Efficient Parallel Algorithms for Graph Problems.

Article *in* Algorithmica · June 1990

DOI: 10.1007/BF01840376 · Source: DBLP

Clyde Kruskal
University of Maryland
117 PUBLICATIONS **3,570** CITATIONS
SEE PROFILE

Larry Rudolph
Two Sigma Investments
203 PUBLICATIONS **7,138** CITATIONS
SEE PROFILE

Parallel Algorithms for Graph Problems

Panagiotis Metaxas

Department of Mathematics
and Computer Science
Dartmouth College

Abstract of the Thesis

In this thesis we examine three problems in graph theory and propose efficient parallel algorithms for solving them. We also introduce a number of parallel algorithmic techniques.

Computing the *connected components* of an undirected graph $G = (V, E)$ is the first problem we examine. We propose an efficient algorithm which runs in $O(\log^{3/2}|V|)$ parallel time using $|V| + |E|$ CREW PRAM processors. This result settles a question that remained unresolved for many years: a connectivity algorithm for this model with running time $o(\log^2|V|)$ was a challenge that had thus far eluded researchers.

The second problem examined is the *vertex updating for a minimum spanning tree* of a graph $G = (V, E)$. We give optimal parallel algorithms (as well as linear-time sequential ones) which run in $O(\log|V|)$ time using $\frac{|V|}{\log|V|}$ EREW PRAM processors. This result is then used in the solution of the *multiple updates of a minimum spanning tree* problem. We present an algorithm for this problem which runs in $O(\log k \cdot \log|V|)$ parallel time using $\frac{k \cdot |V|}{\log k \cdot \log|V|}$ EREW PRAM processors, where k is the number of updates.

In the process of solving these problems, we introduce a number of parallel algorithmic techniques. In particular, we give an algorithm for the *pseudotree contraction* problem having running time $O(\log h)$, where h is the length of the longest simple path in the pseudotree. Also, we propose the edge-plugging scheme which solves the *edge-list augmentation* problem in constant time without concurrent writing.

Finally, we introduce the *growth-control scheduling* technique. This technique balances carefully the work-performed versus progress-made ratio of an algorithm giving a better running time.

Acknowledgements

Several years ago, Professors Dimitris Papoulias and Haralambos Papageorgiou persuaded a doubtful and hesitant student of the Athens University to forgo a life of ease and pleasure and instead to venture onto the icy path of graduate studies. Perhaps I am acquiring by osmosis Puritan New England blood, but I have discovered on this path the greatest rewards of my life and have met here a host of remarkable mentors, friends, and colleagues.

My deepest gratitude goes to my brother Atha and sister Mina Salpoglou, whose initial support and encouragement helped me survive the early stages of my studies, far from home, without the comforts of ouzo, winter warmth, old friends, and the gladdening chaos of Greece.

Of all my New World friends, professors, and colleagues who helped me, I would like to single out my advisor, Donald B. Johnson, for setting consistently high standards for my work, for his patience during this long process, his faith in my abilities, and for his subtle brand of humor. I would also like to thank the members of the thesis committee: Sam Bent, Jim Driscoll, and Elias Manolakos for their help and advice throughout the thesis preparation. Special thanks go to Jeff Shallit for his eagerness to listen to my little problems and to provide help at any time, and to Sam Bent for passing on to me his love for algorithms.

Regarding the thesis itself, I want to thank Paul Pritchard, Larry Raab, and Don Nielsen for their careful reading and their deft and diplomatic suggestions. Don in particular, despite not having the faintest idea of what my work was about, was a skillful gardener to the tangled mess of words which grew between the mathematical symbols like weeds.

During my education, I was fortunate to have great teachers who I would also like to thank. To those I have already mentioned, I should add Matt Bishop, Scot Drysdale, Paris Kanellakis, Jeff Vitter, Cliff Walinsky and Stan Zdonik.

Next, I have been fortunate in having fallen under the influence of a most extraordinary faculty, institution, and group of colleagues. Where I had been told by friends at other universities to expect professional jealousy, territorial intrigues, and neglect, I found only generosity and support. The professional and human standards of my professors, and the support and kindness of my fellow graduate students are difficult to adequately acknowledge.

