PARALLEL DYNAMIC PROGRAMMING

by

Phillip Gnassi Bradford

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

December 15, 1994

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Gregory J. E. Rawlins, Ph.D.
(Principal Adviser)

Paul W. Purdom, Jr., Ph.D.

Edward L. Robertson, Ph.D.

Larry S. Moss, Ph.D.

December 15, 1994

DEDICATION

To my family and friends.

Acknowledgements

On to thanking those who had much influence on me and therefore on this work.

> "Never doubt that a small group of thoughtful, committed citizens can change the world. Indeed, it is the only thing that ever has." —Margaret Mead

First, I sincerely thank Gregory J. E. Rawlins, my advisor. His advice has always been solid and prudent. He has stood by me, time and time again and has had the faith and courage to believe in me—even when I didn't take his excellent advice. His non-conventional methods are refreshing and gave me the free reign I really enjoy. His great writing is something I will always try to emulate. His subtle intensity, great sense of humor, sharp insights, and his friendship are all great!

It is a pleasure to work with Paul Purdom. Paul has taught me a great deal. All projects I have seen him work on, he always does more than his share of the work. His relentless search for the truth and knowledge is certainly refreshing. Further, I really enjoy his congeniality, hospitality and good nature.

Larry Moss helped in a multitude of ways. His understanding of me, mathematics and computer science made this work much easier and definitely very fun. Larry's open mindedness, his love of theory, his devotion to hard work and his generosity towards people sparkles in my mind.

Ed Robertson has been very generous by serving on my committee. Ed's sharp mind and generosity have certainly had much positive influence generally on the

Indiana's Computer Science Department and in particular on me. Further, he really cares about the community.

I sincerely thank Kurt Mehlhorn and the whole gang at the Max-Planck-Institut für Informatik for the wonderful research environment they have introduced me to in Saarbrücken.

Larry Larmore of the University of Nevada at Las Vegas has been a great mentor and has stood by me in times of need. He is a true scholar who has given me much needed advice, etc. As with many famous researchers I recall meeting Larry first through his prolific writings. Then later, when I got to know "Larry the person," as opposed to "Lawrence L. Larmore" the researcher, I was even more impressed!

Mike Atallah of Purdue University was always a pleasure to talk to and is a star in my eyes. He is a true team player and a very great person whose pursuit of knowledge is certainly worthy of substantial note. He also has a very clear high-level perspective. He has given me excellent advice and his eagerness to know what I will do in the future has been extremely encouraging.

Mike Loui of University of Illinois has always had an extremely positive influence on me and my work. His ever positive attitude and great encouragement of me and my work has been extremely invigorating and delightful. He has also been very eager to see what I do in the future—which is quite encouraging to me. I have always really enjoyed talking to him at conferences all over the country.

T. C. Hu has already had an profound influence on my work. He has been very generous to me, especially when I visited him at the University of California at San Diego. He gave me a "tour of the town" and of the University. We also met at the University of Wisconsin at Madison for a conference on optimization where he was equally generous. I recall being asked by his chairman to write a letter for his promotion to distinguished professor. Later I asked T. C. who else was writing such letters for him. He replied something like: "The 3 Ks: Knuth, Karp and Kleitman."

Knowing T. C. certainly put me in great company!

Tom Spencer of the University of Nebraska has been extremely pleasant to talk to and to discuss theory with. I have always looked forward to seeing Tom at various conferences.

Ming Kao of Duke University has been kind and generous to me on several occasions. Ming has given me very good advice and has been a good friend since I met him. He certainly is a person with high standards.

I first met Dan Friedman through reading his book "The Little Lisper" many years ago when I was in college. One of the reasons I thought so highly about Indiana University was because I long knew that Dan and several other "Scheme-rs" were here. I recall spending a very pleasant Saturday afternoon at his house in October of 1990 that solidified my decision to come to IU. This pleasant afternoon made a very significant impression on me that will last for many years to come.

Dirk Van Gucht has always showed interest in me and my work. I have had many very pleasant conversations with Dirk and I really appreciate his excellent influence.

Andy Hanson has also always showed interest in me and my work. It is always nice to talk with Andy. Andy is one of those people who is comfortable working in physics, mathematics, computer science and probably a few other areas to boot. His great depth and breadth is something only the best scholars can shoot for.

Daniel Leivant has been very generous and pleasant while showing an interest in me. His high standard for excellence is something I will always try to emulate.

Several times David Wise has given me advice that is solid and on the money. I appreciate this greatly.

Steve Johnson has also given me advice that is solid and on the money. Now and then, Steve has also written a useful memos for me, which made things a lot easier. I appreciate all of this greatly.

Pete Shirley has been everything from a graphics-lab buddy to one of my best advisors ever. I really hope we cross paths again soon! Together, Pete and Jean certainly make two great friends.

Randall Bramley has always been very interesting to talk to and get advice from. He works very hard and his high ethical standards and his intellectual momentum are superb.

Greg Shannon went way beyond the call of duty and helped me in many ways. One of the central reasons I worked on a Ph.D. at Indiana University is because of Greg Shannon, and I am greatful for this. He was my first advisor at Indiana.

Alok Aggarwal of IBM Research Labs at Yorktown Heights has always been friendly and interesting to talk to. Talking with Alok has always been enjoyable and encouraging.

Zamir Bavel of the University of Kansas is a very good teacher and a fine scholar.

I must thank the many participants in the “Thursday Theory Thing.” They ranged from computer science theory types to several people from the Biology Department. We always had lots of fun and I'm happy to have organized this group. I hope it continues long after I leave.

I sincerely thanks Pam Larson for all of her work in helping me get through the paperwork for my degree. Pam has certainly helped me out to a great extent at several very important junctures.

John MacCuish has really made my day many times. He is a great friend who really helped with many things in lots of ways. We have also shared many hearty laughs together. John's insights and perspective have really been wonderful. I'll never forget the cold winter nights with John, Kay, Emily in her cat suit, and, of course, me in my great innocence. There also was the time we bumped into a friend of ours in Bloomington wearing his 18-th century French wig and packing a pistol in a coffee shop.

I was particularly happy that he could schedule several things in Bloomington so he could be around when I defended this dissertation.

Sushil Louis was always supportive, has a great positive attitude and read through my work and gave me comments that improved it greatly. His help and perspective are lasting. Suresh Srinivas was a great help and a good neighbor. Jiyoung Chang was always encouraging and always had pleasant things to say. Ken Chiu has helped with this work in many ways both as a friend and graphics lab comrade. He also read countless things for me always adding quality and conciseness to their presentation.

Shankar Swamy has been a great friend. His hard work and dedication is excellent and to be admired. He has come through on several occasions that have made a very big difference. It was Shankar who generously helped with the last details of this dissertation while I was in Germany. Shankar is sometimes too generous. Shankar and I have had many great times together and I anticipate many more.

I should also thank all those students who were in classes that the department gave me the privilege of teaching. In particular, there was three classes in the Design and Analysis of Algorithms and a class in Data Structures. Teaching these classes was a great learning experience in many ways.

Of particular note are the following people: Jeff Bass, Wenfang Chang, Gordon Diamant, Keith Maull, Joe Povelari, Beata Winnicka, John Zuckerman.

I owe many thanks to my mother, my father, Camille, aunt A, uncle Charles, aunt Barbara, uncle Paul, Anthony, Ron, John, Suzanne, Anne, Syd, Teresa, Fred, Alex, and all of the rest of my large extended family. And, of course, who could ever forget Ota Benga.

Andrea Rafael has helped with this work in several ways. First, several times she read several drafts of papers I wrote and gave me excellent comments on them. Further, her support and friendship has certainly been delightful.

Emily Nedell has always been lots of fun. She is very well-rounded and is very pleasant to be around.

Jenni McDaniel has been a great friend and more almost since my arrival at IU. The extra efforts she often exerted will always be remembered. I recall spending many pleasant times with Jenni.

Jean-Yves and Cecil Marion have always been lots of fun. Perhaps, too much (unbearable[1]) fun! I recall a party JY and Cecil had in Bloomington where about eight of us were doing shots of Pete Shirley's "Lizard Juice." I also recollect many other great times with JY and Cecil. Now that I am in Saarbrücken Germany and they are conveniently located in Nancy Frence, the fun continues.

Neil Haven has always helped out in many regards. He has been very helpful on several key occasions. Neil has been an inspiration for me through his everlasting hard work and dedication. He still knows how to have fun, while excelling in his academic work and at the same time running a computer vision company.

Mike Wollowski has always been a great friend. I have enjoyed many times when we studied together, had dinner, picnicked, watched movies, etc. It's only rock-and-roll, but I like it.

Kate Ksiazek has always been fun to talk with. Her broad knowledge base is something to be admired. I have spent many pleasant times with Kate.

Tom Loos has helped me a lot. His calmness and solid perspective do the world wonders. I recall on several different times talking with Tom 'till early hours of the morning. He was always generous with his time and perspective. Emily is great too!

I have shared many warm laughs with Venkatesh Choppella. Venk is always fun to be around.

Yue-Herng Lin has always been lots of fun. Y.-H.'s hard work and his perspective on life is to be admired.

[1] Thanks to Dimitri Gusev for introducing me to the notion of "unbearable fun."

Steve Ryner has also been lots of fun. Steve's autonomous character is great.

Raja Sooriamurthi is a great friend. Since I was in Bloomington, Raja has been helping me out. He is very warm and friendly—a great friend indeed!

I must thank the following people for helping me though the language requirement.

Neil Haven helped by giving me many practice exams and torturing himself by grading them. Further, he taught me lots of the grammar and idioms. Kate Ksiazek helped with grammar, translations, and by giving me practice exams. Dan Jacobson helped me translate some passages and taught me grammar and idioms. Paul Purdom helped in many ways, ranging from sending me email in German to encouraging me. Mike Wollowski helped me several times with translations and various other things from German. Of course, a few times he was guilty of trying to hand me large philosophical treatises in German. John Gnassi who on a few occasions corresponded with me in German.

It was Andrew Lenard who actually administrated some of my exams in German. He was always very pleasant and generous with his time and energy.

It was my uncle Charles, a psychiatrist, who I first heard say something like: "Why do you expect humans to be logical ?" Wow...I must say that taking this as an axiom makes my deduction about humans much more complete.

Many thanks everyone!

Abstract

Algorithm design paradigms are particularly useful for designing new and efficient algorithms. However, several sequential algorithm design paradigms seem to fail in the design of efficient parallel algorithms. This dissertation focuses on the dynamic programming paradigm, which until recently has only been used to design sequential algorithms.

A graph structure is given that allows the efficient parallel solution of some problems amenable to the dynamic programming paradigm. Using these graphs we show that dynamic programming is a viable parallel algorithm design paradigm. Several new parallel algorithms are given for two well-known optimization problems. First an approximation algorithm is given. Then an algorithm that works by finding shortest paths in special graphs is given. Finally, the last two and most efficient of these parallel algorithms are given. These algorithms work by using new and efficient techniques for exploiting monotonic problem constraints.

Contents

List of Figures

Chapter 1

Introduction

This chapter contains a brief introduction to this dissertation. It motivates the entire dissertation by discussing some of the new challenges of parallel computation. In doing so, it emphasizes the usefulness of algorithm design paradigms such as dynamic programming. The main focus of this dissertation is the efficient parallelization of certain classical problems which are amenable to the (sequential) dynamic programming paradigm.

1.1 Algorithm Design Paradigms

Parallel computers promise to solve many problems much faster than their sequential counterparts, but to realize this increase in speed some challenges must be overcome. This dissertation addresses the challenge of designing efficient parallel algorithms for problems that have elementary and efficient sequential dynamic programming solutions.

Algorithm design paradigms, such as divide and conquer, the greedy method, and dynamic programming often aid the design of efficient sequential algorithms. However, some sequential algorithm design paradigms may not lead to efficient parallel algorithms. This dissertation focuses on dynamic programming, which until recently has only been used to design sequential algorithms, and demonstrates how to use it to design efficient parallel algorithms for several well-known problems.

Employing new graph structures leads to the efficient parallel solution of some problems amenable to dynamic programming. Designing parallel algorithms with these graphs shows that dynamic programming is a viable parallel algorithm design paradigm. Several new parallel algorithms are given for two well-known optimization problems. After the appropriate background is given, this dissertation contains an efficient approximation algorithm and two algorithms that work by finding shortest paths in these special graphs. Next, the last two and most efficient of these parallel algorithms are given. The last of these algorithms is the most efficient parallel algorithm for solving these problems to date.

These algorithms work by using new and efficient parallel techniques for exploiting monotonicity. An optimal log-time algorithm for solving searching problems in special structured matrices will improve our algorithm to have the same work as the best sequential algorithms for solving these problems.

In the most general terms, this dissertation contributes to the dissection of certain problems into independent parallel components. These independent components run very quickly and taken together they solve the original problem. Dissecting a problem like this could lead to drastic speed up of many algorithms. On the other hand, all of this work is done on a particular parallel model which may, in the long run, contribute to the viability of this model.

1.2 Parallelism

Parallel algorithms are hard to build. There are many parallel architectures to consider, many theoretical parallel models and many factors such as memory conflicts and communication costs that are generally not encountered in sequential algorithm design. To focus on the cost of parallel algorithms in terms of their sequential counterparts, this dissertation chooses a theoretical parallel model called the PRAM, or Parallel Random Access Machine. The PRAM has a sequential counterpart, the RAM or Random Access Machine. The RAM model is extensively used for the design of sequential algorithms due to its correspondence with actual computers.

The PRAM allows us to abstract from available computer architectures and simultaneously design algorithms suitable for many parallel computers. In addition, the PRAM allows the design of parallel algorithms to focus on the inherent parallel or sequential nature of the problem at hand. It seems that any model stronger than a PRAM is too powerful and anything weaker is too restrictive for our purposes.

The *parallel computation hypothesis* basically states that a parallel model of computation is "reasonable" iff the parallel time to solve a problem on this parallel model is proportional to the sequential space on a "reasonable sequential model." At the same time, the total amount of work done is about the same.

On a PRAM the *parallel computation hypothesis* relates the parallel time it takes to solve a problem with the sequential space it takes to solve this problem on a RAM. The parallel computation hypothesis is a theorem for various parallel models, including variants of the PRAM, given that the RAM is a reasonable sequential model of computation.

The parallel computation hypothesis gives us a notion of inherently sequential problems. If a problem takes lots of space to solve sequentially, then it will take lots of time to solve in parallel. No known algorithm for inherently sequential problems speeds up significantly even with reasonable[1] numbers of additional processors. These inherently sequential problems are the bane of parallel algorithm design, particularly

[1]Here "reasonable" means polynomially bounded by the input size of the problem instance, see Chapter 2 for more details.

on the PRAM model. On the other hand, there are problems that have inherently parallel algorithms and some of these problems are the work-horses of parallel algorithm design. The results given in this dissertation use several new inherently parallel algorithms and their design paradigms. Therefore, these new parallel algorithm design paradigms are suitable for the design of parallel dynamic programming algorithms. (These parallel algorithm paradigms will be given as they are needed.) This means the dynamic programming paradigm is, in some sense, useful for designing parallel algorithms.

1.3 Dynamic Programming

Dynamic programming has become a work-horse in a number of areas. Transportation and optimization problems are routinely solved using dynamic programming. Also problems such as string editing (Cormen et al., 1990), context-free grammar recognition (Hopcroft and Ullman, 1979), and optimal static search tree construction (Baase, 1988; Cormen et al., 1990) have efficient sequential dynamic programming solutions. There is an efficient parallel algorithm for string editing (Apostolico et al., 1990), and there are good parallel algorithms for context-free grammar recognition (Klein and Reif, 1988; Rytter, 1988).

The dynamic programming solution of applied problems has a rich history. The first context-free grammar recognition algorithms were intractable because of their time costs. Then the problem of recognizing a context-free grammar became feasible due to elementary and small dynamic programming algorithms. Surprisingly, the good parallel algorithms for context-free grammar recognition are still quite complex.

The rich history, applications, and ease of sequential solution contribute to our interest in studying optimization problems. Further, the high cost of parallel algorithms for solving problems amenable to simple sequential dynamic programming solutions all contribute to our motivation for studying parallel dynamic programming.

The dynamic programming paradigm is based on the *principle of optimality*. This principle is that for a structure to be optimal all of its well-formed substructures must also be optimal. Hence, the dynamic programming paradigm is essentially a top-down design method. Conversely, the *greedy principle* basically is that if a substructure is optimal then it is part of some optimal superstructure. In some sense this is a bottom-up design method. The *lexicographical* greedy principle is when every optimal substructure is in some optimal superstructure and this substructure is built lexicographically into the superstructure. Many problems amenable to the lexicographical greedy principle seem to be inherently sequential (Anderson and Mayr, 1987; Anderson and Mayr, 1987a). This sequential algorithm design paradigm does not seem to give inherently parallel algorithms. On the other hand, this dissertation gives some extensions of the dynamic programming paradigm for designing efficient parallel algorithms.

The following problems will be addressed in this dissertation and they are representative of those to which dynamic programming is often applied, see for instance (Aho et al., 1974; Baase, 1988; Cormen et al., 1990):

- *matrix chain ordering problem* (MCOP): find an optimal way to multiply a chain of n matrices, where the matrices are pairwise compatible but of varying dimensions.

- *optimal convex polygon triangulation problem*: find an optimal triangularization of a convex polygon of n points given the following triangle cost metric: a triangle with node values t_1, t_2, and t_3 has cost $t_1 t_2 t_3$.

- *optimal binary search tree construction problem*: build a static binary search tree with minimal average lookup time given a totally ordered set of n elements and their access probabilities.

These three problems have $O(n^3)$ time sequential solutions with elementary dynamic programming algorithms, although there are faster, but more complex, algorithms for each of these problems. (The notation $O(n^3)$ will be defined in Chapter 2.)

The bulk of this dissertation focuses on the matrix chain ordering problem and the optimal convex polygon triangulation problem.

These problems are so prevalent in textbooks on algorithms and optimization that we refrain from listing them. Only the books that are cogent to the research in this dissertation are cited.

1.4 On the Origins of Dynamic Programming

The term "dynamic programming" came from economic modeling jargon. Just as in the term "linear programming" the word "programming" refers to economic planning rather than computer programming. Dynamic programming became an algorithm design paradigm. Linear programming was and remains a method of solving simultaneous linear equations under certain constraints.

The first book on dynamic programming is (Bellman, 1957). The authors of (Cormen et al., 1990) credit Bellman and his 1957 book as laying the foundations of modern dynamic programming.

Several important problems were first solved efficiently using dynamic programming algorithms. For example, the problem of parsing context-free grammars had no efficient solution until the CKY algorithm (Hopcroft and Ullman, 1979). The CKY algorithm is a simple and efficient dynamic programming algorithm.

1.5 The Structure of this Dissertation

Chapter 1 (this chapter) contains a brief introduction to this dissertation. Further, it motivates algorithm design paradigms, parallelism, and dynamic programming.

Chapter 2 contains the parallel model the rest of this dissertation relies on. It also contains formal notions of inherently parallel and inherently sequential problems.

Chapter 3 contains the classical sequential algorithms for solving the matrix chain ordering problem. Efficient parallel solutions to the matrix chain ordering problem are the main focus of this dissertation.

Chapter 4 contains the parallel graph model that this dissertation relies on. This model is given in full generality. A straightforward approach gives three almost identical $O(\lg^2 n)$ time and $n^6/\lg n$ processor algorithms for the three problems outlined in this chapter.

Chapter 5 contains a specialization of the graphs given in Chapter 4 that are suited for solving the matrix chain ordering problem. These specializations are also suitable for an optimal triangularization problem on convex polygons.

Chapter 6 contains a parallel approximation algorithm for the MCOP. This gives a solution to the MCOP that is within about 15% from optimal.

Chapter 7 contains an $\tilde{O}(n^3)$ work polylog-time algorithm for the MCOP. This algorithm is based on special properties of the MCOP-specific graphs.

Chapter 8 contains several progressively more efficient algorithms culminating with an $O(\lg^2 n)$-time and n-processor algorithm for the EREW PRAM model.

Chapter 9 contains conclusions and discusses further directions.

Chapter 2

Definitions and Foundations

This chapter contains a brief intuitive sketch of the PRAM model. Using this model basic notions of inherently parallel and inherently sequential problems are given through a brief introduction to the problem classes $\mathcal{NC}$ and $\mathcal{P}$-Complete. More details about all of the topics of this chapter can be found in (Gibbons and Rytter, 1988; Karp and Ramachandran, 1990; Johnson, 1990; Kumar et al., 1994; JáJá, 1992; Parberry, 1987; Reif, 1993).

2.1 The PRAM Model

In this dissertation the parallel model is the PRAM (see Figure 1). As stated in Chapter 1, PRAM is an acronym that stands for Parallel Random Access Machine. The PRAM is motivated in Chapter 1.

A PRAM has an unbounded shared common memory and polynomially many processors indexed from 1 to $p(n)$, where $p(n)$ is a polynomial in the size of the input n. Each processor has its own local memory, and each processor has the same instruction set as the single processor in the RAM model (Aho et al., 1974; Papadimitriou, 1994). All processors run the same program and each processor has access to its own index

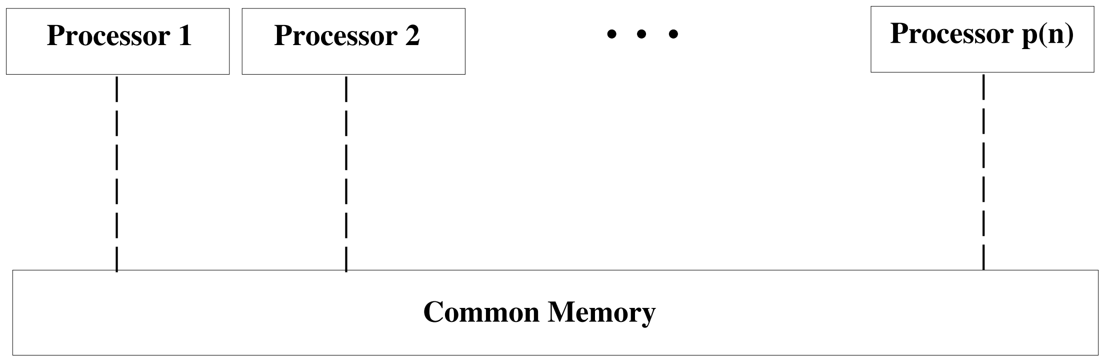

Figure 1: A PRAM with $p(n)$ Processors

so any processor can execute different instructions based on its index.

The PRAM model allows a polynomial number of processors to vary with the input size because:

- A polynomial number of polynomial time bounded processors can be simulated in polynomial time using one processor.

- The power of processors is increasing while their expense and size is decreasing. Therefore, designing algorithms expecting a large numbers of available processors is not unreasonable.

The common memory shared by all of the processors forces us to consider simultaneous memory conflicts. The most common distinctions between memory conflict are, exclusive reads (ER) versus concurrent reads (CR), and exclusive writes (EW) versus concurrent writes (CW). Therefore, the CRCW PRAM allows simultaneous reads and simultaneous writes, where the EREW PRAM does not allow simultaneous reads or simultaneous writes.

It's easy to imagine several processors simultaneously reading from the same memory location, but to allow simultaneous writes we must decide either which processor "wins" or how to combine the different values being simultaneously written.

Here are several well-known models of simultaneous write contention:

- Priority-CW PRAM: In this model each processor has some given priority, so memory contentions are resolved by letting the highest priority contesting processor win.
- Max-CW PRAM: In this model the processor that is attempting to write the maximum value in the contended memory location wins.
- Common-CW PRAM: In this model all processors must write the same value into the contended memory location.

All of the above models can simulate each other within minor time and space factors (Karp and Ramachandran, 1990). Because these parallel models are so closely related, we use the most convenient model at hand.

2.2 Efficient and Optimal Parallel Algorithms

The set $\mathcal{P}$ is the class of problems that have polynomial time bounded algorithms on a RAM. These problems are often thought to be tractable on modern computers (Garey and Johnson, 1979). Since a PRAM can only have a polynomial number of processors, a RAM can simulate a PRAM in polynomial time. This means the only problems that have any hope of being tractable on a PRAM are those that are tractable on a RAM. Therefore, we will examine the problems in $\mathcal{P}$ very closely.

2.2.1 Asymptotic Notation

Given an algorithm let I_n denote the set of all valid inputs of size n. The *time complexity* of an algorithm $\mathcal{A}$ is a function $T_{\mathcal{A}}$ from the set of all its inputs to the natural numbers, such that for all $i \in I_n$ the value $T_{\mathcal{A}}(i)$ is the number of steps algorithm $\mathcal{A}$ uses in computing an answer given input i. We write $t(n)$ to denote

$\max_{i \in I_n}\{ T_{\mathcal{A}}(i) \}$ and call this the *worst case* time complexity for inputs of size n. Time complexity reflects the amount of time it takes an algorithm to solve an instance of a problem.

The *space complexity* of an algorithm is defined analogously to its time complexity. Space complexity reflects the amount of space it takes an algorithm to solve an instance of a problem. Space complexity includes the space used for all data structures during a computation, but it does not include the space used for the input or the output. We write $s(n)$ to denote the *worst case* space complexity for inputs of size n.

The *processor complexity* of a parallel algorithm is defined analogously to space and time complexities. Processor complexity reflects the number of processors it takes a parallel algorithm to solve an instance of a problem. We write $p(n)$ to denote the *worst case* processor complexity for inputs of size n.

In this dissertation the processor, time, and space complexities are always polynomially bounded.

The worst case time complexity of a parallel algorithm is the worst case time it takes all processors to finish. The worst case parallel time complexity is also written $t(n)$ for inputs of size n.

The following definitions comprise asymptotic notation.

Here all functions[1] are from the positive integers onto the positive integers and all constants are positive.

Given two functions f and g, we write $f = O(g)$, if there are two constants c and d such that $f(n) \leq cg(n)$ for all $n \geq d$. We write $f = \Omega(g)$, if there are two constants c and d such that $f(n) \geq cg(n)$ for all $n \geq d$. We write $f = \Theta(g)$ when both $f = O(g)$ and $g = O(f)$.

Call a function f *polylog* iff $f(n) = \Theta(\lg^k n)$ for some constant $k > 0$. A function f is at most polylog iff $f(n) = O(\lg^k n)$ for some constant $k > 0$, etc. We will always

[1] See, for example (Marcus, 1978), for a definition of a function.

attempt to build parallel algorithms that work in polylog time. Further, all polylog and polynomial functions are in terms of the input size n.

We write $f(n) = \tilde{O}(g(n))$ iff $f(n) = O(g(n) \lg^k n)$ for some constant $k \geq 0$. The expressions $f(n) = \tilde{\Theta}(g(n))$ and $f(n) = \tilde{\Omega}(g(n))$ have the expected meanings.

The function f is within a polynomial of g iff $f(n) = O(g^k(n))$ for some constant $k \geq 1$.

2.2.2 Optimality and Work

Given a problem π, an algorithm is *optimal* iff there is no algorithm that can solve π with fewer operations. Likewise, given a problem π, an algorithm is *asymptotically* optimal iff there is no algorithm that can solve π with asymptotically fewer operations.

A problem π has time or space complexity f if there is no *known* algorithm that can solve π in better than f time or space, respectively.

For a given problem, a parallel algorithm's performance is often compared with the performance of a sequential algorithm that solves the same problem. The most common measure of performance of a parallel algorithm is the algorithm's *processor-time* product. This is the product of the processor complexity and the time complexity of a parallel algorithm. The processor-time product is sometimes called the *work* of a parallel algorithm. Comparing the work of a parallel algorithm with the work of a sequential algorithm is often done to within a polylog factor to make up for differences between the models. If the processor-time product of a parallel algorithm is significantly less than the time complexity of a sequential algorithm for the same problem, then simulating the parallel algorithm sequentially gives a new and more efficient sequential solution.

An *efficient* parallel algorithm runs in polylogarithmic time and has processor-time product within a polylog factor of the best known sequential solution. We allow the processor-time product to be within a polylog time factor of the best sequential solution because different variations of the PRAM model are equivalent to within log-time factors.

Suppose a problem has an asymptotically optimal sequential solution that costs f, then a polylog time parallel algorithm is asymptotically *optimal*, if its processor-time product is $O(f)$. There are other notions of optimality for parallel algorithms. A parallel algorithm has optimal speed if reducing its parallel time complexity forces its processor-time product to increase.

2.3 The Parallel Computation Hypothesis

A few technical details are omitted in the following statement of the parallel computation hypothesis since they are not germane to our discussion.

Assuming all time and space complexities are at least asymptotically logarithmic, the parallel computation hypothesis is:

A parallel model of computation is "reasonable" if the time complexity of a problem on this model is within a polynomial of the space complexity of the same problem on a (sequential) Turing machine, assuming the total work of the parallel and sequential solutions are within a polynomial of each other.

Figure 2: The Parallel Computation Hypothesis

This is a hypothesis since it assumes that a Turing machine is a "reasonable" sequential model. That is, this hypothesis defines a "reasonable" parallel model in terms of a "reasonable" sequential model. The Turing machine model is polynomially equivalent to the RAM model, (Aho et al., 1974; Papadimitriou, 1994). Making a few assumptions about the processor word-sizes gives the next theorem (Parberry, 1987),

The time complexity of a problem on a PRAM is polynomially equivalent to the space complexity of the same problem on a RAM.

As before, this assumes the amount of work on the PRAM is within a polynomial of the work on the RAM.

For many different parallel models, including variations of the PRAM model, the parallel computation hypothesis is a theorem assuming the RAM model is a "reasonable sequential model." In other words, given a polynomial number of processors a PRAM is a reasonable parallel model.

A *decision* problem is a problem that has only two possible solutions **True** or **False**. Decision problems free us from details such as the cost of writing the output.

The class $\mathcal{NC}$ contains all decision problems solvable in polylog time on a PRAM with a polynomial number of processors.

A problem is *inherently parallel* if it is in $\mathcal{NC}$. An algorithm is $\mathcal{NC}$ if it runs in polylog time on a PRAM using a polynomial number of processors.

It is unknown whether any problem in $\mathcal{P}$ requires polynomial space, although it seems likely that some do. Suppose some problems in $\mathcal{P}$ require polynomial space to run. Then by the parallel computation hypothesis these problems cannot run in polylog time using a polynomial number of processors. This is the basis of the theory behind inherently sequential problems.

If some problem π_1 has polynomial space complexity and there is a log-space transformation from π_1 to π_2, then π_2 is at least as hard as π_1 in terms of the space it uses. (All log-space algorithms run in polynomial time, see (Papadimitriou, 1994).) This is because any instance of π_1 can be solved by transforming it, in log-space, to an instance of π_2, and then by solving this instance of π_2 we have solved the initial instance of π_1. Therefore, if π_2 has log-space complexity, then π_1 must also have log-space complexity. On the other hand, say all problems in $\mathcal{P}$ can be log-space transformed to π_2, then if any problem in $\mathcal{P}$ requires polynomial space, then π_2 also requires polynomial space. Alternatively, if π_2 can be solved in polylog space, then all problems in $\mathcal{P}$ can be too. Of course, if a problem in $\mathcal{P}$ takes polynomial space to solve on a RAM, then by the parallel computation hypothesis it takes polynomial time to solve on a PRAM (with a polynomial number of processors).

Next is a sketch of how to show a problem is inherently sequential. Given two decision problems π_1 and π_2, the notation $\pi_1 \propto \pi_2$ means π_1 is *reducible* to π_2.

Saying π_1 is *reducible* to π_2 means that:

1. There is an algorithm α that transforms any instance of π_1 to an instance of π_2. And for any input to π_1, say $i \in I_n$, then $\alpha(i)$ is an input to π_2 of polynomial size in n.

