36]. Without loss of generality, all vertices are assumed to have distinct x-coordinates.

Monotone Polygon Triangulation :

1. Decomposition into OSMPs:

(a) Merge the vertices of the lower and upper chains according to their x-coordinates.

(b) For each edge (v_i, v_{i+1}) of the lower chain, the segments of the upper chain consisting of those points whose x-coordinates lie between v_i and v_{i+1}, together with the edge (v_i, v_{i+1}) itself (as the distinguished edge) form an OSMP. Given the merged array in 1(a), the OSMP of an edge (v_i, v_{i+1}) of the lower chain is described by the sub-array extending from v_i to v_{i+1}, if not empty.

(c) OSMPs with respect to edges of the upper chain can similarly be constructed.

2. Triangulating OSMPs:

Let $Q = (u_0, u_1, \cdots, u_{m-1})$ be an OSMP. We assume, without loss of generality, all vertices are above or on the distinguished edge (u_0, u_{m-1}) and that $x(u_0) < x(u_{m-1})$ and $y(u_0) \leq y(u_{m-1})$.

(a) Rotate Q so that the distinguished edge is parallel to the x axis without moving vertex u_0. (The rotation is done in an angle with absolute value smaller than $\pi/2$).

(b) For each i, where $0 \leq i \leq m-1$, add the following edges:

- An edge between u_i and u_j, where $j < i$ is the largest index for which u_j is not *higher* than u_i.
- An edge between u_i and u_k, where $k > i$ is the smallest index for which u_k is lower than u_i.

Steps 1 (a) and 2 (b) can be implemented with the ANSV algorithms as follows:

- For step 1 (a), the input array of the ANSV algorithm is the concatenated array $Q\bar{Q}'$, where $Q = (u_0, u_1, \cdots, u_{m-1})$ and $Q' = (u_0', u_1', \cdots, u_{n-1}')$ are the upper and lower polygonal chains, and $\bar{Q}'$ is the array of Q' in reversed order.

  If u_y' is the right match of u_x, the location of u_x in the merged list $Q\bar{Q}'$ is $x+y$. The locations of u_x''s can be found similarly.

- For step 2 (b), the input array for the ANSV algorithm is an array of pairs of numbers, $((y(u_0), 0), (y(u_1), 1), \cdots, (y(u_{m-1}), m-1))$, and the comparisons during the algorithms are done lexicographically.

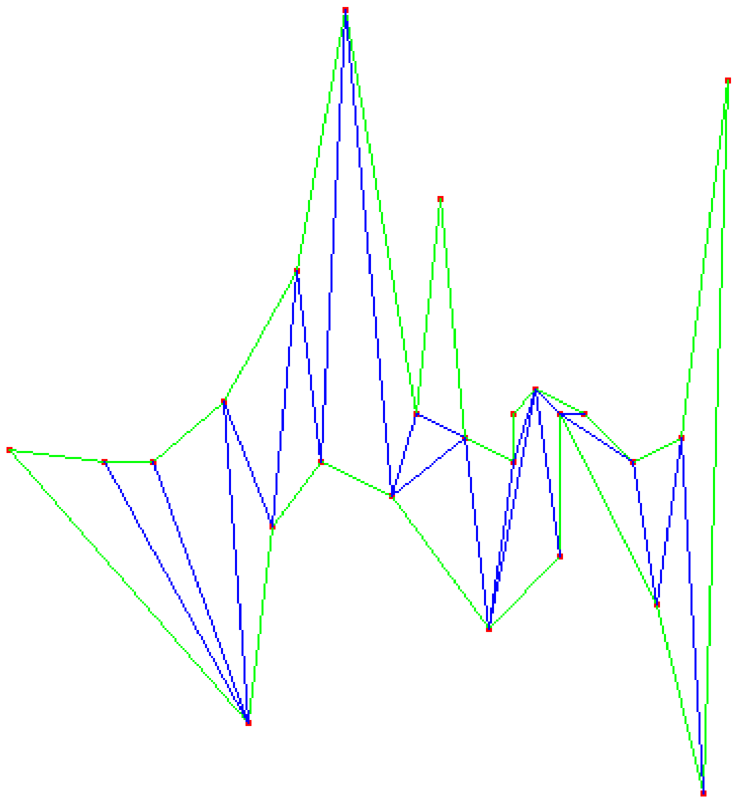

Figure 3.1: A triangulated monotone polygon

3.2 Binary Tree Reconstruction

Given the inorder and preorder (postorder) traversals of a binary tree $T(V,E)$, where $V=\{1,\cdots,n\}$, we are asked to reconstruct the binary tree from these traversals. In other words, the problem is to determine each node's parent. The input to this problem is the traversal information, described as two arrays:

1. $\mathcal{P}$, containing the numbering of the nodes in a preorder traversal;
2. I, containing the numbering of the nodes in an inorder traversal.

The output is the binary tree with each node pointing to its left and right child, if such exist.

We now define arrays $\mathcal{P}^{-1}$, I^{-1} and $\bar{\mathcal{P}}$ as follows:

- $\mathcal{P}^{-1}(i) = j$, where $\mathcal{P}(j) = i$.
- $I^{-1}(i) = j$, where $I(j) = i$.
- $\bar{\mathcal{P}}(i) = \mathcal{P}(I^{-1}(i))$.

Lemma 3.2.1 *[13] Let $1 \le i \le n$ and $j = \mathcal{P}^{-1}(P(i)+1)$. If $I(j) < I(i)$, j is the left child of i. Otherwise, i does not have a left child.*

Lemma 3.2.2 *[13] Let j $(1 \le j \le n)$ be the inorder number of node v. Assume $\bar{\mathcal{P}}(j) \neq 1$ (i.e. v is not the root) and v is not a left child of any node in T (so v must be a right child of some node in T). Let $l(j)$ denote the index of the left match of x_j in $\bar{\mathcal{P}}$. Then, v is the right match of $I^{-1}(l(j))$.*

The above lemmas clearly show that ANSVP serves as the kernel routine for reconstructing the original binary tree.

3.3 String Matching

The string matching problem is to find all occurrences of a given m-character pattern, *PATTERN* $[1 \cdots m]$, in a given n-character text, *TEXT* $[1 \cdots n]$. Many important computational problems can be formulated as a string matching problem. Specific application areas where such problems arise include text editing, computer vision, speech recognition, and molecular biology.

Most of the algorithmic techniques for exact string matching seem to revolve around the *periodic properties* of strings [22, 96]. We introduce some definitions first. The *prefix* of a string u is a substring of u starting at the beginning of u. The concatenation of two strings u and v are denoted as uv. A string u is called a *period* of a string w if w is a prefix of u^k for some positive integer k or equivalently if w is a prefix if uw. The shortest period of w is called the *period* of w. We say w is *periodic* if w is at least twice its period length. Some properties of string periodicity are summarized as follows [57]:

(1) If w has two periods of lengths p and q and $|w| \geq p+q$, then w has a period of length $\gcd(p,q)$.

(2) If w occurs in positions p and q of some string and $0 < q-p < |w|$, then w has a period of length $q-p$. Therefore we cannot have two occurrences of the pattern at positions p and q if $0 < q-p < |u|$ and u is the period of the pattern.

A method proposed in [96] efficiently eliminates many possible occurrences by computing array *WITNESS* $(1 \cdots m)$ defined below and using this information for text analysis.

Let u be the period of the pattern w and v be a prefix of w. It follows immediately from the periodicity property that if $|u|$ does not divide $|v|$ and $|v| < \max(|u|, |w| - |u|)$, then v is not a period, and hence w is not a prefix of vw. In that case, we can find an index k such that *PATTERN* $[k] \neq$ *PATTERN* $[k - |v|]$. We call this k *a witness to the mismatch of w and vw*, and define *WITNESS* $(|v| + 1) = k$. We are interested only in *WITNESS* (i) for $1 < i \leq |u|$, which by the periodicity properties mentioned above can be based only on the first $2|u| - 1$ characters of the pattern. See Figure 3.2 for determining the locations of the *WITNESS* array. (If some *WITNESS* (i) is greater than $2|u|$, it can be modified to be in the desired range: Let $r =$ *WITNESS* $(i) \bmod |u|$; then if $r < i$ set *WITNESS* (i) to $r + |u|$; otherwise we set *WITNESS* (i) to r.)

Having computed the *WITNESS* array, the following method was further suggested in [96] to eliminate close possible occurrences. Suppose we suspect that the pattern may start at positions i and j of the text where $0 < j - i < |u|$, thus, since $r =$*WITNESS* $(j - i + 1)$ has been computed, we can find a character in the text in constant time that will eliminate at least one of the possible occurrences. More specifically, since *PATTERN* $[r] \neq$ *PATTERN* $[r - j + i]$, at most one of them can be equal to *TEXT* $[r + i - 1]$. (Refer to Figure 3.3.)

For the case when the pattern w is not periodic, the pattern can not occur at both positions i and j of the text if $|j - i| < m/2$. This implies that the pattern can occur in

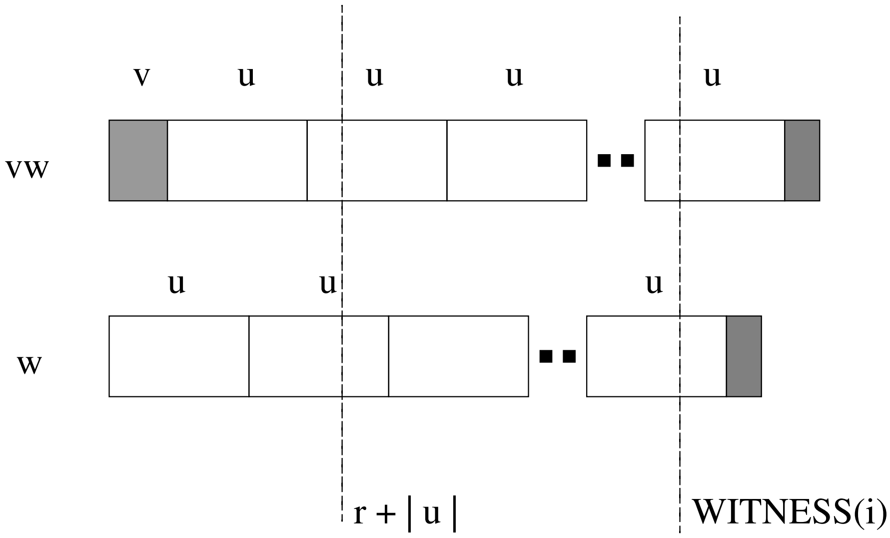

Figure 3.2: Determining witness locations

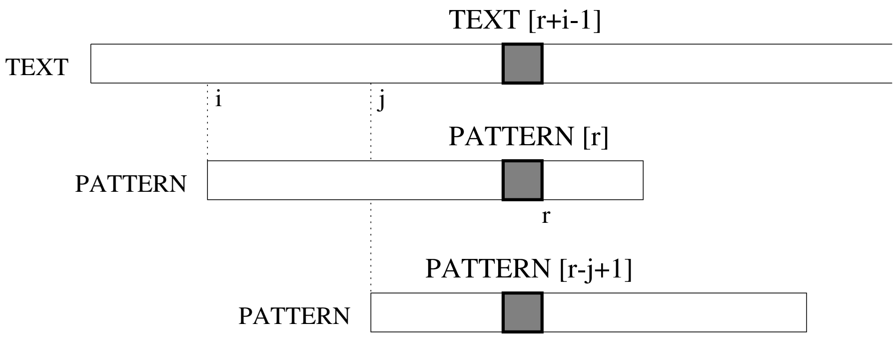

Figure 3.3: Eliminating close possible occurrences

the text at most $2n/m$ times. Namely, if the text is evenly divided into blocks of size $m/2$, only one occurrence of the pattern might start in each block. The *WITNESS* array now of size $m/2$ comes in handy eliminating posisble occurrences.

The cases with periodic patterns extend from the above approach. Now the pattern w is of the form $u^k v$ where $k > 1$ and v is the proper prefix of u. Consider the prefix *PATTERN* $[1 \cdots 2|u| - 1]$ as a new pattern, which is not periodic, whose occurrences in *TEXT* are to be found using the abovementioned procedure. The occurrences of u^2 hence are easily determined. We call an occurrence of u^2 at position i a *final occurrence* if there is no occurrence of u^2 at position $i+|u|$. For an occurrence of u^2, define its *right match* to be the nearest final occurrence to its right. If u^2 occurs at position i, its right match must be at position $i+l|u|$ for some integer $l \geq 0$. This implies that the number of consecutive occurrences of u starting at position i is $l+2$. And here is where the ANSV algorithm can be applied. For each final occurrence we need to verify whether v occurs after it. Note that v occurs after each non-final occurrence since v is a prefix of u. This information also helps decide for each occurrence of u^2 whether it is the beginning of an occurrence of the pattern.

3.4 Range Minimum Search

The *range minimum search* problem is defined as follows: Preprocess an array of real numbers $A = (a_1, \cdots, a_n)$ so that given $i, j,\ 1 \leq i \leq j \leq n$, it takes constant time to find the minimum element, MIN(i, j), in the subarray $A_{i,j} = (a_i, \cdots, a_j)$.

The preprocessing algorithm is based on the sequential algorithm in [38], which uses the *Cartesian tree* data structure.

Definition 3.4.1 *[98] The Cartesian tree for an array $A = (a_1, \cdots, a_n)$ of n distinct real numbers is a binary tree with vertices labeled by the numbers. The root has label a_m, where $a_m = min\{a_1, \cdots, a_n\}$. Its left subtree is a Cartesian tree for $A_{1,m-1} = (a_1, \cdots, a_{m-1})$ and its right subtree is a Cartesian tree for $A_{m+1,n} = (a_{m+1}, \cdots, a_n)$. (The tree for an empty sub-array is the empty tree.)*

The preprocessing procedure starts with the construction of the Cartesian tree for A, and then proceeds with answering queries for *lowest common ancestors* [89, 14] in trees. The definition of the Cartesian tree implies that MIN(i, j) is the lowest common ancestor of a_i and a_j in the Cartesian tree. Therefore, each range minimum query can be answered in constant time by answering the corresponding query for lowest common ancestor in the Cartesian tree.

Let a_i be an element in A. Recall that the left (right, respectively) match of a_i is the nearest element to its left (right, respectively) with a smaller value, if such exists. The following lemma shows how ANSV algorithm is useful in constructing the Cartesian tree for A.