And finally, for all the uniqueness of this fine College, the beauty of Vermont and New Hampshire, and the special friends I have made here, my life would have been poorer, and this thesis perhaps would have been abandoned for its difficulty – if it were not for the comfort, friendship, and inspiration of my new wife, Stella Kakavouli.

Contents

List of Figures

From the discussion of Theseus and Minotaur in the Labyrinth:

> "... Let's concentrate on the maze problem. Two nodes can be joined by a continuous path if and only if they lie in the same connected component of the graph."
>
> "Hmm," Theseus murmured doubtfully. "What is a connected component?"
>
> "It is the set of all nodes that can be reached from a given one by a continuous path," the Minotaur recited proudly.
>
> "Ah! Let me see if I've got this right. What you're saying is that two nodes can be joined by a continuous path if and only if there exists a continuous path that joins them?"
>
> The Minotaur was deeply offended. "Well, you could put it like that, but it seems to me you're trivializing an important concept."

The true story of how Theseus found his way out of the Labyrinth by Ian Stewart, SCIENTIFIC AMERICAN, February 1991, page 137.

Chapter 1

Introduction

1.1 Why Parallel Computation?

Almost all computation done during the first forty years of the history of computers could be called *sequential.* One of the characteristics of sequential computation is that it employs one processor to solve some problem.

However, due to the limit the speed of light imposes on us, it seems extremely unlikely that we can build uni-processor computers that can achieve performance significantly higher than 1 GFLOPS (Giga FLoating-point Operations Per Second)[Den86]. If you consider the always-increasing human appetite for computational power (i.e. for solving larger problems faster), a need for an alternative route to sequential computation becomes apparent. Parallel computation seems the most promising (if not the only) alternative.

Parallel computation is defined as the practice of employing a large number of cooperating processors, communicating among themselves to solve large problems fast, and it is quickly becoming an important area in computer science. It is possible that during the next ten years this area will have grown so wide and strong that most of the research conducted in the fields of design and analysis of algorithms, computer languages, computer applications and computer architectures will be within the context of parallel computation.

New parallel machines with novel architectures are being built every year. The demand for laying the theoretical foundations and, more generally, understanding the nature of parallel computation becomes continuously greater. The issues that arise in this process are many. To mention only a few of them:

- What computational models can provide both realizable architectures and a basis for abstraction simple enough for understanding parallel computation?
- What sort of problems are naturally suited for large scale parallelization?
- What are the basic techniques for designing parallel algorithms?

While no definite answer to these problems exists today, the efforts of the research community have provided a better understanding of parallel computation.

In particular, several models have been introduced; a large number of problems have been identified as amenable to parallel computation, and new techniques have been developed to help the design of parallel algorithms.

1.2 Contents and Organization of the Thesis

We will introduce some new parallel algorithmic techniques and discuss where they fall in the existing framework. Then we will show how they can help design algorithms for some combinatorial optimization problems.

This thesis is organized as follows:

In the next chapter we review the existing models of parallel computation. We begin with a description of the PRAM model, the parallel model most examined today, which we describe along with some of its variations. We also briefly describe some other related parallel models, namely the XRAM and the PPM models. Finally, we introduce the notion of efficient PRAM algorithms and define NC, the class of problems having fast parallel solutions.

In Chapter 3 we discuss some of the existing parallel algorithmic techniques, namely prefix sum, list ranking and tree contraction. Then we introduce some new ones: the pseudotree contraction and the edge-plugging scheme. These techniques will be essential in the development of the graph algorithms in the following chapter.

In Chapter 4 we discuss some parallel algorithms. We begin by reviewing the existing algorithms for computing the connected components of a graph. Then we introduce the growth-control schedule which we apply to create a connectivity algorithm for the CREW PRAM model. This algorithm works in $O(\lg^{3/2}|V|)$ parallel time using $|V|+|E|$ processors. An algorithm for updating a minimum spanning tree follows, and this result is used to derive an algorithm for the multiple parallel updates of a minimum spanning tree.

In the final chapter we give the conclusions and discuss some of the open problems arising from our work.

Chapter 2

Models of Parallel Computation

2.1 The Existing Parallel Models

Several models for parallel computation have been developed in the last fifteen years. One can distinguish three major categories into which these models fall:

In the first category we have the VLSI models, which are closest to today's technological limits. Systolic arrays are examples of these models.