2. If $\mathcal{A}$ solves π_1 and $\mathcal{B}$ solves π_2 then $\mathcal{A}(i)$ iff $\mathcal{B}(\alpha(i))$, since π_1 and π_2 are decision problems the programs $\mathcal{A}$ and $\mathcal{B}$ can only output either **True** or **False**.

The notation $\pi_1 \propto_t \pi_2$ means the reduction algorithm $\propto$ is polynomially time bounded. In this case, $\propto_t$ is a polynomial-time reduction. Similarly, the notation $\pi_1 \propto_s \pi_2$ means the reduction algorithm $\propto$ is logarithmically space bounded.

The class $\mathcal{P}$-Complete contains all decision problems in $\mathcal{P}$ that appear to require polynomial space on a RAM and are log-space reducible to each other.

A problem $\pi \in \mathcal{P}$ is log-space complete for $\mathcal{P}$ ($\mathcal{P}$-Complete) iff for each $\gamma \in \mathcal{P}$ there is a log-space bounded reduction $\propto_s$ such that $\gamma \propto_s \pi$. Generally, a problem π is shown to be $\mathcal{P}$-Complete by first showing that it is in $\mathcal{P}$, then by giving a log-space transformation from some other $\mathcal{P}$-Complete problem to π. In terms of parallel computation, this log-space transformation can be replaced by an $\mathcal{NC}$ algorithm as a consequence of the parallel computation hypothesis.

The first log-space complete problem in $\mathcal{P}$ was given in (Cook, 1974). Such problems are in some sense among the "hardest" in $\mathcal{P}$ in that solving them takes as much space as does solving any other problem in $\mathcal{P}$.

By the parallel computation hypothesis, a problem in $\mathcal{P}$ that appears to take polynomial space on a RAM also appears to take a polynomial amount of parallel

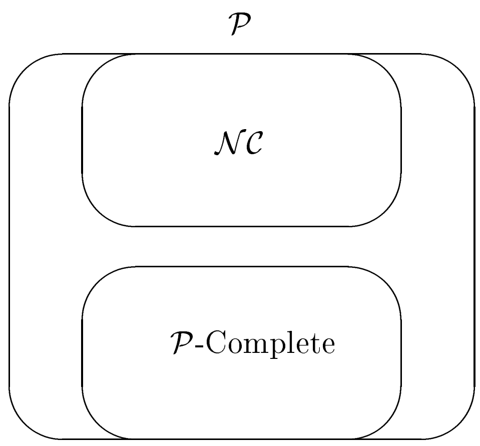

Figure 3: A Hypothetical View of $\mathcal{P}$

time on a PRAM. It seems that only if we augment a PRAM to have an exponential number of processors, then we can solve $\mathcal{P}$-Complete problems on it in polylog time.

See Figure 3 for a hypothetical relationship of $\mathcal{P}$-Complete and $\mathcal{NC}$. For ease of exposition, we simply say a problem is $\mathcal{NC}$ or $\mathcal{P}$-Complete when it is in $\mathcal{NC}$ or it is in $\mathcal{P}$-Complete respectively. If any problem is both $\mathcal{NC}$ and $\mathcal{P}$-Complete then $\mathcal{NC} = \mathcal{P}$-Complete $= \mathcal{P}$.

A problem *inherently sequential* if it, or its restriction to a decision problem, is $\mathcal{P}$-Complete.

2.4 The Nature of Some Paradigms

Not all algorithm design paradigms give efficient parallel algorithms. Standard sequential algorithm design tools, such as depth first search, and variations of the greedy method, do not seem to give efficient parallel algorithms.

The greedy principle applies when every optimal subsolution is in some optimal solution, but there is no method specified for getting from the optimal subsolution to a total solution. On the other hand, the lexicographical greedy principle applies when every optimal subsolution is in some optimal solution *and* this subsolution is built lexicographically into a solution.

Here depth first search to refers to the problem of labeling a graph's nodes in the order they are traversed by some depth first search. The problem of lexicographical depth first search is inherently sequential, though the general problem of depth first search is not known to be either inherently sequential or inherently parallel (Reif, 1985).

In (Anderson and Mayr, 1987; Anderson and Mayr, 1987a) it is shown that many problems that have elementary solutions by the lexicographical greedy principle are inherently sequential. This makes many problems that are solvable sequentially using lexicographical reachability intractable to solve in polylog time. Non-lexicographical greedyness is not necessarily bogged down in the same way.

On the other hand, since (Valiant et al., 1983), many problems amenable to dynamic programming were known to be $\mathcal{NC}$, but their processor complexities were very high (asymptotically ninth degree polynomials). More recent results have lowered this processor complexity. This dissertation shows that some of these problems have efficient parallel solutions.

2.5 Historical Notes

The PRAM model was first given in (Fortune and Wyllie, 1978).

One of the first renditions of the parallel computation hypothesis was in (Chandra et al., 1981) in terms of alternating Turing machines. Here the main focus was $\mathcal{P}$Space and the polynomial hierarchy to describe a notion of parallelism for Turing machines. This work inspired much work showing the asymptotic equivalence of parallel time and sequential space on the Turing machine model.

The authors of (Balcázar et al., 1988 and 1990) say that the conference paper (Chandra and Stockmeyer, 1976) is where the parallel computation hypothesis was first explicitly mentioned by name. In the same conference (Kozen, 1976) gave a similar rendition of a parallel Turing machine model. Together, both (Chandra and Stockmeyer, 1976) and (Kozen, 1976) lead to the journal article (Chandra et al., 1981).

The authors of (Karp and Ramachandran, 1990) point out that the parallel computation hypothesis is quite robust in that most theoretical parallel models of computation abide by it such as those from circuit complexity, parallel vector machines, alternating Turing machines, etc.

The class $\mathcal{NC}$ stands for "Nick's Class" in honor of Nick Pippenger. The first $\mathcal{P}$-Complete problem was given in (Cook, 1974) while the author was discussing the question of the space complexity of parsing context-free grammars relative to the space complexity of all problems in $\mathcal{P}$.

Chapter 3

Sequential Dynamic Programming

This chapter contains sequential dynamic programming algorithms for the matrix chain ordering problem and an optimal convex polygon triangulation problem. More details about these and other related problems and their sequential solutions can be found in several standard textbooks such as (Aho et al., 1974; Baase, 1988; Cormen et al., 1990; Purdom and Brown, 1985).

A *dynamic programming problem* is a problem that is amenable to a simple and efficient sequential dynamic programming solution.

3.1 The Basics

The dynamic programming paradigm is based on the *principle of optimality.* This principle is that for a structure to be optimal all of its well-formed substructures must also be optimal. Hence, the dynamic programming paradigm is essentially a top down design method. Conversely, the *greedy principle* basically is that if a substructure is optimal, then it is in some optimal superstructure. In some sense this is a bottom up design method.

Some foundations for the principle of optimality and the greedy principle can be found in (Bellman, 1957) and (Cormen et al., 1990; Korte et al., 1991), respectively.

3.2 The Matrix Chain Ordering Problem

This section contains an in-depth discussion of the matrix chain ordering problem (MCOP). Its focus is on the classical sequential solution. An optimal convex polygon triangulation problem is also given and is cast in an almost identical framework.

A solution of the MCOP can be expressed as a parenthesization of the n given matrices giving an order to optimally multiply them. There are a Catalan number of ways to parenthesize any n element associative product, which is $\Theta(4^n/n^{3/2})$, hence an exhaustive search algorithm is not feasible.

Let $\bullet$ denote matrix multiplication[1] and take a chain of n matrices $M_1 \bullet M_2 \bullet \cdots \bullet M_n$, then there are $n(n-1)/2$ possible subproducts of the form $M_{i,j} = M_i \bullet \cdots \bullet M_j$. Clearly, the final product $M_{1,n}$ must be made up of one of these subproducts. These subproducts, in turn, are made up of such subproducts with the single matrix base case when $i = j$.

Taking the definition of $M_{i,j}$ and applying the principle of optimality gives a dynamic programming algorithm with a polynomial time solution. This was observed by several researchers in the early 1970s.

In 1973, the first polynomial time solutions of the MCOP were given independently in (Godbole, 1973) and in (Muraoka and Kuck, 1973). Godbole's algorithm has become the classical $O(n^3)$ dynamic programming solution, and it is where we begin.

Start by letting $M[i,j]$ be the *minimal* cost of multiplying matrices i through j, therefore our final goal is to compute $M[1,n]$. Also $M[i,i]$ is the cost of multiplying

[1] The basic matrix multiplication algorithm can be found in almost any standard algorithms textbook, take for example those listed in the beginning of this chapter.

matrix i through i, thus $M[i,i]=0$. For simplicity, let $d_i d_{j+1} d_{k+1}$ be the cost of multiplying two matrices of dimensions $d_i \times d_{j+1}$ and $d_{j+1} \times d_{k+1}$. The cost of multiplying these two matrices is written $f(i,j,k) = d_i d_{j+1} d_{k+1}$. Generalizations of this matrix product function reflecting asymptotically better matrix multiplication algorithms, such as those in (Pan, 1984; Coppersmith and Winograd, 1990), are also acceptable, see (Chandra, 1975).

Given that the matrix $M_{i,j}$ is of dimensions $d_i \times d_{j+1}$, and taking $M[i,i]=0$ as a base case suggests the following recurrence for solving the MCOP:

$$M[i,k] = \min_{i \le j < k} \{ \ M[i,j] + M[j+1,k] + d_i d_{j+1} d_{k+1} \ \} \tag{1}$$

That is, to find a minimal product of matrices i through k, find the minimal cost of creating and combining all well-formed subproducts. This natural use of the principle of optimality is characteristic of dynamic programming algorithms.

The top-down definition of Recurrence 1 can be efficiently solved by the bottom-up algorithm given in Figure 4. This algorithm assumes a table set up as in Figure 5, where $M[i,j]$ is in diagonal D iff $j-i=D$. Therefore, $M[1,1]$, $M[2,2]$, $M[3,3]$ and $M[4,4]$ are all in diagonal 0 and $M[1,4]$ is in diagonal 3.

> In diagonal 0 initialize all elements to 0
> **for** each diagonal D from 1 **to** $n-1$ **do**
>   **for** each element $M[i,k]$ in diagonal D **do**
> $M[i,k] \Leftarrow \min_{i \le j < k} \{M[i,j] + M[j+1,k] + f(i,j,k)\}$

Figure 4: Sequential Matrix Chain Ordering Algorithm

The algorithm in Figure 4 has time complexity $\Theta(n^3)$ because of the two nested **for**-loops and the implicit loop in the min operation. The sequential nature of the diagonal-by-diagonal computation prevents such an algorithm from running in polylog time on a PRAM. The sequential nature of this classical solution should not be taken lightly, since it has taken about a decade for researchers to give an efficient polylog time parallel algorithm to solve the MCOP.

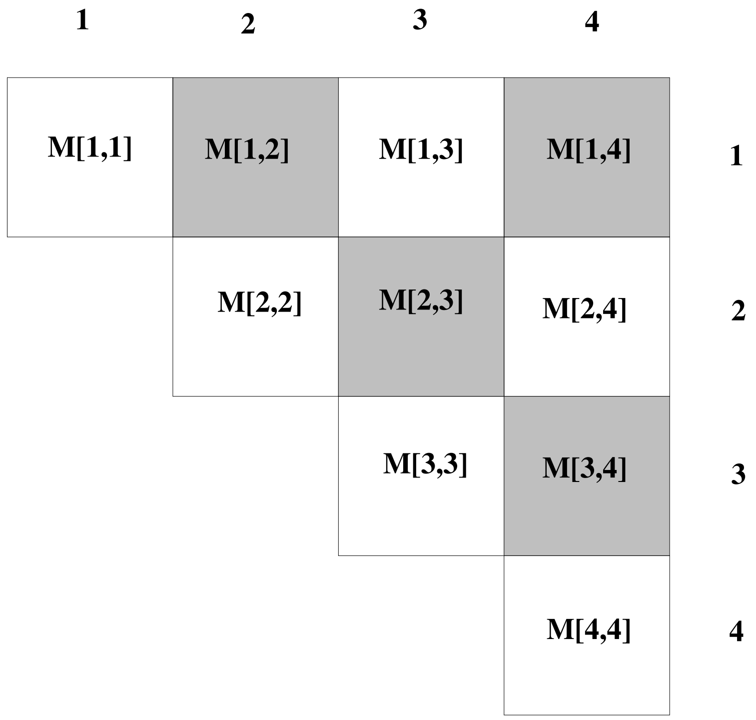

Figure 5: A Dynamic Programming Table for the MCOP

3.3 An Instance of the MCOP

This section contains an instance of the MCOP and its sequential solution. This example is used throughout this dissertation.

Take the four-matrix instance of the MCOP: $M_1 \bullet M_2 \bullet M_3 \bullet M_4$. Say these matrices are of dimensions 5×10, 10×3, 3×20, and 20×6, respectively. That is:

$\boxed{d_1 = 5}$ $\boxed{d_2 = 10}$ $\boxed{d_3 = 3}$ $\boxed{d_4 = 20}$ $\boxed{d_5 = 6}$

In other words,

Matrices	M_1	M_2	M_3	M_4
Dimensions	5×10	10×3	3×20	20×6

	1	**2**	**3**	**4**
1	**0**	**150**	**450**	**600**
2		**0**	**600**	**540**
3			**0**	**360**
4				**0**

Figure 6: The Dynamic Programming Table of $M_1 \bullet M_2 \bullet M_3 \bullet M_4$

The rest of this section follows the algorithm in Figure 4 giving a solution to this four-matrix instance of the MCOP.

First calculate the elements in diagonal 1 of the table of Figure 6, this is the cost of multiplying matrices i though $i+1$, for all $i, 1 \leq i < 4$, which is:

$$
\begin{aligned}
M[1,2] &\leftarrow d_1 d_2 d_3 = 5 \times 10 \times 3 = 150 \\
M[2,3] &\leftarrow d_2 d_3 d_4 = 10 \times 3 \times 20 = 600 \\
M[3,4] &\leftarrow d_3 d_4 d_5 = 3 \times 20 \times 6 = 360
\end{aligned}
$$

With this information in hand compute the elements in the next diagonal:

$$\begin{aligned}
M[1,3] &\leftarrow \min\{\ M[1,2]+d_1d_3d_4,\ M[2,3]+d_1d_2d_4\ \} \\
&= \min\{\ 150+300,\ 600+1000\ \} \\
&= 450 \\
M[2,4] &\leftarrow \min\{\ M[2,3]+d_2d_4d_5,\ M[3,4]+d_2d_3d_5\ \} \\
&= \min\{\ 600+1200,\ 360+180\ \} \\
&= 540
\end{aligned}$$

Both $M[1,3]$ and $M[2,4]$ depend on elements from the prior diagonal. In particular, $M[1,3]$ and $M[2,4]$ are in diagonal 2 and computing them uses elements $M[1,2]$, $M[2,3]$ and $M[3,4]$ which are all in diagonal 1.

Finally, the last value in the table, $M[1,4]$, is computed as:

$$\begin{aligned}
M[1,4] &\leftarrow \min\{\ M[1,2]+M[3,4]+d_1d_3d_5,\ M[1,3]+d_1d_4d_5,\ M[2,4]+d_1d_2d_5\ \} \\
&= \min\{\ 150+360+90,\ 450+600,\ 540+300\ \} \\
&= 600
\end{aligned}$$

Therefore, the optimal cost of multiplying the four given matrices is 600. An optimal order to multiply these matrices is $(M_1 \bullet M_2) \bullet (M_3 \bullet M_4)$ because $M[1,2]+M[3,4]+d_1d_3d_5 = 600$. Other matrix products have much higher costs, for example, the product $((M_1 \bullet (M_2 \bullet M_3)) \bullet M_4)$ has a total cost of 2200.

Tracking the elements of the table contributing to the minimal value in $M[1,n]$ gives a minimal ordering for the matrix product.

The computation of $M[1,4]$ depends on elements from the two prior diagonals, particularly the elements $M[1,2]$, $M[3,4]$, $M[1,3]$ and $M[2,4]$ are in diagonals 1 and 2. In general, using the algorithm in Figure 4, to find the minimum for an element in diagonal D this algorithm uses elements from all $D-1$ previous diagonals. Given n matrices to order, such "inter-diagonal dependencies" make it difficult for the algorithm in Figure 4 to run in polylog time since there is a linear number of diagonals in such a dynamic programming table (linear in the number of given matrices).

3.4 Triangulating Convex Polygons

This section contains a triangulation problem that is very closely related to the MCOP.

A polygon is a closed piece-wise linear geometric structure (Cormen et al., 1990; O'Rourke, 1994). A convex polygon is a polygon such that for any two points on the border of the polygon if a line connects them together, then this line goes through either the border or the interior of the polygon, but never through the outside the polygon.

Given a set of (integer) points that form a convex polygon, such as in Figure 7, then take the following problem:

- Find a minimal cost triangulation of this polygon, given a triangle with the node values d_i, d_j, and d_k costs $d_i d_j d_k$.

This is also a popular optimization problem that is very closely related to the MCOP (Cormen et al., 1990; Hu and Shing, 1982; Hu, 1982). In a triangulated polygon there may be triangles that do not have any sides that are in the borders of the polygon. Further, triangles are not allowed to overlap.

Take the vertices of these polygons and associate the following costs with them,

Vertices	V_1	V_2	V_3	V_4	V_5
Costs	5	10	3	20	6

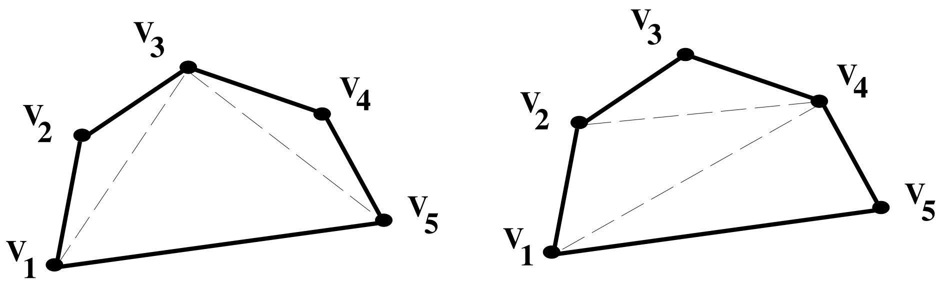

Figure 7: Two Different Triangulations of the Same Convex Polygon

If these are the costs of the vertices of the polygons in Figure 7, then solving this instance of the optimal triangulation problem will solve the instance of the MCOP in the last section. This is because the triangle cost metric is the same as the assumed matrix product cost.

It is common practice to associate the height (y coordinate) of a vertex in a convex polygon with its associated cost. In this case, the matrix instance from the last section would not give a convex polygon, since its dimensions are $5, 10, 3, 20$, and 6. But we may consider a convex polygon with these values associated with its nodes anyway. This is acceptable, since forgetting the geometry in this way only removes problems of finding valid triangulations.

The following recurrence gives the minimal cost of solving the optimal triangulation problem. As with the matrix chain ordering problem, let $T[i, j]$ denote the minimal cost of triangulating points i through j in a convex polygon. This forces us to assume there is a line connecting vertex i and vertex j. Taken together, all of these assumptions naturally lead to the base cases $T[i, i+1] = 0$ and $T[i, i+2] = t_i t_{i+1} t_{t+2}$ to the following recurrence:

$$T[i, k] = \min_{i<j<k} \{ \ T[i, j] + T[j, k] + t_i t_j t_k \ \} \tag{2}$$

This recurrence finds the minimum cost to triangulate the convex subpolygon from node i through node j and the convex subpolygon from j through k. Then add in the cost of the triangle formed by nodes i, j, and k, this triangle joins the two subpolygons to form one larger convex subpolygon from node i through node k.

Recurrence 2 is basically the same as Recurrence 1, the recurrence for solving the MCOP. This can be seen by the following simple transformation:

1. Relabel the nodes from 0 to $n-1$.
2. Let $T'[i,j]$ denote the cost of triangulating the convex subpolygon consisting of nodes $i-1, i, i+1, \cdots, j$.
3. Change base cases to $T'[i,i] = 0$ and $T'[i,i+1] = t_{i-1}t_it_{i+1}$.

Now, Recurrence 2 can be rewritten as

$$T'[i,k] = \min_{i \le j < k} \{ \ T'[i,j] + T'[j+1,k] + t_{i-1}t_jt_{k-1} \ \}$$

This recurrence has a very close form to the recurrence for solving the MCOP, except that indices triangle cost function differ from the indices for the matrix product cost function. In any event, the algorithm in Figure 4 solves this recurrence.

3.5 Historical Notes

A recent survey and classification of modern dynamic programming problems is (Galil and Park, 1992). This paper classifies dynamic programming problems in terms of size of the dynamic programming table and the number of table entries a new table entry depends on. The standard $O(n^3)$ MCOP algorithm has a table of size $\Theta(n^2)$ and each new table entry can depend on up to $\Theta(n)$ other elements. This paper also gives other interesting discussion of convexity, concavity and sparsity in dynamic programming tables.

According to the authors of (Aho et al., 1974) the MCOP first appeared in (Godbole, 1973) and (Muraoka and Kuck, 1973). The authors of these papers on the MCOP were apparently motivated by the rich nature of the MCOP. As years passed the MCOP became a very popular problem for textbooks on algorithm design and analysis.

The MCOP is one of the standard examples of the dynamic programming algorithm design paradigm since it has an elementary dynamic programming solution. Between 1980 and 1984 T. C. Hu and M. T. Shing published several papers (Hu and Shing, 1980; Hu and Shing, 1982; Hu and Shing, 1984) on an $O(n \lg n)$ sequential solution to the MCOP. During this time, F. F. Yao also published several papers on improving dynamic programming algorithms using monotonicity conditions (Yao, 1980; Yao, 1982). In the process, she gave an $O(n^2)$ sequential algorithm for the MCOP.

Since these papers, there was much interest in parallel algorithms for problems with elementary sequential dynamic programming solutions.

Also, lower bounds for the MCOP appear in (Bradford et al., 1993b; Bradford et al., 1995) and lower bounds for problems very close to the MCOP appear in (Ramanan, 1991; Ramanan, 1994).

Chapter 4

A Dynamic Graph Model

This chapter contains a dynamic graph model for solving dynamic programming problems in parallel. These graphs are analogous to classical dynamic programming tables, except they allow fast parallel computation more readily. This is done by interpreting dynamic programming problems as shortest path problems in these graphs. Finding shortest paths in these graphs can be done very fast in parallel, so the related dynamic programming problems can be solved fast too. These problems take $O(\lg^2 n)$ time to solve in parallel.

However, without other considerations this speed comes with a cost of $n^6/\lg n$ processors. The subsequent chapters give ways to maintain the speed while drastically reducing the number of processors.

One of the key insights of this chapter is that in graphs of size n we only need edges of size up to $\lceil n/2 \rceil$. This is based on symmetry in these graphs.

4.1 Theoretical Foundations

This section contains theoretical background for the graph model given in this chapter. These foundations are algebraic in nature and model variations on associative products. The generality given by the abstractions in this section allows broad applications of the graph model.

A (finite) *semigroupoid* $(S, R, \bullet)$ is a nonempty finite set S, a binary relation $R \subseteq S \times S$, and an associative binary operator "$\bullet$" satisfying the following conditions (Marcus, 1978):

1. If $(a, b) \in R$, then $a \bullet b \in S$.

2. $(a \bullet b, c) \in R$ iff $(a, b \bullet c) \in R$ and $(a \bullet b) \bullet c = a \bullet (b \bullet c)$.

3. If $(a, b) \in R$ and $(b, c) \in R$, then $(a \bullet b, c) \in R$.

An *associative product* is any product of the form $a_1 \bullet a_2 \bullet \cdots \bullet a_n$ such that $(a_i, a_{i+1}) \in R$, for $1 \leq i < n$. A *linear* product is a product of the form

$$((\cdots(a_1 \bullet a_2) \bullet \cdots) \bullet a_n)$$

or

$$(a_1 \bullet (\cdots \bullet (a_{n-1} \bullet a_n)) \cdots).$$

A *weighted* semigroupoid $(S, R, \bullet, pc)$ is a semigroupoid $(S, R, \bullet)$ with a nonnegative product cost function pc such that if $(a_i, a_k) \in R$ then $pc(a_i, a_k)$ is the cost of evaluating $a_i \bullet a_k$. The minimal cost of evaluating an associative product $a_i \bullet a_{i+1} \bullet \cdots \bullet a_k$ is denoted by $sp(i, k)$. That is,

$$sp(i, k) = \min_{i \leq j < k} \{ \ sp(i, j) + sp(j + 1, k) + f(i, j, k) \ \}$$

where $f(i, j, k) = pc(a_i \bullet a_{i+1} \bullet \cdots \bullet a_j, \ a_{j+1} \bullet a_{j+2} \bullet \cdots \bullet a_k)$ and and the base case is $sp(i, i) = 0$.

Given a weighted semigroupoid and an associative product $a_1 \bullet a_2 \bullet \cdots \bullet a_n$ the problem of finding $sp(1, n)$ is the *minimum parenthesization problem* (MPP).

Assume a globally accessible 4-tuple $(S, R, \bullet, pc)$ represents a weighted semigroupoid. Represent pc and R by the array, $f(1..n, 1..n, 1..n)$, where

$$f(i,j,k) \quad = \quad \begin{cases} pc(a_i \bullet \cdots \bullet a_j,\ a_{j+1} \bullet \cdots \bullet a_k) & \text{if } (a_i \bullet \cdots \bullet a_j) \bullet (a_{j+1} \bullet \cdots \bullet a_k) \in S \\ \infty & \text{otherwise} \end{cases}$$

Given a_i and a_j assume that both $a_i \bullet a_j$ and $pc(a_i, a_j)$ can be computed in constant time. That is, for any values of i, j, and k the cost $f(i,j,k)$ can be computed in constant time.

4.2 Greedy Minimum Cost Parenthesizations

This section contains a subproblem of the MPP called the greedy parenthesization problem (GPP). We use this problem to develop the MPP and to discuss some previous results.

$$\begin{array}{llll} S_{1,n} & & \text{is the start symbol} & \\ S_{i,i+1} & \rightarrow & i \bullet j & \text{iff} \quad i+1 = j \text{ and } i+1 \leq n \\ S_{i,j} & \rightarrow & i \bullet (S_{i+1,j}) \mid (S_{i,j-1}) \bullet j & \text{iff} \quad i < j-1 \end{array}$$

Figure 8: The Grammar L_1

Any string derived from L_1 is a *greedy parenthesization* of n elements. We consider this to be greedy since this grammar describes an associative product that characterizes "local product growth."

Given a weighted semigroupoid, the greedy parenthesization problem (GPP) is to find minimal products that are generated by the grammar in Figure 8.

For every weighted semigroupoid with a greedy associative product of n elements we can construct a corresponding weighted digraph. Finding a shortest path in such a graph solves the GPP for the given associative product.

Denote vertices by (i, j), where $1 \le i \le j \le n$, and edges by $\rightarrow$, $\uparrow$ or $\nearrow$. Edge $(i, j) \uparrow (i-1, j)$ represents the product $a_{i-1} \bullet (a_i \bullet \cdots \bullet a_j)$, therefore it weighs $f(i-1, i-1, j)$. Similarly, $(i, j) \rightarrow (i, j+1)$ represents the product $(a_i \bullet \cdots \bullet a_j) \bullet a_{j+1}$ and it weighs $f(i, j, j+1)$. Also, for all $i, 1 \le i \le n$, the arrows $\nearrow$ represent edges from $(0, 0)$ to (i, i).

Definition Given an n-element weighted semigroupoid, the graph $G_n = (V, E)$ has vertices,

$$V \;=\; \{(i, j) : 1 \le i \le j \le n\} \cup \{(0, 0)\}$$

and *unit* edges,

$$\begin{aligned} E \;=\; & \{(i, j) \rightarrow (i, j+1) : 1 \le i \le j < n\} \cup \\ & \{(i, j) \uparrow (i-1, j) : 1 < i \le j \le n\} \cup \\ & \{(0, 0) \nearrow (i, i) : 1 \le i \le n\} \end{aligned}$$

and a weight function W where

$$\begin{array}{lcll} W((i, j) \rightarrow (i, j+1)) & = & f(i, j, j+1) & 1 \le i \le j < n \\ W((i, j) \uparrow (i-1, j)) & = & f(i-1, i-1, j) & 1 < i \le j \le n \\ W((0, 0) \nearrow (i, i)) & = & 0 & 1 \le i \le n \end{array}$$

For example, see the graph G_4 in Figure 9.

Given a weighted semigroupoid, n^2 processors can construct a corresponding G_n graph in constant time, since each vertex has in-degree and out-degree of at most two. The communication costs are being ignored here.

The following restricted instance of the matrix chain ordering problem is a special case of the GPP: As before, "$\bullet$" denotes matrix multiplication in the four-matrix instance, $M_1 \bullet M_2 \bullet M_3 \bullet M_4$ of the GPP. The challenge is to minimize the cost of multiplying this chain while *excluding* the product $(M_1 \bullet M_2) \bullet (M_3 \bullet M_4)$, since it is not a greedy parenthesization.

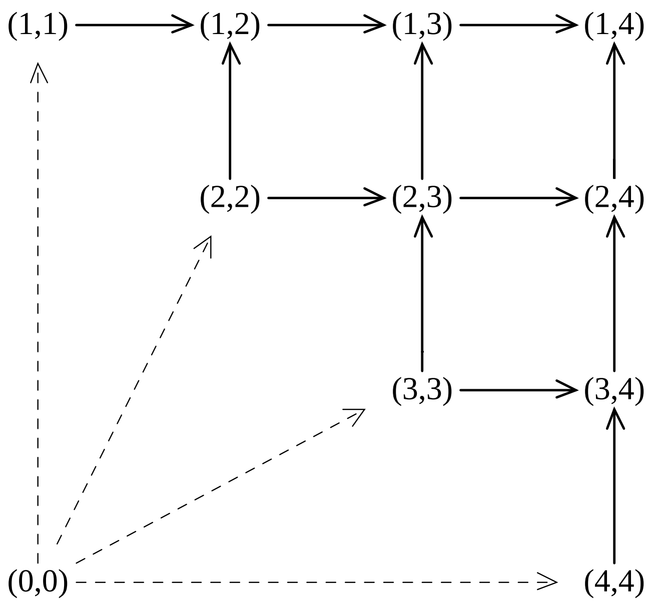

Figure 9: The Greedy Weighted Digraph G_4

Given a vertex (i, j) of G_n, finding a shortest path from $(0, 0)$ to (i, j) solves the minimum cost parenthesization problem for the *greedy* associative product $a_i \bullet a_{i+1} \bullet \cdots \bullet a_j$.

Theorem 1 Finding a shortest path from the vertex $(0, 0)$ to vertex $(1, n)$ in G_n solves the minimal cost greedy parenthesization problem for an associative product of n elements.

A proof is by induction on the distance of minimal paths up to each diagonal.

Since G_n has $\Theta(n^2)$ vertices, computing a minimum path in $O(\lg^2 n)$ time by a parallel matrix multiplication based minimum path algorithm takes $n^6/\lg n$ processors. A $(\min, +)$-matrix multiplication based shortest path algorithm is a matrix multiplication algorithm where the usual matrix operations of $(+, \times)$ are replaced by $(\min, +)$. That is, the vector sums are replaced by minimizations and the dot-products are replaced by dot-sums, see (Cormen et al., 1990). A $(\min, +)$-matrix multiplication based shortest path algorithm is easily run in parallel on a common-CRCW PRAM in $O(\lg^2 n)$ time using $n^3/\lg n$ processors (JáJá, 1992; Kumar et al., 1994).