Lemma 3.4.1 *[13] The parent of a vertex a_i in the Cartesian tree for A is the larger of its left and right matches, if such exist. The vertex a_i is a right child if its parent is its left match, and a left child otherwise.*

Besides Cartesian tree, a complete binary tree can also be used for range minimum search [57]. The preprocessing part now consists of constructing a complete binary tree T with n leaves, and associating the elements of the input sequence with the leaves of T in order. For each vertex v of T having leaves $L_v = (a_{i \cdot 2^k+1}, \cdots, a_{(i+1)2^k})$ in its rooted subtree, compute the prefix minima and suffix minima with respect to L_v. Namely, compute $P_v(q) = min\{a_{i \cdot 2^k+1}, \cdots, a_{i \cdot 2^k+q}\}$ and $S_v(q) = min\{a_{(i+1)2^k-q+1}, \cdots, a_{(i+1)2^k}\}$, for $q = 1, \cdots, 2^k$.

The query MIN(i, j) is answered as follows. We first identify w, the lowest common ancestor of a_i and a_j in T. Let v and u be the left and right children of w, respectively.

Let l be the index of the leftmost leaf in the subtree of u. Then, MIN(i, j) = $min\{S_v(l-i), P_u(j-l+1)\}$. The lowest common ancestor of any two given vertices in T can be found using the inorder numbering of T [89].

Range minimum search has been used to solve the *parallel triconnectivity* [82] and *pattern matching with scaling* problems [7]. The first problem deals with determining in parallel the 3-vertex connectivity for graphs and further decomposing graphs into triconnected components. The second problem is an extended version of string matching problem arising from the field of computer vision. The string $aaa\cdots a$ where the symbol a is repeated k times (denoted a^k) is referred to as *scaling of string a by multiplicative factor k* or simply as *a scaled to k*. Similarly, consider a string $A = a_1 \cdots a_l$. *A scaled to k* (A_k) is the string ${a_1}^k \cdots {a_l}^k$. The input to the problem of *one-dimensional string matching with scaling* is the *PATTERN* $[p_1 \cdots p_m]$ and *TEXT* $[t_1 \cdots t_n]$. The output is all the positions in *TEXT* where an occurrence of *PATTERN* scaled to k starts, where $1 \leq k \leq \lfloor \frac{n}{m} \rfloor$. The *two-dimensional string matching with scaling* is defined analogously.

3.5 Finding All Nearest Neighbors for Convex Polygons

A convex polygon is given as $P = (v_0, \cdots, v_{n-1})$, where $(v_i, v_{i+1}), 0 \leq i \leq n-1$, is an edge of P and each vertex v is given by its x- and y- coordinates, denoted $X(v)$ and $Y(v)$. Here, without loss of generality, we assume the vertices are given in counter-clockwise order. The *all nearest neighbors (ANN) problem for convex polygon* is defined as follows : For each vertex v_i of P, find its nearest neighbor. That is, find a vertex $v_j, j \neq i, 0 \leq j < n$, whose (Euclidean) distance from v_i is minimal. The *ANN* problem is considered a basic problem in computational geometry, and has a number of applications [79].

The relationship between the *ANN* and the merging problems was proposed in [88]. The approach consists of two stages: the decomposition stage and the merge stage. In the decomposition stage, the convex polygon P is partitioned into four convex sub-polygons and the *ANN* problem with respect to each of the sub-polygons is solved separately. In the merge stage, the solutions to the four sub-polygons are extended into a solution to the *ANN* problem with respect to P. More details are given below.

Let $P = (v_0, \cdots, v_{n-1})$ be a convex polygon, where (v_i, v_{i+1}) is an edge of P, for $0 \leq i \leq n-1$. Also, for any two vertices v_i and v_j, let $D_E(v_i, v_j)$ denote the Euclidean distance between v_i and v_j. And, for a vertex v_i of P, let $NN(P, v_i)$ denote its nearest neighbor in P. The *semicircle* property of a polygon is defined as follows:

Definition 3.5.1 *The polygon P has the semicircle property if it satisfies the following two conditions:*

(1) The two farthest vertices, say v_i and v_{i+1}, of P are the endpoints of an edge of P.

(2) All the vertices of P lie inside the circle with diameter $D_E(v_i, v_{i+1})$, and with v_i and v_{i+1} on the circle.

The *ANN* problem for convex polygons having the semicircle property can be solved using the following theorem [71]:

Theorem 3.5.1 *If a convex polygon P has the semicircle property, then , for any vertex v_i of P, $NN(P, v_i)$ is either v_{i-1} or v_{i+1} of P.*

Let v_a, v_b, v_c and v_d be the vertices of P with the smallest x-coordinate, smallest y-coordinate, largest x-coordinate, and largest y-coordinate, respectively. Without loss of generality, assume that $a \leq b \leq c \leq d$. Let the convex sub-polygons P_1, P_2, P_3 and P_4 be $(v_a, \cdots, v_b)$, $(v_b, \cdots, v_c)$, $(v_c, \cdots, v_d)$ and $(v_d, \cdots, v_a)$, respectively.

Theorem 3.5.2 *[99] Each convex sub-polygon P_1, P_2, P_3 and P_4 has the semicircle property.*

Figure 3.4 shows a convex polygon P and its decomposition into four convex polygons each having the semicircle property. The *ANN* problem with respect to each of these sub-polygons can be solved by theorem 3.5.1.

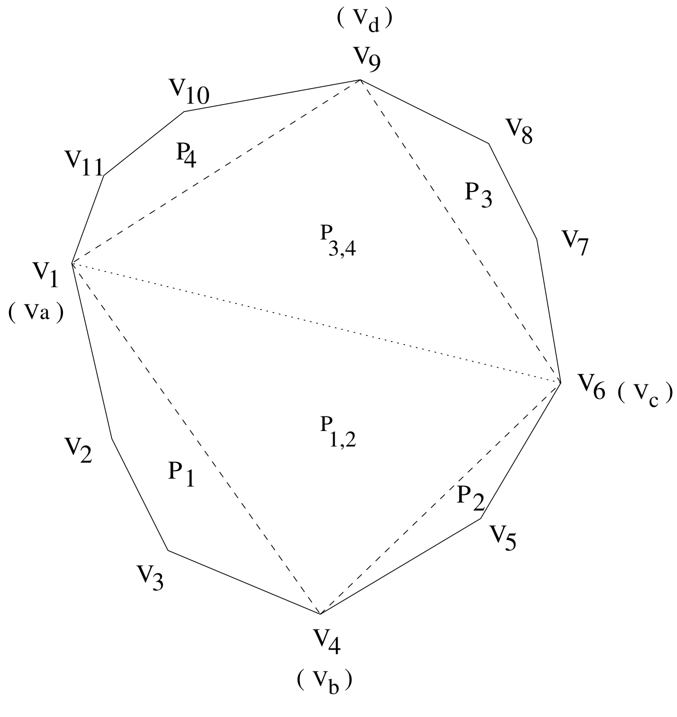

Figure 3.4: Stage (1) – convex polygon decomposition

The next stage combines the solutions to the four *ANN* subproblems into the solution to the *ANN* problem for P. Let $P_{1,2}$ and $P_{3,4}$ be the convex polygons composed by the sub-polygons P_1,P_2 and P_3,P_4. P therefore is composed by $P_{1,2}$ and $P_{3,4}$. The merge stage consists of two steps: (1) Solve the *ANN* problem for $P_{1,2}$ and $P_{3,4}$. (2) Solve the *ANN* problem for P.

The algorithm for the *ANN* problem for $P_{1,2}$ is described first. The algorithm for $P_{3,4}$ can be derived similarly. Recall that $P_1 = (v_a, \cdots, v_b)$, $P_2 = (v_b, \cdots, v_c)$ and $P_{1,2} = (v_a, \cdots, v_b, \cdots, v_c)$. We first describe how to compute $NN(P_{1,2}, v)$ for each $v \neq v_b$ of P_1. $NN(P_{1,2}, v)$, for each $v \neq v_b$ of P_2, is computed similarly. Note that $NN(P_{1,2}, v_b)$ is either $NN(P_1, v_b)$ or $NN(P_2, v_b)$.

For each vertex v of P_1, we define the *circle* of v to be the circle of radius $D_E(v,NN(P_1, v))$ centered at v. Every vertex of P_2 that is closer to v than $NN(P_1, v))$ must lie in the circle of v. Let L be the straight line parallel to the y-axis which goes through v_b. Obviously, L separates P_1 and P_2. That is, all the vertices of P_1 lie on one side of L, while those vertices of P_2 lie on the other. For a vertex v_i, let w_i denote the projection of v_i on L. Note that v_i and w_i have the same y-coordinates, and that w_i and v_b have the same x-coordinates. That is, $X(w_i) = X(v_b)$ and $Y(w_i) = Y(v_b)$. Also note that if a vertex v_i of P_2 lies in the circle of v of P_1, then w_i also lies in this circle. (See Figure 3.5.)

Since the vertices of P are assumed to be given in a counterclockwise cyclic ordering, the vertices of P_1 and P_2 are given in descending and ascending, respectively, orders of their y-coordinates. We first reverse the order of the vertices of P_1 and then merge the two sequences of vertices. The resulting merged list is denoted by S.

For every v_j, $b < j \leq c$, of P_2, the following theorem characterizes the vertices of P_1 such that v_j is contained in their circles [71, 88].

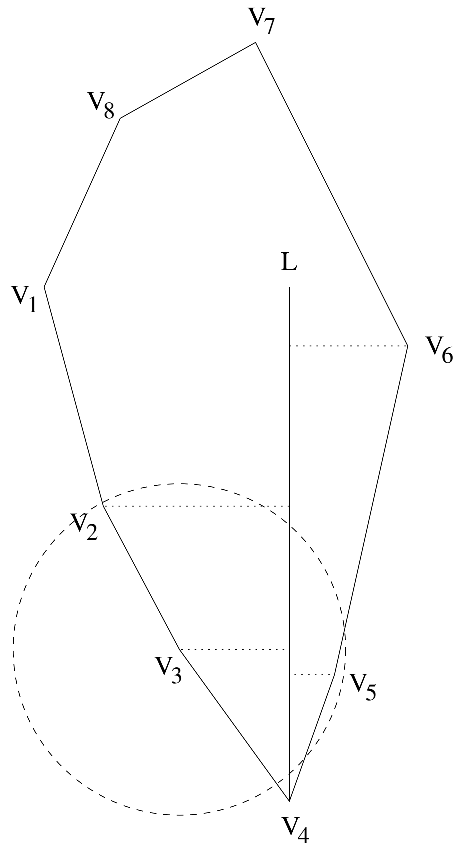

Figure 3.5: Graphical illustration of the properties of "circles"

Theorem 3.5.3 *For every vertex v_j, $b < j \leq c$, of P_2, w_j is contained in the circles of at most two vertices of P_1, and these two vertices of P_1 are those that are adjacent to v_j in S. That is, the last vertex of P_1 which precedes v_j in S (denoted PRED(v_j)) and the first vertex of P_1 which succeeds v_j in S (denoted SUCC(v_j)).*

The algorithm for the *ANN* problem for $P_{1,2}$ is now summarized as follows:

1. Merge the vertices of P_2 and vertices (in the reversed order) of P_1.

2. For each vertex $v \neq v_b$ of P_2, compute $PRED(v)$ and $SUCC(v)$.

3. For each vertex $v \neq v_b$ of P_1, compute the minimum of $D_E(v,u)$, among all ver-

tices u of P_2 such that v =*PRED*(u). Similarly, for each vertex v of P_1, compute the minimum of $D_E(v,u)$, among all vertices u of P_2 such that v =*SUCC*(u). Compare these two minima to D_E(v, *NN*(P_1, v)) and assign a value to *NN*($P_{1,2}, v$) accordingly.

The solutions to the *ANN* problems for $P_{1,2} = (v_a, \cdots, v_c)$ and $P_{3,4} = (v_c, \cdots, v_a)$ are used to compute *NN*(P, v), for each vertex v of P, in a similar manner. And, the computation of *NN*(P, v) for each vertex of $P_{3,4}$ is similar to computation of *NN*(P, v) for each vertex of $P_{1,2}$, which is described below.

Here we need a straight line L', similar to the line L used in solving *ANN* for $P_{1,2}$, which separates $P_{1,2}$ and $P_{3,4}$. The only such straight line is the line segment L'' going through v_a and v_c. However, L'' may not necessarily be parallel to either x- or y- axis. We need to rotate L'' so that it is parallel to the x-axis. (See Figure 3.6.) Let the resulting straight line after rotation be L', and L' will be used to separate $P_{1,2}$ and $P_{3,4}$.

The rotation may turn out destroying the order of some vertices. To reuse the approach used in solving *ANN* problems for $P_{1,2}$ and $P_{3,4}$, some additional work is needed. Without loss of generality, assume that $Y(v_a) > Y(v_c)$ before the rotation. Also, let v_e and v_f denote the vertices with smallest and largest, respectively, x-coordinates of P after rotation. Now that, before the rotation, $Y(v_a) > Y(v_c)$, it's obvious that v_e belongs to $P_{3,4}$ and v_f belongs to $P_{1,2}$. Note that the angle of v_a-v_c-v_i is greater than 90°, for $f \leq i < c$. Therefore, *NN*(P, v_i) = *NN*($P_{1,2}, v_i$), for $f \leq i < c$, implying that only vertices $v_a, \cdots, v_f$ need be taken care of. Note also that the angle of v_c-v_a-v_j is greater than 90°, for $e \leq j < a$. Therefore, $D_E(v_i, v_j) > D_E(v_i$,*NN*$(P_{1,2}, v_i))$, for $a < i \leq f$ and $e \leq j < a$.

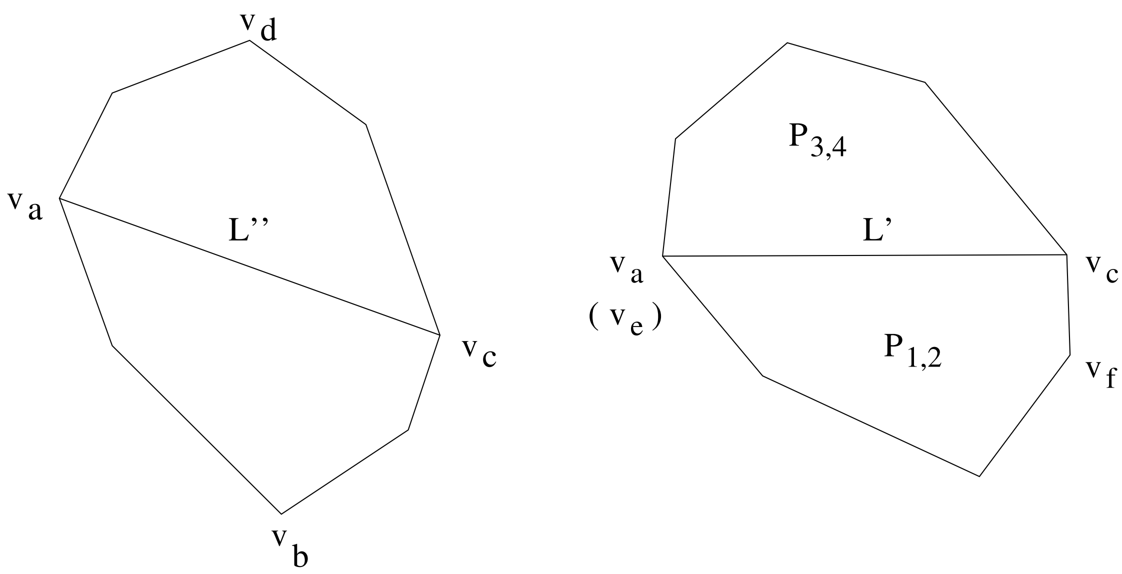

Figure 3.6: The rotation of L'' to derive the horizontal L'

That is, when computing the minimum distance for vertices $v_a, \cdots, v_f$, the only candidate vertices in $P_{3,4}$ are $v_c, \cdots, v_e$. Since all the vertices of $v_a, \cdots, v_f$ of $P_{1,2}$ are given in the order of their x-coordinates and all the vertices $v_c, \cdots, v_e$ are given in the reversed order, the above-mentioned approach to solve *ANN* problems for $P_{1,2}$ and $P_{3,4}$ can be applied for finding $NN(P, v_i)$, for $a < i \leq f$.