The second category contains models that employ a large number of processors communicating using a fixed interconnection network. Hypercube, cycle-connected-cube, shuffle-exchange and butterfly networks fall in this category.

In the final and most abstract category are models that do not restrict the way processors communicate. The best known model in this category is the *parallel random access machine* (PRAM) model. PRAM is an idealized parallel machine which was developed as a straightforward generalization of the sequential RAM [CR73]. It first appeared in Fortune and Wyllie's work [FW78] and has been used extensively ever since. Because this is the model that we will be using, we will give a detailed description of it.

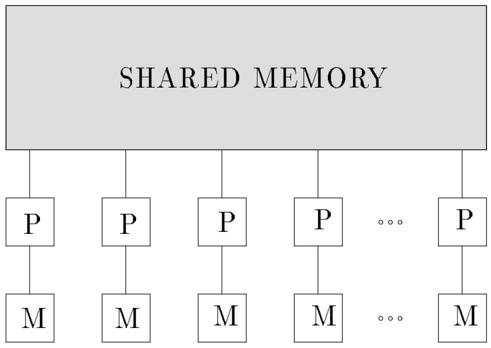

Figure 2.1: The PRAM model.

2.2 The PRAM Model

2.2.1 Description

A PRAM [KR90] uses p identical processors, each one able to perform the usual computation of a sequential RAM using a finite amount of local memory. The processors communicate through some shared global memory (Figure 2.1) to which all are connected. There is a global clock, and in one time-unit period each processor can perform the following three steps:

1. Read from one memory location, global or local;
2. Execute a single RAM operation, and
3. Write to one global or local memory location.

If processor p_i wants to communicate with processor p_j, it can do so by writing to some memory location from which p_j will read. So, the model makes the assumption that communication between processors takes unit time.

2.2.2 Handling Concurrent Accesses

The fact that the communication between the processors in the PRAM is implemented via accesses to the shared memory makes access conflicts possible. These conflicts may be caused by concurrent accesses to a memory cell by many processors for either reading or writing.

Depending on the way the access of the processors to the global memory is handled, PRAMs are classified as EREW, CREW and CRCW.

EREW In the exclusive-read-exclusive-write (EREW) PRAM model, no conflicts are permitted for either reading or writing. If, during the execution of a program on this model, some conflict occurs, the program's behavior is undefined.

CREW In the concurrent-read-exclusive-write (CREW) PRAM model, simultaneous readings by many processors from some memory cell are permitted. However, if a writing conflict occurs, the behavior of the program is, as before, undefined.

CRCW Finally, the concurrent-read-concurrent-write (CRCW) PRAM, the strongest of these models, permits simultaneous accesses for both reading and writing. In the case of multiple processors trying to write to some memory cell, one must define which of the processors eventually does the writing. There are several answers that researchers have given to this question. The most frequent [1] of them are:

[1]Two more CRCW PRAM models appear sometimes in the literature: The Weak and the Strong CREW PRAM [EG88]. In the former, all processors trying to write to a memory location must write the value zero. No other value is permitted. In the Strong model, the value that is written in the memory cell is the largest (or equivalently the smallest) of all the values attempting to be written.

COMMON All processors attempting to write to a memory location must write the same value, so the outcome of some writing is the same, no matter which processor wins. This is called the COMMON CRCW PRAM model.

ARBITRARY Any processor may win. No restriction is placed on the values written. This is called the ARBITRARY CRCW PRAM. Note that, if the program runs a second time, the outcome of a concurrent writing may be different. However, the outcome of the computation in *every* case must be correct.

PRIORITY In this model we assume that each processor has some unique value *id* associated with it, the *id*'s being drawn from a totally ordered set. In case of a concurrent write, the processor with the lowest *id* wins. This is called the PRIORITY CRCW PRAM model.