The processor complexity of special instances of the GPP was improved dramatically in (Apostolico et al., 1990; Aggarwal and Park, 1988; Ibarra et al., 1988). These methods rely on either divide and conquer or monotonicity properties of the matrices that are derived from the structure of the particular G_n graphs. These methods can compute a shortest path in a G_n graph in $O(\lg^2 n)$ time with $n^2/\lg n$ processors on a common-CREW PRAM.

In (Apostolico et al., 1990; Aggarwal and Park, 1988; Ibarra et al., 1988) dynamic programming problems are also transformed into graph search problems; these variations of the GPP are used to solve string edit problems. For example, given two strings of n characters each, find the longest common subsequence of them. The longest common subsequence of two strings is the longest string that is a subsequence of both given strings, see for example (Cormen et al., 1990). Applications of this problem can be found in computational biology and operating systems.

4.3 Minimum Cost Parenthesizations

This section contains a generalization of the G_n graphs that accounts for split parenthesizations. These new graphs are particularly suited to problems that seem to require the principle of optimality for their solution and not the greedy principle.

The grammar in Figure 10 characterizes all well-formed parenthesizations of an n element associative product. The grammar L_2's terminals and nonterminals are the

$$\begin{array}{llll} S_{1,n} & & \text{is the start symbol} & \\ S_{i,i+1} & \rightarrow & i \bullet j & \text{iff} \quad i+1=j \text{ and } i+1 \leq n \\ S_{i,j} & \rightarrow & i \bullet (S_{i+1,j}) \mid (S_{i,j-1}) \bullet j & \text{iff} \quad i < j-1 \\ S_{i,j} & \rightarrow & (S_{i,k}) \bullet (S_{k+1,j}) & \text{iff} \quad i < k < j-1 \end{array}$$

Figure 10: The Grammar L_2

same as L_1's, however the last derivation rule now lets us generate products of the form $(a_i \bullet \cdots \bullet a_j) \bullet (a_{j+1} \bullet \cdots \bullet a_k)$.

To represent split parenthesizations *jumpers* are added to G_n graphs. The jumper $(i,j) \Longrightarrow (i,k)$ represents the product $(a_i \bullet \cdots \bullet a_j) \bullet (a_{j+1} \bullet \cdots \bullet a_k)$ and this jumper weighs $sp(j+1,k)$ plus $f(i,j,k)$. See Figure 11.

Horizontal jumpers are denoted by $\Longrightarrow$, and vertical jumpers by $\Uparrow$ and split parenthesizations can be expressed as either a vertical jumper or a horizontal jumper. This leads directly to a balancing argument on jumper length that says for any split parenthesization of n matrices, one of the jumpers representing it is $\lceil n/2 \rceil$ units long. The vertical jumper $(i,j) \Uparrow (s,j)$ is $i-s$ units long and the horizontal jumper $(i,j) \Longrightarrow (i,t)$ is $t-j$ units long, where all non-jumper edges are unit edges 1 unit long.

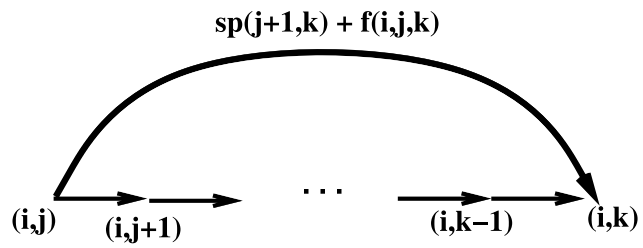

Figure 11: A Horizontal Jumper with its Associated Weight

For the moment, when finding a shortest path to (i,k) we take it on faith that a shortest path to $(j+1,k)$ will be computed in time for $sp(j+1,k)$ to be added to $f(i,j,k)$, see Theorem 3. We make similar assumptions for vertical jumpers.

Definition The weighted digraph $D_n = (V, E \cup E')$, is a weighted digraph $G_n = (V, E)$ together with the *jumpers*,

$$\begin{aligned} E' &= \{(i,j) \Longrightarrow (i,t) : 1 \leq i < j < t \leq n\} \cup \\ & \quad \{(s,t) \Uparrow (i,t) : 1 \leq i < s < t \leq n\} \end{aligned}$$

and each jumper has weight

$$\begin{array}{lcll} W((i,j) \Longrightarrow (i,t)) & = & sp(j+1,t) + f(i,j,t) & 1 < i < j < t \leq n \\ W((s,t) \Uparrow (i,t)) & = & sp(i,s-1) + f(i,s-1,t) & 1 \leq i < s < t \leq n \end{array}$$

For example, see the graph D_4 in Figure 12.

The next lemma and the theorem following it are central for the rest of this dissertation. These results are based on the symmetry of D_n graphs.

Lemma 1 For all vertices (i,k) in a D_n graph, $sp(i,k)$ can be computed by a path having edges of length no larger than $\lceil (k-i)/2 \rceil$.

Proof: Suppose that $(i,j) \Longrightarrow (i,k)$ is in a shortest path to (i,k) and $k - j > \lceil (k-i)/2 \rceil$. (This jumper terminates at (i,k) without loss.) Hence,

$$\begin{aligned} sp(i,k) &= sp(i,j) + W((i,j) \Longrightarrow (i,k)) \\ &= sp(i,j) + sp(j+1,k) + f(i,j,k) \end{aligned}$$

But $$W((j+1,k) \Uparrow (i,k)) = sp(i,j) + f(i,j,k)$$

so,

$$sp(i,k) = sp(j+1,k) + W((j+1,k) \Uparrow (i,k))$$

The jumper $(j+1,k) \Uparrow (i,k)$ is of length $j+1-i$. Therefore, since

$$j+1-i+k-j = k-i+1$$

and $k-j > \lceil (k-i)/2 \rceil$, it must be that $j+1-i \leq \lceil (k-i)/2 \rceil$.

On the other hand, a shortest path to $(j+1,k)$ cannot contain a jumper longer than $k-(j+1)$. Since $k-(j+1) < k-j$ this lemma follows inductively. □

The proof of this last lemma leads directly to the following theorem.

Theorem 2 (Duality Theorem) If a shortest path from $(0,0)$ to (i,k) contains the jumper $(i,j) \Longrightarrow (i,k)$, then there is a *dual* shortest path containing the jumper $(j+1,k) \Uparrow (i,k)$.

The following instance of the matrix chain ordering problem is a special case of the MPP: As before, "$\bullet$" denotes matrix multiplication. Take the four-matrix instance of the MCOP $M_1 \bullet M_2 \bullet M_3 \bullet M_4$. Say these matrices are of dimensions 5×10, 10×3, 3×20, and 20×6, respectively. These repeat the same example given in Chapter 3:

Matrices	M_1	M_2	M_3	M_4
Dimensions	5×10	10×3	3×20	20×6

The optimal product of matrices M_1, M_2, and M_3 is $(M_1 \bullet M_2) \bullet M_3$. But this is *not* a well-formed subproduct of the optimal matrix product of all four matrices, that is $(M_1 \bullet M_2) \bullet (M_3 \bullet M_4)$. This apparent lack of greediness seems to make techniques such as those of (Apostolico et al., 1990; Aggarwal and Park, 1988; Ibarra et al., 1988) fail to work for the MCOP.

Note the similarity of a D_n graph and a classical dynamic programming table, T, for the matrix chain ordering problem. The value of $sp(i,k)$ in a D_n graph is the same as $T[i,k]$ in the equivalent dynamic programming table.

Calculating a shortest path to (i, k) gives the minimum cost parenthesization of $a_i \bullet \cdots \bullet a_k$, for $1 \leq i < k \leq n$. So finding a shortest path from (i, k) to $(1, n)$ gives a minimal parenthesization of $a_1 \bullet \cdots \bullet a_{i-1} \bullet P \bullet a_{k+1} \bullet \cdots \bullet a_n$ where $P = (a_i \bullet \cdots \bullet a_k)$.

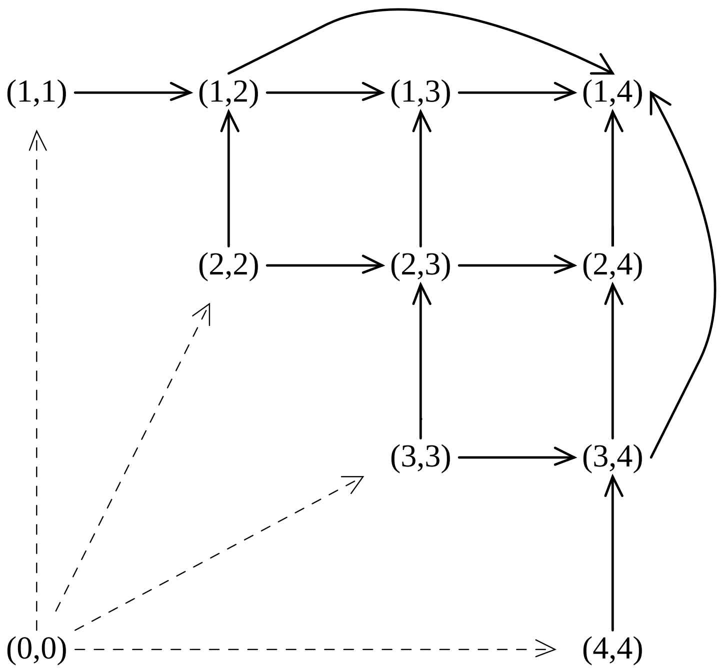

Figure 12: The Weighted Graph D_4

Theorem 3 Finding a shortest path from $(0, 0)$ to $(1, n)$ in D_n solves the minimal cost parenthesization problem for an associative product of n elements.

Proof: To prove this theorem it is sufficient to show that the cost of every path from $(0, 0)$ to $(1, n)$ in a D_n graph corresponds to the cost of a parenthesization of an n element associative product. In addition, for every parenthesization of an n element associative product there is a corresponding path in D_n where both the path and the associative product have the same cost.

The theorem holds for paths in the underlying G_n graph and their associated greedy parenthesizations, by Theorem 1. So we only consider jumpers.

We only show that for each path from $(0, 0)$ to $(1, n)$ in a D_n graph there is a corresponding associative product of n elements (or $n - 1$ operators "$\bullet$-s"). This proof is by induction on the lengths of the associative products. D_2 and D_3 are trivial so consider D_4 and the product $a_1 \bullet a_2 \bullet a_3 \bullet a_4$. In D_4 the only derivation of the string $1 \bullet 2 \bullet 3 \bullet 4$ that can't be derived in L_1 is $(1 \bullet 2) \bullet (3 \bullet 4)$. The jumper $(1, 2) \Longrightarrow (1, 4)$ has weight $sp(3, 4) + f(1, 2, 4)$. Therefore, the cost of a path using this jumper corresponds to the cost of the associative product $(a_1 \bullet a_2) \bullet (a_3 \bullet a_4)$. A symmetric argument holds for the jumper $(3, 4) \Uparrow (1, 4)$. Hence for each associative product there is a path and for each path there is an associative product.

Now suppose that the theorem holds for all $n \leq k$, where $k \geq 4$. And n is the number of associative operators in a given associative product. Thus, the inductive hypothesis is for any path in D_n from $(0, 0)$ to $(1, n)$, where $n \leq k$ there is a corresponding associative product of n elements with the same cost.

Without loss of generality, we only consider horizontal jumpers. For $m = 2k$ take a path in D_m from $(0, 0)$ to $(1, m)$. We will show that for all $m \leq 2k$ there is a corresponding m element associative product of the same cost. (This proof holds for both even and odd length products because D_{k-1} is a proper subgraph of D_k.)

The structure of D_m along with the inductive hypothesis gives:

1. The cost of a path from $(0, 0)$ to any node (i, j), such that $j - i \leq k$, corresponds to the cost of a parenthesization of $a_i \bullet \cdots \bullet a_j$ by the inductive hypothesis. Since the product $a_i \bullet \cdots \bullet a_j$ contains at most $k - 1$ associative operators ("$\bullet$-s").

2. The cost of a path from (i, t) to $(1, 2k) = (1, m)$, where $k \leq t - i < m$, corresponds to the cost of a parenthesization of $a_1 \bullet \cdots \bullet a_{i-1} \bullet P \bullet a_{t+1} \bullet \cdots \bullet a_m$ where $P = (a_i \bullet \cdots \bullet a_t)$, by the inductive hypothesis. Since P consists of at least k elements, so the product $a_1 \bullet \cdots \bullet a_{i-1} \bullet P \bullet a_{t+1} \bullet \cdots \bullet a_m$ consists of at most $k - 1$ associative operators ("$\bullet$-s").

Suppose a path from $(0,0)$ to $(1,2k)$ includes the jumper $(i,j) \Longrightarrow (i,t)$. Then assume that $j-i < k \leq t-i$, otherwise by the inductive hypothesis and the two facts above the proof is complete.

Therefore, consider the jumper $(i,j) \Longrightarrow (i,t)$, where $j-i < k \leq t-i$. By the inductive hypothesis and Fact 2, the cost of a path from (i,t) to $(1,m)$ corresponds to the cost of a parenthesization of $a_1 \bullet \cdots \bullet a_{i-1} \bullet P \bullet a_{t+1} \bullet \cdots \bullet a_m$, where $P = (a_i \bullet \cdots \bullet a_t)$. Again by the inductive hypothesis and Fact 1 above, the cost of a path to (i,j) corresponds to the cost of a parenthesization of $a_i \bullet \cdots \bullet a_j$, where $a_i \bullet \cdots \bullet a_j$ is a subproduct of P. Furthermore, the jumper $(i,j) \Longrightarrow (i,t)$ is a part of a path from $(0,0)$ to $(1,m)$ whose cost corresponds to the cost of the product $(a_i \bullet \cdots \bullet a_j) \bullet (a_{j+1} \bullet \cdots \bullet a_t)$. This is because its cost is $sp(j+1,t) + f(i,j,t)$ and because $t-(j+1) < j-i$ we can apply the inductive hypothesis. Vertical jumpers follow similarly. $\Box$

The converse of this theorem also holds. That is, for any optimal parenthesization of a weighted semigroupoid there is a shortest path in a corresponding D_n graph.

It is not clear how to generalize the shortest path algorithms in (Apostolico et al., 1990; Aggarwal and Park, 1988; Ibarra et al., 1988) to D_n graphs. This is at least partially due to the jumper's weights; computing these weights seems difficult. This difficulty seems to correspond to the difference between the principle of optimality and the greedy principle.

4.4 Constructing a D_n Graph

This section contains a method for constructing D_n graphs. In the process, several key results are given that are used in the following chapters. Jumper lengths are the basis of the arguments in this section.

In order to construct a D_n graph by starting with a weighted digraph G_n and then perform incremental *path relaxation* (Cormen et al., 1990) while adding new jumpers. A shortest path in D_n is found simultaneously. This is done by using a variation of a $(\min, +)$-matrix multiplication based all pairs shortest path algorithm.

Although G_n can be constructed in constant time with n^2 processors, it does not seem possible to construct D_n in as little time with as few processors. This is because the weight of a jumper $(i, j) \Longrightarrow (i, k)$ cannot be computed until $sp(j+1, k)$ becomes available.

Constructing a D_n graph starting with a G_n graph is done as follows. For any vertex (i, j) the value $j - i$ is the distance from $(0, 0)$ to (i, j). Here distance means simply the minimal number of unit edges it takes to get from $(0, 0)$ to (i, j). So starting with a G_n graph, one $(\min, +)$-matrix multiplication computes all $sp(i, j)$ for any (i, j) where $j - i \leq 2^1 - 1$. Also, the minimum distances between all pairs of nodes in G_n up to 2 nodes apart are now available. With this, construct all jumpers of length 2. For length 2 horizontal jumpers, this is done by relaxing (Cormen et al., 1990, Pages 520-527) them with the two horizontal edges they are directly above. In Figure 13, the min operations are path relaxations.

Another, $(\min, +)$-matrix multiplication gives $sp(i, j)$ for all (i, j) such that $j - i \leq 2^2 - 1$. Continue this process inductively. Suppose that for all (i, j), where $j - i \leq 2^r - 1$, the values $sp(i, j)$ have been computed. At the same time the minimum distances between all pairs of nodes in G_n up to 2^r nodes apart have been computed. Now we can compute all jumpers of lengths ranging from $2^{r-1} + 1$ through 2^r and then relax them with the appropriate straight paths of lengths from $2^{r-1} + 1$ through 2^r. Another $(\min, +)$-matrix multiplication gives the values for $sp(i, j)$, for all (i, j) where $j - i \leq 2^{r+1} - 1$.

Lemma 2 Assume all shortest paths have been calculated between each pair of vertices 2^{r-1} units apart. Suppose, for all (j, k) where $k - j \leq 2^{r-1} - 1$, the value $sp(j, k)$ is available. Then we can calculate $sp(i, t)$ with one $(\min, +)$-matrix multiplication for all

(i, t) such that $t - i \leq 2^r - 1$.

Proof: Suppose $sp(j, k)$ is available for all (j, k) where $k - j < 2^{r-1}$. Also, assume that the all pairs shortest paths have been calculated for all pairs of vertices of distance up to 2^{r-1}.

Now construct every jumper in D_n of length 2^{r-1} or smaller. Placing these jumpers and at the same time relaxing these paths in D_n and performing one $(\min, +)$-matrix multiplication supplies $sp(i, t)$ for all $t - i < 2^r$ by Lemma 1. □

The algorithm in Figure 13 is a modified $(\min, +)$-matrix multiplication all pairs shortest path algorithm. It is modified in that minimum paths are dynamically relaxed with new jumpers. This algorithm is basically the same as Rytter's algorithm (Rytter, 1988). Here $(i, j) \leq (s, t)$ means that $s \leq i \leq j \leq t$, which means there is a path in G_n from (i, j) to (s, t).

for all $1 \leq i, j, u, v \leq n$ **in parallel do**
  $W[(i,j),(u,v)] \leftarrow \infty$
  $W[(0,0),(i,i)] \leftarrow 0$
for all $1 \leq i, j \leq n$ **in parallel do**
  if $j + 1 \leq n$ **then** $W[(i,j),(i,j+1)] \leftarrow f(i,j,j+1)$
  if $i - 1 \geq 1$ **then** $W[(i,j),(i-1,j)] \leftarrow f(i,i-1,j)$
loop $\lceil \lg n \rceil$ **times**
  for all $(0,0) \leq (i,j) \leq (s,t) \leq (u,v) \leq (n,n)$ **in parallel do**
    $W[(i,j),(u,v)] \leftarrow \min\{\ W[(i,j),(s,t)] + W[(s,t),(u,v)],\ W[(i,j),(u,v)]\ \}$
    $W[(i,j),(u,j)] \leftarrow \min\{\ W[(i,j),(u,j)],\ W[(0,0),(u,i-1)] + f(u,i-1,j)\ \}$
    $W[(i,j),(i,v)] \leftarrow \min\{\ W[(i,j),(i,v)],\ W[(0,0),(j+1,v)] + f(i,j,v)\ \}$

Figure 13: Modified $(\min, +)$-All-Pairs-Shortest-Path Algorithm

For the next theorem, the adjacency matrix begins as the adjacency matrix of the appropriate G_n graph.

Theorem 4 Immediately after iteration r, for $1 \leq r \leq \lceil \lg n \rceil$, the algorithm in Figure 13, has computed $sp(i, t)$ for all $t - i \leq 2^r - 1$.

Proof: The correctness of the first two loops is straightforward, so consider the third loop.

The outer loop iterates $\lceil \lg n \rceil$ times because of Lemma 2. The first line of the inner loop is the standard $(\min, +)$-matrix multiplication all pairs shortest path algorithm; provided we do not violate the adjacency matrix, its correctness follows from the correctness of such shortest path algorithms. An adjacency matrix has been violated when it can no longer be used to correctly determine shortest paths using the same $(\min, +)$-shortest-path algorithm. That is, we could violate an adjacency matrix by using edges that are *not* in $E(D_n)$ or edges that are to long at the wrong iteration of the algorithm.

Justification of the next two lines follows by induction on jumper length. We only consider horizontal jumpers here, vertical jumpers follow immediately.

After the first iteration of the algorithm $sp(j,k)$ has been correctly calculated for all (j,k) where $k-j<2$. After iteration r, for some $r \geq 1$, $sp(j,k)$ has been calculated for all (j,k) such that $k-j<2^r$. At the start of iteration $r+1$, $sp(j,k)$ has already been computed for all (j,k) such that $k-j<2^r$ by the inductive hypothesis. During iteration $r+1$ the algorithm computes shortest paths of length between 2^r-1 and 2^{r+1}. By Lemma 2, this gives $sp(i,t)$, for all (i,t) such that $t-i<2^{r+1}$.

Next in line 2, during iteration $r+1$ the relaxation of straight vertical unit paths of length ranging from 2^r-1 to 2^{r+1} with vertical jumpers occurs. (A *unit path* is a path without any jumpers. A unit path has only unit edges.)

This is done by replacing $W[(i,j),(i,t)]$ with

$$\min\{\ W[(i,j),(i,t)],\ W[(0,0),(j+1,t)] + f(i,j,t)\ \}$$

for all $(j+1,t)$ such that $t-(j+1)<2^{r+1}$. This does not violate the adjacency matrix since any new distance cost may be replaced by smaller, but positive, values because they have not been used yet in the calculation of other shortest paths. So relax the paths of length from $2^{r-1} - 1$ to 2^r with the appropriate jumpers of the same lengths. □

The algorithm in Figure 13 solves the three problems given in Chapter 2 in $O(\lg^2 n)$ time with $n^6/\lg n$ processors.

Theorem 4 shows that D_n graphs can be constructed quickly in parallel. These results are similar to those in (Rytter, 1988).

Any cost function $f(i, j, k)$ can be used to solve the MPP as long as the shortest paths in the appropriate D_n graph remain the same.

4.5 Historical Notes

There is an interesting history of parallel solutions to the MCOP in the 1980s.

In 1983, (Valiant et al., 1983) showed that many problems with simple $O(n^3)$ sequential dynamic programming solutions are in the class $\mathcal{NC}$. One of the problems they were considering was the MCOP. They used straight line programs to solve all of these problems in $O(\lg^2 n)$ time and with n^9 processors. In 1988, Rytter used pebbling games to show that these same problems can be solved in $O(\lg^2 n)$ time with $n^6/\lg n$ processors.

Most of these models, especially Rytter's, are similar to ours. Our model (and similar ones) have lots of merit in their own right. Furthermore, for various reasons it is important to have a re-characterization of Hu and Shing's work on the MCOP. Also, our model is general enough to capture more than just Hu and Shing's work on the MCOP.

In 1990, the paper (Huang et al., 1990) improved Rytters result by giving an $O(\lg^2 n)$ time algorithm that uses $n^6/\lg^5 n$ processors. Further, (Huang et al., 1992) gives an $O(\sqrt{n} \lg n)$ time algorithm using $O(n^{3.5}/\lg n)$ processors on a CREW PRAM.

Also (Galil and Park, 1992a) gave algorithms that can solve the MCOP (and other problems) in $O(\lg^2 n)$ time and $n^6/\lg^6 n$ processors.

We will continue discussing the more recent parallel solutions to the MCOP in the next chapters as they are relevant to our work.

Chapter 5

Special D_n Graphs for the MCOP

This chapter contains results that allow the decomposition of D_n graphs specially for solving the MCOP. As before, the associative product costs are computed as $f(i,j,k) = w_i w_{j+1} w_{k+1}$ where $w_s = w_t$ iff $s = t$. All weights are positive integers and they are distinct only for convenience. These associative product costs model the matrix chain order problem and the minimum cost triangulation of convex polygons.

The key insight in this chapter is the matrix dimensions of a given instance of the MCOP in some sense approximate the depths of parenthesis in an optimal solution to the MCOP. Construct a D_n graph suited to solve an instance of the MCOP. This chapter shows the matrix dimensions of this MCOP instance give important structural information about the D_n graph.

5.1 Nesting Levels of Matching Parentheses

This section contains an algorithmic technique that is central to the rest of this dissertation. Intuitively, this technique allows matrix dimensions to approximate the nesting level of matched parentheses.

Subsection 5.1.1 gives an important invariance theorem that makes the exposition of these results much easier. Subsection 5.1.2 gives results that show matrix dimensions can approximate the nesting level of matched parentheses in some sense.

5.1.1 An Invariance Theorem

From here on D_n graphs are the central focus since the greedy version of this problem has been solved efficiently in (Apostolico et al., 1990; Aggarwal and Park, 1988; Ibarra et al., 1988).

Let γ be a cyclic rotation function; in other words γ is a one-to-one and onto function on the set $\{1, 2, \cdots, n+1\}$ such that for some $k : 0 \leq k \leq n$ we have $\gamma(i) = (i + k \bmod n + 1) + 1$.

Theorem 5 (Deimel and Lampe, 1979) and (Hu and Shing, 1982) Given an instance of the MCOP with the weight list $l_1 = w_1, w_2, \ldots, w_{n+1}$ and cyclically rotating it getting $l_2 = w_{\gamma(1)}, w_{\gamma(2)}, \ldots, w_{\gamma(n+1)}$, then finding an optimal parenthesization with l_2 provides an optimal solution to the original instance of the MCOP with l_1.

Directly from this last theorem and Theorem 3 we have the next corollary.

Corollary 1 Finding a shortest path in a D_n graph whose edge weights are constructed from either a weight list l or a cyclic rotation of l gives an optimal solution to the MCOP.

In the rest of this dissertation let w_1 denote the smallest weight in any weight list.

5.1.2 Matrix Dimensions as Nesting Levels of Matching Parentheses

Given an associative product where the level of each parenthesis in an optimal product is known, we can compute the parenthesization of this associative product by solving the following all nearest smaller value (ANSV) problem (Berkman et al., 1989; Berkman et al., 1993):

Given $w_1, w_2, \ldots, w_n$ drawn from a totally ordered set, for each w_i find the largest j, where $1 \leq j < i$, and smallest k where $i < k \leq n$, so that $w_j < w_i$ and $w_k < w_i$ if such values exist.

A weight list may contain weights without any matches. For example, in a monotone weight list there are no ANSV matches for any weight.

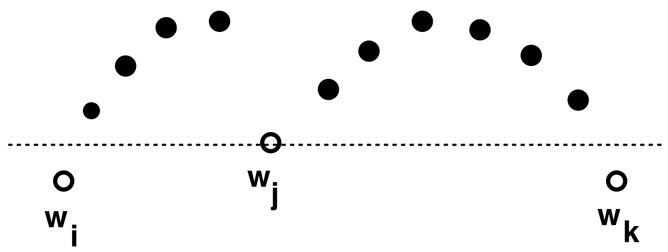

Figure 14: The Weight w_j with its Match $[w_i, w_k]$

Given only a sequence of integers representing the depth of parentheses in an associative product (for example see the bottom row in Figure 15), then by solving the ANSV problem for each parenthesis we can compute its matching parenthesis (see the top row in Figure 15). The correspondence between the ANSV and the nesting levels of matching parenthesis can be seen by comparing Figures 14 and 15.

Now we investigate the effect monotonicity has on weight lists. This is interesting because we can solve any instance of the MCOP very efficiently if its weight list is monotonic.

(	(	(	)	)	(	(	(	)	)	)	)
0	1	2	2	1	1	2	3	3	2	1	0

Figure 15: Parentheses and their Depths

As in (Hu and Shing, 1980) let

$$\|w_i : w_k\| = \sum_{j=i}^{k-1} w_j w_{j+1},$$

and all such pair-wise sums can be computed by performing one parallel partial prefix.

A partial prefix sum of the $n+1$ weights $w_1, \cdots, w_{n+1}$ is the list of values:

$$w_1, \quad w_1 + w_2, \quad w_1 + w_2 + w_3, \; \cdots \; , \; \sum_{i=1}^{n+1} w_i$$

That is, the ith partial prefix element is

$$\sum_{j=1}^{i} w_j.$$

Sequentially, a partial prefix sum can be computed easily in $O(n)$ time which is asymptotically optimal. Also, a partial prefix sum can be computed easily in parallel with $O(n)$ work and in $O(\lg n)$ time (JáJá, 1992; Kumar et al., 1994).

Theorem 6 (Hu and Shing, 1980) The vector $F[i] = \|w_1 : w_i\|$ can be computed by a parallel partial prefix.

As suggested by this last theorem, we can compute $\|w_i : w_k\|$ by performing one subtraction $\|w_i : w_k\| = F[k] - F[i]$. Therefore calculating the cost of any horizontal or vertical unit path in D_n can be done by multiplying such a sum by the appropriate weight.

Let the i^{th} row of D_n be all vertices of the form (i, k) for $1 \leq k \leq n$ and the k^{th} column be vertices of the form (i, k) for $1 \leq i \leq k$. So, the unit path along the i^{th} row costs $w_i \| w_{i+1} : w_{n+1} \|$ where the unit path in the k^{th} column costs $w_{k+1} \| w_1 : w_k \|$.

Lemma 3 Given a D_n graph with a weight list containing a sublist of increasing weights $w_i < w_{i+1} < \cdots < w_{k+1}$, in D_n the unit path $(0,0) \nearrow (i,i) \rightarrow \cdots \rightarrow (i,k)$ is cheaper than the unit path $(0,0) \nearrow (k,k) \uparrow (k-1,k) \uparrow \cdots \uparrow (i,k)$.

Proof: Column k costs

$$w_{k+1} \| w_i : w_k \| = w_{k+1} \sum_{j=i}^{k-1} w_j w_{j+1}$$

and row i costs

$$w_i \sum_{j=i+1}^{k} w_j w_{j+1}$$

which is

$$w_i \sum_{j=i}^{k-1} w_{j+1} w_{j+2}.$$

Clearly, $w_{k+1} > w_i$ and

$$\sum_{j=i}^{k-1} w_{j+1} w_{j+2} > \sum_{j=i}^{k-1} w_j w_{j+1},$$

hence the lemma follows. □

Symmetrically to the above we have the following.

Given a sublist of decreasing product weights

$$w_i > w_{i+1} > \cdots > w_{k+1}$$

then the horizontal unit path

$$(0,0) \nearrow (i,i) \rightarrow \cdots \rightarrow (i,k)$$

is more expensive than

$$(0,0) \nearrow (k,k) \uparrow (k-1,k) \uparrow \cdots \uparrow (i,k).$$

The relative expense of such paths is key to the intuition that leads to the development of the rest of this chapter.

The two unit paths,

$$\mathcal{A} = (i,i) \rightarrow \cdots \rightarrow (i,k)$$

and

$$\mathcal{B} = (i+1,i+1) \rightarrow \cdots \rightarrow (i+1,k) \uparrow (i,k)$$

are *adjacent* horizontal unit paths.

Lemma 4 (Row Trade Off Lemma) Given two adjacent horizontal unit paths $\mathcal{A}$ and $\mathcal{B}$ going to (i,k), which cost $w_i \| w_{i+1} : w_{k+1} \|$ and $w_{i+1} \| w_{i+2} : w_{k+1} \| + w_i w_{i+1} w_{k+1}$ respectively, one of the following conditions may hold:

- if $w_i < w_{i+1}$ and $w_{i+2} < w_{k+1}$, then $\mathcal{A}$ is cheaper
- if $w_i > w_{i+1}$ and $w_{i+2} > w_{k+1}$, then $\mathcal{B}$ is cheaper

Proof: Since $\mathcal{A}$ costs $w_i \| w_{i+1} : w_{k+1} \|$ and $\mathcal{B}$ costs $w_{i+1} \| w_{i+2} : w_{k+1} \| + w_i w_{i+1} w_{k+1}$, the difference in their costs is

$$d = w_i \| w_{i+1} : w_{k+1} \| - w_{i+1} \| w_{i+2} : w_{k+1} \| - w_i w_{i+1} w_{k+1}.$$

That is, $d = (w_i - w_{i+1}) \| w_{i+2} : w_{k+1} \| + (w_{i+2} - w_{k+1}) w_i w_{i+1}$ and when $w_i > w_{i+1}$ and $w_{i+2} > w_{k+1}$ then d is positive, hence $\mathcal{B}$ is cheaper.