3.6 Concluding Remarks

We have discussed various applications for which the ANSV algorithm either serves as the kernel routine or plays an important role. These applications are generally considered to be communication-intensive, since most of them don't have regular structures for parallelization. A portable and communication-efficient parallel ANSV algorithm therefore is important in the attempt to design practically efficient parallel algorithms for these applications. And these practical parallel algorithms have been made possible based on the design, analysis and implementation of our BSP ANSV algorithm.

Chapter 4

Fully Scalable Interval Graph Algorithms

4.1 Introduction

An *interval graph* is an undirected graph G whose vertices can be put into one-to-one correspondence with a set of intervals $\mathfrak{I}$ of a linearly ordered set such that two vertices are connected by an edge of G if and only if their corresponding intervals intersect [45]. ($\mathfrak{I}$ is called an *interval model* of G.) Due to richness in real world applications from various fields, such as biology, psychology, archaeology, protein sequencing, VLSI design and many scheduling problems, interval graphs have been intensely researched [15, 48, 65, 75, 77, 83, 87, 92, 100, 101]. Many NP-complete problems on general graphs, such as finding maximum clique, maximum independent set and minimum coloring, etc., have efficient sequential algorithms and efficient parallel algorithms on PRAM model.

Throughout this chapter, we will use $G_{\Im}$ to denote an interval graph whose interval model is $\Im$. I_k=[l_k, r_k] ($1 \leq k \leq n$) specifies an interval in $\Im$. Without loss of generality [15, 92, 101], all the endpoints of the intervals in $\Im$ are assumed distinct.

In Section 4.2, we present an algorithm for parallel *prefix sum* on CGM. Our algorithm solves this basic problem in optimal $O(1)$ rounds with full scalability. It is used in later sections and may find applications for solving other problems. Section 4.3 briefly summarizes the sorting algorithms on BSP-like models. The CGM sorting algorithm [47] will be used as a common subroutine in later sections. In Sections 4.4 through 4.7, we present algorithms for finding maximum independent set, maximum weighted clique, minimum coloring, and cut vertices and bridges, respectively. All these problems have important practical applications [45]. Our algorithms in this chapter are for CGM models. Yet, these algorithms can be easily modified to work on Valiant's original EREW BSP model. Section 4.8 concludes this chapter.

4.2 Prefix Sum on CGM

As mentioned in Section 2.5, prefix sum is defined in terms of a binary, associative operator $\otimes$. The computation takes as input a sequence $< a_1, a_2, \cdots, a_n >$ and produces as output a sequence $< b_1, b_2, \cdots, b_n >$ such that $b_1 = a_1$ and $b_k = b_{k-1} \otimes a_k$ for $k = 2, 3, \cdots, n$.

The prefix sum can trivially be solved on the CGM machine using $O(1)$ communication rounds and optimal local computation time with scalability at $\frac{n}{p} \geq p$. Our algorithm further achieves scalability at $\frac{n}{p} \geq p^{\varepsilon}$ for any $\varepsilon > 0$, while preserving both computation and communication optimality.

Meijer and Akl [75] presented a parallel prefix sum algorithm on binary tree machine. We conceptually think of a p-processor CGM model as a complete *g-ary* tree machine, with g^k leaf processors and $\frac{g^{k+1}-1}{g-1}$ processors in total. (Here g is a parameter used for grouping processors which will be determined later.) Each non-leaf processor and its rightmost child are actually the same processor. An example using 16 processors with $g = 4$ is shown in Figure 4.1.

Assume that the initial input is $a_1, a_2, \cdots, a_n$ and the processors are labeled $P_1, P_2, \cdots, P_p$. Processor P_i initially contains $a_{\frac{n}{p}(i-1)+1}, a_{\frac{n}{p}(i-1)+2}, \cdots, a_{\frac{n}{p}i}$. For simplicity, we also assume $n \equiv 0 \bmod p$ and $p = g^k$. Our algorithm can easily be extended for any n and p.

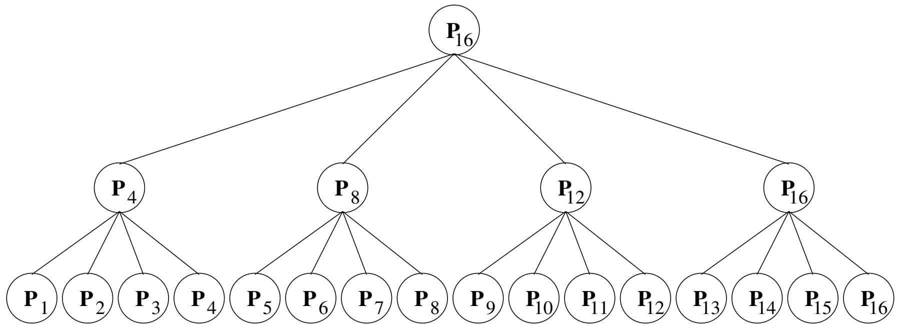

Figure 4.1: A conceptual 4-ary tree with respect to a 16-processor CGM model

First each processor computes all prefix sums of $a_{\frac{n}{p}(i-1)+1}, a_{\frac{n}{p}(i-1)+2}, \cdots, a_{\frac{n}{p}i}$ and stores the results in local memory locations $s_{\frac{n}{p}(i-1)+1}, s_{\frac{n}{p}(i-1)+2}, \cdots, s_{\frac{n}{p}i}$. Then, starting from the leaves, for each tree level, we perform the following steps:

1. Each P_i sends $s_{\frac{n}{p}i}$ to its parent processor.

2. Each parent processor computes prefix sums of the g values received from its children and store these values at $t_1, t_2, \cdots, t_g$.

3. Each parent processor broadcasts t_i to all the leaf processors of the subtree rooted at C_{i+1}, for $1 \leq i \leq g-1$. (Here $C_1, \cdots, C_g$ denote the child processors ordered from left to right.)

4. Each leaf processor adds received value to $s_{\frac{n}{p}(i-1)+1)}, s_{\frac{n}{p}(i-1)+2}, \cdots, s_{\frac{n}{p}i}$.

For general n and p, slight modification for computing parent processor number is needed but the algorithm remains intact. An example with $n = 26$ and $p = 9$ is shown in Figures 4.2, 4.3 and 4.4.

We let $g = \min\{\lceil \frac{n}{p} \rceil, p\}$. The required number of communication rounds is the height of the conceptual tree, with scalability measure $\frac{n}{p} \geq p^{\varepsilon}$, which is $\lceil \log_g p \rceil \leq \log_g p + 1 \leq \frac{\log p}{\varepsilon \log p} + 1 = \frac{1}{\varepsilon} + 1 = O(1)$. In each round, the local computation time is obviously $O(\frac{n}{p})$. This gives:

Theorem 4.2.1 *Parallel prefix sum on n input items can be optimally implemented on a p-processor CGM computer in $O(1)$ communication rounds and optimal local computation time in each round, with scalability at $\frac{n}{p} \geq p^{\varepsilon}$, $\forall \varepsilon > 0$.*

Next we describe two examples using prefix sum operations, which will appear as sub-procedures in some algorithms in the incoming sections.

Array Packing [5]: Given an array A= $\{a_1, a_2, \cdots, a_n\}$, some of whose elements are labeled. We need to pack all labeled elements at the front of the array. The corresponding

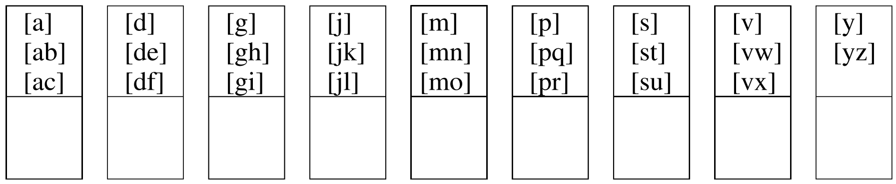

Figure 4.2: $(n = 26, p = 9)$ Initially each processor stores $\lceil \frac{n}{p} \rceil$ data items. This diagram shows the result after the first computation round. ([ac] denotes the prefix sum $a \otimes b \otimes c$, where $\otimes$ stands for any prefix computation operator.)

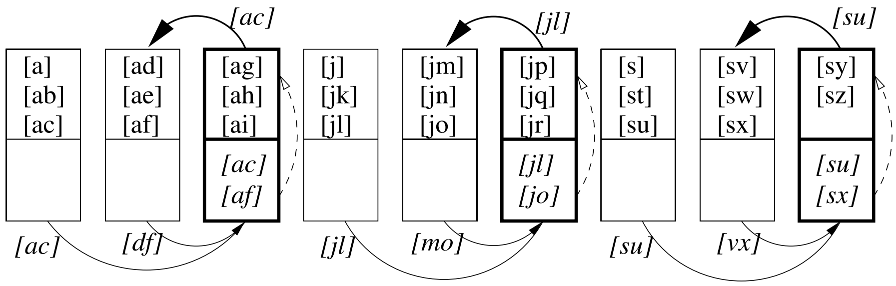

Figure 4.3: In each level, each parent processor collects (with one g-relation), computes, and broadcasts (with one g-relation) values to its child processors.

CGM algorithm is a straightforward application of the prefix sum algorithm.

Interval Broadcasting [5]: Given an array A, some of whose elements are "leaders", broadcast those values associated with the leaders to the subsequent elements, up to, but not including, the next leader. The CGM solution is as below:

1. Each leader in A holds a pair (i, d_i), where i is the leader's index in A and d_i is its value. Each non-leader holds a pair $(-1, \#)$, where # is a dummy value.

2. Perform CGM prefix operation. The prefix operator is defined as:

   $(i, a) \circ (j, b) \rightarrow (i, a)$ if $j < i$;

   $(i, a) \circ (j, b) \rightarrow (j, b)$ otherwise.

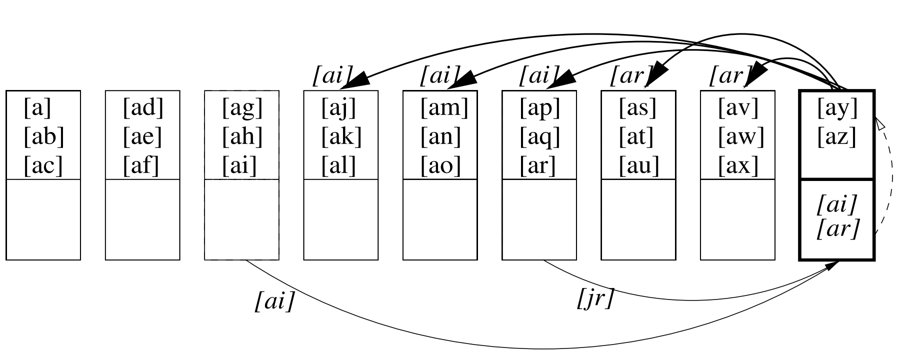

Figure 4.4: In another level, the parent processor also collects, computes and broadcasts values. (Here we are employing the *weak-CREW* [47] property of CGM model.)

4.3 Coarse-Grained Sorting Algorithms

Coarse-grained sorting algorithms have been designed and analyzed on various parallel models. Coarse-Grained Batcher's bitonic sort, radix sort and sample sort were first discussed in [12]. Another coarse-grained sorting algorithm by Li and Sevcik [72] uses a single-step approach, *i.e.* merging of the sorted sublists or partitioning into sublists is done in one step. An over-partitioning scheme is used to reduce load imbalance. Their approach was also applied to quicksort and radix sort.

Deterministic sorting on a variant of the BSP model, XPRAM, was proposed in [1]. The algorithm is based on a version of Column-sort and imposes an upper bound on the sorting algorithm. For all values of n and p such that $p = n^{1-\varepsilon}, 0 < \varepsilon < 1$, this algorithm takes $\lceil e^{-3.42} \rceil \cdot O(n \log \frac{n}{p})$ computation time and $\lceil e^{-3.42} \rceil \cdot O(gn \log \frac{n}{p})$ communication time.

Another deterministic sort is proposed by Gerbessiotis *et. al.* in [41] on the BSP model, which improves the upper bound in [1]. For $p = n^{1-\varepsilon}, 0 < \varepsilon < 1$, this algorithm uses $(1 + \lfloor \varepsilon^{-1}(1-\varepsilon) \rfloor (\frac{1-\varepsilon}{2} + o(1))\frac{T_{seq}}{p}$ computation time and $O(g\frac{n}{p})$ communication time, where T_{seq} is the sequential sorting time.

Gerbessiotis and Valiant presented in [43] a randomized sorting algorithm on BSP which is based on Sample-sort. The algorithm is one-optimal for a range of parameters p, n and L, and also one-optimal in communication if g is bounded appropriately. The algorithm runs in $O(n \log \frac{n}{p})$ computation time and $O(\frac{gn \log n}{p \log \frac{n}{p}})$ communication time with high probability, provided $p = \frac{n}{\log^{1+\varepsilon} n}$, for any $\varepsilon > 0$. The authors later improved the scalability of the algorithm to $p = \frac{n \log \log n}{\log^{1+\varepsilon} n}$, for any $\varepsilon \leq 0$.

Goodrich [47] proposed a sorting scheme on the weak-CREW BSP (CGM) and EREW BSP models, which can be done in $O(\frac{\log n}{\log(h+1)})$ communication rounds and $O(\frac{n \log n}{p})$ computation time, where $h = \theta(\frac{n}{p})$. The computation is optimal for any comparison-based algorithm. When $p < n^{1-\frac{1}{c}}$, for any constant $c \geq 1$, these algorithms have constant communication rounds. It can also be shown that, for the full range of p, the number of communication rounds is asymptotically optimal.

We will use this sorting scheme in the later sections as a subroutine to design fully scalable coarse-grained interval graph algorithms.

4.4 Maximum Independent Set

Given $G = (V, E)$, a subset S of V is called an *independent set* of G if no two vertices of S are adjacent in G. An independent set S is *maximum* if G has no independent set S' with $|S'| > |S|$. We are interested in finding a maximum independent set (MIS) of an interval graph $G_{\Im}$. The following sequential algorithm for this purpose was presented in [48].

Seq-MIS Algorithm:

> First sort the $2n$ endpoints. Scan in ascending order until the first right endpoint is encountered. Output the interval I with this right endpoint as a member of the MIS and delete all intervals containing this point (including I). Proceed this process until no interval is left.

A PRAM algorithm for finding MIS of interval graphs requiring $O(\log n)$ time and $O(\frac{n^2}{\log n})$ processors was presented in [15].

In this section, we give a scalable CGM algorithm for this purpose, using $O(\log p)$ communication rounds and optimal local computation time. Our algorithm also implies a cost-optimal PRAM algorithm using $O(\log n)$ time and $O(n)$ processors which improves the algorithm in [15].

An interval $I_i \in \Im$ is *redundant* if there exists $I_j \in \Im$, such that $l_i < l_j$ and $r_j < r_i$. Given $\Im$, $\Im'$ denotes the interval set after removing all redundant intervals in $\Im$. Next we prove two technical lemmas.

Lemma 4.4.1 *An MIS of $G_{\Im'}$ is also an MIS of $G_{\Im}$.*

Proof: The lemma follows directly from the fact that when the algorithm Seq-MIS is performed on $\Im$, the redundant intervals never appear in MIS. Thus the behavior of Seq-MIS on $\Im$ and $\Im'$ is identical. $\square$

Lemma 4.4.2 *The order of the intervals of $\Im'$ sorted by left endpoints and that by right endpoints are identical.*

Proof: Let $I_1, \cdots, I_m$ be the intervals in $\Im'$ sorted by left endpoints. If the right endpoints of $I_1, \cdots, I_m$ are not strictly increasing, there must exist at least one redundant interval in $\Im'$, leading to a contradiction. □

Let $I_1, I_2, \cdots, I_n$ be the intervals in $\Im$ sorted by left endpoints. Let $R(I_i)$ be the index of I_i in $\Im$ when sorted by right endpoints. By Lemma 4.4.2, an interval I_i is redundant if and only if there exists j $(j > i)$ such that $R(I_j) < R(I_i)$.

Based on Lemmas 4.4.1 and 4.4.2, it is clear that the intervals selected by the Seq-MIS algorithm on $\Im$ are exactly those selected by the following:

Modified-Seq-MIS Algorithm:

- Construct $\Im'$ from $\Im$.
- Sort the remaining intervals by their left endpoints. Let $I_1, \ldots, I_m$ be the sorted intervals in $\Im'$.
- For each index i, let $\mathrm{link}(i)$ be the smallest index such that $r_i < l_{\mathrm{link}(i)}$. If no such index exists, let $\mathrm{link}(i) = null$. Note that if we consider $I_{\mathrm{link}(i)}$ as the parent of I_i, then the intervals in $\Im'$ are arranged into a forest structure F.
- Select the intervals on the path from I_1 to the root of the tree in F that contains I_1.

Our CGM MIS algorithm is based on the algorithm Modified-Seq-MIS. The basic data structure in our algorithm is an array $L[1 \cdots n]$. Each $L[i]$ $(1 \le i \le n)$ specifies an interval in $\Im$ and is a record consisting of the following fields:

- $L[i].id$: the interval id, within $[1 \cdots n]$,
- $L[i].l$: the coordinate of the left endpoint of interval $L[i]$, and
- $L[i].r$: the coordinate of the right endpoint of interval $L[i]$.

CGM-MIS Algorithm:

Input: Records of L are evenly distributed among the p-processors.

1. Renaming intervals:
   (a) Sort L by l fields. (Each processor P_i now contains $\frac{n}{p}$ records of sorted L, according to left endpoints.)
   (b) Each processor replaces the *id* fields of its local $\frac{n}{p}$ records by their corresponding indices in sorted L.
2. Computing $\Im'$:
   (a) Sort L by r fields. (Each processor P_i now contains $\frac{n}{p}$ records of sorted L, according to right endpoints.)
   (b) Perform prefix sum on field *id* for all intervals with operator max.
   (c) Discard redundant intervals, which are those whose *id* fields themselves are not prefix maximum values.
   (d) Sort the remaining intervals by l fields and update the *id* field values as done in Step 1. These intervals form $\Im'$.
3. Computing MIS of $G_{\Im'}$:

(a) For each $L[i]$ of $\Im'$, create two records: $L_l[i] = (L[i].l, 0, L[i].id, L[i].id)$ and $L_r[i] = (L[i].r, 1, L[i].id, 0)$. In addition, create a dummy record $(\infty, 1, null, null)$.

(b) Sort these $2|\Im'| + 1$ records by their first fields.

(c) Perform backward interval broadcasting for these $2|\Im'| + 1$ records on the 4th field, with nonzero values as leaders. (For each interval $L[i]$ in $\Im'$, the 4th field of the record $L_r[i]$ now contains the index link(i).)

(d) Perform array packing to pack, among the $2|\Im'| + 1$ records, those $|\Im'| + 1$ ones, whose second fields are 1's. These $(|\Im'| + 1)$ records thus conceptually form a forest F linked by the 4th field.

(e) Among those $(|\Im'| + 1)$ records in F, identify the records on the path from the first record $L_r[1]$ to the root of the tree in F that contains $L_r[1]$. Those records correspond to the intervals (whose *id*'s are stored in the 3rd fields) in MIS of $G_{\Im}$.

Analysis: The sorting steps can be done by using the CGM parallel merge sort in [47], taking $O(1)$ communication rounds and optimal local computation time. Prefix sum in step 2 (b), backward interval broadcasting in step 3 (c) and array packing in step 3 (d) are applications of our CGM prefix sum algorithms.

Step 3 (e) is the only step that requires $O(\log p)$ communication rounds. For each record $L_r[i]$ in the forest F, let $L_r[i].min$ denote the minimum of interval id's (the 3rd field of $L_r[i]$) of all records in the subtree rooted at $L_r[i]$. Those records with $L_r[i].min = 1$ are the records to be identified. All $L_r[i].min$ can be computed by using pointer jumping technique.

Here, in order to enforce the weak-CREW property of CGM, before performing pointer jumping operations, we would need to relocate the records in such a way that those of the same depths are stored in consecutive processors. To accomplish this, we apply the well known method by Tarjan and Vishkin [94] to compute Euler Tours of trees in our forest F and further compute depths for all records. Another sorting by depth of each record finishes our record relocation.

Caceres [23] presented CGM algorithms for list ranking and Euler Tour computation, both requiring $O(\log p)$ communication rounds. So far, for CGM model, list operations usually contribute the dominating part of required communication rounds. Thus we have:

Theorem 4.4.1 *Finding maximum independent set of interval graphs containing n intervals can be optimally solved on a p-processor CGM computer, using $O(\log p)$ communication rounds and optimal local computation time in each round, given $\frac{n}{p} \geq p^{\varepsilon}$, $\forall \varepsilon > 0$.*

4.5 Maximum Weighted Clique

A set $S \subseteq \Im$ is a *clique* of $G_{\Im}$ if every pair of intervals in S intersect. A *maximum cardinality clique* is a clique with maximum number of elements. An interval graph is *weighted* if a positive real number w_i, the *weight*, is associated with each interval I_i. A *maximum weighted clique* (MWC) of a weighted interval graph is the clique with maximum total weight. When all intervals carry unit weights, it is easily seen that a MWC is actually a maximum cardinality clique.

Our MWC algorithm uses an array $L[1..2n]$. Each record $L[i]$ $(1 \leq i \leq 2n)$ specifies either a left or right endpoint of some interval, and has the following fields:

- $L[i].coord$: coordinate of the end point specified by $L[i]$,
- $L[i].end = \begin{cases} l & \text{if } L[i] \text{ is a left endpoint} \\ r & \text{if } L[i] \text{ is a right endpoint,} \end{cases}$
- $L[i].weight = \begin{cases} \text{the weight of the interval corresponding to } L[i] & \text{if } L[i].end = l \\ -\ (\text{the weight of the interval corresponding to } L[i]) & \text{if } L[i].end = r, \end{cases}$
- $L[i].id$: the *id* of the interval corresponding to $L[i]$, within $[1 \cdots n]$, and
- $L[i]$.sum: weight sum (initially 0).

MWC Algorithm:

Input: Each processor P_i stores $\frac{2n}{p}$ endpoints of L.

1. Sort L, using *coord* as key.
2. Perform prefix sum on field $L[i].weight$ and store the result at $L[i].sum$.

   (*i.e.*, $L[i].sum = \sum_{j=1}^{i} L[j].weight$. Note that if $L[i].end = l$, $L[i].sum$ is the total weight of the clique consisting of all intervals that contain the point $L[i].coord$.)
3. Compute prefix max on $L[i].sum$, for all $L[i]$'s with $L[i].end = l$.
4. Assume $L[k].sum$ is the maximum of the *sum* fields among all records. Broadcast $L[k].coord$ to all other processors.

5. In parallel, each processor determines the intervals containing the point $L[k].coord$. Those intervals are in the MWC.

Analysis: Step 1 can be done by using the CGM parallel merge sort in [47], taking $O(1)$ communication rounds and optimal local computation time. Steps 2 and 3 are applications of prefix sum operation. Step 4 uses one communication round for all processors to receive the broadcasted endpoints. Step 5 uses a single computation round in time $O(\frac{2n}{p})$. This gives:

Theorem 4.5.1 *Finding maximum weighted clique of interval graphs containing n intervals can optimally be solved on a p-processor CGM computer, using $O(1)$ communication rounds and optimal local computation time in each round, given $\frac{n}{p} \geq p^{\varepsilon}$, $\forall \varepsilon > 0$.*

4.6 Minimum Coloring

Given $\Im$, a partition $\wp = \{S_1, S_2, \cdots, S_k\}$ of $\Im$ is a *coloring* of $G_{\Im}$ if each S_i $(1 \leq i \leq k)$ is an independent set of $G_{\Im}$. A coloring with minimum k is called a *minimum coloring* [45]. It is known that $\omega(G_{\Im})$, the maximum clique size, is equal to $\chi(G_{\Im})$, the chromatic number (the fewest number of colors needed to properly color vertices of $G_{\Im}$) [45].

Yu [101] devised an optimal algorithm for computing minimum coloring of interval graphs on EREW PRAM. It works as follows: Let $\Im = \{I_1 \cdots I_n\}$ be an interval set sorted by left endpoints. Let $I_{R(i)}$ denote the interval with i-th smallest right endpoint.

For each i, compute link(i):

$$\text{link}(i) = \begin{cases} i + \omega(G_{\Im}) & \text{if } i + \omega(G_{\Im}) < n+1; \\ n+1 & \text{otherwise.} \end{cases}$$

Then, for any i $(1 \leq i \leq n)$, link the interval $I_{R(i)}$ to the interval $I_{\text{link}(i)}$ if link$(i) \neq n+1$; otherwise, link $I_{R(i)}$ to $null$. There will be $\omega(G_{\Im})$ separate linked lists and by assigning each list a distinct color, we have a minimum coloring. The correctness proof can be found in [101].

Our CGM algorithm for finding minimum coloring of interval graphs is based on the above observation. In addition to prefix computation and sorting, in order to propagate colors in these $\omega(G_{\Im})$ lists, our algorithm also needs pointer jumping techniques, which requires $O(\log p)$ communication rounds. The algorithm uses an array $L[1..n]$. Each record $L[i]$ specifies an interval in $\Im$ and contains the following fields:

- $L[i].id$: the interval id, within $[1 \cdots n]$,
- $L[i].l$: the coordinate of left endpoint of interval corresponding to $L[i]$,
- $L[i].r$: the coordinate of right endpoint of interval corresponding to $L[i]$,
- $L[i].link = \begin{cases} i + \omega(G_{\Im}) & \text{if } i + \omega(G_{\Im}) < n+1 \\ n+1 & \text{otherwise} \end{cases}$ (initially null), and
- $L[i].color$: the color assigned to interval $L[i]$ (initially i).

Another array $R[1 \cdots n]$ is needed in our algorithm:

- $R[i]$: the interval id containing the i-th smallest right endpoint.

Minimum Coloring Algorithm:

Input: L and R are evenly distributed among the p processors.

1. Renaming intervals:

   Sort L using l as key field. Each processor replaces the *id* fields of its local $\frac{n}{p}$ records by their corresponding indices in sorted L.

2. Computing array R:

   (a) Sort L using r as key field.

   (b) Each processor computes $R[i] = L[i].id$ for all its local records.

3. All records of L resume their positions in step 1.

4. Compute $\omega(G_{\mathfrak{Z}})$ by using the MWC algorithm in Section 4.5.

5. Each processor computes $L[i].link$ for all its local intervals, using the above formula.

6. Assign color to each interval:

   (a) Consider the record corresponding to $I_{R[i]}$ "linked" to that corresponding to $I_{L[i].link}$. These records form $\omega(G_{\mathfrak{Z}})$ linked lists.

   (b) In parallel, each record searches for the root of its list.

   (c) In parallel, each record replaces its color value by the color of root record.

Analysis: Steps 1 and 2 both involve global sorting and, as mentioned in previous sections, can be finished using optimal local computation time and $O(1)$ communication rounds. Step 3 consumes a single communication round with a $\frac{n}{p}$-relation. Step

4 takes $O(1)$ communication rounds and optimal local computation time by Theorem 4.5.1. Broadcasting $\omega(G_{\Im})$ to all other processors takes another single communication round. Step 5 takes obviously $O(\frac{n}{p})$ local computation time. Step 6 (a) finishes in a single computation round. Step 6 (b) uses pointer jumping on CGM, and takes $O(\log p)$ communication rounds (for inter-processor pointer jumps) and $O(\frac{n}{p})$ local computation time in each round (for intra-processor pointer jumps). Step 6 (c) obviously finishes in one single computation round. This concludes:

Theorem 4.6.1 *Finding minimum coloring of interval graphs containing n intervals can optimally be solved on a p-processor CGM computer, using $O(\log p)$ communication rounds and optimal local computation time in each round, given $\frac{n}{p} \geq p^{\varepsilon}, \forall \varepsilon > 0$.*

4.7 Cut Vertices and Bridges

A vertex v of $G_{\Im}$ is a *cut vertex* if removal of v results in a graph having more components than $G_{\Im}$ has. Likewise, an edge with such property is called a *bridge*. A *biconnected component* of $G_{\Im}$ is a maximal connected component of $G_{\Im}$ which contains no cut vertices.