The other case follows similarly. $\square$

There is a similar Column Trade Off Lemma, and its proof is almost identical. Cases such as $w_i < w_{i+1}$ and $w_{i+2} > w_{k+1}$ are not dealt with in either the Row or Column Trade Off Lemmas.

5.2 Critical Nodes in D_n

This section relates the ANSV problem to the Row and Column Trade Off Lemmas. By solving the ANSV problem it is possible to isolate certain nodes that are central to finding shortest paths in D_n graphs.

The next definition is originally due to (Hu and Shing, 1980), although they present it in a different framework:

In D_n, a *critical node* is a node (i, k) such that $w_j > \max\{w_i, w_{k+1}\}$ for all $i < j \leq k$.

There are no critical nodes of the form (j, j) for $1 \leq j \leq n$.

The row and column equations are,

$$\begin{array}{rcll} \mathcal{R} & = & (w_i - w_{i+1})\|w_{i+2} : w_{k+1}\| + (w_{i+2} - w_{k+1})w_i w_{i+1} & k+1 \neq i+2 \\ \mathcal{C} & = & (w_{k+1} - w_k)\|w_{k-1} : w_i\| + (w_{k-1} - w_i)w_{k+1} w_k & k-1 \neq i \end{array}$$

Lemma 4 elucidates a key observation about the row and column equations. Particularly, only the order relationships of four weights is enough to determine whether equation $\mathcal{R}$ will be *necessarily* positive or *necessarily* negative. A similar observation holds for $\mathcal{C}$. This cannot be done for either $\mathcal{C}$ or $\mathcal{R}$ when both conditions $w_i < w_j$ and $w_j > w_{k+1}$, for $i < j \leq k$, hold. Generalizing for the possibility of an edge-connected path of critical nodes and associated row and column equations we say that $\mathcal{C}$ and $\mathcal{R}$ are order *indeterminate* when $w_j > \max\{w_i, w_{k+1}\}$ for all $j, i < j \leq k$. Critical nodes determine where both the row and column equations fail to provide any information. Further, critical nodes indicate where magnitude can overtake order in the row and column equations.

By solving the ANSV problem we can compute all critical nodes of a D_n graph. In our nomenclature, (Berkman et al., 1989) shows on a common-CRCW PRAM and in (Chen 1990; Kim 1990) on a EREW PRAM:

Theorem 7 On a common-CRCW PRAM all critical nodes can be computed in $O(\lg\lg n)$ and $O(\lg n)$ time using $n/\lg\lg n$ and $n/\lg n$ processors, respectively. On an EREW PRAM all critical nodes can be computed in $O(\lg n)$ time using $n/\lg n$ processors.

In our context, this theorem follows trivially as a result of the relationship of matches in the weight list and critical nodes in D_n graphs.

Critical nodes (i, s) and (j, t) are not *compatible* when $s-i = t-j$ and there exists some vertex other than $(0, 0)$ that can reach both (i, s) and (j, t) by a unit path.

The next theorem was originally proved by Hu and Shing in a different framework and follows from the properties of ANSV matches.

Theorem 8 (Hu and Shing, 1982) In any instance of the matrix chain ordering problem all critical nodes are compatible.

Proof: Suppose D_n represents an instance of the matrix chain ordering problem with two non-compatible critical nodes (i, s) and (j, t). Since these are critical nodes, it must be that either $j < s < t$ or $i < j < s$. Either case gives a contradiction since both $w_k > \max\{w_i, w_{s+1}\}$, for $i < k < s+1$, and $w_r > \max\{w_j, w_{t+1}\}$, for $j < r < t+1$, cannot hold. $\Box$

A D_n graph can have as many as $n-1$ and as few as zero critical nodes. For example, a monotone weight list does not have any critical nodes.

The next lemma follows from the ANSV characterization of critical nodes.

Lemma 5 (Hu and Shing, 1982) Any D_n graph has at most $n-1$ critical nodes.

Proof: Take a list of $n+1$ weights $w_1, w_2, \ldots, w_{n+1}$ making up the edge weights of some D_n graph.

A *representative weight* for the critical node (i, k) is the weight w_j with match $[w_i, w_{k+1}]$. Each unique critical node has a unique representative weight. Further, a list of $n + 1$ weights can have at most $n - 1$ representative weights, since neither w_1 nor w_{n+1} can be representative weights. Hence, there can be at most $n - 1$ critical nodes. $\Box$

A jumper $(i, j) \Longrightarrow (i, v)$ is said to *include* all critical nodes that are in some path r from $(0, 0)$ to $(j + 1, v)$, where r may contain jumpers too. Suppose there is a path r from $(0, 0)$ to $(j + 1, v)$ that includes all critical nodes in the set $\{ (k, u) \}$, for all $j + 1 \leq k < u \leq v$. Then we say any path q, containing only this one jumper $(i, j) \Longrightarrow (i, v)$, *includes* all critical nodes that are vertices of q and all critical nodes in r. Now generalize this notion recursively to paths with more than one jumper. Going even further, we can prove the next theorem.

Theorem 9 In a D_n graph there is at least one path from $(0, 0)$ to $(1, n)$ that includes all critical nodes.

A proof of this theorem follows from the fact that all ANSV matches are compatible and each match represents a pair of parentheses. If a solution of the ANSV problem gives a parenthesization of all associative elements, then we are done. So consider the case where a solution of the ANSV problem only provides a partial parenthesization of the associative product. In this case, any arbitrary but legal completion of the parenthesization describes a path in the associated D_n graph.

5.3 Canonical Subgraphs of D_n

This section investigates the interaction between monotonicity and critical nodes. Weight lists can be broken into monotone sublists and sublists that have ANSV matches. Such sublists naturally lead to subgraphs that are useful for finding shortest paths.

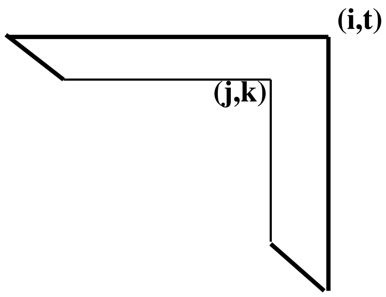

Figure 16: A $D^{(i,t)}_{(j,k)}$ Graph without $(0,0)$ and with No Jumpers Shown

A subgraph $D(i,t)$ of D_n is all vertices and edges of D_n that can reach (i,t) by a unit path; this includes $(0,0)$. Formally, $D(i,t)$ is all vertices $(j,k) \in V[D_n]$ such that $i \leq j \leq k \leq t$ in addition to the associated edges. The node $(0,0)$ is also in each subgraph. A *graph* $D(i,j)$ is monotonic iff the weights $w_i, \ldots, w_{j+1}$ are monotonic.

Critical nodes may form a unit edge-connected maximal path p that isolates a subgraph. Suppose that p forms a contiguous maximal path starting at some critical node (j,k), where $k - j \geq 1$ and terminating at the critical node (i,t), then p isolates $D^{(i,t)}_{(j,k)}$. Saying the unit path p is maximal means that if there is any unit edge-connected path q of critical nodes that p is a subpath of, then $p = q$.

A *canonical subgraph* $D^{(i,t)}_{(j,k)}$ is the subgraph containing the maximal contiguous edge-connected path of critical nodes that begins at critical node (j,k) and terminates at critical node (i,t).

The canonical subgraph $D^{(i,t)}_{(j,k)}$ has vertex set

$$V[D(i,t)] - V[D(j+1,k-1)] \cup \{(0,0)\}$$

and the associated edges, see Figure 16. We write $D^{(i,t)}$ for a canonical subgraph of the form $D^{(i,t)}_{(j,j+1)}$.

A $D^{(i,t)}$ canonical graph is a *leaf* graph. A $D^{(i,t)}_{(j,k)}$ canonical graph, where $k > j+1$, is a *band* graph.

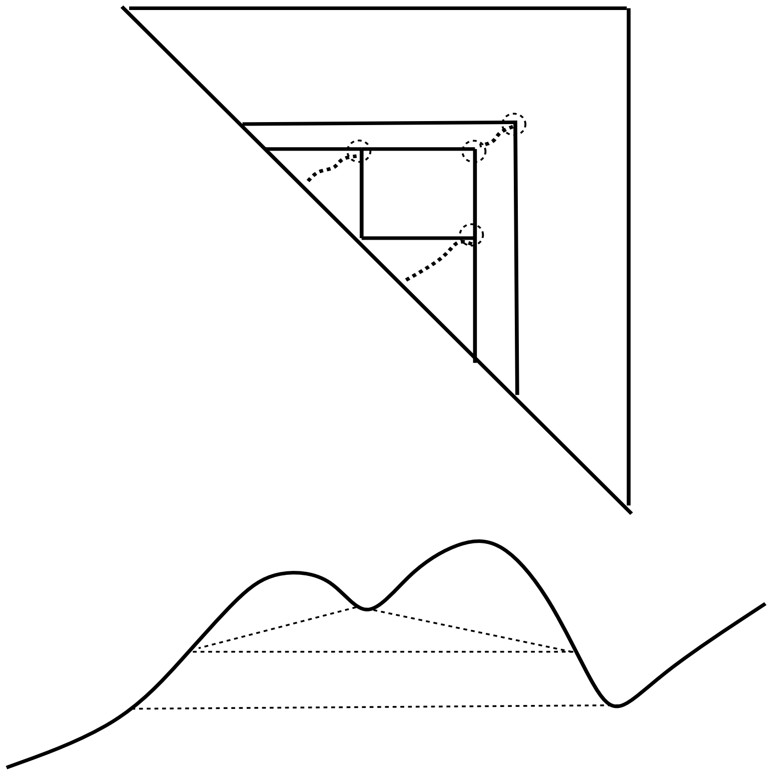

Figure 17: Several Canonical Subgraphs and Their Weight List

Generally p denotes the path of critical nodes in a canonical graph.

For instance, a leaf graph $D^{(i,t)}$ is a graph that in some sense isolates the weight list $w_i, w_{i+1}, \ldots, w_{t+1}$ where $w_k > \max\{w_j, w_s\}$ for all k, such that $i \leq j < k < s \leq t+1$.

In Figure 17 the weight list is represented as the contour below the D_n graph. In this contour, four key ANSV matches are represented by dotted lines. The four corresponding critical nodes are circled in the related D_n graph.

Canonical subgraphs are easily distinguishable by properties of their critical nodes. So by Theorem 7, we can find all canonical subgraphs in $O(\lg\lg n)$ time using $n/\lg\lg n$ processors on a common-CRCW PRAM or in $O(\lg n)$ time using $n/\lg n$ processors on a common-CRCW PRAM or EREW PRAM. Notice that there are only two basic kinds of canonical subgraphs.

Theorem 10 (Monotonicity Theorem) Given a $D(i,u)$ graph with a monotone list of weights $w_i < w_{i+1} < \cdots < w_{u+1}$, the shortest path from $(0,0)$ to (i,u) is along the straight unit path $(0,0) \nearrow (i,i) \to (i,i+1) \to \cdots \to (i,u)$.

Proof: The proof is in two cases. In the first Case, the theorem is shown to be true for G_n graphs while the second Case shows that the theorem also holds for D_n graphs.

Case i: Take a unit path in $G(i,u)$

Let $(0,0) \nearrow (i,i) \to (i,i+1) \to \cdots \to (i,u)$ be a horizontal path with the associated vertical path $(0,0) \nearrow (u,u) \uparrow (u-1,u) \uparrow \cdots \uparrow (i,u)$.

By Lemma 3, the horizontal path is cheaper. Inductive application of the Row and Column Trade Off Lemmas (see Lemma 4) completes this case.

Case ii: Take a path with jumpers in $D(i,u)$

This case only addresses horizontal jumpers, since vertical jumpers follow by symmetry and Lemma 3.

Given an arbitrary path r from $(0,0)$ to (i,u) in $D(i,u)$ that includes one jumper, the unit path from (i,i) to (i,u) is shown to be cheaper than r. Suppose the jumper in this path is $(k,s) \Longrightarrow (k,t)$, for $i < k < s < t < u$. Since this is the only jumper in r, the path to (k,s) is $(k,k) \to \cdots \to (k,s)$.

Therefore, this problem has been reduced to finding a shortest path to (k,t). For, if the shortest path to (k,t) is a unit path, then r cannot be a shortest path by Case i, because $(k,s) \Longrightarrow (k,t)$ is the *only* jumper in r.

Still, since $W((k,s) \Longrightarrow (k,t)) = sp(s+1,t) + f(k,s,t)$ we must consider jumpers in the shortest path to $(s+1,t)$. Say a shortest path to $(s+1,t)$ is a unit path, then by Case i it is the straight unit path $(s+1,s+1) \to \cdots \to (s+1,t)$. Since $f(k,s,t) = w_k w_{s+1} w_{t+1}$ and $W((k,s) \to (k,s+1)) =$

$w_k w_{s+1} w_{s+2}$, it must be that $t > s+1$ so $f(k,s,t) > W((k,s) \to (k,s+1))$. Intuitively, we are trading the associative product cost $f(k,s,t)$ for the first edge of $(k,s) \to \cdots \to (k,t)$. This first unit edge is cheaper than the associative product cost.

With this, because $(k,s+1) \to \cdots \to (k,t)$ costs $w_k \| w_{s+2} : w_{t+1} \|$ and the path $(s+1,s+1) \to \cdots \to (s+1,t)$ costs $w_{s+1} \| w_{s+2} : w_{t+1} \|$ and since $w_k < w_{s+1}$ the theorem holds.

Otherwise, say there is a horizontal jumper in r to $(s+1,t)$. Apply this case again inductively, until there is a jumper that derives its weight from a straight unit path. We are trading each jumper's associative product cost for the cost of the first unit edge in the associated unit path. These first unit edges are always cheaper than the related associative product cost. Eventually, the shortest path to (k,t) is shown to be $(k,k) \to \cdots \to (k,t)$. Hence, r is a unit path, but this means that it must be the straight unit path in row i by Case i.

Handling a path with more than one jumper is straightforward. Inductively applying these two cases to jumpers successively farther away from $(0,0)$ completes the proof. $\square$

This theorem also holds for a monotone list of weights having the relation

$$w_i > w_{i+1} > \cdots > w_{j+1}$$

where the shortest path is

$$(0,0) \nearrow (j,j) \uparrow (j-1,j) \uparrow \cdots \uparrow (i,j).$$

Therefore, if the list of weights $w_i, w_{i+1}, \ldots, w_{j+1}$ is monotone, then we do not have to construct any jumpers in $D(i, j)$.

We say that $D(i, t)$ *intersects* $D(j, v)$ iff $V[D(i, t)] \cap V[D(j, v)] - \{(0, 0)\} \neq \emptyset$. From here on, references to monotone subgraphs assume that the monotone subgraphs do not intersect any canonical subgraphs. This is because canonical subgraphs contain monotone subgraphs, but such monotone subgraphs are not of particular interest to us.

Theorem 11 If $D(i, t)$ does not intersect any canonical graphs and has no critical nodes, then the weight list $w_i, w_{i+1}, \ldots, w_{t+1}$ is monotonic.

Proof: Say there are no critical nodes in $D(i, t)$. Then there is at most *one* weight w_{j+1} where $j + 1 \neq 1$ such that

$$w_i > \cdots w_j > w_{j+1} < w_{j+2} < \cdots < w_{t+1},$$

otherwise $D(i, t)$ would contain critical nodes. But in this case, since $w_1 = \min_{1 \leq i \leq n+1} \{w_i\}$ it must be that $D(1, j + 1)$ contains the critical node $(1, j)$, which means that $D(i, t)$ would intersect with a canonical subgraph.

On the other hand, this means both the row and column equations begin and remain indeterminate so the next fact holds.

> Fact 1: For all $(j, j + 2) \in V[D(i, t)]$, either $w_j < w_{j+1} < w_{j+2} < w_{j+3}$ or $w_j > w_{j+1} > w_{j+2} > w_{j+3}$.

Applying the row or column equations to the nodes $(j, j + 2)$, for all $j, i < j \leq t$, establishes this fact. For instance, let

$$\mathcal{R} = (w_j - w_{j+1})\|w_{j+2} : w_{j+3}\| + (w_{j+2} - w_{j+3})w_j w_{j+1},$$

then $\mathcal{R}$ is order determinant so it must be either $w_j < w_{j+1}$ and $w_{j+2} < w_{j+3}$, or $w_j > w_{j+1}$ and $w_{j+2} > w_{j+3}$, but not both.

Suppose $w_j < w_{j+1}$ and $w_{j+2} < w_{j+3}$, and assume that $w_{j+1} > w_{j+2}$, otherwise if $w_{j+1} < w_{j+2}$, then $w_j < w_{j+1} < w_{j+2} < w_{j+3}$ so we are done. This means $w_j < w_{j+1}$ and $w_{j+1} > w_{j+2}$, therefore $w_{j+1} > \max\{w_j, w_{j+2}\}$ and this indicates $(j, j+1)$ is a critical node. This is a contradiction, hence it must be that $w_j < w_{j+1} < w_{j+2} < w_{j+3}$.

Using Fact 1, the theorem follows inductively. $\square$

A lack of critical nodes implies the existence of a monotone subgraph. Just the same, a lack of ANSV matches in a section of a weight list indicates a monotone sublist. Assuming that $D(i,t)$ does not intersect with any canonical graphs gives the next corollary.

Corollary 2 If $D(i,t)$ contains no critical nodes, then there are no jumpers in a shortest path from $(0,0)$ to (i,t). Moreover, if $D(i,t)$ contains no critical nodes, then the shortest path from $(0,0)$ to (i,t) is a straight unit path.

A proof of this follows immediately from Theorems 10 and 11.

Take a canonical subgraph $D^{(1,m)}$, where all critical nodes have been found and form a unit edge-connected path p. Removing the nodes and adjacent edges of p splits $D^{(1,m)}$ in two. These two pieces of $D^{(1,m)}$ are $\mathcal{U}$ for *upper* and $\mathcal{L}$ for *lower*. See Figure 18.

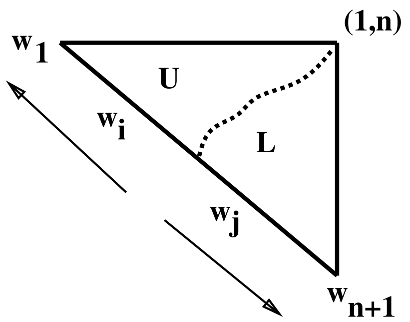

Figure 18: A D_n Graph Split by a Path of Critical Nodes, Arrows Point Toward Smaller Weights

Let $D(1,s)$ be the maximal well-formed subgraph of $\mathcal{U}$ and let $D(s+1,m)$ be the maximal well-formed subgraph of $\mathcal{L}$. By the maximality of $D(i,t)$ in $\mathcal{U}$ we mean that for any other well-formed subgraph $D(j,k)$, if $D(j,k) \subseteq \mathcal{U}$ then $D(j,k) \subseteq D(i,t)$.

Theorem 12 (Modality Theorem) If $D(1,s) \subseteq \mathcal{U}$ and $D(s+1,n) \subseteq \mathcal{L}$, where both $D(1,s)$ and $D(s+1,m)$ are maximal, then $w_1 < w_2 < \cdots < w_{s+1}$ and $w_{s+2} > w_{s+3} > \cdots > w_m > w_{m+1}$.

Proof: The path p splits $D^{(1,m)}$ into $\mathcal{U}$ and $\mathcal{L}$ where $D(1,s)$ and $D(s+1,n)$ are maximal, so $(s,s+1)$ is a critical node. Therefore, it must be that $w_{s+1} > \max\{w_s, w_{s+2}\}$, so $w_s < w_{s+1}$ and $w_{s+1} > w_{s+2}$.

By Theorem 11, the weight lists $w_1, w_2, \ldots, w_s, w_{s+1}$ and $w_{s+1}, w_{s+2}, \ldots, w_m, w_{m+1}$ are both monotonic. Thus $w_s < w_{s+1}$, so it must be that $w_1 < w_2 < \cdots < w_s < w_{s+1}$. In addition, because $w_{s+1} > w_{s+2}$, so $w_{s+1} > w_{s+2} > \cdots > w_m > w_{m+1}$. □

Take a $D^{(i,t)}_{(j,k)}$ canonical graph, then both

$$w_i < w_{i+1} < \cdots < w_j \text{ and } w_k > w_{k+1} > \cdots > w_{t+1}$$

follow from Theorem 12.

5.4 Historical Notes

Theorem 5 was first proved by (Deimel and Lampe, 1979) and a simpler proof was later given by (Hu and Shing, 1982).

There seems to be two general approaches to improving the complexity of finding minimal paths in special graphs. The first is to use specific properties of the graphs to get more efficient renditions of the $(\min,+)$-matrix based shortest-path algorithms. This approach can be seen in (Aggarwal et al., 1987; Aggarwal and Park, 1988; Apostolico et al., 1990), where they use Monge and monotone properties to improve the processor complexity of the standard $(\min,+)$-matrix multiplication. The second approach breaks the graph up while only keeping a very small fraction of the pieces. These pieces can be worked on in parallel and the results of each of these computations can be joined giving a complete solution to the original problem. The work in this dissertation is based on a divide-and-conquer approach.

Chapter 6

Approximating the MCOP

This chapter contains a fast parallel approximation algorithm for the MCOP. This algorithm can run in $O(\lg n)$ time using only $n/\lg n$ processors on both a common-CRCW PRAM and a EREW PRAM. Alternatively it can run in $O(\lg\lg n)$ time using $n/\lg\lg n$ processors on a common-CRCW PRAM.

The algorithm given in this section is based on two applications of the ANSV problem.

6.1 A Parallel Approximation Algorithm for the MCOP

This section contains a $O(\lg\lg n)$ time and $n/\lg\lg n$ processor approximation algorithm for the MCOP. This algorithm is built by combining results of (Chin, 1979), and (Hu and Shing, 1981) with those of (Berkman et al., 1989). This algorithm approximates the MCOP to within 15.5% of optimality. In addition, the processor-time product of this algorithm is linear.

This algorithm is not much more than several applications of the ANSV problem, so various processor complexities and times result in applications of Theorem 7.

The approximation algorithm consists of two stages. The first stage isolates relatively heavy weights by finding matrix products that must be in an optimal parenthesization. The isolation of such heavy weights provides optimal substructures that are in optimal superstructures—essentially giving a converse to the principle of optimality. The second stage is simply a greedy approach for finding a parenthesization once we have applied the first stage of the algorithm. Therefore, this is basically a greedy algorithm, but there are no lexicographical constraints on it.

By Corollary 1 rotate any given weight list so that w_1 is the smallest weight. For the next theorem let w_i, w_{i+1}, and w_{i+2} be three adjacent weights in a weight list of an instance of the MCOP where $w_i < w_{i+1}$ and $w_{i+1} > w_{i+2}$ which together means that $w_{i+1} > \max\{w_i, w_{i+2}\}$.

Theorem 13 (Hu and Shing, 1981) If

$$w_1 w_i w_{i+2} + w_i w_{i+1} w_{i+2} < w_1 w_i w_{i+1} + w_1 w_{i+1} w_{i+2} \tag{3}$$

then the product $(a_i \bullet a_{i+1})$ is in an optimal parenthesization.

Proof of this Theorem is left to the literature, see (Chin, 1979) and (Hu and Shing, 1981) for different proofs. When $w_{i+1} > \max\{w_i, w_{i+2}\}$ fails to hold Equation 3 cannot hold, so there is no gain in assuming $w_{i+1} > \max\{w_i, w_{i+2}\}$.

Corollary 3 If Equation 3 holds, then $w_{i+1} > \max\{w_i, w_{i+2}\}$.

A proof follows from Equation 3 with

$$w_1 = \min_{1 \le i \le n+1} \{w_i\},$$

and

$$w_1 w_i (w_{i+2} - w_{i+1}) + w_{i+1} w_{i+2} (w_i - w_1) < 0$$

so it must be that $w_{i+2} - w_{i+1} < 0$. Also, starting with Equation 3 again and reassociating gives

$$w_1 w_{i+2}(w_i - w_{i+1}) + w_i w_{i+2}(w_{i+2} - w_1) < 0$$

so $w_i - w_{i+1} < 0$.

Unfortunately, the converse of this last corollary is not true. An ANSV match may not represent a minimal parenthesization in the MCOP. But any product that is in a minimal parenthesization by way of Equation 3 has been isolated by some match. Therefore, using the ANSV problem, the values in the weight list *approximate* the optimal level of parentheses.

A list of weights is *reduced* iff for all weights, say w_{i+1}, with ANSV match $[w_i, w_{i+2}]$ Equation 3 fails to hold (Chin, 1979). A reduced weight list may be non-monotonic.

Generalizing Equation 3 is done as follows. Suppose by Theorem 13 that $(a_i \bullet a_{i+1})$ is in an optimal parenthesization. Applying Theorem 13 to the list

$$l = w_1, \ldots w_{i-1}, w_i, w_{i+2}, w_{i+3}, \ldots, w_{n+1}$$

works in the same way. That is, if $w_i > \max\{w_{i-1}, w_{i+2}\}$ and $w_1 w_{i-1} w_{i+2} + w_{i-1} w_i w_{i+2} < w_1 w_{i-1} w_i + w_1 w_i w_{i+2}$, then the parenthesization given by the solution of the ANSV problem on l indicates that $(a_{i-1} \bullet \cdots \bullet a_{i+1})$ is optimal.

Given the weight list $l = w_1, w_2, \ldots, w_{n+1}$ the approximation algorithm is (Chin, 1979; Hu, 1982; Hu and Shing, 1981):

1. Reduce the weight list l giving the weight list l_2, renumbering l_2 to be $l_2 = w_1, w_2, \ldots, w_{r+1}$ where $w_1 = \min_{1 \le i \le r+1} \{w_i\}$.
2. If l_2 has more than two weights, then compute the depths of the parentheses for the linear product $((\cdots(a_1 \bullet a_2) \bullet \cdots) \bullet a_r)$ of cost $w_1 \| w_2 : w_{r+1} \|$. With this,

make the parenthesis discovered in Step 1 the appropriate amount deeper.

The depth of the parentheses determines the order to multiply the matrices.

Next techniques are given to run this algorithm efficiently in parallel. Intuitively, by Theorem 13, if the match $[w_{i-1}, w_{i+1}]$ represents the nesting level of two parentheses in an optimal product, then we have characterized w_i's influence. Remove w_i from the weight list and recursively apply Theorem 13.

Suppose in solving the ANSV problem the weight w_j has the match $[w_i, w_k]$. Then, if

$$w_1 w_i w_k + w_i w_j w_k < w_1 w_j w_k + w_1 w_i w_j \tag{4}$$

and products $(a_i \bullet \cdots \bullet a_{j-1})$ and $(a_j \bullet \cdots \bullet a_{k-1})$ are both in an optimal parenthesization, then the product $(a_i \bullet \cdots \bullet a_{j-1}) \bullet (a_j \bullet \cdots \bullet a_{k-1})$ is also in an optimal parenthesization. Certainly, by Theorem 13 this is true when $i = j - 1$ and $j = k - 1$. In addition, Corollary 3 generalizes to suit Equation 4.

A weight list can be reduced by two applications of the ANSV problem as follows.

Given the weight list l the next algorithm outputs a reduced weight list. Let $A[1 \ldots n+1]$ be an array of $n+1$ integers all initialized to zero.

1. Solve the ANSV problem on the weight list l. Next check to see if there are any weights satisfying the condition described by Equation 3. If there are none, then output l since it is reduced, then stop.

2. For all weights w_j in l that have matches, say $[w_i, w_k]$, if w_j and w_i, w_k, satisfy Equation 4, then assign a 1 to $A[j]$.

3. Now solve the ANSV problem on $A[1 \ldots n+1]$. If the nearest smaller values of $A[j]$ are in the match $[A[i], A[k]]$, then $(a_i \bullet \cdots \bullet a_{k-1})$ is in an optimal parenthesization. Removing all of the weights isolated by optimal parenthesizations

gives a reduced weight list, which is output.

This algorithm produces a reduced weight list and optimal parenthesizations that have been isolated by the conditions of Equation 3.

The first step of this algorithm is correct by Theorem 13 and Corollary 3. The next theorem establishes the correctness of the last two steps of the algorithm.

Theorem 14 If the ANSV match of $A[j]$ is $[A[i], A[k]]$ where $i < k$, then the product $(a_i \bullet \cdots \bullet a_{k-1})$ is in an optimal parenthesization.

Proof: The array A contains values from the set $\{0, 1\}$, so if $A[j] = 0$ then $A[j]$ does not have an ANSV match. On the other hand, if $A[j] = 1$ then $A[j]$ must have an ANSV match since $A[1] = 0$ and $A[n+1] = 0$.

Now consider the case where $A[j]$ has match $[A[i], A[k]]$. This means for all t such that $i < t < k$, $A[t]$ also has match $[A[i], A[k]]$. All of these matches are compatible, consequently all $A[t] = 1$ for $i < t < k$ are nested ANSV matches. This means there must be at least one list of three consecutive weights, say w_t, w_{t+1}, and w_{t+2}, that satisfy Equation 3. Now remove the middle such weight, w_{t+1}, and recursively continue this argument knowing that Equation 4 has marked the other such weights. $\Box$

By Theorem 7, the three steps of this algorithm cost $O(\lg\lg n)$ time using $n/\lg\lg n$ processors on the common-CRCW PRAM or in $O(\lg n)$ time using $n/\lg n$ processors on a EREW PRAM or a common-CRCW PRAM.

Assume that $r+1$ weights remain after reduction. Then renumbering and rotating the list of remaining weights gives $w_1, w_2, \cdots, w_{r+1}$ where $w_1 = \min_{1 \le i \le r+1}\{w_i\}$. The second step of the approximation algorithm requires that we form the appropriate linear product with the remaining matrices.

The depth of the parentheses provides an approximation to within 15.5% of optimal for the MCOP. This is due to (Chandra, 1975), (Chin, 1979), and (Hu and Shing, 1981).

Theorem 15 (Hu and Shing, 1981) If a weight list $w_1, w_2, \cdots, w_{r+1}$ is reduced, then the MCOP can be solved to within a multiplicative factor of 1.155 from optimal in constant time using n processors. This is done by choosing the linear parenthesization $((\cdots(a_1 \bullet a_2) \bullet \cdots) \bullet a_{r-1}) \bullet a_r$.

That is, after a weight list is reduced choosing a linear parenthesization with cost $w_1 \| w_2 : w_{r+1} \|$ gives a matrix chain product that is within a multiplicative factor of 1.155 from optimal.

The approximation algorithm given here is another problem whose solution is built on the ANSV problem. This algorithm also shows that only a linear number of entries of a dynamic programming table give a nice approximate solution to the MCOP. That is, a minor variation of the path of critical nodes in the canonical subgraphs supply a good approximate solution for the matrix chain ordering problem.

This algorithm is built on the greedy principle more than the dynamic programming paradigm. In terms of the processor-time product, this algorithm is optimal.

6.2 Historical Notes

Both (Chandra, 1975) and (Chin, 1979) gave approximations for the MCOP that were not quite as good as the 1.155 of Theorem 15. Theorem 15 was conjectured in (Chin, 1979) and finally proved in (Hu and Shing, 1981).