In [92], Sprague proposed an optimal EREW PRAM algorithm for finding cut vertices and bridges of interval graphs, using $\frac{n}{\log n}$ processors in $O(\log n)$ time. This algorithm heavily relies on the PRAM prefix sum operations. Here, by employing our CGM prefix sum procedure, we can easily implement an optimal CGM algorithm for finding cut vertices, bridges, and biconnected components in interval graphs.

We use an array $L[1 \cdots 2n]$ for endpoints of intervals in $\Im$. Each $L[i]$ $(1 \leq i \leq 2n)$

specifies either a left or a right endpoint of some interval. Later in the algorithm, after sorting the $2n$ coordinates, we will further replace these original coordinates by their indices in sorted array L. From then on, all coordinates are within range $[1\cdots 2n]$. Detailed data structures are described as below:

- $L[i].coord$: coordinate of $L[i]$,
- $L[i].end = \begin{cases} l & \text{if } L[i] \text{ is a left endpoint} \\ r & \text{if } L[i] \text{ is a right endpoint,} \end{cases}$
- $L[i].type = \begin{cases} 1 & \text{if } L[i].end = l \\ -1 & \text{if } L[i].end = r, \end{cases}$
- $L[i].id$: the id of the interval corresponding to $L[i]$, within $[1\cdots n]$,
- $L[i].density$: the number of intervals containing $L[i].coord$ (initially 0),
- $L[i].e = \begin{cases} (id, r_{id}) & \text{if } L[i].end = l \\ (-\infty, -\infty) & \text{if } L[i].end = r, \end{cases}$
  where r_{id} is the right endpoint coordinate of interval I_{id},
- $L[i].f = L[1].e \oplus L[2].e \oplus \cdots \oplus L[i].e$, where $(x,a) \oplus (y,b) = \begin{cases} (x,a) & \text{if } a > b \\ (y,b) & \text{otherwise} \end{cases}$
  (initially e, and to be computed in step 4), and
- $L[i].g$ is to be assigned the cut vertex (denoted as interval id) at step 5 (initially -1).

An important property on which our CGM cut vertex algorithm is based is uncovered in [92] as follows::

Lemma 4.7.1 *An interval I_i is a cut vertex if and only if there exists j, such that $l_i \leq j < r_i$ and $L[j-1].density = L[j+1].density = 2$; $L[j].density = 1$.*

Cut Vertex Algorithm:

Input: Each processor P_i stores $\frac{2n}{p}$ records of L.

1. Globally sort L, using *coord* as key.

2. Each processor replaces the *coord* fields of its local $\frac{2n}{p}$ records by their corresponding indices in sorted L.

3. Compute density by using prefix sum: $L[i].density = \sum_{j=0}^{i} L[j].type$.

   (An interval I_k is called a *covering interval* of a point x if $l_k \leq x < r_k$. $L[i].density$ now contains the number of covering intervals of $L[i].coord$).

4. For each i such that $L[i].density > 0$, find a covering interval as follows:

   Compute f field for each $L[i]$. Then the first coordinate of f specifies the interval whose left end appears right at or before i and whose right end extends to the rightmost.

5. Assign the first component of f to g for those $L[j]$'s satisfying

   (a) $L[j].density = 1$ and

   (b) $L[j-1].density = L[j+1].density$=2.

6. Generate all cut vertices (using array packing for repetition removal and packing).

Analysis Step 1 is an application of Goodrich's CGM sorting and uses $O(1)$ communication rounds and optimal local computation time. Now each processor contains its part of $\frac{2n}{p}$ endpoints in sorted L. Step 2 assigns index i to $L[i].coord$, making these $2n$ endpoints consecutively numbered from 1 to $2n$. Each processor sequentially finishes this step in one computation round, using $O(\frac{n}{p})$ computation time. Step 3 is also a prefix sum operation and takes $O(1)$ communication rounds. Step 4 is a prefix maxima operation. Step 5 needs one communication round for those $L[i]$'s whose density fields are 1's to get either $L[i-1].density$ or $L[i+1].density$. So obviously the required communication rounds are dominated by sorting and prefix sum operations, which are $O(1)$. And the local computation time is also optimal. Therefore, we conclude:

Theorem 4.7.1 *Finding cut vertices on interval graphs containing n intervals can optimally be solved on a p-processor CGM computer, using $O(1)$ communication rounds and optimal local computation time in each round, given $\frac{n}{p} \geq p^{\varepsilon}, \forall \varepsilon > 0$.*

Optimal CGM algorithms for finding bridges and biconnected components can be designed in a similar fashion.

4.8 Concluding Remarks

We have designed the fully scalable CGM prefix sum algorithm in $O(1)$ communication rounds and, by incorporating the fully scalable CGM sorting subroutine, provided algorithms for finding maximum weighted cliques, cut vertices and bridges in interval graphs, also in $O(1)$ communication rounds. Since prefix computation and sorting have widely been applied in interval graph algorithms, many other problems can be solved on CGM computers in similar approaches. Pointer jumping is another important procedure commonly used in many parallel algorithms. On the CGM computers, algorithms using pointer jumping usually take $O(\log p)$ communication rounds, which is independent of the input size and grows slowly only with the number of processors. Our algorithms for finding maximum independent set and minimum coloring of interval graphs employ prefix computation, sorting and pointer jumping, and thus use $O(\log p)$ communication rounds.

Chapter 5

Coarse-Grained Parallel Divide-and-Conquer

5.1 Introduction

A large number of parallel computing problems in many fields are defined in terms of graphs. However, graph problems have been shown to have considerably less internal structures than many other problems studied. This results in highly data-dependent communication patterns and makes it difficult to achieve communication efficiency.

Balancing the load assigned to different processors and minimizing the communication overhead are the core problems in achieving high performance on parallel or distributed systems. Typically, algorithms based on divide-and-conquer need to trade off between processor and communication overhead. However, parallel divide-and-conquer specifies an important class of problems in many fields, such as computational geometry [2, 4], graph theory [44, 85], numerical analysis [70, 37] and optimization [10]. Therefore, designing an approach that reduces the interprocessor communication overhead while balancing the workload becomes essential, especially when such algorithms are to be implemented on fully distributed-memory environments.

This chapter focuses on one such typical example, *finding Hamiltonian paths in tournaments*, a graph problem that has a divide-and-conquer algorithm in the sequential setting. A *tournament* is a directed graph $\Im = (V, E)$ in which, for any pair of vertices u, v, either $(u, v) \in E$ or $(v, u) \in E$, but not both. This models a competition involving n players, where every player competes against every other one. A *Hamiltonian path* in a graph is a simple path that visits every vertex exactly once.

We present load-balanced and communication efficient partitioning strategies that generate sub-tournaments as evenly as possible for each processor. The computation for the Hamiltonian path of each sub-tournament is then carried out using the existing sequential algorithm.

The major features of the main algorithm in this chapter are as follows:

1. Inter-processor communication overhead in the "divide" stage (partitioning) is reduced by routing data only after the final destination processor has been determined, reducing large amount of data movement.

2. Code reuse from existing sequential algorithm is maximized in the "conquer" stage.

3. No additional communication overhead is required in the "merge" stage.

For theoretical completeness, this algorithm has also been revised for parallel computing platforms where either each individual processor has extremely limited local memory or the ratios of computation and communication throughputs are low.

Throughout this chapter, we design algorithms based on the original EREW BSP model; that is, neither network broadcast nor combining capability is assumed. In addition, although the original BSP model definition does not limit the local memory size, we follow the CGM constraint that the local memory size of each processor is only $O(\frac{N}{p})$. We also follow the CGM constraint that, in each communication round, only an $O(\frac{N}{p})$-relation is routed.

The rest of this chapter is organized as follows: Section 5.2 presents the BSP-style (EREW) algorithm, Algorithm HPT-1, for finding Hamiltonian paths in tournaments and details the BSP cost analysis for both local computation and interprocessor communication. Algorithm HPT-2, revised from Algorithm HPT-1, for platforms in more theoretical settings are given in Section 5.3. In this section, two operations: prefix sum and array packing are employed in the partitioning stage so that the revised algorithm fits in these environments. Section 5.4 concludes this chapter.

5.2 The BSP Algorithm and Cost Analysis

5.2.1 The BSP Algorithm

Assume a tournament $\Im = (V, E)$ is given, where $|V| = n$, and $|E| = n^2 = N$. Throughout this chapter, the size of a tournament $\Im = (V, E)$ will always refer to $|E|$. A trivial, but useful, fact is that any induced subgraph of a tournament is also a tournament.

Given $\Im = (V, E)$, define $\Im(V')$ to be the tournament (induced subgraph) on V', where $V' \subseteq V$. We say u *dominates* v if $(u, v) \in E$, and denote this property by $u > v$. Note that since the directions of the arcs are arbitrary, the domination relation is not necessarily transitive. The notion of domination is extended to sets of vertices: Let A, B be subsets of V. A *dominates* B ($A > B$) if every vertex in A dominates every vertex in B. For a given vertex v in $\Im$, the rest of the vertices of $\Im$ are categorized according to their relations with v: $W(\Im, v)$ is the set of vertices that are dominated by v and $L(\Im, v)$ is the set of vertices that dominate v.

Much work has been done on tournaments [11]. Here we concentrate on a classical result: every tournament has a Hamiltonian path [84, 91] and start by stating the theorem for Hamiltonian path.

Theorem 5.2.1 *Every tournament contains a Hamiltonian path.*

Proof: By induction on the number of vertices, n, the result is clear for $n = 2$. Assume the theorem holds for tournaments on n vertices. Considering a tournament $\Im$ on $n+1$ vertices, let v be an arbitrary vertex of V. By the induction hypothesis, $\Im(V - \{v\})$ has a Hamiltonian path $v_1, v_2, \cdots, v_n$. If $v > v_1$, then $v, v_1, \cdots, v_n$ is a Hamiltonian path of $\Im$. Otherwise, let i be the largest index such that $v_i > v$. If $i = n$ then $v_1, \cdots, v_n, v$ is a Hamiltonian path. If not, $v_1, \cdots, v_i, v, v_{i+1}, \cdots, v_n$ is the desired Hamiltonian path.

□

For each vertex v in $\Im$, we define $d_{in}(\Im, v)$ (respectively, $d_{out}(\Im, v)$) to be the *in-degree* (respectively, *out-degree*) of v in $\Im$.

Theorem 5.2.2 *In a tournament $\Im = (V, E)$ on n vertices, there exists a vertex v, referred to as mediocre player, for which both $L(v)$ and $W(v)$ have at least $\lfloor \frac{n}{4} \rfloor$ vertices.*

Proof: Let $I = \{u \in V \mid d_{in}(\Im, u) \geq d_{out}(\Im, u)\}$ and $O = V - I$. Assume without loss of generality that $|I| \geq |O|$. Since the sum of the in-degrees of the vertices in $\Im(I)$ equals to the sum of the out-degrees of the vertices in $\Im(I)$, there exists a vertex $v \in I$ whose out-degree in $\Im(I)$ is no less than its in-degree in $\Im(I)$. Thus: $d_{out}(\Im(I), v) \geq \lfloor \frac{|I|}{2} \rfloor \geq \lfloor \frac{n}{4} \rfloor$ and $d_{in}(\Im, v) \geq d_{out}(\Im, v) \geq d_{out}(\Im(I), v) \geq \lfloor \frac{n}{4} \rfloor$ as to be shown. □

Throughout this chapter, $H_{\Im}$ will be used to denote a Hamiltonian path of a tournament $\Im$. Assuming a mediocre player of $\Im$ is v_m, an observation reveals the fact that the concatenation $H_{\Im(L(\Im, v_m))}, v_m, H_{\Im(W(\Im, v_m))}$ is a $H_{\Im}$. This observation, along with Theorem 5.2.2, motivates the algorithm design. The ideas of the algorithm are sketched below.

Denote the initial tournament $\Im$ by $\Im_0^0$. In the first partitioning step, we identify the mediocre player, v_0^1, of $\Im_0^0$ and split $\Im_0^0$ into $\Im_0^1 = \Im(L(\Im_0^0, v_0^1)$ and $\Im_1^1 = \Im(W(\Im_0^0, v_0^1)$. Similarly, during the second partitioning step, we identify the mediocre players v_0^2 and v_1^2 of $\Im_0^1$ and $\Im_1^1$, respectively. We then split $\Im_0^1$ into $\Im_0^2 = \Im(L(\Im_0^1, v_0^2)$, $\Im_1^2 = \Im(L(\Im_0^1, v_0^2)$ and $\Im_1^1$ into $\Im_2^2 = \Im(L(\Im_1^1, v_1^2)$, $\Im_3^2 = \Im(W(\Im_1^1, v_1^2)$. Inductively, during the i-th partitioning step, we need to identify the mediocre players $v_0^i, v_1^i, \cdots, v_{2^{i-1}-1}^i$ of $\Im_0^{i-1}, \Im_1^{i-1}, \cdots, \Im_{2^{i-1}-1}^{i-1}$, respectively. Then split $\Im_0^{i-1}$ into $\Im_0^i = \Im(L(\Im_0^{i-1}, v_0^i)$, $\Im_1^i = \Im(W(\Im_0^{i-1}, v_0^i)$; and split $\Im_1^{i-1}$ into $\Im_2^i = \Im(L(\Im_0^{i-1}, v_1^i)$, $\Im_3^i = \Im(W(\Im_0^{i-1}, v_1^i)$, etc. The partitioning stage proceeds no more than $\log_{\frac{4}{3}} p$ iterations and no less than $\log_4 p$ iterations. At this point, each sub-tournament has at most n/p vertices. Then each sub-tournament $\Im_i$ is sent to a processor, and a Hamiltonian path $H_{\Im_i}$ is locally found by the processor. The concatenation of these Hamiltonian paths $H_{\Im_i}$'s and the selected mediocre players is a Hamiltonian path $H_{\Im}$.

For ease of description, the number of iterations in the partitioning stage of our algorithm is denoted as $\log p$.

Note that the mediocre player of a tournament is not unique. In order to split the tournaments as evenly as possible in the partitioning stage, in each round, the mediocre player whose in-degree and out-degree are closest is selected and used to split the tournaments from the preceding partitioning round. An important idea here is, during the "divide" stage, only the mediocre players are communicated among processors. Sub-tournaments are moved to destination processors only after the splitting process ends, when each sub-tournament can now fit in the storage of a single processor. Sequential algorithm for finding Hamiltonian path is now applied in parallel, and the results, along with all the mediocre players selected during the last partitioning step form the final Hamiltonian path.

The BSP algorithm for finding Hamiltonian paths in tournaments uses the following major data structures:

(1) Adjacency matrix:

$$\varepsilon[i][j]\ (0 \le i, j \le n-1)\ = \begin{cases} 1 & \text{if } v_i > v_j\ ; \\ -1 & \text{if } v_i < v_j. \end{cases}$$

Note that ε is anti-symmetric. Also note that the total number of 1's (-1's, respectively) in row i is the in-degree (out-degree, respectively) of v_i.