The algorithm given in this chapter appeared in (Bradford, 1992; Bradford, 1992a) and a very similar algorithm was given in (Czumaj, 1992). The content of this chapter is almost identical to that which the author of this dissertation submitted to the *Symposium on Parallel Algorithms and Architectures* (SPAA) 1992.

Chapter 7

An $\tilde{O}(n^3)$-Work Polylog-Time Algorithm

This chapter contains an $O(\lg^3 n)$-time algorithm for solving the matrix chain ordering problem that uses $n^3/\lg n$ processors. Throughout this chapter leaf subgraphs of the form $D^{(i,j)}$ are written as $D^{(1,m)}$ where $1 \leq m \leq n$. In addition, assume that D_n contains critical nodes, otherwise by Corollary 2 there is an immediate exact solution.

One of the key insights of this chapter is that canonical subgraphs can be treated atomically while finding a shortest path in a D_n graph. Further, $(\min, +)$-matrix multiplication joins these subgraphs together to form a shortest path in an entire D_n graph.

7.1 Shortest Paths Without Critical Nodes

This section culminates with Theorem 18 which basically states that in a $D^{(1,m)}$ graph shortest paths have a very rigid structure; this result supplies the first step for finding shortest paths in canonical subgraphs. All results in this section apply to shortest paths from $(0,0)$ to $(1,m)$ in $D^{(1,m)}$ (leaf) graphs and to shortest paths from (j,k) to

(i, v) in $D_{(j,k)}^{(i,v)}$ (band) graphs.

A path q with one jumper contains no critical nodes iff there are no critical nodes in q and there are no critical nodes in q's dual path. That is, a jumper $(i, j) \Longrightarrow (i, k)$ contains no critical nodes if both (i, j) and (i, k) are not critical nodes and there are no critical nodes in a shortest path contributing to this jumper's weight. This generalizes to paths with more than one jumper.

In our terminology the following result of (Hu and Shing, 1982, Corollary 3) can be stated as:

Theorem 16 In any canonical graph, the sum of the two products $w_i w_{j+1} w_{k+1} + w_{j+1} w_{j+2} w_{k+1}$ where $i < j < k$, cannot contribute to the weight of any shortest path iff $w_{k+1} > w_{j+1} > w_i$.

Next is a useful technical lemma.

Lemma 6 In a $D^{(1,m)}$ graph if $(i, t) \in V[\mathcal{U}]$, then $w_i < w_{t+1}$.

Proof: Since $(i, t) \in V[\mathcal{U}]$ and $i < t$, there must be some critical node $(i, u) \in V[p]$ where $t < u$. This means that $w_j > \max\{w_i, w_{u+1}\}$, for all $j, i < j \leq u$. Since $i < t < u$ it must be that $w_i < w_{t+1}$. $\Box$

A symmetric argument to that of Lemma 6 shows that $w_i > w_{j+1}$ for all nodes $(i, j) \in V[\mathcal{L}]$.

Theorem 17 Any shortest path q to some vertex (i, j) in $D^{(1,m)}$ where q contains no critical nodes, except possibly (i, j), is a straight unit path.

A proof of this theorem follows from an inductive application of Lemma 4 and Theorem 12.

The last theorem and all previous results of this section also apply to shortest paths from (j, k) to (i, v) in $D_{(j,k)}^{(i,v)}$ canonical subgraphs.

Jumpers of the form $(i,j) \Longrightarrow (i,k)$ such that $(j+1,k) \in V[p]$ are very important. Such jumpers contain at least one critical node, namely $(j+1,k)$.

Lemma 7 If a horizontal shortest path q to $(i,u) \in V[\mathcal{U}] \cup V[p]$, is such that q has one jumper $(i,j) \Longrightarrow (i,k)$ and $(j+1,k) \in V[p]$, then q is equivalent to a shortest path to $(j+1,k)$ followed by $(j+1,k) \Uparrow (i,k) \rightarrow \cdots \rightarrow (i,u)$.

The same holds for any such vertical path. A proof of this lemma follows from Lemma 1 and Theorem 2. That is, this lemma is based on the Duality Theorem.

Suppose (j,k) and (i,t) are two critical nodes in a canonical graph $D^{(1,m)}$, where $i \le j \le k \le t$. Then there is a unit path of critical nodes from (j,k) to (i,t). With this, there are two important symmetric paths between (j,k) and (i,t):
The *upper angular* path of (j,k) and (i,t) is

$$(j,k) \Uparrow (i,k) \rightarrow \cdots \rightarrow (i,t)$$

and the *lower angular* path of (j,k) and (i,t) is

$$(j,k) \Longrightarrow (j,t) \uparrow \cdots \uparrow (i,t)$$

A *trivial* angular path has only two unit edges and no jumpers. That is, if the three unit edge connected critical nodes

$$(j,k) \rightarrow (j,k+1) \uparrow (j-1,k+1)$$

then there is a trivial angular path

$$(j,k) \uparrow (j-1,k) \rightarrow (j-1,k-1)$$

where $(j-1,k)$ is not a critical node. For examples of angular paths see Figure 19.

In a canonical subgraph the shortest path between any two critical nodes that contains no other critical nodes is an *angular* path. Angular paths are central to the rest of this dissertation.

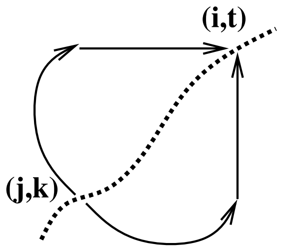

Figure 19: Two Angular Paths

A central result of this section is that a shortest path to a critical node (i,j) in a canonical graph may be along a path of critical nodes, then through an angular path then back to a subpath of critical nodes, then through an angular path and back to a subpath of critical nodes, etc.

In our terminology, Hu and Shing gave the following important theorem.

Theorem 18 (Hu and Shing, 1982) A shortest path to a critical node (i,j) in a $D^{(1,m)}$ graph is either along a straight unit path to (i,j), along the path of critical nodes to (i,j), or along subpaths of critical nodes connected together by angular paths and finally to (i,j).

A proof of this theorem follows inductively from Theorem 17 and the following. Say that there is some jumper $(i,j) \Longrightarrow (i,k)$ such that $(j+1,k) \notin V[p]$. Then, since $W((i,j) \Longrightarrow (i,k)) = f(i,j,k) + sp(j+1,k)$ and assume without loss that $(j+1,k) \in V[\mathcal{L}]$, see Figure 20**a**. Now, by Theorem 17 we know the shortest path to $(j+1,k)$ includes the unit edge $(j+2,k) \uparrow (j+1,k)$ which has cost $w_j w_{j+1} w_{k+1}$. Additionally, assume that $(j+2,k) \notin V[p]$. But, because $f(i,j,k) = w_i w_{j+1} w_{k+1}$ and since (i,j) and (i,k) are in $\mathcal{U}$ Lemma 6 indicates that $w_i < w_{j+1} < w_{k+1}$. Therefore applying Theorem 16 shows that the jumper $(i,j) \Longrightarrow (i,k)$ can't be in a minimal path in D_n. A similar case holds for Figure 20**b**.

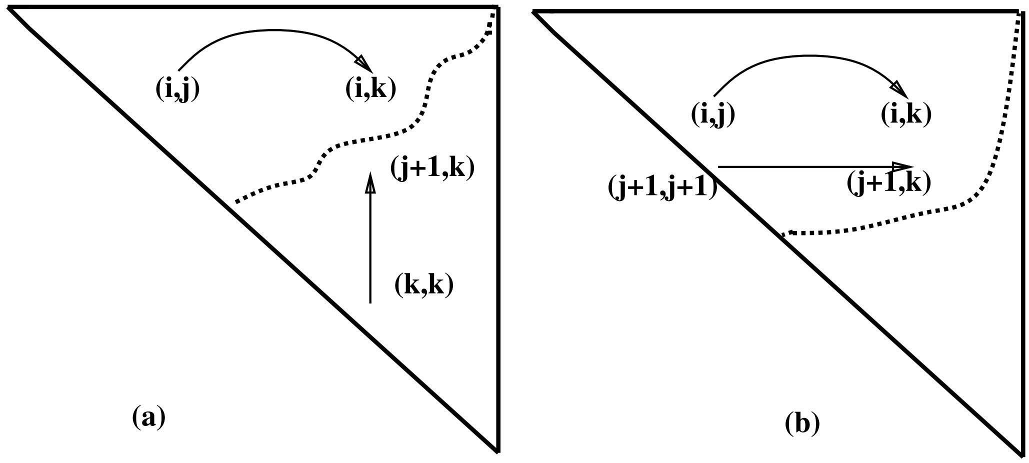

Figure 20: Two Jumpers and their Complimentary Paths

By the Duality Theorem and the compatibility of critical nodes, any jumpers over p have dual paths containing no jumpers over p giving the last case just discussed.

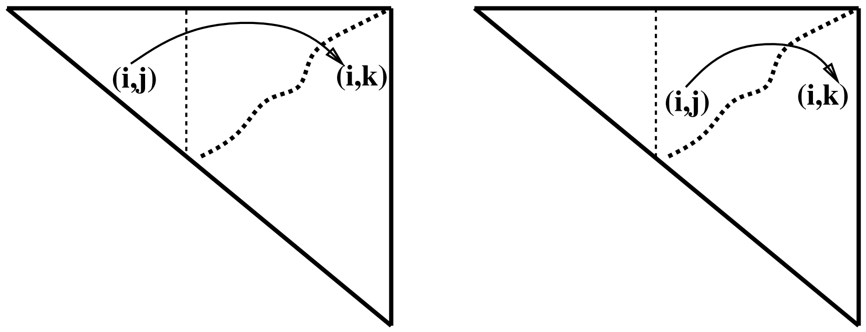

Figure 21: Two Jumpers Over the Path p

This last theorem and the next corollary also hold for shortest paths from (j,k) to (i,v) in $D^{(i,v)}_{(j,k)}$ graphs.

Corollary 4 A shortest path from $(0,0)$ to a *non*-critical node (s,t) in a $D^{(1,m)}$ graph is either a straight unit path, or it is a minimal path to some critical node (i,j) and from (i,j) to (s,t) by an angular path.

7.2 Combining the Canonical Graphs

This section gives a parallel divide and conquer tool for finding minimal paths in D_n graphs based on work of Hu and Shing. This is done by isolating an underlying tree structure connecting the canonical subgraphs, so we can find shortest paths in these subgraphs individually while essentially ignoring the effect of the monotone subgraphs. This allows the computation of a shortest path in a D_n graph by applying variations of tree contraction techniques. These techniques incorporate special "leaf pruning" operations.

As before, by Corollary 1 rotate any given weight list so that w_1 is the smallest weight.

Next we present, without proof, a central result of Hu and Shing. In essence, this result gives some of the power of the greedy principle together with the principle of optimality. That is, this result allows the isolation of some substructures that are necessarily in an optimal superstructure.

Theorem 19 (Hu and Shing, 1982) Given a weight list $w_1, \ldots, w_{n+1}$ with the three smallest weights $w_1 < w_{i+1} < w_{v+1}$ and $i + 1 < v$, the products $w_1 w_{i+1}$ and $w_1 w_{v+1}$ are in some associative product(s) in an optimal parenthesization.

There may be one or two fs that contain the products $w_1 w_{i+1}$ and $w_1 w_{v+1}$.

The next corollary is central to the results of this section. It essentially guarantees that certain easily computable nodes are a part of a shortest path in a given D_n graph. Where Hu and Shing use the results of the last theorem as a sequential divide and conquer tool, here it is made into a parallel tool.

Corollary 5 (Atomicity Corollary) Given a weight list $w_1, \ldots, w_{n+1}$ with the three smallest weights $w_1 < w_{i+1} < w_{v+1}$ and $i + 1 < v$, the critical nodes $(1, i)$ and $(1, v)$ are in a shortest path from $(0, 0)$ to $(1, n)$ in D_n.

A proof follows directly from Theorem 19.

Suppose $w_1 < w_{i+1} < w_{v+1}$ are the three smallest weights in D_n and both $D^{(1,i)}$ and $D^{(i+1,v)}$ are leaf subgraphs. Then there is a shortest path from $(0,0)$ to $(1,i)$ in $D^{(1,i)}$. Applying Corollary 5 to the subgraph $D(1,v)$, which also has the three smallest weights $w_1 < w_{i+1} < w_{v+1}$, shows that $(1,i)$ is in a minimal path to $(1,v)$. Therefore by the structure of D_n graphs the only contribution that $D^{(i+1,v)}$ can make to a shortest path to $(1,v)$ is by providing *sp* values for jumpers along the unit path $(1,i) \to \cdots \to (1,v)$. That is, there *may* be some jumper $(1,j) \Longrightarrow (1,k)$ such that $(j+1,k) \in V[D^{(i+1,v)}]$ and

$$(1,i) \to \cdots \to (1,j) \Longrightarrow (1,k) \to \cdots \to (i,v)$$

is cheaper than

$$(1,i) \to \cdots \to (1,v).$$

Shortly, in Lemma 9, we will see that we only have to consider jumpers $(1,j) \Longrightarrow (1,k)$ such that $(j+1,k)$ is a critical node in $D^{(i+1,v)}$.

7.2.1 Canonical Trees

Dividing a D_n graph into a tree of canonical subgraphs using Corollary 5 is easily done in $O(\lg n)$ time with $n/\lg n$ processors by solving the ANSV problem.

In a D_n graph the structure joining all of the critical nodes is a *canonical tree*, see also Hu and Shing (Hu and Shing, 1980; Hu and Shing, 1982; Hu and Shing, 1984). Define the leaves, edges, and internal nodes of a canonical tree as follows. Initially, in every canonical subgraph $D^{(1,m)}$ the critical node $(1,m)$ is a leaf and is denoted by $\overline{(1,m)}$ to distinguish it from other critical nodes. A $D^{(1,m)}$ canonical subgraph only contains one tree node, namely $\overline{(1,m)}$. All other tree nodes will also be overlined.

An *isolated* critical node is a critical node with no critical nodes that are one unit edge away. The internal tree nodes are isolated critical nodes or $\overline{(i,v)}$ and $\overline{(j,k)}$ for $D^{(i,v)}_{(j,k)}$ canonical graphs, where $k \neq j+1$. A $D^{(i,v)}_{(j,k)}$ graph only contains the two tree nodes $\overline{(i,v)}$ and $\overline{(j,k)}$. Notice that all tree nodes are also critical nodes, thus they are compatible.

The edges of a canonical tree are the straight unit paths that connect tree nodes. Jumpers may reduce the cost of tree edges. An edge from $\overline{(i,j)}$ to $\overline{(i,k)}$ is denoted by $\overline{(i,j)} \rightarrow \cdots \rightarrow \overline{(i,k)}$ and all edges are directed towards $(1,n)$.

Since tree nodes are critical nodes with easily discernible properties, they are efficiently distinguishable in parallel. In addition, we can discard all monotone subgraphs since they have no influence on a shortest path to $(1,n)$, except if $D(1,j)$ is monotone and $w_1 = \min\limits_{1\le i\le j}\{w_i\}$. That is, since $w_1 = \min\limits_{1\le i\le n+1}\{w_i\}$, if $D(1,i)$ is monotone for some i, then a shortest path to $(1,n)$ will travel along the path $(1,1) \rightarrow \cdots \rightarrow (1,i)$. But this is the only case when a monotone graph contains a piece of a minimal path from $(0,0)$ to $(1,n)$.

For the next lemma, assume $w_1 = \min\limits_{1\le i\le n+1}\{w_i\}$.

Lemma 8 In a monotone subgraph $D(i,k)$ of D_n, the cost of a shortest path to (i,k) for $i > 1$ plus the associative weight $f(1,i-1,k)$ is more than the unit path $(1,i-1) \rightarrow \cdots \rightarrow (1,k)$.

Proof: Since $D(i,k)$ is monotone, we either have $w_i < \cdots < w_{k+1}$ or $w_i > \cdots > w_{k+1}$. So by Theorem 10, the shortest path to (i,k) in $D(i,k)$ is either $(i,i) \rightarrow \cdots \rightarrow (i,k)$ or $(k,k) \uparrow \cdots \uparrow (i,k)$. Therefore, taking the jumper $(1,i-1) \Longrightarrow (1,k)$ with weight $sp(i,k) + f(1,i-1,k)$ we have two cases.

Case i: The ordering of the weight list is $w_i < \cdots < w_{k+1}$.

In this case, the shortest path to (i,k) is $(i,i) \rightarrow \cdots \rightarrow (i,k)$. Clearly, $f(1,i-1,k) = w_1 w_i w_{k+1}$ and $W((1,i-1) \rightarrow (1,i)) = w_1 w_i w_{i+1}$. But since

$w_{i+1} \leq w_{k+1}$, it must be $f(1, i-1, k) \geq W((1, i-1) \to (1, i))$. Along the same lines, $W((1, j) \to (1, j+1)) < W((i, j) \to (i, j+1))$ holds for all $j, i \leq j < k$.

CASE ii: The ordering of the weight list is $w_i > \cdots > w_{k+1}$.

In this case, the shortest path to (i, k) is $(k, k) \uparrow \cdots \uparrow (i, k)$. Since $f(1, i-1, k) = w_1 w_i w_{k+1}$, $W((1, k-1) \to (1, k)) = w_1 w_k w_{k+1}$, and $w_i \geq w_k$, it must be that $f(1, i-1, k) \geq W((1, k-1) \to (1, k))$. Along the same lines, $W((1, i+j-1) \to (1, i+j)) < W((i+j+1, k) \uparrow (i+j, k))$ for all $j, 0 \leq j < k-i$.

□

Suppose D_n has fewer than $n-1$ critical nodes. Then D_n may have several disconnected canonical trees and one or more monotone subgraphs by Corollary 2. But, by Theorem 9 there is at least one path from $(0, 0)$ to $(1, n)$ joining these canonical subtrees. From Lemma 8, after discarding irrelevant monotone subgraphs and for the moment ignoring $D^{(i,v)}_{(j,k)}$ graphs, there are several structures that $D^{(1,m)}$ graphs may form together. The relationships of tree nodes is the basis of all of these structures.

Let $\overline{(i, j)}, \overline{(j+1, k)}, \ldots, \overline{(u+1, v)}$ be neighboring leaves in $D(i, v)$ such that they are all in a canonical tree rooted at (i, v) or in no canonical tree at all. All of these leaves together can have the next relationships, or combinations of them.

CASE 1: The leaves are not joined together by internal tree nodes. For instance, this case occurs if $w_i < w_{j+1} < \cdots < w_{u+1} < w_{v+1}$.

CASE 2: The leaves form a binary canonical tree, see Figure 22. In this case, there are internal tree nodes in $D(i, v)$ that connect the leaves together.

Solving Case 1 is done by treating it as an instance of Case 2 by building a surrogate tree. The viability of treating these situations as Case 2 is now shown.

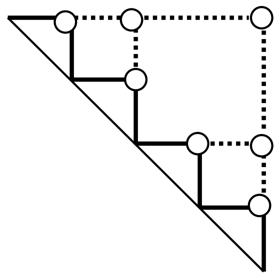

Figure 22: A Canonical Tree of $D^{(1,m)}$ Graphs, the Circles Denote Tree Nodes

Take a list of r monotone leaves, none of which are in a canonical tree. Label the leaves $\overline{(i,j)}, \overline{(j+1,k)}, \overline{(k+1,t)}, \ldots, \overline{(u+1,v)}$ and without loss assume that $w_i < w_{j+1} < \cdots < w_{u+1} < w_{v+1}$ so these leaves are all in $D(i,v)$. Now, by Corollary 5 tree node $\overline{(i,j)}$ must be in a shortest path from $(0,0)$ to $\overline{(i,v)}$. Therefore, applying the Theorem 2, a shortest path from $(0,0)$ to $\overline{(i,v)}$ goes from $(0,0)$ to $\overline{(j+1,v)}$ and then over the tree edge $\overline{(j+1,v)} \uparrow \cdots \uparrow \overline{(i,v)}$. Now, complete this argument by induction.

Band subgraphs also form trees in the same way. Trees containing band subgraphs are treated in the same way as above.

7.3 Finding Shortest Paths to All Critical Nodes in Canonical Subgraphs

This section discusses an algorithm for finding shortest paths in $D^{(1,m)}$ and $D^{(i,v)}_{(j,k)}$ canonical graphs. As before, p denotes the continuous path of critical nodes in a $D^{(1,m)}$ or $D^{(i,v)}_{(j,k)}$ graph.

First an $m^3/\lg m$ processor and $O(\lg^2 m)$-time algorithm for finding a shortest path to all critical nodes in a $D^{(1,m)}$ leaf subgraph is given. This is done by treating angular paths as edges so p is now an $(m+1)$-node graph with $\Theta(m^2)$ edges. That is, any angular path $(j,k) \Uparrow (i,k) \rightarrow \cdots \rightarrow (i,t)$ connecting the critical nodes (j,k) and (i,t) becomes *one edge* from (j,k) to (i,t) costing $W((j,k) \Uparrow (i,k)) + w_i \| w_{k+1} : w_{t+1} \|$.

Of course, since (j,k) is a critical node but (i,k) is not a critical node and both are in the same canonical graph, it must be that

$$W((j,k) \Uparrow (i,k)) = w_i \| w_{i+1} : w_j \| + f(i, j-1, k).$$

Now by Theorem 17, this also holds for $D_{(j,k)}^{(i,v)}$ canonical graphs. Further, using a $(\min,+)$-matrix multiplication shortest path algorithm, finding shortest paths in such an $(m+1)$-node graph costs $O(\lg^2 m)$ time with $m^3/\lg m$ processors. This results in a polylog time and $n^3/\lg n$ processor MCOP algorithm at the end of this chapter.

Next is the $O(\lg^2 m)$ time and $m^3/\lg m$ processor algorithm for finding a shortest path to each critical node in a $D^{(1,m)}$ graph. First compute all of the unit paths to nodes in p in constant time using m processors. Perhaps, we can best view these paths as edges from $(0,0)$ to the nodes in p. Further, the cost of each of the $\Theta(m^2)$ angular paths can be computed in constant time using m^2 processors, with preprocessing costing $O(\lg m)$ time and $m/\lg m$ processors. The preprocessing is just computing parallel partial prefix minima.

Compute the shortest path to each node in p by treating every angular path as a weighted edge and applying a parallel $(\min,+)$-matrix multiplication all pairs shortest path algorithm to the nodes in p. This algorithm costs $O(\lg^2 m)$ time and $m^3/\lg m$ processors, and provides a shortest path from $(0,0)$ to every critical node in a $D^{(1,m)}$ graph. Compute the shortest paths from $(0,0)$ to all critical nodes in a $D_{(j,k)}^{(i,v)}$ canonical graph in the same way.

Theorem 20 Given a $D^{(1,m)}$ graph computing a shortest path from $(0,0)$ to all nodes in p takes $O(\lg^2 m)$ time using $m^3/\lg m$ processors.

This theorem also holds for finding shortest paths from all critical nodes in a $D_{(j,k)}^{(i,v)}$ canonical graph to (i,v).

7.3.1 Leaf Pruning and Band Merging

Here a basic technique for joining band canonical subgraphs quickly in parallel is given. This technique is based on edge minimization and builds shortest paths in D_n graphs.

Take the two jumpers $(i, j) \Longrightarrow (i, t)$ and $(i, k) \Longrightarrow (i, u)$ in row i and without loss say $j < k$. Then these jumpers are not *compatible* iff $k < t < u$. If $(j+1, t) \in V[p]$ and $(k+1, u) \in V[p]$, then $(j+1, t)$ and $(k+1, u)$ are compatible. Consequently, any two jumpers in row i such as $(i, j) \Longrightarrow (i, t)$ and $(i, k) \Longrightarrow (i, u)$, where $(j+1, t) \in V[p]$ and $(k+1, u) \in V[p']$, must be compatible, where p and p' are possibly distinct paths of critical nodes. Notice that if p and p' are distinct, then they are still compatible. Given a D_n graph with the jumpers $(i, j) \Longrightarrow (i, t)$ and $(i, k) \Longrightarrow (i, u)$, let $\overline{p}$ be a minimal path from $(0, 0)$ to $(1, n)$ in D_n. Then all jumpers $(i, j) \Longrightarrow (i, t)$ and $(i, k) \Longrightarrow (i, u)$ such that $(j+1, t) \in V[\overline{p}]$ and $(k+1, u) \in V[\overline{p}]$ are compatible.

Minimizing the cost of a straight unit edge path in a canonical tree by using jumpers is *edge minimizing*, and we will show that the jumpers only have to get their *sp* values from critical nodes. *Only* tree edges or straight unit paths will be edge minimized in $D^{(i,v)}_{(j,k)}$ graphs. For example, let p be the path of critical nodes in $D(k, t)$ and consider the straight unit path $(i, j) \rightarrow \cdots \rightarrow (i, v)$. If the jumper $(i, k) \Longrightarrow (i, t)$ is such that $(k+1, t) \in V[p]$ and

$$(i, j) \rightarrow \cdots \rightarrow (i, k) \Longrightarrow (i, t) \rightarrow \cdots \rightarrow (i, v)$$

is cheaper than $(i, j) \rightarrow \cdots \rightarrow (i, v)$, then we edge minimize $(i, j) \rightarrow \cdots \rightarrow (i, v)$ with $(i, k) \Longrightarrow (i, t)$.

The next procedure *edge minimizes* the unit path along the i^{th} row to the critical node (i, v) with all jumpers that get their *sp* values from the critical nodes $V[p]$,

$$A[i,v] \;=\; \min_{\forall (k+1,u)\in V[p]} \{\; w_i \| w_{i+1} : w_{v+1} \|,\; w_i \| w_{k+1} : w_{u+1} \| - w_i \| w_{k+1} : w_{u+1} \| \\ + W((i,k) \Longrightarrow (i,u)) \;\}$$

and the same can be done for straight vertical unit paths. The minimal cost along the tree edge i to (i, v) is in $A[i, v]$, assuming only one connected path of critical nodes p. Notice by Theorem 6 that we can compute the cost of the straight unit (sub)paths in constant time with one processor with the appropriate preprocessing.

Lemma 9 When edge minimizing a tree edge $\overline{(i,j)} \to \cdots \to \overline{(i,v)}$ in a canonical subgraph we only have to consider jumpers $(i,k) \Longrightarrow (i,t)$ such that $(k+1,t) \in V[p]$.

Proof: Take row i, and $(i,i) \to \cdots \to (i,v)$ assuming that some jumper $(i,s) \Longrightarrow (i,t)$ minimizes row i where $(s+1,t) \notin V[p]$ and $i < s < t \leq u$. Without loss, say $(s+1,t) \in V[\mathcal{U}]$, so a shortest path to $(s+1,t)$ is either the straight unit path $(s+1,s+1) \to \cdots \to (s+1,t)$ or this unit path with jumpers by Corollary 4.

All jumpers getting their *sp* value from a critical node in p must be compatible. Therefore, there is only one jumper that gets its *sp* value from a critical node in a shortest path from $(s+1,s+1)$ to $(s+1,t)$. Now there are three cases to consider.

Case i: A minimal path to $(s+1,t)$ is the straight unit path $(s+1,s+1) \to \cdots \to (s+1,t)$.

By the Duality Theorem the minimal path along

$$(i,i) \to \cdots \to (i,s) \Longrightarrow (i,t)$$

is equivalent to the path

$$(s+1,s+1) \to \cdots \to (s+1,t) \Uparrow (i,t).$$

But

$$(s+1, s+1) \rightarrow \cdots \rightarrow (s+1, t) \Uparrow (i, t)$$

is not an angular path, a straight unit path, or a path of critical nodes intermixed with angular paths getting their *sp* value from p. Therefore, we arrive at a contradiction of Corollary 4.

CASE ii: A shortest path to $(s+1, t)$ is along the unit path from $(s+1, s+1)$ to $(s+1, t)$ and contains one or more jumpers whose *sp* values are *not* from $V[p]$.

This case would imply that a shortest path from $(0, 0)$ to nodes in $\mathcal{U}$ are not angular paths or straight line unit paths contradicting Corollary 4.

CASE iii: A shortest path to $(s+1, t)$ is along the unit path from $(s+1, s+1)$ to $(s+1, t)$ and contains one jumper that gets its *sp* value from a critical node in p.

Let $(s+1, j) \Longrightarrow (s+1, k)$ be a jumper such that $(j+1, k) \in V[p]$. Thus by the Duality Theorem the path

$$(s+1, s+1) \rightarrow \cdots \rightarrow (s+1, j) \Longrightarrow (s+1, k) \rightarrow \cdots \rightarrow (s+1, t)$$

followed by the jumper $(s+1, t) \Uparrow (i, t)$ costs the same as the path

$$(i, i) \rightarrow \cdots \rightarrow (i, s) \Longrightarrow (i, t).$$

At the same time, the path

$$(s+1, s+1) \rightarrow \cdots \rightarrow (s+1, j) \Longrightarrow (s+1, k) \rightarrow \cdots \rightarrow (s+1, t)$$

is equivalent to a shortest path to $(j+1,k)$ followed by the angular path $(j+1,k) \Uparrow (s+1,k) \rightarrow \cdots \rightarrow (s+1,t)$, by the Duality Theorem. This means a shortest path from $(0,0)$ to (i,t) is from $(0,0)$ to $(j+1,k)$ then

$$(j+1,k) \Uparrow (s+1,k) \rightarrow \cdots \rightarrow (s+1,t) \Uparrow (s+1,t),$$

but this is a contradiction by Corollary 4, since this path is not an angular path.

Cases i and ii can occur at the same time, though the above arguments still hold. □
This lemma easily generalizes to the case where p is a path of critical nodes in a conglomerate of canonical subgraphs. Lemma 9 also highlights the role well-formed subsolutions play in the dynamic programming solution of the MCOP.

Let $\overline{p}$ be a minimal path from $(0,0)$ to (k,t) in $D(k,t)$. Then Lemma 9 extends to the case of only jumpers $(i,k) \Longrightarrow (i,t)$ along tree edges $(i,j) \rightarrow \cdots \rightarrow (i,v)$ such that $(k+1,t) \in V[\overline{p}]$.

Theorem 21 When edge minimizing a tree edge $\overline{(i,j)} \rightarrow \cdots \rightarrow \overline{(i,v)}$ in a canonical subgraph we only have to consider jumpers $(i,k) \Longrightarrow (i,t)$ such that $(k+1,t) \in V[\overline{p}]$.

Proof: Since $\overline{(i,j)} \rightarrow \cdots \rightarrow \overline{(i,v)}$ is a tree edge it must be that $\max\{w_i, w_{v+1}\} < w_s$ for all $s, i \leq s \leq v+1$.

For the sake of a contradiction suppose the jumper $(i,k) \Longrightarrow (i,t)$, such that $(k+1,t) \notin V[\overline{p}]$ but $(k+1,t) \in V[p]$, is the jumper that minimizes the tree edge $(i,j) \rightarrow \cdots \rightarrow (i,v)$ more than any other jumper.

But since $(k+1,t) \notin V[\overline{p}]$, it's safe to assume that the angular path

$$(r,s) \Uparrow (q,s) \rightarrow \cdots \rightarrow (q,u)$$

is the angular path *in* $\overline{p}$ that goes around $(k+1,t)$, so $(r,s) \in V[\overline{p}]$ and $(q,u) \in V[\overline{p}]$. In this case, by Theorem 2 a minimal path to (q,u) is

$$(q,q) \to \cdots \to (q,r-1) \Longrightarrow (q,s) \to \cdots \to (q,u)$$

such that $(r,s) \in V[\overline{p}]$. In particular, notice that the jumper $(q,k) \Longrightarrow (q,t)$ saves no more than any other jumper by edge minimizing the unit path $(q,q) \to \cdots \to (q,u)$ and, since $(i,j) \to \cdots \to (i,v)$ is a tree edge, we know that $w_i < w_q$ and $f(i,k,t) < f(q,k,t)$. Thus, if $(q,k) \Longrightarrow (q,t)$ saves no more than the jumper $(q,r-1) \Longrightarrow (q,s)$ in row q, then $(i,k) \Longrightarrow (i,t)$ saves no more than any other jumper in row i. $\square$

This last theorem is very important. It says once a minimal path $\overline{p}$ is discovered in a subgraph $D(j,k)$, then during edge minimization we only have to consider jumpers with *sp* values from $\overline{p}$. Of course $V[\overline{p}] \subseteq V[p]$, where p is the path of critical nodes.

Theorem 22 Given a tree node $\overline{(i,u)}$ where the graph $D(i,u)$ contains the leaves $D(i,j)$ and $D(j+1,u)$ so $\overline{(i,j)}$ and $\overline{(j+1,u)}$ are tree nodes, if $D(j+1,u)$ is a canonical subgraph, then a shortest path from $\overline{(i,j)}$ to $\overline{(i,u)}$ can be found in $O(u-j)$ operations.

Proof: Since (i,j), (i,u), and $(j+1,u)$ are all critical nodes, the three smallest weights in $w_i,\ldots,w_{u+1}$ are w_i, w_{j+1}, and w_{u+1}. Now assume without loss that $w_i < w_{u+1} < w_{j+1}$. Hence, by Corollary 5, tree node $\overline{(i,j)}$ must be in a shortest path from $(0,0)$ to (i,u). Therefore, edge minimize the unit path $(i,j) \to \cdots \to (i,u)$. Otherwise, if $w_{u+1} < w_i < w_{j+1}$ then $(j+1,u)$ is in a shortest path from $(0,0)$ to (i,u). Therefore by the Duality Theorem we can edge minimize the path $(i,j) \to \cdots \to (i,u)$.

There could be a quadratic number of jumpers of the form $(i,k) \Longrightarrow (i,t)$ such that $j \leq k < t \leq u$. But by Theorem 21 we only have to consider jumpers along row i that get their *sp* values from $\overline{p}$. That is, only jumpers such as $(i,k) \Longrightarrow (i,t)$ where $(k+1,t) \in V[\overline{p}]$. The appropriate value of *sp*, which has been computed for each node in $\overline{p}$, can be retrieved and added to the appropriate value of f in constant

time. Therefore finding the minimal values for the paths between nodes (i,j) and (i,u) takes $O(u-j)$ operations, since there are $O(u-j)$ such compatible jumpers. $\square$
The $O(u-j)$ operations are easily done in $O(\lg(u-j))$ time using $(u-j)/\lg(u-j)$ processors. In $O(\lg(u-j))$ time and with $(u-j)/\lg(u-j)$ processors first get all of the jumper values and then find the minimal such jumper.

The last theorem holds for leaves in the canonical tree that have been combined and become conglomerates of other leaves and internal nodes. In this situation, jumpers derived from critical nodes in different subtrees are independent and compatible, so we can minimize tree edges with them simultaneously. But, canonical subgraphs of the form $D_{(j,k)}^{(i,v)}$ must be considered. In this case, take the leaf $D(j,k)$ that must be pruned in $D_{(j,k)}^{(i,v)}$, where $D(j,k)$ consists of the pruned subgraphs $D(j,t)$ and $D(t+1,k)$.

If there is a shortest path through (j,k) to (i,v), then by Corollary 5 and depending on whether $w_j < w_{k+1}$ or $w_{k+1} < w_j$ either (j,t) or $(t+1,k)$, respectively, are in shortest paths to (j,k). Therefore, by Theorem 22 we would be done. But we must address the possibility that the cost of paths from $(0,0)$ to critical nodes throughout $D(j,k)$ can contribute to shortest paths to critical nodes from (j,k) to (i,v). In fact, we must combine the information about shortest paths in $D(j,k)$ with information about shortest paths in $D_{(j,k)}^{(i,v)}$. The combining of this shortest path information is done by edge minimizing unit paths in $D_{(j,k)}^{(i,v)}$ with *sp* values from critical nodes of $D(j,k)$.

The next theorem is another parallel divide and conquer tool, but it is for merging a $D(j,k)$ leaf into a $D_{(j,k)}^{(i,v)}$ graph.

Theorem 23 In a $D_{(j,k)}^{(i,v)}$ graph with a shortest path $\overline{p}$ from (j,k) to (i,v) and suppose the four smallest weights are $w_i < w_{v+1} < w_{i+1} < w_v$, then $\overline{p}$ goes through one of

1. Either $(i+1,v)$ or $(i+1,v-1)$ or both

2. $(i, i+1)$ and $(i, v-1)$

3. $(v-1, v)$ and $(i+1, v)$

Proof: Since $w_i < w_{v+1} < w_{i+1} < w_v$ and we are in a $D^{(i,v)}_{(j,k)}$ graph, it must be that $(i, v), (i+1, v)$, and $(i+1, v-1)$ are all critical nodes, so $\overline{p}$ *may* go through them. Clearly, if $\overline{p}$ goes through $(i+1, v-1)$, then $\overline{p}$ cannot go through $(i, i+1)$ or $(v-1, v)$.

If the shortest path $\overline{p}$ has a jumper up to row i, then $(i, i+1)$ will be in $\overline{p}$ in the sense of Theorem 9. Of course, if $\overline{p}$ goes through $(i, i+1)$, then $\overline{p}$ cannot go through either $(v-1, v)$ or $(i+1, v)$.

Finally, if $(v-1, v)$ is in a shortest path to (i, v), then $(i+1, v)$ is also and both $(i, i+1)$ and $(i+1, v-1)$ are not in this minimal path. □

This theorem also holds for $D^{(1,m)}$ graphs.

By Theorem 23 take a shortest path from $(0, 0)$ to (i, v) that goes through $(i, i+1)$. Then there might be a straight unit path $(i, i+1) \rightarrow \cdots \rightarrow (i, v)$ connecting $(i, i+1)$ and (i, v). On the other hand, there *may* be some jumper $(i, j) \Longrightarrow (i, k)$ such that

$$(i, i+1) \rightarrow \cdots \rightarrow (i, j) \Longrightarrow (i, k) \rightarrow \cdots \rightarrow (i, v)$$

is cheaper than $(i, i+1) \rightarrow \cdots \rightarrow (i, v)$.

The only possible ways to merge canonical graphs together are in Figure 23. Merging the two leaf canonical graphs as in Figure 23**a** is leaf pruning, and Figures 23**b,c** are band merging. Leaf pruning can be accomplished by edge minimizing after all shortest paths to critical nodes in the leaves have been found.

In Figure 23**c**, contracted trees **A** and **B** are used to edge minimize the unit paths marked by "**Min-A**" and "**Min-B**." Edge minimizing the unit paths in the outer band with the contracted trees gives an instance of Figure 23**b**.

Merging $D^{(i,v)}_{(j,t)}$ with $D^{(j,t)}_{(k,s)}$ takes $O(\lg^2 m)$ time using $m^3/\lg m$ processors, assuming

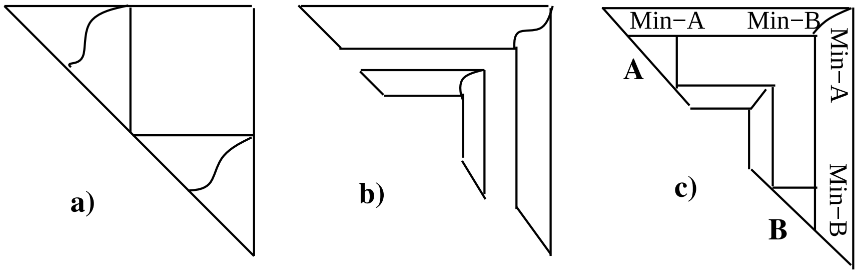

Figure 23: The Variations of Band Merging or Leaf Pruning

$D^{(i,v)}_{(k,s)}$ has a total of m critical nodes. Let $\overline{p_1}$ be a minimal path from (i,v) back to $(0,0)$ *only* through $D^{(i,v)}_{(j,t)}$, and let $\overline{p_2}$ be a minimal path from (j,t) back to $(0,0)$ *only* through $D^{(j,t)}_{(k,s)}$. Now, merging these two graphs by finding all the angular paths from critical nodes in $\overline{p_2}$ forward to any critical nodes in $D^{(i,v)}_{(j,t)}$. Next, applying an all pairs shortest path algorithm combines these bands giving a shortest path from (i,v) back to $(0,0)$ through $D^{(i,v)}_{(k,s)}$.

7.3.2 Contracting a Canonical Tree

This subsection shows how to use edge minimization and band merging as the prune operations for a standard tree contraction algorithm. Together these algorithms complete a good algorithm for the MCOP. This algorithm has work $\tilde{O}(n^3)$ and runs in polylog time.

In a tree that contains only $D^{(1,m)}$ canonical subgraphs, each prune operation is an edge minimization that joins leaves together, until the entire canonical tree is one leaf. Every critical node (i,j) has an associated variable $sp(i,j)$ where $sp(i,j)$ denotes the cost of a shortest path from $(0,0)$ to (i,j). Further, a canonical tree has at most $n-1$ critical nodes, so only $O(n)$ such variables are necessary.

Initially, all internal tree nodes have $sp(i,j) = \infty$ and all nodes in $\overline{p}$ which are tree leaves have the minimum value from $(0,0)$ to (i,j) stored in $sp(i,j)$. That is, in the tree leaves the minimal paths $\overline{p}$ have been computed. In addition, for all $D^{(i,v)}_{(j,k)}$

graphs compute and store the value of the shortest path from all critical nodes in $D^{(i,v)}_{(j,k)}$ to (i,v). Compute these initial values using the methods of Subsection 7.3. Also, each tree edge $\overline{(i,j)} \rightarrow \cdots \rightarrow \overline{(i,k)}$ initially has weight $w_i \| w_{j+1} : w_{k+1} \|$. After the appropriate preprocessing, by Theorem 6, computing the initial cost of each of these $O(n)$ tree edges costs constant time with one processor.

Assume the standard tree contraction algorithm (JáJá, 1992; Karp and Ramachandran, 1990; Kumar et al., 1994) except for the prune operation. Along with the new prune operation, there is an ordering of the leaves that prevents the simultaneous pruning of two adjacent leaves. Take two neighboring canonical graphs $D^{(i,j)}$ and $D^{(j+1,k)}$ with the two leaves $\overline{(i,j)}$ and $\overline{(j+1,k)}$ and the internal node $\overline{(i,k)}$, say $w_i < w_{k+1} < w_{j+1}$. Then prune leaf $\overline{(j+1,k)}$ since $\overline{(i,j)}$ is in a shortest path from $(0,0)$ to $\overline{(i,k)}$ by Corollary 5. Otherwise, say $w_{k+1} < w_i < w_{j+1}$, then prune leaf $\overline{(i,j)}$. While, standard tree contraction algorithms use the Euler tour technique (JáJá, 1992; Karp and Ramachandran, 1990) to number the leaves appropriately, as just seen the canonical nodes often provide a natural prune ordering. The viability of this natural leaf numbering follows by induction. Thus apply the Euler tour technique in case the leaf ordering is arbitrary, otherwise take the natural pruning order. Take the tree made by connecting the two leaves $\overline{(i,j)}, \overline{(j+1,k)}$ to the internal node $\overline{(i,k)}$, then the pruning order is arbitrary if $w_i = w_{j+1} = w_{k+1}$.

Also, the numbering of tree nodes by the Euler tour technique prevents a band graph from being merged simultaneously with both the band inside it and the band around it. See standard tree contraction techniques for details of this use of the Euler tour technique.

Figure 24 depicts a canonical tree rooted at $\overline{(i,k)}$ that is the parent of tree nodes $\overline{(i,j)}$ and $\overline{(j+1,k)}$ which are siblings representing $D(i,j)$ and $D(j+1,k)$, respectively. Assume that $D(j+1,k)$ has been pruned into a leaf, and if $w_{k+1} < w_i < w_{j+1}$, then compute what contribution, if any, $\overline{(j+1,k)}$ makes to the shortest path to (i,k).

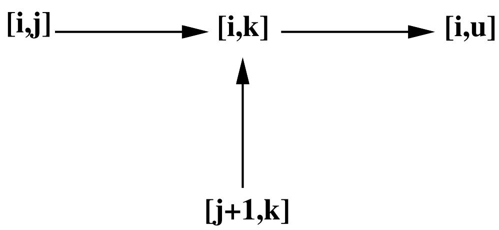

Figure 24: A Small Canonical Tree

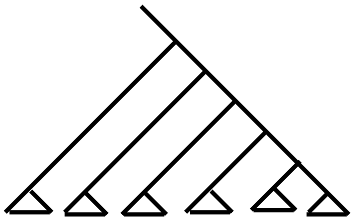

Figure 25: A Linear List of Tree Leaves

This is determined by edge minimizing the tree edge $\overline{(i,j)} \to \cdots \to \overline{(i,k)}$ with all nodes in a minimal path from $(0,0)$ to $(j+1,k)$. After this edge minimization, inactivate $\overline{(j+1,k)}$ and its parent $\overline{(i,k)}$, so $\overline{(j+1,k)}$ will not be pruned again.

The tree leaves depicted in Figure 25 can be pruned in any order. They are "pruned into" the edge joining them by first finding minimal paths to all critical nodes in each leaf. And then edge minimizing the edge joining them all with jumpers that get their *sp* values from the critical nodes in the leaves.

A linear list of leaves as those in Figure 25 can be pruned in any order. Therefore, choosing to do them simultaneously makes most sense. But, any number of nested bands must be merged in a way to avoid conflicts. This is easily done using the Euler tour technique for numbering appropriately for tree contraction, see (Karp and Ramachandran, 1990).

The next lemma shows that pruning allows the construction of a shortest path from $(0,0)$ to $(1,n)$ in a tree made of leaf canonical graphs and isolated critical nodes. This is necessary for canonical subtrees as in Figure 22. Let D'_n only contain leaf subgraphs and isolated tree nodes.

Lemma 10 Tree contraction of a tree of $D^{(1,m)}$ graphs with isolated internal tree nodes correctly computes a shortest path from $(0,0)$ to $(1,n)$ in D'_n.

Proof: The proof is by induction on the tree node depth, where a leaf is of depth 1.

The case where the depth $d = 1$ is trivial, so consider when the depth is $d = 2$. Without loss say the canonical tree of depth 2 has nodes $\overline{(i,j)}, \overline{(j+1,k)}$, and $\overline{(i,k)}$, (see the subgraph $D(i,k)$ in Figure 24). Now by Corollary 5 and since w_i, w_{j+1}, and w_{k+1} are the three smallest weights in the list $w_i, \ldots, w_{k+1}$ it must be that $\overline{(i,j)}$ or $\overline{(j+1,k)}$ is in a shortest path from $(0,0)$ to $\overline{(i,k)}$. Thus, the prune operation above processes this properly.

Suppose the prune operation is correct for all trees T_d of depth d, and take the tree T_{d+1} of depth $d+1$. Without loss, say T_{d+1} contains two subtrees of depth d and the root of T_{d+1} is $\overline{(i,k)}$. In addition, if the two depth d subtrees of T_{d+1} have roots $\overline{(i,j)}$ and $\overline{(j+1,k)}$, then by the properties of critical nodes, we know that the three smallest weights in $w_i, \ldots, w_{k+1}$ are w_i, w_{j+1}, and w_{k+1}. Without loss, say $w_{k+1} < w_i < w_{j+1}$, which by Corollary 5 tree node $\overline{(i,j)}$ must be in a shortest path from $(0,0)$ to $\overline{(i,k)}$.

By the inductive hypothesis, if all the subtrees rooted at $\overline{(i,j)}$ and $\overline{(j+1,k)}$ have been pruned, then there is a shortest path from $(0,0)$ to $\overline{(i,j)}$ and another to $\overline{(j+1,k)}$. Pruning leaf $\overline{(j+1,k)}$ finds all critical nodes that are in each of the two subtrees rooted at $\overline{(i,j)}$ and $\overline{(j+1,k)}$. The pruning operation is just an edge minimization to see which combinations of critical nodes may from a shortest path form $(0,0)$ to $\overline{(i,k)}$. $\square$

The tree pruning of Lemma 10 takes $O(\lg^2 n)$ time using $n/\lg n$ processors following standard tree contraction techniques. Either Corollary 5 gives the leaf pruning ordering, or they can be pruned arbitrarily where we would choose an ordering like the one given by the Euler tour technique.

Lemma 11 Given two nested canonical graphs $D^{(i,v)}_{(j,u)}$ and $D^{(k,t)}_{(r,s)}$, where $j \leq k < t \leq u$, with no such canonical subgraph between them, we can join them with one merge operation.

A proof of this follows the proof of Lemma 10 in a straightforward manner.

Assume that the shortest paths in all canonical subgraphs are computed first at a cost of $O(\lg^2 n)$ time and $n^3/\lg n$ processors. Independently, compute the shortest paths of all of these canonical subgraphs. Then the pruning algorithm takes $O(\lg^2 n)$ time using $n^3/\lg n$ processors.

Now considering Lemmas 11 and 10 we can solve the MCOP by performing tree contraction with the prune operation.

Analyzing this algorithm gives the following theorem.

Theorem 24 We can solve the MCOP in $O(\lg^3 n)$ time using $n^3/\lg n$ processors.

This algorithm uses $O(n)$ nodes in a D_n graph to solve the MCOP. Thus we can solve the MCOP by using only $O(n)$ elements of a classical dynamic programming table.

Theorem 24 also applies to the optimal convex triangulation problem with the standard triangle cost metrics (Cormen et al., 1990; Hu and Shing, 1982).

7.4 Historical Notes

The algorithm outlined here is basically from (Bradford, 1992). An extended abstract was published in (Bradford, 1992a) and the full version is in (Bradford, 1992b). The full version was available and circulated since May of 1992. Other parallel algorithms for the MCOP with better work were also reported in: (Czumaj, 1993; Ramanan, 1992; Ramanan, 1993; Bradford et al., 1992).

The model used in this chapter is important for the study of efficient parallel algorithms for the MCOP and other problems. Furthermore, it is nice to have a re-characterization of Hu and Shing's work on the MCOP.

Chapter 8

An $O(\lg^2 n)$ Time and n Processor Algorithm

This chapter contains an $O(\lg^2 n)$ time algorithm for solving the matrix chain ordering problem that uses n processors on either the common-CRCW PRAM or an EREW PRAM. This algorithm can be adapted to take $O(\lg^3 n)$ time and use $n/\lg n$ processors on either of these models. The best algorithms in this chapter have processor-time products of $O(n \lg^2 n)$ just a $\lg n$ factor from the work of the sequential algorithm of Hu and Shing. Further, our algorithms will have processor-time products of $O(n \lg n)$ if an algorithm is discovered that has takes $O(\lg n)$ time and $n/\lg n$ processors for finding the row minima of totally monotone matrices.

This chapter starts by isolating the $n^3/\lg n$ processor bottlenecks of the $O(\lg^2 n)$ time and $\tilde{O}(n^3)$ work algorithm from the last chapter. The first key insight of this chapter is that a carefully constructed inductive invariant allows the replacement of the brute force all-pairs shortest paths algorithm by successive applications of parallel partial prefix and binary search algorithms. This blasts the process complexity from $n^3/\lg n$ down to $n/\lg n$ or $n/\lg\lg n$, or even n depending on the desired time complexity. The algorithms with these processor complexities have $O(n \lg^3 n)$ total work.

The next central insight is that both the time and work can be improved by using efficient parallel algorithms for computing row minima of totally monotone matrices.

8.1 The $n^3/\lg n$ Processor Bottlenecks

This section gives the $n^3/\lg n$ processor bottlenecks of the main algorithm in Chapter 7.

Three parts of the algorithm in Chapter 7 use $n^3/\lg n$ processors. All other parts of this algorithm use a total of $n/\lg n$ processors and take $O(\lg n)$ time. The three bottlenecks are: Finding shortest paths from all critical nodes in leaf graphs back to $(0,0)$, see Figure 26**a**. Merging two bands, see Figure 26**b**, and merging two bands that have contracted canonical trees between them, see Figure 26**c**.

In Figure 26**c**, contracted trees **A** and **B** are used to edge minimize the unit paths marked by "**Min-A**" and "**Min-B**." Edge minimizing the unit paths in the outer band with the contracted trees gives an instance of the second bottleneck, see Figure 26**b**. Edge minimizing the unit paths in the outer band with the contracted trees costs $O(\lg n)$ time with $n^2/\lg n$ processors. Section 8.5 shows how to perform such edge minimization in $O(\lg^2 n)$ time with $n/\lg n$ processors.

Finding shortest paths back to $(0,0)$ from all critical nodes in a leaf graph, as in Figure 26**a**, is done by breaking it into nested bands. Therefore, finding efficient parallel methods of band merging and edge minimization gives an efficient parallel algorithm for the MCOP. So, finding efficient ways to get shortest paths from all critical nodes back to $(0,0)$ by edge minimization in leaf subgraphs partitioned as bands is the focus of the rest of this chapter.

Given the band $D_{(j,t)}^{(i,v)}$, let $\overline{p}_{(j,t)}^{(i,v)}$ denote a shortest path from (i,v) back to $(0,0)$ totally contained in $D_{(j,t)}^{(i,v)}$, see Figure 27. When there is no ambiguity, $\overline{p}_{(j,t)}^{(i,v)}$ will be written as $\overline{p}$.

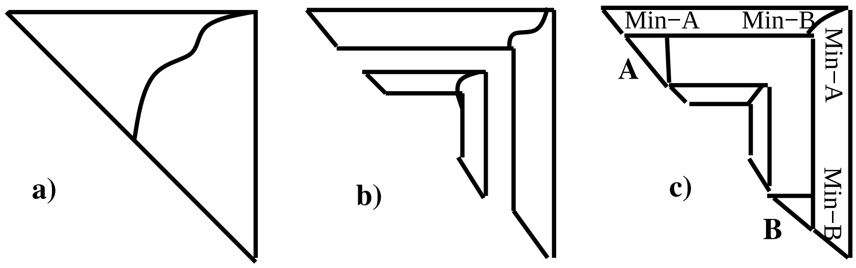

Figure 26: Bottlenecks 1, 2, and 3 for the $n^3/\lg n$ Processor Algorithm

Given a band $D_{(j,t)}^{(i,v)}$, whenever $\overline{p} = \overline{p}_{(j,t)}^{(i,v)}$ starts from the front critical node of the band, then the nodes $V[\overline{p}]$ are *super*-critical nodes.

Take the minimal path back from the front critical node in Figure 27, only the two black critical nodes are super-critical nodes. Super-critical nodes of a band are all of the critical nodes in some minimal path from the front critical node back to $(0, 0)$. Any two super-critical nodes in $\overline{p}$ are connected by super-critical nodes interspersed with angular paths by Theorem 18.

When a canonical tree of D_n is totally contracted then the final path $\overline{p}$ from $(1, n)$ back to $(0, 0)$ gives the optimal order to multiply the set of n matrices. In addition, the cost of $\overline{p}$ is the minimal cost of multiplying the given chain of n matrices.

8.2 A Metric for Minimal Cost Angular Paths

This section gives a metric for finding a minimal cost angular paths by using the equivalence of angular paths and jumpers along unit paths. This equivalence comes directly from Theorem 2.

When merging two bands, a unit path has at most one jumper minimizing it since all of the relevant jumpers are nested. These jumpers get their *sp* values from super-critical nodes of the inner band.

The influence of an angular edge can be taken as a jumper in a straight unit path

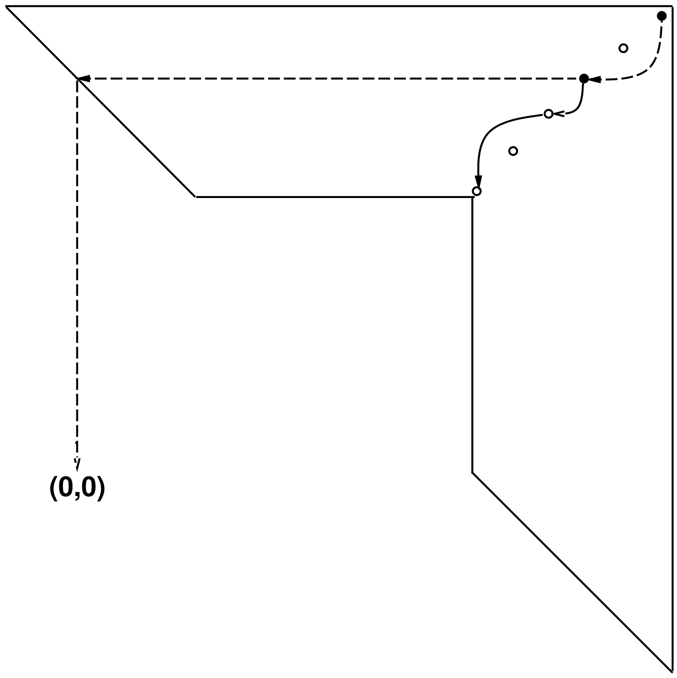

Figure 27: The Dashed Path IS $\overline{p}$ and the Two Black Nodes Are Super-Critical Nodes

by Theorem 2. In the case of Figure 26**c**, any unit edge minimization using sp values from **A** or **B** is independent of unit edge minimization using sp values from the inner band. Therefore, measuring the potential contribution of angular edges to straight unit shortest paths is done by measuring the potential contribution of jumpers to shortest paths along straight unit paths.

Take a node $(s,t) \in V[\overline{p}]$, where $sp(s,t)$ is the cost of a shortest path back to $(0,0)$, with respect to a band, then in row i we want to compare the cost of the jumper $(i, s-1) \Longrightarrow (i,t)$ with the cost of the associated unit path $(i, s-1) \rightarrow \cdots \rightarrow (i,t)$.

Given $(s,t) \in V[\overline{p}]$, take row i above p with the jumper $(i, s-1) \Longrightarrow (i,t)$, define

$$\Delta_i(s,t) \quad = \quad w_i \| w_s : w_{t+1} \| - [\ sp(s,t) + f(i, s-1, t)\ ].$$

If $\Delta_i(s,t) > 0$, then the jumper $(i, s-1) \Longrightarrow (i,t)$ provides a *cheaper* path along row i than the unit path $(i, s-1) \rightarrow \cdots \rightarrow (i,t)$. In particular, take both $(s,t) \in V[\overline{p}]$ and

$(x, y) \in V[\overline{p}]$, and the two possible jumper nestings of Figure 28.

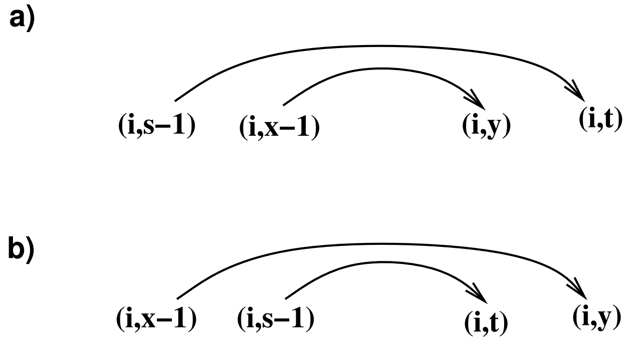

Figure 28: Two Different Nestings of Two Jumpers

For the nesting of jumpers of Figure 28**a**, if $\Delta_i(s,t) > \Delta_i(x,y) > 0$, then the jumper $(i, s-1) \Longrightarrow (i,t)$ "saves more" than the jumper $(i, x-1) \Longrightarrow (i,y)$ along row i because $(i, s-1) \Longrightarrow (i,t)$ doesn't have to deal with the paths $(i, s-1) \to \cdots \to (i, x-1)$ and $(i,y) \to \cdots \to (i,t)$ as well as $\Delta_i(s,t) > \Delta_i(x,y) > 0$. Similarly, for the nesting of Figure 28**b**, if $\Delta_i(x,y) > \Delta_i(s,t) > 0$, then the jumper $(i, x-1) \Longrightarrow (i,y)$ "saves more" than the jumper $(i, s-1) \Longrightarrow (i,t)$ along row i. Notice, in nesting of Figure 28**b**, if $\Delta_i(s,t) > \Delta_i(x,y) > 0$, then the jumper $(i, s-1) \Longrightarrow (i,t)$ may or may not make row i cheaper than the jumper $(i, x-1) \Longrightarrow (i,y)$. But on the other hand, for Figure 28**b**, if $(i, s-1) \Longrightarrow (i,t)$ makes row i cheaper than the jumper $(i, x-1) \Longrightarrow (i,y)$ does, then $\Delta_i(s,t) > \Delta_i(x,y) > 0$.

In $D^{(1,m)}$, if $(s,t) \in V[p]$, then above the path of critical nodes p, the function $\Delta_i(s,t)$ is defined for all rows i such that $s > i \geq 1$.

Edge minimizing a unit paths is only half of the game, since we must also consider the shortest paths forward.

Figure 29 is for the next theorem, see also (Hu and Shing, 1982).

Theorem 25 Given a leaf graph $D^{(i,v)}_{(r,s)}$ and any two critical nodes (j,u) and (k,t) in $D^{(i,v)}_{(r,s)}$ such that there is a unit path from (k,t) to (j,u), then a shortest path from (j,u) to (i,v) costs less than a shortest path from (k,t) to (i,v).

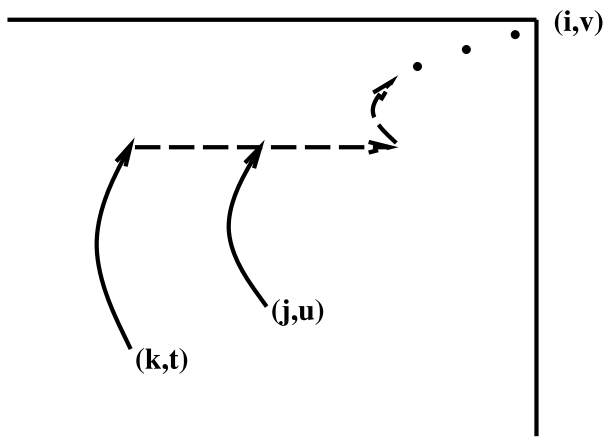

Figure 29: (j, u) Shadowing (k, t)'s Shortest Path Forward

A proof follows inductively by shadowing trivial angular paths without any jumpers then showing that any shortest path from (k, t) forward to (i, v) can be shadowed by a shorter path from (j, u) forward to (i, v) then taking into account the f values. See Figure 29. Theorem 25 also holds for shortest paths forward in leaf graphs.

The next theorem will also be useful.

Theorem 26 Say $(i, s-1) \Longrightarrow (i, t)$ is in a shortest path forward, and in the next band merging the value of $sp(s, t)$ decreases due to an edge minimization of row s or a lower row, then $(i, s-1) \Longrightarrow (i, t)$ is still in a shortest path forward.

A proof of this theorem follows directly from the basic notions of shortest paths. In particular, if the shortest path forward from the critical node (s, t) goes through $(s, t) \Uparrow (i, t)$, then making the path to (s, t) shorter will not effect the jumper $(s, t) \Uparrow (i, t)$ or the path from (i, t) to the front node of the present band.

8.3 A Polylog-Time and $n^2/\lg n$ Processor MCOP Algorithm

This section contains an $O(\lg^2 n)$ time and $n^2/\lg n$ processor algorithm for the MCOP. This algorithm works by using a key induction invariant that allows recursive doubling techniques to break through the bottlenecks given in the last section.