(2) $\wp[i][j]\ (0 \le i \le \log p - 1, 0 \le j \le n-1)$

$$= \begin{cases} 0 & \text{if } v_j \in \bigcup_{k=0}^{2^{i-1}-1} L(\Im_k^{i-1}, v_k^i) \text{ in partitioning round } i\ ; \\ 1 & \text{if } v_j \in \bigcup_{k=0}^{2^{i-1}-1} W(\Im_k^{i-1}, v_k^i) \text{ in partitioning round } i. \end{cases}$$

Note that $\wp[i][j]$ keeps track of the sub-tournament v_j belongs to in partitioning round i. When the partitioning stage finishes, each v_j can determine, via column j of $\wp[i][j]$ the target processor $P_{T(j)}$, where $T(j) = \sum_{t=0}^{\log p-1} 2^t \times \wp[t][j]$.

(3) $\iota[i](\vartheta[i], \text{respectively})\ (0 \le i \le n-1)$ = in-degree (out-degree, respectively) of v_i in the current sub-tournament to which v_i belongs.

Assume BSP processors are labeled $P_0, P_1, \cdots, P_{p-1}$. Processor P_i initially contains

(a) $\varepsilon[i(\frac{n}{p}) \ldots i(\frac{n}{p}) + (\frac{n}{p}) - 1][0 \ldots n-1]$,

(b) $\wp[0 \ldots \log p - 1][0 \ldots n-1]$ (initially all 0's) and

(c) ι, $\vartheta[0 \ldots n-1]$ (initially all 0's).

The algorithm is described as below:

Algorithm HPT-1 :

Input: A tournament $\Im = (V,E)$, where $|V| = n$ and $|E| = n^2 = N$, given as $\varepsilon[i][j]$ $(0 \leq i, j \leq n-1)$. Each processor P_i initially stores $\varepsilon[i(\frac{n}{p}) \ldots i(\frac{n}{p}) + (\frac{n}{p}) - 1][0 \ldots n-1]$.

Output: $H_{\Im}$, a Hamiltonian path of the given tournament.

1. Each processor computes $\iota[j], \vartheta[j]$ for all local v_j's.

   (Note that, since P_i contains the row j of the adjacency matrix, $\iota[j], \vartheta[j]$ can be computed locally.)

2. Each processor performs the following steps in partitioning round i ($i = 1$ to $\log p$):

   (Note: Before the i-th iteration starts, for each vertex v, we have computed the sub-tournament (in partition round $(i-1)$) containing v and the in- and out-degrees of v in that sub-tournament. This information is stored in each processor.)

   (a) Identify the mediocre players $v_0^i, \cdots, v_{2^{i-1}-1}^i$ in tournaments $\Im_0^{i-1}, \cdots, \Im_{2^{i-1}-1}^{i-1}$, respectively: (Note that the initial $\Im = \Im_0^0$.)

      i. Each processor P_k $(0 \leq k \leq p-1)$ computes (if any) the (most) mediocre player $v_{k_t}^i$ contained in P_k, where $\iota[k_t] \geq \frac{|\Im_t^{i-1}|}{4}$ and $\vartheta[k_t] \geq \frac{|\Im_t^{i-1}|}{4}$, for every $\Im_t^{i-1}$ $(0 \leq t \leq 2^{i-1}-1)$.

      ii. Each processor P_k $(0 \leq k \leq p-1)$ sends v_{k_t} to P_t $(0 \leq t \leq 2^{i-1}-1)$.

      iii. Each processor P_t $(0 \leq t \leq 2^{i-1}-1)$ computes the (most) mediocre player (the one with the closest in-degree and out-degree) v_t^i for $\Im_t^{i-1}$.

      iv. Each processor P_t $(0 \leq t \leq 2^{i-1}-1)$ broadcasts v_t^i to all the other $p-1$ processors.

   (b) Split tournaments $\Im_0^{i-1}, \cdots, \Im_{2^{i-1}-1}^{i-1}$:

i. Each processor P_k $(0 \leq k \leq p-1)$ locally performs the following:

Assign 0 (1, respectively) to $\wp[i][j]$ if $v_j \in \bigcup_{k=0}^{2^{i-1}-1} L(\Im_k^{i-1}, v_k^i)$ $(v_j \in \bigcup_{k=0}^{2^{i-1}-1} W(\Im_k^{i-1}, v_k^i)$, respectively).

(c) Update incoming and outgoing degrees for all vertices:

i. Mark all entries in ε corresponding to the selected mediocre players.

ii. Recompute $\vartheta[j]$ and $\iota[j]$ to be the in- and out-degrees of v_j in the newly created sub-tournament containing v_j.

3. Re-distribute data of each sub-tournament to proper processor:

Each processor P_k $(0 \leq k \leq p-1)$ sequentially sends its local data $\varepsilon[x][0 \ldots n-1]$ (for $k(\frac{n}{p}) \leq x \leq k(\frac{n}{p}) + \frac{n}{p} - 1$) to $P_{T(x)}$ where $T(x) = \sum_{y=0}^{\log p-1} 2^y \times \wp[y][x]$.

4. Each processor P_k $(0 \leq k \leq p-1)$ performs the sequential algorithm to compute $H_{\Im_k}$ for the sub-tournament received from step 3.

(All $H_{\Im_k}$'s, along with the mediocre players selected from the last partitioning step, form the $H_{\Im}$, when concatenated according to the observation stated earlier.)

5.2.2 Computation and Communication Complexities

Assume the sequential algorithm for finding Hamiltonian paths in tournaments of size N is $T_s(N)$. Then the BSP cost breakdown of Algorithm HPT-1 can be derived as Table 5.1.

Therefore, the following theorem is derived:

Table 5.1: BSP cost breakdown of *Algorithm HPT*-1

	Cost		
Step	comp.	comm.(in relation)	synch.
1	$\frac{N}{p}$		L
2(a)i	$(\frac{N}{p})\log p$		
2(a)ii		$p\log p$	$L\log p$
2(a)iii	$(\frac{N}{p})\log p$		
2(a)iv		$p\log p$	$L\log p$
2b	$(\frac{N}{p})\log p$		
2(c)i	$(\frac{N}{p})\log p$		
2(c)ii	$(\frac{N}{p})\log p$		$L\log p$
3	$\frac{N}{p}$	$\frac{N}{p}$	L
4	$T_s(\frac{N}{p})$		L

Theorem 5.2.3 *The Hamiltonian path of a tournament $\Im = (V,E)$, where $|V|$=n and $|E|$=n^2=N, can be determined on a p-processor BSP computer with $O(\frac{N}{p})$ local memory, using $T_s(\frac{N}{p}) + \theta(\frac{N}{p}\log p)$ local computation time and $3(\log p + 1)$ supersteps, in which the total communication time is $g \times (\frac{N}{p} + 2p\log p)$.*

The number of supersteps needed is $3(\log p + 1)$, independent of the tournament size N. And, in total, the routed relations is only of size $(\frac{N}{p} + 2p\log p)$. This approach can be applied to solve problems cited in Section 5.1 in the same computation and communication efficient manners.

5.3 For Theoretical Environments

The data partitioning approach stated in Section 5.2 generally favors those platforms where the ratios of computation and communication throughputs are high. Typically, such ratios of commercially available parallel machines are high. For those machines where such ratios are low, an alternative partitioning strategy is provided as follows: In each partitioning round, a well-known basic operation *array packing* [5] is used to distribute sub-tournaments after the mediocre players have been identified. Array packing is a typical application of the general parallel operation *prefix sum* mentioned in Chapter 4. The details are to be included in Section 5.3.1.

Besides, step 2 of Algorithm HPT-1 assumes the local memory size $\frac{N}{p} = \Omega(p)$, which is practically valid for all existing parallel machines. Yet, for environments where each individual processor has extremely limited local memory, *i.e.* $\frac{N}{p} = o(p)$, Algorithm HPT-1 needs to be revised to extend the scalability factor to $\frac{N}{p} = \Omega(p^{\varepsilon})$, $\forall \varepsilon > 0$ [47]. Array packing operation turns out also useful while improving the scalability of Algorithm HPT-1.

In Section 5.3.1, we mention the fully scalable array packing algorithm. Using this algorithm as a subroutine for splitting tournaments in the partitioning stage, Algorithm HPT-2, revised from Algorithm HPT-1, can fit in those environments where (1) the ratios of computation and communication throughputs are low, or (2) the local memory of each processor is extremely limited. Algorithm HPT-2, along with the BSP cost analysis, are presented in Section 5.3.2.

5.3.1 Array Packing

As mentioned in Chapter 4, prefix sum is defined in terms of a binary, associative operator $\otimes$. The computation takes as input a sequence $< a_1, a_2, \cdots, a_n >$ and produces as output a sequence $< b_1, b_2, \cdots, b_n >$ such that $b_1 = a_1$ and $b_k = b_{k-1} \otimes a_k$ for $k = 2, 3, \cdots, n$.

The prefix sum can easily be solved on BSP using $O(1)$ supersteps and optimal local computation time with scalability at $\frac{n}{p} \geq p$. Theorem 4.2.1 further achieves scalability at $\frac{n}{p} \geq p^{\varepsilon}$ for any $\varepsilon > 0$, while preserving both computation and communication optimality.

Array packing using prefix sum operations is described next.

Array Packing [5]: Given an array $A = \{a_1, a_2, \cdots, a_n\}$, some of whose elements are labeled. All labeled elements need be packed at the front of the array. The corresponding BSP algorithm is a straightforward application of the prefix sum algorithm.

1. In parallel, use a secondary array of n elements $S = \{s_1, s_2, \cdots, s_n\}$ to compute the destination of each labeled element of A. Initially $s_i = 1$ if a_i is labeled; $s_i = 0$ if a_i is unlabeled.

2. Perform prefix sum on S; then broadcast the maximum value. Each labeled element now knows its correct position.

3. Reverse the 1's and 0's in S and perform prefix sum again. Also broadcast the maximum value. Each unlabeled element now has its position w.r.t the other unlabeled elements. By adding the broadcasted value from 2, each unlabeled element can determine its correct position.

4. Using another communication round with $\frac{n}{p}$-relation, all elements reach their destination processors.

5.3.2 The Revised BSP Algorithm

Algorithm HPT-1 attempts to reduce the inter-processor communication overhead in the "divide" stage (partitioning) by routing data only after the final destination processor has been determined. That is, the sub-tournaments are sent to the proper processors only when the partitioning process finishes. During the partitioning steps, only the selected mediocre players are communicated among the processors, which reduces the order of relations from $O(\frac{N}{p})$ to $O(p)$. This introduces some computation overhead in the partitioning stage while computing the mediocre players, updating the in- and out-degrees, etc. In addition, the broadcasting operation used in steps 2(a)ii and 2(a)ii assume $\frac{N}{p} = \Omega(p)$; that is, the local memory size of each processor is significantly more than the number of BSP processors. For theoretical parallel computing platforms, where either the ratio of computation and communication throughputs are low, or the local memory of each processor is extremely limited ($\frac{N}{p} = \o(p)$, we need to revise Algorithm HPT-1. Here we employ the fully scalable prefix sum and array packing procedures in the partitioning stage to make sure, before each iteration starts, all sub-tournaments are properly distributed.

For ease of algorithm description, we define $\mathcal{G}(\Im_k^i)$ to be the set of processors which contain vertices in $\Im_k^i$ in partitioning step i. Inheriting concepts from Sections 5.2 and 5.3.1, Algorithm HPT-2, revised from Algorithm HPT-1, is outlined as follows:

Algorithm HPT-2 :

1. Each processor computes $\iota[j], \vartheta[j]$ for all local v_j's.

2. Perform the following steps in partitioning round i ($i = 1$ to $\log p$):

   (a) Identify the mediocre players $v_0^i, \ldots, v_{2^{i-1}-1}^i$ in tournaments $\mathfrak{I}_0^{i-1}, \ldots, \mathfrak{I}_{2^{i-1}-1}^{i-1}$: (Note that the initial $\mathfrak{I} = \mathfrak{I}_0^0$.)

      i. Apply prefix minimum operation among processors in $\mathcal{G}(\mathfrak{I}_k^i)$ to compute the (most) mediocre player (the one with the closest in-degree and out-degree) v_k^i for $\mathfrak{I}_k^{i-1}$. ($0 \leq k \leq 2^{i-1} - 1$, in parallel)

      ii. v_k^i ($0 \leq k \leq 2^{i-1} - 1$) is globally communicated among processors in $\mathcal{G}(\mathfrak{I}_k^i)$.

   (b) Split tournaments $\mathfrak{I}_0^{i-1}, \ldots, \mathfrak{I}_{2^{i-1}-1}^{i-1}$:

      i. All processors of $\mathcal{G}(\mathfrak{I}_k)$ ($0 \leq k \leq 2^{i-1} - 1$) locally perform the following:

         Assign 0 (1, respectively) to $\wp[i][j]$ if $v_j \in L(v_{m_k})$ ($v_j \in W(v_{m_k})$, respectively), for all local v_j's.

      ii. Apply array packing on $\wp[i][j]$ to relocate sub-tournaments $\mathfrak{I}_k^i$'s.

   (c) Update incoming and outgoing degrees for all vertices:

      i. Mark all entries in ε corresponding to the selected mediocre players.

      ii. Recompute $\vartheta[j]$ and $\iota[j]$ to be the in- and out-degrees of v_j in the newly created sub-tournament containing v_j.

3. Now that all sub-tournaments are properly positioned, each processor P_k ($0 \leq k \leq p - 1$) performs sequential algorithm to compute $H_{\mathfrak{I}_k}$.

   (All $H_{\mathfrak{I}_k}$'s, along with the mediocre players selected from the last partitioning

Table 5.2: BSP cost breakdown of *Algorithm HPT*-2

	Cost		
Step	comp.	comm.(in relation)	synch.
1	$\frac{N}{p}$		L
2(a)i	$\theta(\frac{N}{p})\log p$	$\theta(\frac{1}{\varepsilon})(\frac{N}{p}+p)\log p$	$\theta(\frac{1}{\varepsilon})L\log p$
2(a)ii		$p\log p$	$L\log p$
2(b)i	$\theta(\frac{N}{p})\log p$		
2(b)ii	$\theta(\frac{N}{p})\log p$	$\theta(\frac{1}{\varepsilon})(\frac{N}{p}+p)\log p$	$\theta(\frac{1}{\varepsilon})L\log p$
2(c)i	$(\frac{N}{p})\log p$		
2(c)ii	$(\frac{N}{p})\log p$		
3	$T_s(\frac{N}{p})$		L

step, form the $H_{\Im}$, when concatenated according to the observation stated earlier.)