The basic idea of the algorithm is the following. All critical nodes know their shortest paths to the front of the present bands they are in. Only super-critical nodes have their shortest paths back to $(0,0)$ thorough their present bands. When merging two bands, by Theorem 21, we only have to consider shortest paths from super-critical nodes in the inner band to any critical node in the outer band. Therefore, all critical nodes must maintain a shortest path to the front of the band they are in. At the same time, all super-critical nodes must maintain a shortest path backwards to $(0,0)$ through the band they are in. This section supplies the details and correctness of this algorithm.

Each critical node in D_n has two pointers called *front-ptr* and *back-ptr* that represent angular edges. *Back-ptr*-s are only used by super-critical nodes. With each *front-ptr* there are two values, *cost-of-front-ptr* and *cost-to-front*; and with each *back-ptr* there is one value *cost-to-back*. *Cost-of-front-ptr* is the cost of the single angular edge going forward from the present node to the front critical node in the present band. Where, the value of *cost-to-front* is the cost to the front critical node of the present band containing *front-ptr*. Similarly, the value of *cost-to-back* is the cost from the super-critical node at hand back to $(0,0)$ through the present band. Initially, these pointers connect critical nodes and tree edges in the canonical tree.

Let $D_{(j,t)}^{(i,v)}$ and $D_{(k,s)}^{(j,t)}$ be nested bands with paths of critical nodes labeled $p_{(j,t)}^{(i,v)}$ and $p_{(k,s)}^{(j,t)}$, respectively. Note (i,v) and (j,t) are the front critical nodes of these bands. As before, $\overline{p}_{(j,t)}^{(i,v)}$ and $\overline{p}_{(k,s)}^{(j,t)}$ are shortest paths from the front critical nodes back to $(0,0)$

through the bands $D^{(i,v)}_{(j,t)}$ and $D^{(j,t)}_{(k,s)}$, respectively. Let $\overline{p}^{(i,v)}_{(j,t)}$ and $\overline{p}^{(j,t)}_{(k,s)}$ be made by two linked lists of *back-ptr*-s along super-critical nodes back to $(0,0)$ in their bands. It turns out that the shortest paths forward from all critical nodes in each of these bands are made up of linked lists of trees of *front-ptr*-s. We will see that this linked list of trees of *front-ptr*-s is interconnected through the super-critical nodes as in Figure 31.

1. All critical nodes in both bands have their *front-ptr*-s in trees of shortest paths that eventually go to super-critical nodes which go to the front critical nodes of their respective bands.

2. In the two bands both shortest paths back to $(0,0)$ of super-critical nodes are known. These shortest paths of super-critical nodes are made of linked lists of *back-ptr*-s from the front critical nodes of each band back through their respective bands to $(0,0)$.

Figure 30: An Inductive Invariant for Band Merging

Figure 30 gives the induction invariant for merging two bands. Figure 31 gives an example of the data structures for maintaining the inductive invariant. In this figure only critical nodes are shown and the super-critical nodes are black. The solid arrows are *front-ptr*-s and the dashed arrows are *back-ptr*-s.

Suppose (s,t) is a critical node but not a super-critical node, that is $(s,t) \in V[p]$ and $(s,t) \notin V[\overline{p}]$. There is a unique angular edge

$$(x,y) \Uparrow (r,y) \rightarrow \cdots \rightarrow (r,u)$$

in $\overline{p}$ that "goes around" (s,t), see Figure 32. Given a canonical graph and take all rows above p then $w_i < w_r$ implies that row i is "above" row r as in Figure 32. From here on the focus is on finding shortest paths above the path p of critical nodes, the

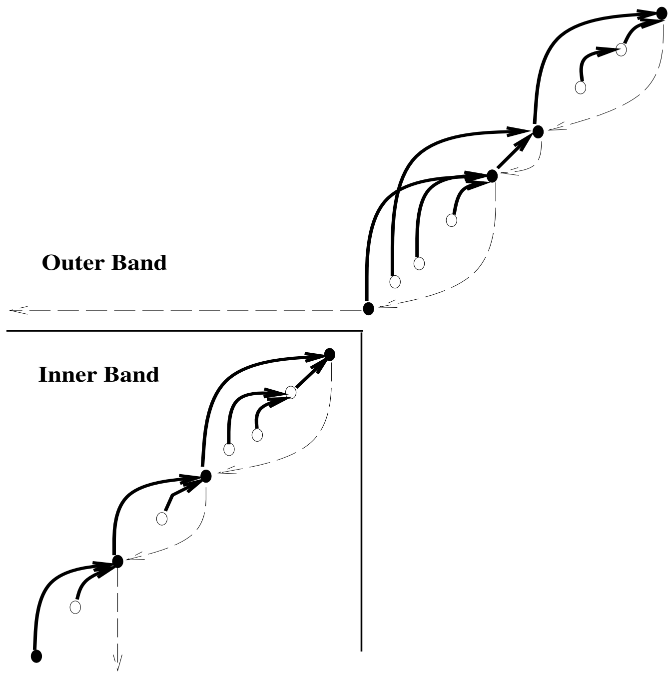

Figure 31: Solid Arrows: Forward Linked Lists of Trees; Dashed Arrows: Backward Linked Lists $\overline{p}$

symmetric case of shortest paths below the path p of critical nodes follows.

If all unit paths in $D^{(i,v)}_{(j,k)}$ are edge minimized with jumpers that get their sp values from super-critical nodes in $D^{(j,k)}_{(s,t)}$, then we can find the shortest path from (i,v) back to $(0,0)$ through $D^{(i,v)}_{(s,t)}$. First take one processor at each critical node in $D^{(i,v)}_{(j,k)}$ that sums the cost of the path back to $(0,0)$ possibly through an edge minimized unit path with the cost of its shortest path forward. Next, find the minimal of all of these sums, giving the shortest path from (i,v) back to $(0,0)$ through $D^{(j,k)}_{(s,t)}$.

The basic intuition for the next lemma is if the shorter of two nested jumpers edge minimizes a unit path r, then any unit path above r with both of these jumpers is not minimized by the longer jumper, see Figure 33. Assume there is a unit path of critical nodes from (x,y) to (s,t) to (r,u) as in Figure 32 for the next lemma.

Lemma 12 Given a critical node (s,t) between the *super*-critical node (x,y) and critical

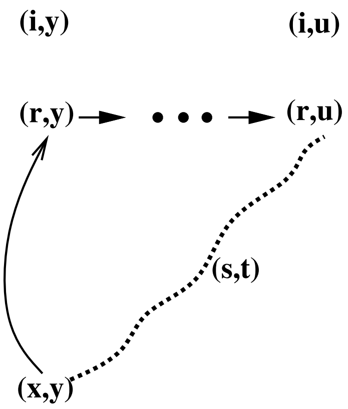

Figure 32: $(s,t) \notin V[\overline{p}]$ and the Angular Edge $(x,y) \Uparrow (r,y) \rightarrow \cdots \rightarrow (r,u)$

node (r,u) and say $i < r < s < x$ and row i is above row r, that is $w_i < w_r$, where rows i and r are above p then

$$\textbf{if } \Delta_r(x,y) \geq \Delta_r(s,t), \textbf{ then } \Delta_i(x,y) \geq \Delta_i(s,t)$$

Proof: Start with $\Delta_r(x,y) \geq \Delta_r(s,t)$ which gives that,

$$w_r \|w_x : w_{y+1}\| - [\ sp(x,y) + f(r,x-1,y)\ ]$$

is larger than or equal to

$$w_r \|w_s : w_{t+1}\| - [\ sp(s,t) + f(r,s-1,t)\ ]$$

and a little algebra gives the following (where $\|w_i : w_i\| = 0$),

$$w_r[\ \|w_s : w_x\| + \|w_{y+1} : w_{t+1}\|\ ] \quad < \quad sp(s,t) - sp(x,y) + w_r(w_s w_{t+1} - w_x w_{y+1})$$

and $sp(s,t) - sp(x,y)$ is always positive because $(r,x-1) \Longrightarrow (r,y)$ is nested inside of $(r,s-1) \Longrightarrow (r,t)$ and $f(r,s-1,t) < f(r,x-1,y)$. Therefore, if $sp(x,y) > sp(s,t)$, then a shortest path $\overline{p}$ would go through (s,t) to (r,u) and *not* over (s,t).

In particular if $sp(x,y) > sp(s,t)$, then since $f(r,x-1,t) > f(r,s-1,t)$, it must be that $sp(x,y) + f(r,x-1,t) > sp(s,t) + f(r,s-1,t)$. Therefore row r would have been edge minimized by jumper $(r,s-1) \Longrightarrow (r,t)$ and *not* by $(r,x-1) \Longrightarrow (r,y)$, see Figure 33.

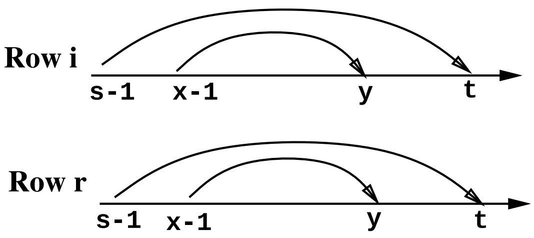

Figure 33: Two Jumpers in Different Rows

In addition, $w_s w_{t+1} - w_x w_{y+1} < 0$ since both (x,y) and (s,t) are critical nodes where $s \le x < y \le t$, so it must be that $w_x w_{y+1} - w_s w_{t+1} > 0$. Therefore, since

$$w_r[\ \|w_s : w_x\| + \|w_{y+1} : w_{t+1}\| + w_x w_{y+1} - w_s w_{t+1}\ ] \quad < \quad sp(s,t) - sp(x,y)$$

holds and because $w_i < w_r$ and the term $sp(s,t) - sp(x,y)$ is independent of i and r then it must be that $\Delta_i(x,y) \ge \Delta_i(s,t)$. $\square$

The next theorem follows from Lemma 12.

Theorem 27 Given a critical node (s,t) between the *super*-critical node (x,y) and critical node (r,u) and say $i < r < s < x$ and row i is above row r, that is $w_i < w_r$, where rows i and r are above p then

> **if** $(r,x-1) \Longrightarrow (r,y)$ makes row r cheaper than $(r,s-1) \Longrightarrow (r,t)$ does,
> **then** $(i,x-1) \Longrightarrow (i,y)$ makes row i cheaper than $(i,s-1) \Longrightarrow (i,t)$ does.

A proof follows from Lemma 12 and by the fact that the rows

$$(i,s-1) \to \cdots \to (i,x-1) \text{ and } (i,y) \to \cdots \to (i,t)$$

are cheaper than

$$(r, s-1) \rightarrow \cdots \rightarrow (r, x-1) \text{ and } (r, y) \rightarrow \cdots \rightarrow (r, t)$$

and the change in f values between $(r, x-1) \Longrightarrow (r, y)$ and $(i, x-1) \Longrightarrow (i, y)$ is greater than the change of f values between $(r, s-1) \Longrightarrow (r, t)$ and $(i, s-1) \Longrightarrow (i, t)$. That is,

$$f(r, x-1, y) - f(i, x-1, y) > f(r, s-1, t) - f(i, s-1, t)$$

since w_x and w_{y+1} are both bigger than w_s and w_{t+1}, in addition $w_r > w_i$ therefore,

$$(w_r - w_i)[w_x w_{y+1} - w_s w_{t+1}] > 0.$$

Given two nested bands with paths of critical nodes p_i for the inner band and p_o for the outer band. Where, $\overline{p_i}$ and $\overline{p_o}$ are shortest paths from the front critical nodes back to $(0,0)$ in each of these bands. Suppose (s,t) is between (x,y) and (r,u) and $(s,t) \in V[\overline{p_i}]$ and if $(x,y) \in V[\overline{p_i}]$ and $(r,u) \in V[p_o]$, then Lemma 12 and Theorem 27 also hold. This is because $sp(s,t) - sp(x,y)$ is positive by an argument similar to that in the proof of Lemma 12.

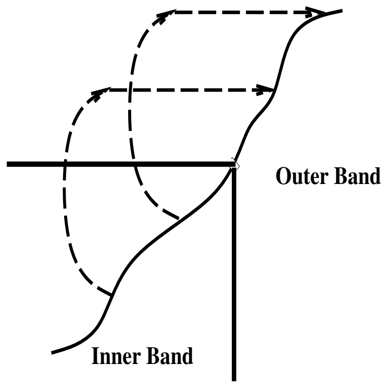

Figure 34: Conflicting Angular Paths *Between* Two Bands Being Merged

Two angular edges above p, say

$$(x, y) \Uparrow (r, y) \to \cdots \to (r, u)$$

and

$$(i, j) \Uparrow (s, j) \to \cdots \to (s, t),$$

are *compatible* if they don't cross each other. Compatibility also holds for angular paths below p. Theorem 28 shows that *when merging* two bands and computing shortest paths forward, only compatible angular edges need to be considered. Figure 34 shows two conflicting angular paths.

Take the canonical graphs $D^{(a,z)}_{(c,x)}$, $D^{(d,v)}_{(e,u)}$, and $D^{(g,t)}$, where $D^{(g,t)}$ is nested inside of $D^{(d,v)}_{(e,u)}$ which is, in turn, inside of $D^{(a,z)}_{(c,x)}$, see Figure 35. And assume that $D^{(a,z)}_{(c,x)}$ and $D^{(d,v)}_{(e,u)}$ are to be merged together then in the next recursive doubling step the new band $D^{(a,z)}_{(e,u)}$ will be merged with the leaf $D^{(g,t)}$. We can assume $D^{(g,t)}$ is a leaf or a band.

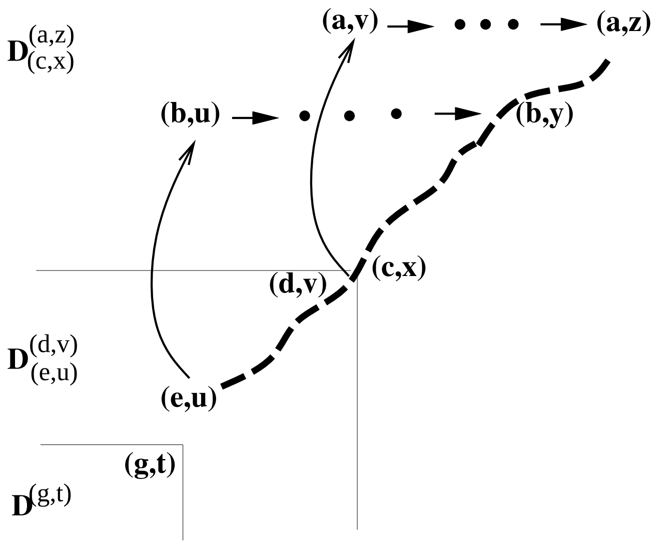

Figure 35: The Bands $D^{(a,z)}_{(c,x)}$, $D^{(d,v)}_{(e,u)}$ and the Leaf $D^{(g,t)}$

The next theorem assumes we have found a shortest path from super-critical nodes in $D^{(d,v)}_{(e,u)}$, through critical nodes in the outer band $D^{(a,z)}_{(c,x)}$, see Figure 35. Of course, $(d,v) \in V[\overline{p}^{(d,v)}_{(e,u)}]$ and without loss assume

$$(e,u) \Uparrow (d,u) \to \cdots \to (d,v)$$

is in $\overline{p}^{(d,v)}_{(e,u)}$. Suppose the angular edge

$$(e,u) \Uparrow (b,u) \to \cdots \to (b,y)$$

is in $\overline{p}^{(a,z)}_{(e,u)}$, that is the minimal path from (a,z) back to $(0,0)$ through $D^{(a,z)}_{(e,u)}$.

Theorem 28 In merging two nested bands computing shortest paths forward from super-critical nodes of the inner band, we can consider only compatibly nested angular edges.

Proof: Suppose for the sake of a contradiction,

$$(e,u) \Uparrow (b,u) \to \cdots \to (b,y)$$

is in $\overline{p}^{(a,z)}_{(e,u)}$, that is the angular edge

$$(e,u) \Uparrow (b,u) \to \cdots \to (b,y)$$

is in a shortest path from (a,z) back to $(0,0)$ through $D^{(a,z)}_{(e,u)}$, see Figure 35. Take (d,v) in the band $D^{(d,v)}_{(e,u)}$, therefore (d,v) is between (e,u) and (a,z) in $D^{(a,z)}_{(e,u)}$. Now, when merging $D^{(g,t)}$ with $D^{(a,z)}_{(e,u)}$ we will show that a shortest path forward to (a,z) that goes through (d,v) must go through a critical node in row b or a critical node in some row below b.

Now assume otherwise, say after merging $D_{(c,x)}^{(a,z)}$ with $D_{(e,u)}^{(d,v)}$ there is some shortest path from $(0,0)$ through (d,v) to (a,z) traveling through an angular path connecting the bands $D_{(c,x)}^{(a,z)}$ and $D_{(e,u)}^{(d,v)}$ and this angular path is conflicting with the angular path

$$(e,u) \Uparrow (b,u) \rightarrow \cdots \rightarrow (b,y).$$

Say, without loss, this conflicting angular path is

$$(d,v) \Uparrow (a,v) \rightarrow \cdots \rightarrow (a,z),$$

see Figure 35. These are conflicting angular paths since the shortest path from (d,v) forward goes through an angular path that terminates above row b, and the shortest path forward from (e,u) goes through an angular path that terminates in row b. But, *in* $D_{(e,u)}^{(a,z)}$ the shortest path from (a,z) back to $(0,0)$ still goes through the angular edge

$$(e,u) \Uparrow (b,u) \rightarrow \cdots \rightarrow (b,y).$$

In $D_{(e,u)}^{(a,z)}$ the angular edge

$$(d,v) \Uparrow (a,v) \rightarrow \cdots \rightarrow (a,z)$$

can't be the shortest path forward from (d,v).

By Theorem 2, the shortest path to (b,y) through the angular edge

$$(e,u) \Uparrow (b,u) \rightarrow \cdots \rightarrow (b,y)$$

is equivalent to the path

$$(b,b) \rightarrow \cdots \rightarrow (b,e-1) \Longrightarrow (b,u) \rightarrow \cdots \rightarrow (b,y).$$

And since $\overline{p}_{(e,u)}^{(a,z)}$ goes through

$$(e,u) \Uparrow (b,u) \rightarrow \cdots \rightarrow (b,y),$$

the jumper $(b, e-1) \Longrightarrow (b,u)$ edge minimizes row b. Thus, the jumper $(b, d-1) \Longrightarrow (b,v)$ saves at most as much as $(b, e-1) \Longrightarrow (b,u)$ and at the same time $(b, d-1) \Longrightarrow (b,v)$ is nested around $(b, e-1) \Longrightarrow (b,u)$ which means

$$\Delta_b(e,u) \geq \Delta_b(d,v).$$

Also, by Theorem 2, the shortest path from (d,v) to (a,z) that is through the angular edge

$$(d,v) \Uparrow (a,v) \rightarrow \cdots \rightarrow (a,z)$$

is equivalent to the path

$$(a,a) \rightarrow \cdots \rightarrow (a, d-1) \Longrightarrow (a,v) \rightarrow \cdots \rightarrow (a,z)$$

But, consider the path,

$$(a,a) \rightarrow \cdots \rightarrow (a, e-1) \Longrightarrow (a,u) \rightarrow \cdots \rightarrow (a,z)$$

and it must be that $(a, e-1) \Longrightarrow (a,u)$ is nested inside of $(a, d-1) \Longrightarrow (a,v)$. In this case, it is possible that $d = e$ or $u = v$ but not both.

Since (d,v) is between (e,u) and (b,y) and $a < b$ and $w_a < w_b$ where row a is above row b and they both are above p, and since the appropriate Δ values are defined, the following holds by Lemma 12,

$$\textbf{if } \Delta_b(e,u) \geq \Delta_b(d,v), \textbf{ then } \Delta_a(e,u) \geq \Delta_a(d,v).$$

Therefore,

$$\Delta_a(e,u) \quad \geq \quad \Delta_a(d,v)$$

which means the jumper $(a, e-1) \Longrightarrow (a,u)$ saves at least as much as the jumper $(a, d-1) \Longrightarrow (a,v)$ in a path to (a,z).

And because $(b, e-1) \Longrightarrow (b,u)$ edge minimizes row b, by Theorem 27 and since $\Delta_a(e,u) \geq \Delta_a(d,v)$ the jumper $(a,e-1) \Longrightarrow (a,u)$ saves more in row a than $(a,d-1) \Longrightarrow (a,v)$.

Now, letting

$$\begin{aligned} \mathcal{A} &= (a,a) \rightarrow \cdots \rightarrow (a,e-1) \Longrightarrow (a,u) \rightarrow \cdots \rightarrow (a,v) \\ \mathcal{D} &= (d,d) \rightarrow \cdots \rightarrow (d,e-1) \Longrightarrow (d,u) \rightarrow \cdots \rightarrow (d,v) \Uparrow (a,v) \end{aligned}$$

see Figure 36.

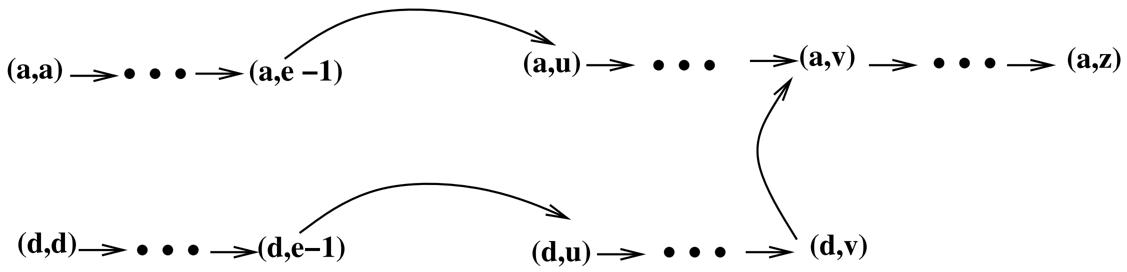

Figure 36: The Two Paths $\mathcal{A}$ and $\mathcal{D}$

Path $\mathcal{A}$ is a cheaper than path $\mathcal{D}$ going from (a,z) back to $(0,0)$ in $D^{(a,z)}_{(e,u)}$ by Theorem 27. Now, if $(d,v) \Uparrow (a,v)$ is in a shortest path forward from (d,v), then the shortest path forward from (e,u) must be through the angular path

$$(e,u) \Uparrow (a,u) \rightarrow \cdots \rightarrow (a,z)$$

and not the angular path

$$(e, u) \Uparrow (b, u) \rightarrow \cdots \rightarrow (b, y)$$

which is a contradiction. This comes from the application of Theorem 27 to the jumpers $(b, d-1) \Longrightarrow (b, v)$ and $(b, e-1) \Longrightarrow (b, u)$ in row b and then up to row a, since (b, y) is between (d, v) and (a, z).

Now, suppose $D^{(g,t)}$ is merged with the outer band $D^{(a,z)}_{(e,u)}$ then none of the angular paths connecting super-critical nodes in $D^{(g,t)}$ with paths forward $D^{(a,z)}_{(e,u)}$ change. This case is a straightforward application of the proof above and Theorem 26. □

It is important to note that Theorem 28 only shows that angular paths that all start from super-critical nodes in the same path back to $(0, 0)$ are compatible. Theorem 28 doesn't say that all angular paths are always compatible.

Suppose, there is some angular path from a super-critical node in the inner band, say (s, t), to the outer band that is in a shortest path from the front node of the outer band back to $(0, 0)$. Then all super-critical nodes from (s, t) back to $(0, 0)$ have their shortest paths forward through the angular path starting at (s, t). On the other hand, all super-critical nodes after (s, t) up to the front super-critical node of the inner band have their shortest paths through nested angular paths connecting the inner and outer bands by Theorem 28. In fact, we can inductively apply this argument together with Theorem 25 giving:

Corollary 6 Take the nested angular paths connecting two bands that are shortest paths forward from the different super-critical nodes of the inner band, then listing the path containing the outermost such angular path to the path containing the innermost such angular path gives more and more costly paths forward.

The next lemma assumes we are merging two nested bands to find a shortest path from the front critical node of the outer band back to $(0, 0)$.

Lemma 13 Given a critical node (s, t) and take the i^{th} and r^{th} rows above p such that $i < r < s$ and $w_i < w_r$, then we have $\Delta_i(s, t) < \Delta_r(s, t)$.

Proof: The function $\Delta_i(s, t)$ measures $(i, s-1) \Longrightarrow (i, t)$'s potential minimizing effect on the path $(i, i) \rightarrow \cdots \rightarrow (i, u)$ where $(i, u) \in V[p]$ and $i < s < t \leq u$. The cost of the jumper $(i, s-1) \Longrightarrow (i, t)$ is $sp(s, t) + f(i, s-1, t)$. Therefore, the difference $\Delta_{i+1}(s, t) - \Delta_i(s, t)$ is,

$$(w_{i+1} - w_i)[\ \|w_s : w_{t+1}\| - w_s w_{t+1}\ ]$$

where $w_{i+1} > w_i$. Since the expression $\|w_s : w_{t+1}\| - w_s w_{t+1}$ is independent of the difference of weights w_i and w_{i+1} and $\|w_s : w_{t+1}\| - w_s w_{t+1} > 0$, because $(s, t) \in V[p]$, also when $s = t - 1$ gives

$$\|w_s : w_{t+1}\| \quad = \quad w_s w_{s+1} + w_{s+1} w_{t+1}.$$

In addition, since $(s, t) \in V[p]$, it must be that $\max\{w_s, w_{t+1}\} < w_u$, for $s < u \leq t$, thus $\max\{w_s, w_{t+1}\} < w_{s+1}$. Therefore,

$$w_s w_{s+1} + w_{s+1} w_{t+1} \quad > \quad w_s w_{t+1}$$

and the proof follows inductively. □

The proof of the next lemma is similar to that of Lemma 12. The basic intuition here is that if the longer of two nested jumpers edge minimizes a unit path r, then any unit path below r, with both of these jumpers, is not minimized by the shorter jumper, see Figure 37.

This next lemma only considers super-critical nodes since the focus here is on merging two nested bands. Assume there is a unit path of critical nodes from (v, z) to (x, y) for the next lemma.

Lemma 14 Given two *super*-critical nodes (v, z) and (x, y) where $r < s < x < v$ and $w_r < w_s$ such that rows s and r are above p, then we have

$$\textbf{if } \Delta_r(x, y) \geq \Delta_r(v, z), \textbf{ then } \Delta_s(x, y) \geq \Delta_s(v, z)$$

Proof: Start with $\Delta_r(x, y) \geq \Delta_r(v, z)$ which can be written out as the next expression,

$$w_r \| w_x : w_{y+1} \| - [\ sp(x, y) + f(r, x-1, y)\ ]$$

which is greater than or equal to

$$w_r \| w_v : w_{z+1} \| - [\ sp(v, z) + f(r, v-1, z)\ ].$$

By Lemma 13 and since each of these jumpers is of length at least 2 it must be that $w_r w_v w_{z+1} < w_r \| w_v : w_{z+1} \|$ and $w_r w_x w_{y+1} < w_r \| w_x : w_{y+1} \|$. In addition, since $w_r < w_s$ it must be that $f(r, x-1, y) < f(r, v-1, z)$ and $w_r \| w_x : w_{y+1} \| > w_r \| w_v : w_{z+1} \|$ and the same holds in row s, therefore $\Delta_s(x, y) \geq \Delta_s(v, z)$. $\square$

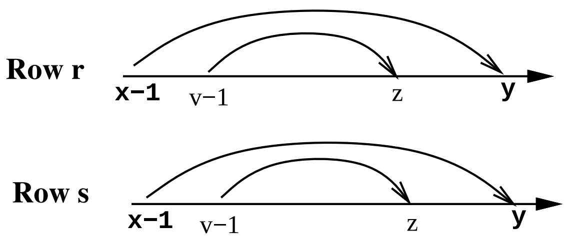

Figure 37: Two Jumpers in Different Rows

Theorem 29 Given two *super*-critical nodes (v, z) and (x, y) where $r < s < x < v$ and $w_r < w_s$ such that rows s and r are above p, then we have

if $(r, x-1) \Longrightarrow (r, y)$ makes row r cheaper than $(r, v-1) \Longrightarrow (r, z)$ does,

then $(s, x-1) \Longrightarrow (s, y)$ makes row s cheaper than $(s, v-1) \Longrightarrow (s, z)$ does.

A proof of this theorem follows from Lemma 14 and the fact that the change of the f values between $(r, v-1) \Longrightarrow (r, z)$ and $(s, v-1) \Longrightarrow (s, z)$ increases faster than the change in the f values between $(r, x-1) \Longrightarrow (r, y)$ and $(s, x-1) \Longrightarrow (s, y)$.

While merging $D^{(i,v)}_{(j,t)}$ and $D^{(j,t)}_{(k,s)}$ to form $\overline{p}^{(i,v)}_{(k,s)}$ the next lemma shows that we only need shortest path values backwards to $(0,0)$ from super-critical nodes. That is, we don't need shortest path values backwards to $(0,0)$ from any other critical nodes. Hence, the *back-ptr*s will form a linked list between super-critical nodes backwards eventually to $(0,0)$ and we can compute the *cost-to-back* weights using a parallel pointer jumping partial prefix.

Lemma 15 Take $\overline{p}^{(i,v)}_{(j,t)}$ and $\overline{p}^{(j,t)}_{(k,s)}$ in $D^{(i,v)}_{(j,t)}$ and $D^{(j,t)}_{(k,s)}$ respectively, for any critical nodes in the outer band, say $(u,z) \in V[p^{(i,v)}_{(j,t)}]$ and $(u,z) \notin V[\overline{p}^{(i,v)}_{(k,s)}]$, we don't need shortest paths *back* to $(0,0)$.

Proof: Take the angular path $(x,y) \Uparrow (q,y) \rightarrow \cdots \rightarrow (u,z)$ between $D^{(j,t)}_{(k,s)}$ and $D^{(i,v)}_{(j,t)}$. That is $(x,y) \in V[\overline{p}^{(j,t)}_{(k,s)}]$ and $(u,z) \in V[p^{(i,v)}_{(j,t)}]$ but suppose, $(u,z) \notin V[\overline{p}^{(i,v)}_{(k,s)}]$. Notice that (u,z) may be a super-critical node in $\overline{p}^{(i,v)}_{(j,t)}$.

Take the next cases,

CASE i: Suppose $D^{(i,v)}_{(k,s)}$ is merged with another band nested around it. Then since (u,z) is not in $V[\overline{p}^{(i,v)}_{(k,s)}]$ by Theorem 21 we do not have to consider any angular paths starting from (u,z) going forward to critical nodes in the band nested around $D^{(i,v)}_{(k,s)}$.

CASE ii: Suppose $D^{(i,v)}_{(k,s)}$ is merged with a smaller band inside $D(k,s)$.

Node (u,z) could be the terminal node of an incoming angular path contributing to a shortest path forward for *some* super-critical node in $D(k,s)$. In this case (u,z) needs to have a shortest path from (u,z) forward. Of course, in this case (u,z) could become a super-critical node and would have a minimal path back to $(0,0)$, but while the critical node (u,z) is not a *super*-critical node it has no need of a shortest path back to $(0,0)$.

These cases complete the proof. $\square$

A shortest path forward must be found for each critical node. This is because some angular path from some future inner band may terminate at any critical node. Therefore, after finding each super-critical node's minimal cost to the front critical node of the outer band then do a tree partial prefix sum from the critical nodes to the super-critical nodes. This allows all critical nodes to know their shortest paths to the front of the outer band.

Suppose recursive doubling generates the band $D_{(j,t)}^{(i,v)}$ and there is a shortest path $\overline{p}_{(j,t)}^{(i,v)}$ from (i,v) back to $(0,0)$. The next theorem shows inductively how to build the appropriate data structures to maintain the inductive invariant through recursive doubling.

Theorem 30 After merging any two nested bands the front pointers of the new band form a tree and the back pointers of the new band form a linked list.

There is a proof by induction based on Theorem 28.

Theorem 30 shows the inductive invariant holds given the appropriate data structures and computations.

8.4 Merging Bands Using $n^2/\lg n$ Processors

This section shows how to merge two bands using $n^2/\lg n$ processors in $O(\lg n)$ time. This algorithm also merges two optimally triangulated convex polygons when all of the weights of one polygon are heavier than all of the weights of the other.

Recursively doubling the band merging algorithm while using the proper data structures and appropriate tree contracting gives the $n^2/\lg n$ processor and $O(\lg^3 n)$ time MCOP algorithm.

Take two adjacent nested bands, say $D^{(i,v)}_{(j,t)}$ nested around $D^{(j,t)}_{(k,s)}$, such that for each band individually the inductive invariant holds.

1. **for all** super-critical nodes $(x, y) \in V[\overline{p}^{(j,t)}_{(k,s)}]$ **in parallel do**
   for all angular edges from (x, y) to all $(u, z) \in V[p^{(i,v)}_{(j,t)}]$ **in parallel do**
   Find the angular edge between the bands that gives a shortest path
   from (x, y) all the way to (i, v), compute the *cost-of-front-ptr*-s
   for these new edges let each super-critical node (x, y) have a pointer
   to a shortest path through $p^{(i,v)}_{(j,t)}$ to (i, v).
   For the super-critical nodes in $\overline{p}^{(j,t)}_{(k,s)}$ put the
   angular edge that gives them a shortest path forward to (i, v) in M.
2. **for all** angular edges in M **in parallel do**
   Find the shortest path N from (i, v) back to $(0, 0)$ through $D^{(i,v)}_{(k,s)}$.
   for all critical nodes in the path N **in parallel do**
   Using pointer jumping build the *back-ptr*-s giving any new super-critical nodes
   and compute the values of *cost-to-back* for each new super-critical node.
3. **for all** *non*-super-critical nodes in $p^{(j,t)}_{(k,s)}$ **in parallel do**
   Using pointer jumping expand the tree of *front-ptr*-s through the new angular
   edges in M and their minimal values to (i, v). This gives trees joined by
   a linked list through the super-critical nodes.
   With this list find the shortest path to (i, v) for all non-super-critical
   nodes in $p^{(j,t)}_{(k,s)}$ by computing a partial prefix in a rooted tree.
   Also compute all of the new *cost-to-front* values using a parallel partial prefix.

Figure 38: A $O(\lg n)$ Time and $n^2/\lg n$ Processor Algorithm for Merging Two Bands

The algorithm in Figure 38 merges two bands in $O(\lg n)$ time using $n^2/\lg n$ processors. Adding the cost of recursive doubling and tree contraction gives a factor of $O(\lg^2 n)$ time to the entire algorithm making the total cost for solving the MCOP $O(\lg^3 n)$ time using $n^2/\lg n$ processors.

The two **for** loops in step 1 of the algorithm in Figure 38 perform the edge minimizing. This is the only part of this algorithm that uses $n^2/\lg n$ processors. In $O(\lg n)$ time using $n^2/\lg n$ processors we can edge minimizing unit paths with contracted trees such as those depicted in the bottleneck of Figure 26**c**.

The **for** loops in step 2 compute the super-critical nodes of the band that is being merged. Step 3 computes the shortest paths forward for all critical nodes in the inner band.

The base case for the recursive doubling can be established by breaking the canonical subgraphs into bands of constant width. Then for each band sequentially, let the $n/\lg n$ processors set up the inductive invariant in $O(\lg n)$ time. Number the nested bands consecutively according to their nestings by the Euler tour technique so the algorithm can track adjacent bands for merging.

The correctness of the algorithm in Figure 39 comes from Theorems 26, 28, and 30. The time complexity of solving the MCOP can be reduced to $O(\lg^2 n)$ time with $n^2/\lg n$ processors by performing band merging and tree contraction basically at the same time.

Theorem 31 Recursive doubling with band merging can be done simultaneously with tree contraction, thereby solving the MCOP in $O(\lg^2 n)$ time with $n^2/\lg n$ processors.

Proof: Take any canonical tree T with non-trivial bands and leaves. Then T has at most $n-1$ critical nodes. In general, for any arithmetic expression tree with $n-1$ nodes, it takes $O(\lg n)$ time to contract it. In a canonical tree we have just seen that each contraction operation (raking) can be done in $O(\lg n)$ time using $n^2/\lg n$ processors. This is because in the worst case a leaf raking operation in a canonical tree is the merging of two bands. Now, each band can be seen as no more than a linked list in the canonical tree that must be contracted where there is one leaf per list node. Now, we can just take every band that has k critical nodes and is in any canonical tree, and we can assume that it has 2^c "linked list nodes" such that $2^{c-1} < k \leq 2^c$. With this, each raking operation will cost at most $O(\lg n)$ using $n^2/\lg n$ processors. In addition, by assuming k is the nearest power of two greater than or equal to k, we are at most doubling the number of critical nodes in T, hence the asymptotic bound claimed holds. □

8.5 Efficient Polylog-Time MCOP Algorithms

This section contains the last details of the $O(\lg^2 n)$ time and n processor MCOP algorithm. Other variations of this algorithm are based on the different complexities of solving the ANSV problem and the row minima problem on totally monotone matrices.

This section reduces the processor complexity of the band merging algorithm of Section 8.3. The results in this section are based on a parallel divide-and-conquer form of binary search which is tied into some classical problems of finding row minima in totally monotone matrices. Theorems 27 and 29 supply the basis for a parallel divide and conquer binary search algorithm that finds the jumpers that minimize each unit path in a canonical graph.

Theorem 32 Suppose that r is the row in the outer band such that the dual of $(r, x) \Longrightarrow (r, y)$ gives a shortest path forward from the super-critical node $(x-1, y)$ of the inner band to the front node of the outer band then to find shortest paths forward from other super-critical nodes,

- It is sufficient to consider only larger nested jumpers in any row s below row r, that is $w_s > w_r$,
- It is sufficient to consider only smaller nested jumpers in any row i above row r, that is $w_i < w_r$.

A proof of this theorem comes directly from Theorems 27 and 29.

The next algorithm replaces the two nested **for** loops in step 1 in the algorithm of Figure 38. This next algorithm gives shortest paths forward for all super-critical nodes originally in the inner band and a shortest path back to $(0,0)$ through the two merged bands.

Say each band has m critical nodes, the next procedure finds shortest paths from all super-critical nodes of the inner band to the front of the outer band. In addition, from the shortest path information before the merging and all shortest paths to the front of the outer band, then the shortest path back from the front node of the outer band is easily computed. As before, begin assuming the inductive invariant. Also, all jumpers in the next algorithm are jumpers that get their sp values from the inner band where the jumpers themselves are in unit rows or columns of the outer band.

The following algorithm is striking similar to those discussed in (Aggarwal et al., 1987) and (Aggarwal and Park, 1988). This key observation leads to some complexity improvements.

Now assigning one processor to each unit path in the outer band and then summing the cost up to the critical node and the cost from the critical node to the front super-critical node of the outer band gives a shortest path backwards from the front node of the outer band to $(0,0)$. Computing these minimal paths costs $O(\lg n)$ time using $n/\lg n$ processors. If a unit path has no edge minimizing jumpers, then this algorithm just finds the shortest path forward for all super-critical nodes in the inner band. Since, in this case, the shortest path back to $(0,0)$ from the front critical node of the outer band does not go through the inner band.

The algorithm in Figure 39 also breaks through the bottleneck of Figure 26**c**. It takes $O(\lg^2 n)$ time and uses $n/\lg n$ processors in the worst case. Considering the cost of the recursive doubling and the tree contraction gives the $O(\lg^3 n)$ time and $n/\lg n$ processor matrix chain ordering algorithm.

1. Find the middle super-critical node in the inner band, say $(x-1, y)$.
2. Using $m/\lg m$ processors and in $O(\lg m)$ time find a shortest path forward from $(x-1, y)$ to the front of the outer band. Say this shortest path forward from the super-critical node $(x-1, y)$ has an angular edge between the two bands that terminates in row r.
3. Split the jumpers into two sets,
   (a) Those smaller than or equal to $(r, x) \Longrightarrow (r, y)$ call them S, they are nested *inside* $(r, x) \Longrightarrow (r, y)$.
   (b) Those larger than or equal to $(r, x) \Longrightarrow (r, y)$ call them L, they are nested *around* $(r, x) \Longrightarrow (r, y)$.
4. Do the following two steps in parallel:
   (a) Assign $|S|$ processors to rows r up through 1 and recursively repeat this algorithm with the jumpers in S.
   (b) Assign $|L|$ processors to rows r down through m and recursively repeat this procedure with the jumpers in L.

Figure 39: An $O(\lg^2 n)$ Time and $n/\lg n$ Processor Band Merging Algorithm

The next corollary shows that the algorithm given here can be improved by using efficient algorithms for finding the row minima in totally monotone matrices. This row minima problem is classical and has been shown to be at the root of many important problems, see for example (Aggarwal et al., 1987; Aggarwal and Park, 1988).

Corollary 7 Solving the row minima problem on totally monotone matrices allows us to merge two bands.

Proof: Given two nested bands to merge, for ease of exposition take only the horizontal straight unit paths of the outer band. Let each of these straight unit paths denote the row of a matrix M. Each column of M represents the jumpers that get their *sp* values from the super-critical nodes of the inner band. The first column represents the effect of the most inner jumper, the second column represents the effect of the immediate jumper containing it, etc. Similarly, several sets of independent jumpers gives several totally monotone matrices.

By Theorem 32, M is a monotone matrix. But, any submatrix of M represents neighboring straight unit paths in the rows and neighboring jumpers along the columns. Since Theorems 27 and 29 still hold it must be that such a submatrix is also monotone since we can again apply Theorem 32. □

Therefore, our algorithm is one of the many known to depend on the row minima problem on a totally monotone matrix. Hence, by the results in (Aggarwal and Park, 1988) and in (Atallah, 1991) our algorithm runs in $O((\lg^2 n)(\lg\lg n))$ time using $n/\lg\lg n$ processors on a common-CRCW PRAM or in $O(\lg^2 n)$ time using n processors on a EREW PRAM. For the EREW PRAM algorithm note that everything from pointer jumping to tree contraction the time complexity stays the same asymptotically.

An asymptotically optimal log-time row minima algorithm for totally monotone matrices would make the work of our MCOP algorithm the same as the work of Hu and Shing's $O(n \lg n)$ sequential algorithm. Hu and Shing's algorithm has the best know work for solving the MCOP to date. In this regard, (Ramanan, 1991; Ramanan, 1994) shows problems closely related to the MCOP have a $\Omega(n \lg n)$ lower bound. Further, (Bradford et al., 1993b; Bradford et al., 1995) give several lower bounds for the MCOP on different models of computation including a simple $\Omega(n \lg n)$ lower bound on the comparison based model with a constrained version of the MCOP.

Recently, (Raman and Vishkin, 1994) gave a Las Vegas (randomized) parallel algorithm for solving the row minima problem on a totally monotone matrix that "almost always" takes $O(\lg n)$ time using $n/\lg n$ processors. Therefore, we can solve the MCOP with high probability in $O(\lg^2 n)$ time and using $n/\lg n$ processors on an EREW PRAM.

8.6 Historical Notes

The algorithms given in this chapter were motivated by algorithms given in (Bradford, 1992) and (Bradford, 1992a; Bradford, 1992b). The algorithms given here improve the algorithms given by many people. In particular, both the extended abstract in (Ramanan, 1992) and the later full version (Ramanan, 1993) give an $O(\lg^4 n)$ time and n processor CREW PRAM MCOP algorithm building directly on the work of Hu and Shing. On the other hand, the full version (Bradford et al., 1992) (versions circulated since the late summer of 1992) gives an $O(\lg^4 n)$ time and $n/\lg n$ processor common-CRCW PRAM MCOP algorithm using the model given in this dissertation and originally in (Bradford, 1992).

Of course, the algorithms outlined in this chapter can run in $O(\lg^2 n)$ time using n processors or in in $O(\lg^3 n)$ time using $n/\lg n$ processors on an EREW PRAM. An extended abstract of the algorithm given in this section is in (Bradford et al., 1994) and the full version has been submitted.

Chapter 9

Directions of Further Research and Conclusions

This chapter contains conclusions and directions of further research. Some extensions to the work in this dissertation have already come to fruition, see (Bradford et al., 1993b; Bradford et al., 1995). In general, the design and analysis of parallel algorithms has lots of open problems that make good research topics. Good parallel algorithms for dynamic programming problems are almost always of interest too.

9.1 Future Directions

Solving dynamic programming problems in parallel is an area with great potential. This is because dynamic programming is often used to solve optimization problems.

It is not unusual for dynamic programming problems to be solved in parallel using dynamic graph algorithms. Dynamic graph algorithms are coming along on their own. The work presented in this dissertation represents strides in this area, hopefully there will be many more.

9.1.1 Optimal Binary Search Trees

The development of algorithms for building optimal binary search trees in polylog parallel time with low processor complexity is cited as an open problem in (Atallah et al., 1989) and (Larmore, Przytycka, and Rytter, 1993). By low processor complexity we mean n^k for some constant $k < 6$, The best sequential algorithm for building optimal binary search trees takes $\Theta(n^2)$ time (Knuth, 1973).

There are efficient parallel algorithms for building Huffman trees, alphabetic trees and approximate binary search trees, see (Teng, 1987; Atallah et al., 1989; Larmore and Przytycka, 1992; Larmore, Przytycka, and Rytter, 1993). But, to date, there have been no efficient parallel algorithms for the construction of optimal binary search trees.

The optimal binary search tree problem (OBST) is: Given n search keys K_i, for $1 \leq i \leq n$ each with an associated probability p_i of being searched for all $i, 1 \leq i \leq n$. Let d_i be the depth of node i in a tree. The problem of building an optimal binary search tree is to build a tree so that each K_i is in a node and the tree minimizes the cost function $\sum_{i=1}^{n} d_i p_i$. This is a classic problem, see for example (Aho et al., 1974; Cormen et al., 1990).

The tree cost function $\sum_{i=1}^{n} d_i p_i$ ensures that we are building a binary search tree that is optimal "on average" given the access probabilities. This is given as a dynamic programming problem in Chapter 1 and a $O(\lg^2 n)$ time and $n^6/\lg n$ processor solution is given in Chapter 4.

9.1.2 Previous Results

In (Atallah et al., 1989), among other things, the authors give a *nearly* optimal algorithm for building optimal *alphabetic* binary search trees in $O(\lg^2 n)$ time and using $n^2/\lg^2 n$ processors.

Alphabetic trees are binary search trees that have their search keys only in the leaves. There are several interesting results for the building of alphabetic trees, take for example (Kirkpatrick and Przytycka, 1990; Larmore and Przytycka, 1992), and (Larmore, Przytycka, and Rytter, 1993), and for alphabetic trees the best of which is (Larmore, Przytycka, and Rytter, 1993) giving a polylog time and $n^2/\lg n$ processor algorithm. Although, the alphabetic tree construction problem can be solved sequentially in $O(n \lg n)$.

Many of the parallel and sequential algorithms for solving problems amenable to dynamic programming solutions depend on monotonicity properties. Early examples of monotonicity properties used to improve sequential dynamic programming algorithms can be found in (Knuth, 1971) and (Yao, 1982). Some of the techniques of (Yao, 1982) have been made parallel by (Czumaj, 1993) for attacking the MCOP.

9.1.3 Some Comments on Solving the OBST on a D_n Graph

As we saw in Chapter 4, a D_n graph suited for solving the problem of building an optimal binary search tree can be constructed in $O(\lg^2 n)$ time using $n^6/\lg n$ processors.

The central research goal is to find a a divide-and-conquer tool with which we could break up the appropriate D_n graph into a tree structure. In the case of the efficient matrix chain ordering algorithm in this dissertation, the ANSV problem of (Berkman et al., 1989; Berkman et al., 1993) in conjunction with some theorems by (Hu and Shing, 1982; Hu and Shing, 1984) is used as such a tool. Unfortunately, the MCOP is quite different in character from the OBST, so it does not appear that the ANSV problem can be used in the same manner to solve the OBST efficiently in parallel. But, other possibilities exist, for example in the parallel algorithm of (Larmore, Przytycka, and Rytter, 1993) for building optimal alphabetic trees both monotonicity and tree contraction techniques are used.

A D_n graph adapted for solving the the OBST exhibits several interesting properties. It turns out that there are some definite relationships between the subgraphs that might be useful for a tree decomposition divide-and-conquer tool. In particular, Knuth's principle (Knuth, 1971) or something close to it, may give monotonicity conditions that lead to a good and cheap tree decomposition allowing efficient construction of OBSTs in parallel.

9.2 Conclusions

In terms of algorithm design paradigms, Chapter 6 gives a greedy approximation algorithm. It is greedy in the sense that it isolates certain locally optimal matrix products. Next in Chapter 7 the principle of optimality comes into play since the canonical graphs are well-formed substructures that may be in an optimal superstructure. In addition, the tree contraction algorithm gives an optimal matrix chain ordering product by combining such well-formed optimal substructures. The same holds for Chapter 8. Therefore, the dynamic programming algorithm design paradigm is a viable design paradigm for parallel computation. Also, there are lots of new and interesting problems to consider regarding these areas of research.

Bibliography

A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilbur: "Geometric Applications of a Matrix Searching Algorithm," *Algorithmica*, Vol. 2, 195–208, 1987.

A. Aggarwal and J. K. Park: "Notes on Searching Multidimensional Monotone Arrays," *Proceedings of the 29th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, IEEE Press, 497–512, 1988. Full versions to appear in the *Journal of Algorithms*.

A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1975.

R. Anderson and E. W. Mayr: "Parallelism and the Minimal Path Problem," *Information Processing Letters,* Vol. 24, 121–126, 1987.

R. Anderson and E. W. Mayr: "Parallelism and Greedy Algorithms," *Advances in Computing Research*, Vol. 4, 17–38, JAI Press, 1987.

A. Apostolico, M. J. Atallah, L. L. Larmore, and S. H. McFaddin: "Efficient Parallel Algorithms for String Editing and Related Problems," *SIAM Journal on Computing*, Vol. 19, No. 5, 968–988, Oct. 1990.

T. Archibald: "Parallel Dynamic Programming," in *Advances in Parallel Algorithms*, L. Kronsjö and D. Shumsheruddin (Editors), John Wiley and Sons, 343–367, 1992.

M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S.-H. Teng: "Constructing Trees in Parallel," *Proc. 1st Symp. on Parallel Algorithms and Architectures (SPAA)*, ACM Press, 499–513, 1989.

M. J. Atallah and S. R. Kosaraju: "An Efficient Parallel Algorithm for the Row Minima of a Totally Monotone Matrix," *Proceedings of the 1st Symposium on Discerete Algorithms (SODA)*, 394–403, ACM Press, 1991.

M. J. Atallah and S. R. Kosaraju: "An Efficient Parallel Algorithm for the Row Minima of a Totally Monotone Matrix," *Journal of Algorithms*, Vol. 13, 394–413, 1992.

S. Baase: *Computer Algorithms*, Second Edition, Addison-Wesley, 1988.

J. L. Balcázar, J. Díaz, and J. Gabarró: *Structural Complexity I and II*, Vols. 11 and 22 of *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, New York, 1988 and 1990.

R. Bellman: *Dynamic Programming*, Princeton University Press, 1957.

O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin: "Highly Parallelizable Problems," *Symposium on the Theory of Computing (STOC)* , ACM Press, 309–319, 1989.

O. Berkman, B. Schieber, and U. Vishkin: "Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values," *Journal of Algorithms*, Vol. 14, 344–370, 1993.

P. G. Bradford: "Efficient Parallel Dynamic Programming," Technical Report # 352, Indiana University, April 1992.

P. G. Bradford: "Efficient Parallel Dynamic Programming," *The* 30*th Allerton Conference on Communication, Control, and Computation*, University of Illinois, 185–194, 1992.

P. G. Bradford: "Efficient Parallel Dynamic Programming," Full Version Manuscript May, 1992: Version with very minor revisions in Revised TR # 352, Indiana University.

P. G. Bradford, G. J. E. Rawlins, and G. E. Shannon: "Matrix Chain Ordering in Polylog Time with $n/\lg n$ Processors," Technical Report # 360, Indiana University, December 1992.

P. G. Bradford, G. J. E. Rawlins, and G. E. Shannon: "Matrix Chain Ordering in Polylog Time with Linear Processors (Extended Abstract)," *Proceedings of the* 8*th Annual IEEE International Parallel Processing Symposium (IPPS)*, Cancun Mexico, H. J. Siegel editor, IEEE Press, 234–241, April 1994.

P. G. Bradford, G. J. E. Rawlins, and G. E. Shannon: "Matrix Chain Ordering in Polylog Time with Linear Processors," Full Version Submitted to *SIAM Journal on Computing*.

P. G. Bradford, V. Choppella, and G. J. E. Rawlins: "On Lower Bounds for the Matrix Chain Ordering Problem," Full version to be Submitted, Earlier version: Technical Report # 391, Indiana University, October 1993.

P. G. Bradford, V. Choppella, and G. J. E. Rawlins: "On Lower Bounds for the Matrix Chain Ordering Problem," Extended Abstract to appear in the proceedings of LATIN '95, Springer Verlag, 1995.

D. Z. Chen: "Efficient Geometric Algorithms on the EREW PRAM," *Proceedings of the 28th Allerton Conference on Communication, Control, and Computation*, Monticello, Illinois, 818–827, 1990. Full version accepted to *IEEE Trans. on Parallel and Distributed Systems.*

F. Y. Chin: "An $O(n)$ Algorithm for Determining Near-Optimal Computation Order of Matrix Chain Products," *Communications of the ACM*, Vol. 21, No. 7, 544–549, July 1978.

A. K. Chandra: "Computing Matrix Chain Products in Near Optimal Time," IBM Research Report RC-5625, IBM T. J. Watson Research Center, Oct. 1975.

A. K. Chandra and L. J. Stockmeyer: "Alternation," *Proceedins of the 17th Anual IEEE Symposium on the Foundations of Computing (FOCS)*, IEEE Press, 98–108, 1976.

A. K. Chandra, D. Kozen, and L. J. Stockmeyer: "Alternation," *Journal of the ACM*, Vol. 28, 114–133, 1981.

S. A. Cook: "An Observation on Time-Storage Trade Off," *Journal of Computer and System Sciences*, Vol. 9, No. 3, 308–316, 1974.

S. A. Cook: "A Taxonomy of Problems with Fast Parallel Algorithms," *Information and Control*, Vol. 64, 2–22, 1985.

D. Coppersmith and S. Winograd: "Matrix Multiplication via Arithmetic Progressions," *Journal of Symbolic Computation*, Vol. 9, 251–280, 1990.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest: *Introduction to Algorithms*, McGraw Hill, 1990.

A. Czumaj: "An Optimal Parallel Algorithm for Computing a Near-Optimal Order of Matrix Multiplications," *Scandinavian Workshop on Algorithms Theory (SWAT)*, Springer Verlag, LNCS # 621 , 62–72, 1992.

A. Czumaj: "Parallel Algorithm for the Matrix Chain Product and the Optimal Triangulation Problem (Extended Abstract)," *Symposium on Theoretical Aspects of Computer Science (STACS)*, Springer Verlag, LNCS # 665 , 294–305, 1993. Full version submitted.

L. E. Deimel Jr. and T. A. Lampe: "An Invariance Theorem Concerning Optimal Computation of Matrix Chain Products," North Carolina State Univ. Tech Report # TR79-14, 1979.

S. Fortune and J. Wyllie: "Parallelism in Random Access Machines," *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, IEEE Press, 114–118, 1978.

Z. Galil and K. Park: "Dynamic Programming with Convexity, Concavity and Sparsity," *Theoretical Computer Science*, Vol. 92, 49–76, 1992.

Z. Galil and K. Park: "Parallel Dynamic Programming," Manuscript, 1992.

M. R. Garey and D. S. Johnson: *Computers and Intractability*, W. H. Freeman, 1979.

A. Gibbons and W. Rytter: *Efficient Parallel Algorithms*, Cambridge University Press, 1988.

S. S. Godbole: "An Efficient Computation of Matrix Chain Products," *IEEE Transactions on Computers*, Vol. C-22, 864–866, 1973.

R. Greenlaw: "Polynomial Completeness and Parallel Computation," 901-953, in Synthesis of Parallel Algorithms, J. H. Reif (Editor), Morgan-Kaufmann, 1993.

R. Greenlaw, H. J. Hoover, and W. L. Ruzzo: *A Compendium of Problems Complete for P*, Revision 1.46, Technical Report from Universities of Alberta (TR 91-11), New Hampshire (TR91-14) and Washington (TR 91-05-01), 1991.

J. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

S.-H. S. Huang, H. Liu, and V. Viswanathan: "Parallel Dynamic Programming," *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, 497–500, 1990.

S.-H. S. Huang, H. Liu, and V. Viswanathan: "A Sublinear Parallel Algorithm for Some Dynamic Programming Problems," *Theoretical Computer Science*, Vol. 106, 361–371, 1992.

T. C. Hu: *Combinatorial Algorithms*, Addison-Wesley, 1982.

T. C. Hu and M. T. Shing: "Some Theorems about Matrix Multiplication", *Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, IEEE Press, 28–35, 1980.

T. C. Hu and M. T. Shing: "An $O(n)$ Algorithm to Find a Near-Optimum Partition of a Convex Polygon," *Journal of Algorithms*, Vol. 2, 122–138, 1981.

T. C. Hu and M. T. Shing: "Computation of Matrix Product Chains. Part I," *SIAM Journal on Computing*, Vol. 11, No. 3, 362–373, 1982.

T. C. Hu and M. T. Shing: "Computation of Matrix Product Chains. Part II," *SIAM Journal on Computing*, Vol. 13, No. 2, 228–251, 1984.

O. H. Ibarra, T.-C. Pong, and S. M. Sohn: "Hypercube Algorithms for Some String Comparison Problems," Proceedings of the *IEEE International Conference on Parallel Processing*, IEEE Press, 190–193, 1988.

J. JáJá: *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

D. S. Johnson: "A Catalog of Complexity Classes," Chapter 2 in *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, V. Van Leeuwen (Editor), Elsevier, 67–161, 1990.

R. M. Karp and V. Ramachandran: "Parallel Algorithms for Shared Memory Machines," Chapter 17 in *Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity* V. Van Leeuwen (Editor), Elsevier, 869–942, 1990.

S. K. Kim: "Optimal Parallel Algorithms on Sorted Intervals," TR 90-01-04, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1990.

D. G. Kirkpatrick and T. Przytycka: "Parallel Construction of Near Optimal Binary Search Trees," *Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures (SPAA)*, 234–243, 1990.

P. N. Klein and J. H. Reif: "Parallel Time $O(\lg n)$ Acceptance of Deterministic CFLs on an Exclusive-Write P-RAM," *SIAM Journal on Computing*, Vol. 17, 463–485, 1988.

D. E. Knuth: "Optimum Binary Search Trees," Acta Informatica, Vol. 5, 14–25, 1971.

D. E. Knuth: *The Art of Computer Programing, Volume 3: Searching and Sorting*, Addison-Wesley, 1971.

B. Korte, L. Loväsz, and R. Schrader: *Greedoids*, Springer-Verlag, 1991.

D. Kozen: "On Parallelism in Turing Machines," *Proceedins of the 17th IEEE Symposium on the Foundations of Computing (FOCS)*, IEEE Press, 89–97, 1976.

V. Kumar, A. Grama, A. Gupta, and G. Karypis: *Introduction to Parallel Computing*, Benjamin/Cummings, 1994.

L. L. Larmore and W. Rytter: "Efficient Sublinear Time Parallel Algorithms for the Recognition of Context-Free Languages," *Scandinavian Workshop on Algorithms Theory, (SWAT)*, Springer Verlag, LNCS #577, 121–132, 1991.

L. L. Larmore and T. M. Przytycka: "Constructing Huffman Trees in Parallel (Extended Abstract)," *Proceedings of the 3rd Symposium on Parallel Algorithms and Architectures (SPAA)*, 71–80, 1991.

L. L. Larmore, T. M. Przytycka, and W. Rytter: "Parallel Construction of Optimal Alphabetic Trees (Extended Abstract)," *Proceedings of the 5th Symposium on Parallel Algorithms and Architectures (SPAA)*, 214–223, 1993.

M. Marcus: *Introduction to Modern Algebra*, Marcel Dekker, 1978.

Y. Muraoka and D. J. Kuck: "On the Time Required for a Sequence of Matrix Products," *Communications of the ACM*, Vol. 16, No. 1, 22–26, 1973.

J. O'Rourke: *Computational Geometry in C*, Cambridge University Press, 1994.

V. Pan: *How to Multiply Matrices Faster*, Lecture Notes in Computer Science, # 179, Springer Verlag, 1984.

C. Papadimitriou: *Computational Complexity*, Addison-Wesley, 1994.

I. Parberry: *Parallel Complexity Theory*, Research Notes in Theoretical Computer Science, Pitman Publishers, London, 1987.

P. W. Purdom Jr. and C. A. Brown: *The Analysis of Algorithms*, Holt, Rinehart and Winston, 1985.

R. Raman and U. Vishkin: "Optimal Randomized Parallel Algorithms for Computing the Row Minima of a Totally Monotone Matrix," *Proceedings of the Fifth Symposium on Discerete Algorithms (SODA)*, ACM Press, 613–621, 1994.

P. Ramanan: "A New Lower Bound Technique and its Application: Tight Lower Bounds for a Polygon Triangularization Problem," *Proceedings of the Second Annual Symposium on Discrete Algorithms, (SODA)*, ACM Press, 281–290, 1991.

P. Ramanan: "An Efficient Parallel Algorithm for Finding an Optimal Order of Computing a Matrix Chain Product," Technical Report, WSUCS-92-2, Wichita State University, June, 1992.

P. Ramanan: "An Efficient Parallel Algorithm for the Matrix Chain Product Problem," Technical Report, WSUCS-93-1, Wichita State University, January, 1993. Submitted.

P. Ramanan: "A New Lower Bound Technique and its Application: Tight Lower Bounds for a Polygon Triangularization Problem," *SIAM Journal on Computing*, Vol. 23, No. 4, 834–851, 1994.

G. J. E. Rawlins: *Compared To What ?*, Computer Science Press/W. H. Freeman, 1992.

J. H. Reif: "Depth First Search is Inherently Sequential," *Information Processing Letters*, Vol. 20, No. 5, 229–234, 1985.

J. H. Reif (Editor): *Synthesis of Parallel Algorithms*, Morgan-Kaufmann, 1993.

W. Rytter: "On Efficient Parallel Computation for Some Dynamic Programming Problems," *Theoretical Computer Science*, Vol. 59, 297–307, 1988.

S-H. Teng: "The Construction of Huffman-Equivalent Prefix Code in $\mathcal{NC}$," SIGACT News, 18(4), 54–61, 1987.

L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff: "Fast Parallel Computation of Polynomials Using Few Processors," *SIAM Journal on Computing*, Vol. 12, No. 4, 641–644, Nov. 1983.

F. F. Yao: "Efficient Dynamic Programming Using Quadrangle Inequalities," *Symposium on the Theory on Computing (STOC)*, ACM Press, 429–435, 1980.

F. F. Yao: "Speed-Up in Dynamic Programming," *SIAM Journal on Algebraic and Discrete Methods*, Vol. 3, No. 4, 532–540, 1982.