The BSP cost breakdown of algorithm HPT-2 can therefore be derived as Table 5.2. This concludes the following theorem:

Theorem 5.3.1 *The Hamiltonian path of a tournament $\Im = (V,E)$, where $|V|$=n and $|E|$=n^2=N, can be determined on a p-processor BSP computer with local memory size $\frac{N}{p}$, where $\frac{N}{p} = \Omega(p^{\varepsilon}), \forall\varepsilon > 0$, using $T_s(\frac{N}{p}) + \theta(\frac{N}{p}\log p)$ local computation time and $\theta(\frac{1}{\varepsilon})\log p$ supersteps, in which the total communication time is $g \times \theta(\frac{1}{\varepsilon})(\frac{N}{p}+p)\log p$.*

5.4 Concluding Remarks

Since parallel divide-and-conquer specifies an important class of problems in various fields, including computational geometry, graph theory, numerical analysis, and optimization, designing an efficient parallelization approach to reduce the inter-processor communication overhead and balance the workload for this class of problems has been essential. In this chapter, we discuss the coarse-grained parallel divide-and-conquer techniques to find a Hamiltonian path for a given tournament graph. We discuss the computation and communication efficiency on the BSP model, which conforms to almost all the commercially available parallel machines. In addition, we revise the algorithm and discuss the efficiency for platforms where either the ratio of computation and communication throughputs is low or the local memory of each processor is extremely limited.

Chapter 6

Future Work

The focus of this dissertation is to design communication-efficient and scalable BSP algorithms for problems that are more communication-intensive. The extensions of our current work include the following: (1) Design communication-efficient (or preferably communication-optimal) BSP algorithms for other fundamental problems, and provide techniques for worst-case and average-case efficiency analyses in both computation and communication; (2) Design BSP algorithms, taking into consideration that some processing nodes may become corrupt, and analyze the efficiency under such circumstances; and (3) Design I/O-efficient BSP algorithms, taking into consideration the memory hierarchy (main memory and storage disks, for example) of each processor, and analyze the scalability under such circumstances.

Section 6.1 presents another fundamental problem in parallel computing: the *General Prefix Computation* (GPC, [93]). Deriving a communication-efficient algorithm for this problem will imply practical parallel algorithms for the various applications it has. Section 6.2 discusses the issue of fault tolerance on BSP by summarizing current results and potential future research work. Section 6.3 addresses issues of parallel algorithms in external memory on a variation of BSP, the EM-BSP* model, which captures the ideas of memory hierarchy of the processors in some parameters used in the algorithm design.

6.1 General Parallel Prefix Computation

General Prefix Computation (GPC, [93]) is a generic problem component that captures the most common, difficult kernel of many types of problems. The definition is as follows: Let $\{f(1), f(2), \cdots, f(n)\}$ and $\{y(1), y(2), \cdots, y(n)\}$ be two sequences of elements with a binary associative operator $*$ defined on sequence f and a linear order $\prec$ defined on sequence y. It is required to compute the sequence $\{D(1), D(2), \cdots, D(n)\}$ whose elements $D(m)$, $m = 1, 2, \cdots, n$ are defined as $D(m) = f(j_1) * f(j_2) * \cdots * f(j_k)$, where $j_1 < j_2 < \cdots < j_k$ and $\{j_1, j_2, \cdots, j_k\}$ is the sequence of indices such that $j_i < m$ and $y(j_i) \prec y(m)$ for $i = 1, 2, \cdots, k$. We are interested in designing a computation and communication optimal (or, at least, efficient) BSP algorithm for solving GPC.

A typical example using the GPC algorithm as a subroutine is the *range searching* problem in large databases [5]. The range searching problem is defined as follows: Let $Q = \{q_1, q_2, \cdots, q_n\}$ be a finite sequence of ordered pairs of real numbers. Each entry q_i = (x_i, y_i) of Q can be viewed as Cartesian coordinates of a point in the plane. A rectangle G is given whose sides are parallel to the axes of the coordinates; thus, G is defined by the two intervals $[a, b]$ and $[c, d]$ on the x and y axes, respectively. For some functions f and a binary operator $*$, it is required to compute the quantity $f(i) * f(j) * \cdots * f(k)$, where $q_i, q_j, \cdots, q_k$ are those elements of Q which fall inside G. More generally, the elements of Q are ordered tuples of the form $(x, y, \cdots, z)$, and it is required to find those elements whose components $x, y, \cdots, z$, fall within given ranges specified in the query $G = \{[a,b],[c,d],\cdots,[e,f]\}$; that is, $a \leq x \leq b, c \leq y \leq d, e \leq z \leq f$.

It has been shown that GPC is applicable to a wide variety of geometric (point set and tree) problems [93], including triangulation of point sets, two-set dominance counting, ECDF searching, finding two- and three-dimensional maximal points, and the (classical) reconstruction of trees from their traversals. Therefore, providing a portable parallel algorithm with predictable communication efficiency for GPC becomes an essential step for solving these applications on practical parallel computing platforms.

6.2 Fault Tolerance on BSP

The fault tolerance research on the BSP model tries to capture situations where some nodes of a target parallel machine become corrupt (become unavailable of stall) during the execution of a BSP algorithm. The goal is to exploit the paradigm of the BSP and the limited synchrony that it assumes, to model fast networks of processing elements (e.g. networks of workstations, [28, 9]), where a workstation may be stalled by a user, may be shutdown, or stalled because of overload. This implies that the faults may occur at any time (*i.e.* they are dynamic), while the input algorithm is suitably expressed as a BSP program.

The BSP model is a purely message passing model, considering only local memory modules, and this is why some very advanced heuristics that exploit the characteristics of shared memory for providing fault tolerance [63, 64, 58, 59, 25] are not efficient in this setting. On the other hand, BSP is not a distributed memory model either, but rather a general purpose bridging model that tries to compromise the local computation cost with the communication throughput of the underlying network infrastructure, taking into account features such as the available bandwidth and the periodicity of the synchronization operations. Therefore, the well-behaving fault tolerant strategies on distributed networks [80, 28, 39] can not exploit the limited asynchrony provided by a BSP algorithm, and thus they are not efficiently applicable in this setting.

In [66], the issue of fault tolerance on BSP computations has been addressed. Simulations for two different cases were considered: The static case, where the faulty or unavailable processors are already known at the beginning of the computation and no processor changes its status afterwards, and a semi-dynamic case, where each processor may fail or become unavailable with a fixed probability during the computation and remains so until the end of the computation - however there were some critical periods during the computation where no processor was allowed to fail. Indeed, the existence of these fault-free periods during the simulation prevented the application of that result in a setting where hardware (processor) faults may dynamically occur. Additionally, the simulations provided in [66] used a Monte Carlo strategy that had a polynomially small failure probability.

Generally speaking, we need to consider fault tolerant BSP computations under "fully dynamic processor faults" without considering any fault-free periods. Namely, the faults may happen on-line at any point of the computation. To tackle the problem, the issue of the fault tolerance on BSP is usually modeled as an independent job scheduling problem, on a dynamically changing working environment. More specifically, consider having a BSP algorithm that is designed for an ideal, fault-free p-processor BSP machine. Each virtual superstep of this algorithm may be thought of as a set of p independent computational threads that impose some communication demands (*i.e.* the implementation of an h-relation among these threads at the end of each superstep), and correspond to the work to be done by each virtual processor during the current virtual superstep. The required task is to assign this amount of work on a dynamically changing set of live processors, in such a way that, as long as there are at least $(1-\alpha)p$ live processors (α is an input parameter of the realistic setting that will be used), the amount of this work will be successfully executed. The goal is to choose an efficient strategy that will achieve an almost work-preserving, robust execution of the BSP algorithm, and will also assure a balanced split of the work load among the operational processors.

A robust, Las Vegas simulation strategy, which eventually leads to the completion of an input BSP algorithm as long as at least a specific number of the initially considered processors remain live, was proposed in [67]. The proposed strategy is a modular and efficient simulation scheme which, compared to an optimal off-line adversarial computation under the same sequence of fault occurrences, achieves an $O((\log p \cdot \log\log p)^2)$-factor times the optimal work. The proposed scheme is Las Vegas, *i.e.* it never fails to complete the computation completely. This is so, due to a *backtracking* process, which retrieves robustly stored instances of the simulation process in case of an interruption to the flow of the computation has occurred, due to locally unrecoverable situations. Moreover, the storage schemes adopted in the simulation achieve space optimality, which is very crucial in the BSP cost model since space overhead is interpreted as communication overhead, when a virtual work load has to migrate to a new live processor (probably because of a new death). Rabin's *Information Dispersal Algorithm* [81] is used in [67] to achieve the space optimality.

Yet, there remain several open questions that arise through fault tolerance issues on BSP. A typical one is the design of efficient simulation strategy, considering a dynamically changing (by means of unreliable nodes or links) network that is continuously imposed computational threads, and the convergence of such a system to a stable state (if it ever converges). Another intriguing issue would be the tighter competitive analysis of such a simulation strategy that would indicate the actual behavior of a simulation strategy over a dynamically changing environment.

6.3 I/O-Efficient Algorithms

In recent years, computer processor speeds have increased dramatically, computer storage device capacities have increased, prices have dropped, and applications have grown in their demands on storage resources. However, disk speeds have not kept pace with main memory speeds. The bottleneck between main memory and "external memory" (EM) devices, such as disks, threatens to limit the size of very large scale applications [86]. While internal memories have also become much more reliable, cheaper, and larger, we see many new and up-sized applications that demand more storage than is feasible through internal memories. The area of external memory (EM) algorithm design aims to create algorithms which perform efficiently when the internal memory of the computer is significantly smaller than the storage required for the application data. Such applications require the bulk of their data to reside in external memory since, for practical machines, main memory is only a fraction of the required size.

Applications in geographic information systems (GIS), astrophysics, virtual reality, computational biology, VLSI design, weather prediction, computerized medical treatment, 3D simulation and modeling, visualization and computational geometry fall naturally into this category. Parallel disk I/O and parallel computing have been identified as critical components of a suitable high performance computer.

The time required to perform an I/O operation is typically several orders of magnitude larger than the time for an internal CPU operation. In addition, the time to set up the data transfer (disk head movement, rotational delay) is much larger than the time to actually transfer the data. Data items are grouped into blocks and accessed blockwise by efficient external memory algorithms in order to amortize the setup time over a large number of data items. Algorithms developed originally for internal memory models, such as the RAM or PRAM models, are frequently found to be inefficient in the EM environment for this reason. Such algorithms typically have poor "locality of reference", causing too many I/O operations to be performed during the execution of an application.

Prior to the evolution of a distinct EM methodology, large problems were often handled via virtual memory techniques, which used demand paging to swap pages between disks and internal memory. Unfortunately such techniques are heuristic and the performance can be very poor in difficult cases. As a result, researchers have developed special algorithms optimized for EM computation.

We have been interested in designing I/O-efficient external memory algorithms, using the EM-BSP* model as the design platform, especially for problems which are generally considered to be irregularly structured or communication-intensive.

Bibliography

[1] ADLER, M., BYERS, J. W., AND KARP, R. M. Parallel Sorting With Limited Bandwidth. In *Proc. ACM Symposium on Parallel Algorithms and Architectures* (1995), pp. 129–136.

[2] AGGARWAL, A., CHAZELLE, B., C.O'DUNLAING, L. G., AND YAP, C. Parallel Computational Geometry. *Algorithmica 3* (1988), 293–327.

[3] AGGARWAL, A., CHAZELLE, B., GUIBAS, L., O'DUNLAING, C., AND YAP, C. Parallel Computational Geometry. In *Proc. 26th IEEE Annual Symposium on Foundations of Computer Science* (1985).

[4] AKL, S. Optimal Parallel Algorithms for Computing Convex Hulls and for Sorting. *Computing 33* (1984), 1–11.

[5] AKL, S. G. *Parallel Computation, Models and Methods*. Prentice Hall, 1997.

[6] ALEXANDRAKIS, A. G., GERBESSIOTIS, A. V., LECOMBER, D. S., AND SINIOLAKIS, C. J. Bandwidth, Space and Computation Efficient PRAM Programming: The BSP Approach. In *Proceedings of SUPEUR'96, Krakow* (1996).

[7] AMIR, A., LANDAU, G. M., AND VISHKIN, U. Efficient Pattern Matching with Scaling. In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 344–357.

[8] ATALLAH, M. J., AND CHEN, D. Z. Parallel Geometric Algorithms in Coarse-Grained Network Models. In *Proc. 4th Annual International Computing and Combinatorics Conference(COCOON)* (1998), pp. 55–64.

[9] AUMANN, Y., KEDEM, Z., PALEM, K., AND RABIN, M. Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs. In *Proc. 34th Annual Symposium on Foundations of Computer Science* (1993).

[10] B.A. CHALMERS AND S.G. AKL. Optimal Parallel Algorithms for a Transportation Problem. In *Proceedings of the Canadian Conference on Electrical and Computing Engineering* (1991), pp. 36.1.1–36.1.6.

[11] BEINEKE, L., AND WILSON, R. *Selected Topics in Graph Theory*. Academic Press, New York/London, 1978.

[12] BELLOCH, G. E., LEISERSON, C. E., MAGGS, B. M., PLAXTON, C. G., SMITH, S. J., AND ZAGHA, M. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proc. ACM Symposium on Parallel Algorithms and Architectures* (1991), pp. 3–16.

[13] BERKMAN, O., SCHIEBER, B., AND VISHKIN, U. Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. *Journal of Algorithms 14* (1993), 344–370.

[14] BERKMAN, O., AND VISHKIN, U. Recursive *-tree Parallel Data Structures. In *Proc. 30th IEEE Symposium on Foundations of Computer Science* (1989), pp. 196–202.

[15] BERTOSSI, A. A., AND BONUCCELLI, M. A. Some Parallel Algorithms on Interval Graphs. *Discrete Applied Mathematics 16* (1987), 101–111.

[16] BÃ̈UMKER, A., AND DITTRICH, W. Fully Dynamic Search Trees for an Extension of the BSP Model. In *Proc. of ACM Symposium on Parallel and Distributed Computing* (1996), pp. 233–242.

[17] BÃ̈UMKER, A., DITTRICH, W., AND HEIDE, F. M. A. D. Efficient Parallel Algorithms: c-optimal multisearch for an Extension of the BSP Model. In *Proc. Annual European Symposium on Algorithms* (1995), pp. 17–30.

[18] BÃ̈UMKER, A., DITTRICH, W., AND PIETRACAPRINA, A. The Deterministic Complexity of Parallel Multisearch. In *Proc. Scandinavian Workshop on Algorithms Theory* (1996), pp. 404–415.

[19] BONDY, J. A., AND MURTY, U. S. R. *Graph Theory with Applications*. The Macmillan Press Limited, 1976.

[20] BORODIN, A., AND HOPCROFT, J. Routing, Merging and Sorting on Parallel Models of Computation. *J. Comput. System Sci. 30* (1985), 130–145.

[21] BOXER, L., MILLER, R., AND RAU-CHAPLIN, A. Some Scalable Parallel Algorithms for Geometric Problems. In *Proc. 8th International Conference on Parallel and Distributed Computing and Systems (PDCS'96), Chicago* (1996).

[22] BRESLAUER, D., AND GALIL, Z. An Optimal $O(\log\log n)$ Time Parallel String Matching Algorithm. *SIAM J. Computing 19* (1990), 1050–1058.

[23] CACERES, E., DEHNE, F., FERREIRA, A., FLOCCHINI, P., RIEPING, I., RONCATO, A., SANTORO, N., AND SONG, S. W. Efficient Parallel Graph Algorithms for Coarse Grained Multicomputers and BSP. In *Proc. 24th International Colloquium on Automata, Languages and Programming* (1997), pp. 390–400.

[24] CHEN, Z.-Z., HE, X., AND HUANG, C.-H. Finding Double Euler Trails of Planar Graphs in Linear Time. *SIAM Journal on Computing* (to appear), Preliminary version in *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science* (*FOCS*),319–329,New York City,1999.

[25] CHLEBUS, B., GASIENIEC, L., AND PELC, A. Fast Deterministic Simulation of Computations on Faulty Parallel Machines. In *Proc. 3rd Annual European Symposium on Algorithms Springer Verlag LNCS 979* (1995), pp. 89–101.

[26] CORMEN, T. H., AND GOODRICH, M. T. Working Group on Storage I/O for Large-Scale Computing. In *Proc. ACM Workshop on Strategic Directions in Computing Research* (1996).

[27] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. McGraw-Hill, 2000.

[28] DASGUPTA, P., KEDEM, Z., AND RABIN, M. Parallel Processing on Networks of Workstations. In *Proc. 15th International Conference on Distributed Systems* (1995).

[29] DEHNE, F., DITTRICH, W., AND HUTCHINSON, D. Efficient External Memory Algorithms by Simulating Coarse-Grained Parallel Algorithms. In *Proc. ACM Symposium on Parallel Algorithms and Architectures* (1997), pp. 106–115.

[30] DEHNE, F., FABRI, A., AND RAU-CHAPLIN, A. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers. In *Proc. 9th ACM Annual Computational Geometry* (1993), pp. 298–307.

[31] DEHNE, F., FABRI, A., AND RAU-CHAPLIN, A. Scalable Parallel Computational Geometry for Coarse Grained Multicomputers. *International Journal on Computational Geometry and Applications 6*, 3 (1996), 379–400.

[32] DEHNE, F., AND GOETZ, S. Practical Parallel Algorithms for Minimum Spanning Trees. In *Proc. 17th IEEE Symp. on Reliable Distributed Systems, Advances in Parallel and Distributed Systems* (1998), pp. 366–371.

[33] DUNNE, P. E. *The Complexity of Boolean Networks*. Academic Press, 1988.

[34] EPPSTEIN, D., AND GALIL, Z. Parallel Algorithmic Techniques for Combinatorial Computation. *Ann. Rev. Comput. Sci. 3* (1988), 233–283.

[35] FOSTER, I., AND KESSELMAN, C. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications 11*, 2 (1997), 115–128.

[36] FOURNIER, A., AND MONTUNO, D. Y. Triangulating Simple Polygons and Equivalent Problems. *ACM Transactions on Graphics 3* (1984), 153–174.

[37] FREEMAN, T., AND PHILLIPS, C. *Parallel Numerical Algorithms*. Prentice Hall International, London, 1992.

[38] GABOW, H. N., BENTLEY, J. L., AND TARJAN, R. E. Scaling and Related Techniques for Geometry Problems. In *Proc. 16th ACM Symposium on Theory of Computing* (1984), pp. 135–143.

[39] GALIL, Z., MAYER, A., AND YUNG, M. Resolving Message Complexity of Byzantine Agreement and Beyond. In *Proc. 36th IEEE Annual Symposium on Foundations of Computer Science* (1995,).

[40] GAREY, M. R., JOHNSON, D. S., PREPARATA, F. P., AND TARJAN, R. E. Triangulating a Simple Polygon. *Information Processing Letters 7* (1978), 175–179.

[41] GERBESSIOTIS, A., AND SINIOLAKIS, C. Communication Efficient Data Structures on the BSP model with Applications. Tech. rep., Oxford University Technical Report PRG-TR-13-96, 1996.

[42] GERBESSIOTIS, A. V., AND VALIANT, L. G. Direct Bulk-Synchronous Parallel Algorithms. Tech. rep., TR10-92, Computer Science Department, Harvard University, 1992.

[43] GERBESSIOTIS, A. V., AND VALIANT, L. G. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing 22* (1994), 251–267.

[44] GIBBONS, A., AND RYTTER, W. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[45] GOLUMBIC, M. C. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 1980.

[46] GOODRICH, M. T. Triangulating a Polygon in Parallel. *Journal of Algorithms 10* (1989), 327–351.

[47] GOODRICH, M. T. Communication-Efficient Parallel Sorting. In *Proc. 28th ACM Symp. on Theory of Computing* (1996), pp. 247–256.

[48] GUPTA, U. I., LEE, D. T., AND LEUNG, Y. Y.-T. Efficient Algorithms for Interval Graphs and Circular-Arc Graphs. *Networks 12* (1982), 459–467.

[49] HE, X., AND HUANG, C.-H. Scalable Coarse Grained Parallel Interval Graph Algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas* (2000), pp. 1369–1375.

[50] HE, X., AND HUANG, C.-H. Communication Efficient BSP Algorithm for All Nearest Smaller Values Problem. *Journal of Parallel and Distributed Computing* (to appear).

[51] HILL, J. M. D., MCCOLL, B., STEFANESCU, D. C., GOUDREAU, M. W., LANG, K., RAO, S. B., SUEL, T., TSANTILAS, T., AND BISSELING, R. H. BSPlib: The BSP programming library. *Parallel Computing 24*, 14 (1998), 1947–1980.

[52] HILL, J. M. D., AND SKILLICORN, D. B. Practical Barrier Synchronization. In *Proceedings Sixth EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)* (Jan. 1998), IEEE Computer Society Press, pp. 438–444.

[53] HUANG, C.-H., AND HE, X. Communication-Efficient Bulk Synchronous Parallel Algorithm for Parentheses Matching. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, VA. unpaginated, 9 pages* (2001).

[54] HUANG, C.-H., AND HE, X. Finding Hamiltonian Paths in Tournaments on Clusters – A Provably Communication-Efficient Approach. In *Proceedings of the 16th ACM Symposium on Applied Computing, Las Vegas* (2001), pp. 549–553.

[55] HUANG, C.-H., AND HE, X. Parallel Range Searching in Large Databases Based on General Parallel Prefix Computation. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, VA. unpaginated, 3 pages* (2001).

[56] HWANG, K. *Advanced Computer Architecture – Parallelism, Scalability and Programmability*. McGraw-Hill, 1993.

[57] JÁJÁ, J. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[58] KANELLAKIS, P., AND SHVARTSMAN, A. Efficient Parallel Algorithms on Restartable Fail-Stop Processors. In *Proc. ACM Symposium on Principles of Distributed Computing* (1991), pp. 23–36.

[59] KANELLAKIS, P., AND SHVARTSMAN, A. Efficient Parallel Algorithms can be Made Robust. pp. 201–217.

[60] KARP, R. M., LUBY, M., AND AUF DER HEIDE, F. M. Efficient PRAM simulation on a Distributed Memory Machine. In *24th Annual ACM Symposium on Theory of Computing* (1992).

[61] KATAJAINEN, J. Finding All Nearest Smaller Values on a Distributed Memory Machine. In *Proc. of Conference on Computing: The Australian Theory Symposium* (1996), pp. 100–107.

[62] KEDEM, Z., LANDAU, G., AND PALEM, K. Optimal Parallel Prefix-Suffix Matching Algorithm And Applications. In *Proceedings 1st ACM Symposium on Parallel Algorithms and Architectures* (1989), pp. 388–398.

[63] KEDEM, Z., PALEM, K., RAGHUNATHAN, A., AND SPIRAKIS, P. Combining Tentative and Definite Executions for Very Fast Dependable Parallel Computing. In *Proc. ACM Symposium on Theory of Computing* (1991), pp. 381–390.

[64] KEDEM, Z., PALEM, K., AND SPIRAKIS, P. Efficient Robust Parallel Computations. In *Proc. ACM Symposium on Theory of Computing* (1990), pp. 138–148.

[65] KEIL, J. M. Finding Hamiltonian Circuits in Interval Graphs. *Information Processing Letters 20* (1985), 201–206.

[66] KONTOGIANNIS, S., PANTZIOU, G., AND SPIRAKIS, P. Efficient Computations on Fault-Prone BSP Machines. In *Proc. ACM Symposium on Parallel Algorithms and Architectures* (1997).

[67] KONTOGIANNIS, S., PANTZIOU, G., SPIRAKIS, P., AND YUNG, M. Dynamic-Fault-Prone BSP: A Paradigm for Robust Computations in Changing Environments. In *Proc. ACM Symposium on Parallel Algorithms and Architectures* (1998).

[68] KRAVETS, D., AND PLAXTON, C. G. All Nearest Smaller Values on the Hypercube. *IEEE Transactions on Parallel and Distributed Systems 7*, 5 (1996), 456–462.

[69] KRUSKAL, C. Searching, Merging and Sorting in Parallel Computation. *IEEE Trans. Comput. C-32* (1983), 942–946.

[70] LAKSHMIVARAHAN, S., AND DHALL, S. *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems*. McGraw-Hill, 1990.

[71] LEE, D. T., AND PREPARATA, F. P. The All Nearest Neighbor Problem for Convex Polygons. *Information Processing Letters 7* (1978), 189–192.

[72] LI, H., AND SEVICK, K. C. Parallel Sorting by Overpartitioning. In *Proc. ACM Symposium on Parallel Algorithms and Architectures* (1994), pp. 46–56.

[73] MCCOLL, W. F. Special Purpose Parallel Computing. In *ALCOM Spring School on Parallel Computing, Cambridge International Series on Parallel Computation. Cambridge University Press.* (1991).

[74] MCCOLL, W. F. Scalable Computing. In *Computer Science Today: Recent Trends and Developments* (1995), J. van Leeuwen, Ed., vol. 1000 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 46–61.

[75] MEIJER, H., AND AKL, S. G. Optimal Computation of Prefix Sums on a Binary Tree of Processors. *International Journal of Parallel Programming 16*, 2 (1987), 127–136.

[76] MILLER, R., GALLO, S. M., KHALAK, H. G., AND WEEKS, C. M. *SnB*: Crystal Structure Determination via *Shake-and-Bake*. *Journal of Applied Crystallography 27* (1994), 613–621.

[77] OLARIU, S. An Optimal Greedy Heuristic to Color Interval Graphs. *Information Processing Letters 37* (1991), 21–25.

[78] PACHECO, P. S. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[79] PREPARATA, F. P., AND SHAMOS, M. I. *Computational Geometry: An Introduction*. Springer, New York, 1985.

[80] PRISO, R. D., MAYER, A., AND YUNG, M. Time-Optimal Message-Efficient Work Performance in the Presence of Faults. In *Proc. ACM Symposium of Distributed Computing* (1994).

[81] RABIN, M. Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance. *Journal of the Association for Computing Machinery 36*, 2 (1989), 335–348.

[82] RAMACHANDRAN, V. L., AND VISHKIN, U. Efficient Parallel Triconnectivity in Logarithmic Parallel Time. In *Proc. AWOC 88, Lecture Notes in Computer Science Vol. 319, Springer-Verlag* (1988), pp. 33–42.

[83] RAMALINGAM, G., AND RANGAN, C. P. New Sequential and Parallel Algorithms for Interval Graph Recognition. *Information Processing Letters 34* (1990), 215–219.

[84] REDEI, L. Ein Kombinatorischer Satz. *Acta Litt. Sci. Szeged 7* (1934), 39–43.

[85] REIF, J. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[86] RUEMMLER, C., AND WILKES, J. An Introduction to Disk Drive Modeling. *IEEE Computer 27*, 3 (1976), 17–28.

[87] SAXENA, S., AND RAO, N. M. Parallel Algorithms for Connectivity problems on Interval Graphs. *Information Processing Letters 56* (1995), 37–44.

[88] SCHIEBER, B., AND VISHKIN, U. Finding All Nearest Neighbors for Convex Polygons in Parallel: A New Lower Bound Technique and A Matching Algorithm. *Discrete App. Math. 29* (1990), 97–111.

[89] SHIEBER, B., AND VISHKIN, U. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM Journal on Computing 17* (1988), 1253–1262.

[90] SKILLICORN, D., HILL, J. M., AND MCCOLL, W. Questions and Answers About BSP. *Oxford University Computing Laboratory* (1996).

[91] SOROKER, D. Fast Parallel Algorithms for Finding Hamiltonian Paths and Cycles in a Tournament. *Journal of Algorithms 9* (1988), 276–286.

[92] SPRAGUE, A. P. Optimal Parallel Algorithms for Finding Cut Vertices and Bridges of Interval Graphs. *Information Processing Letters 42* (1992), 229–234.

[93] SPRINGSTEEL, F., AND STOJMENOVIC, I. Parallel General Prefix Computations with Geometric, Algebraic, and Other Applications. *International Journal of Parallel Programming 18*, 6 (1989), 485–503.

[94] TARJAN, R. E., AND VISHKIN, U. An Efficient Parallel Biconnectivity Algorithm. *SIAM Journal on Computing 14*, 4 (1985), 862–874.

[95] VALIANT, L. G. A Bridging Model for Parallel Computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[96] VISHKIN, U. Optimal Parallel Pattern Matching in Strings. *Information and Computation 67* (1985), 91–113.

[97] VISHKIN, U. Structural Parallel Algorithmics. In *18th International Colloquium on Automata, Languages and Programming, LNCS Vol. 510, Springer-Verlag* (1991), pp. 363–380.

[98] VUILLEMIN, J. A Unified Look at Data Structures. *Communications of the ACM 23* (1980), 229–239.

[99] YANG, C. C., AND LEE, D. T. A Note on the All Nearest Neighbor Problem for Convex Polygons. *Information Processing Letters 8* (1979), 193–194.

[100] YU, M.-S., AND YANG, C.-H. An Optimal Parallel Algorithm for the Domatic Partition Problem on an Interval Graph Given its sorted Model. *Information Processing Letters 44* (1992), 15–22.

[101] YU, M.-S., AND YANG, C.-H. A Simple Optimal Algorithm for the minimum Coloring Problem on Interval Graphs. *Information Processing Letters 48* (1993), 47–51.