2.2.3 Relation Between PRAM Models.

Given the plethora of PRAM models, the need for comparison and simulation among the models arises. Several papers have established that there is a strict hierarchy in the PRAM models [Eck77, Vis83, Kuc82, CDR86]. In particular, the EREW is strictly less powerful than the CREW model, which in turn is strictly less powerful than the CRCW model. The hierarchy is extended inside the CRCW model and their power follows the presentation order. [2]

Obviously, a program written for some model A can run on models stronger than A in the hierarchy without any change: The program will not use any of the extra features that the stronger model provide. Eckstein [Eck77] and Vishkin [Vis83] have

[2]In this hierarchy, Weak is the weakest of the CRCW models and Strong is the strongest of them.

shown that any algorithm written for the PRIORITY CRCW model can be simulated on an EREW model with a slowdown of $O(\lg P)$ where P is the number of processors employed by the algorithm. The slowdown depends on the complexity of sorting on a EREW PRAM [AKS83, Col88].

Another very important result is the following: Cook, Dwork and Reischuk [CDR86] proved that computing the OR of n bits requires $\Omega(\lg n)$ time on a CREW PRAM, no matter how many processors or how much memory is used. The fact that this function is computed in $O(1)$ time on a COMMON CRCW PRAM proves that concurrent write is strictly more powerful than exclusive write.

Finally, Kučera [Kuc82] showed that each algorithm for the p-processor PRIORITY CRCW PRAM can be simulated by a COMMON CRCW PRAM with no loss of parallel time provided that $O(p^2)$ processors are available.

2.3 Other Parallel Models

The PPM Model.

The *parallel pointer machine* (PPM) introduced by Goodrich and Kosaraju [GK89] is the parallel version of Knuth's *linking automaton* [Knu80, page B-295] (called also a *pointer machine*), just as the PRAM is the parallel version of RAM. In the PPM the shared memory can be viewed as a directed graph whose vertices are memory cells, each cell containing some small number of value and pointer fields.

Access of the common storage by some processor is limited to the locations pointed to by the processor's pointer registers. A processor may load a pointer register in one step and perform the usual arithmetic and comparison operations. However, pointer arithmetic is not allowed. As in the PRAM, the communication between processors is done via writing to and reading from the shared memory.

We categorize the PPMs according to how the simultaneous accesses to the shared memory is handled as EREW, CREW and CRCW PPM. Note that these models are weaker than their PRAM counterparts, because a PRAM can easily simulate a PPM but not the other way around, since a PPM does not support array indexing.

The XRAM Model.

In [Val90], Valiant proposed the XRAM model as the most realizable parallel model today. In this model, which can optimally simulate EREW and CRCW PRAMs, a usual sequential execution step costs one, but a communication-between-processors step costs g.

An XRAM executes in *supersteps.* Let's assume that within a superstep i some processor executes α_i local operations, sends β_i messages and receives γ_i messages. Then we say that the superstep lasted for

$$r_i = \frac{\alpha_i}{g} + \beta_i + \gamma_i$$

steps. Let $t = \max\{r_i | i \in < p >\}$ where $< p >$ is the set of the processors employed. The run-time of a superstep is $\lceil t/L \rceil \cdot L$ global operations or time $\lceil t/L \rceil \cdot L \cdot g$. The parameter L is usually taken to have value $\lg p$. At the end of L global operations, all the processors know whether a superstep is complete or not. Within each period, however, there is no synchronization among the processors.

2.4 The Class NC and Efficient PRAM Algorithms

In the search for fast parallel problems, a lot of attention has been given to class NC, the class of problems solvable by fast parallel algorithms. A fast algorithm is one which runs in polylogarithmic parallel time using a polynomial number of PRAM processors. To be more precise we need some definitions.

Let $polylog(n) = \cup_{k>0}(\log^k n)$ and $polynomial(n) = \cup_{l>0}(n^l)$. A problem is in class NC iff it has an algorithm whose time and processor complexity, respectively, are

$t(n) \in polylog(n)$ and

$p(n) \in polynomial(n)$.

Consider a problem P whose fastest known sequential algorithm has running time $T(n)$ for an input of size n. We say that a PRAM algorithm for P performs *work* $w(n)$, if it runs in time $t(n)$ using $p(n)$ processors, and

$$w(n) = t(n) \cdot p(n)$$

Observe that the definition implies $w(n) = \Omega(T(n))$: if this were not true, then one could design a sequential algorithm with running time less than $T(n)$ by simulating the steps executed by the parallel algorithm.

We say that a parallel algorithm is *optimal* iff

$t(n) \in polylog(n)$ and

$w(n) = O(T(n))$.

Note that the second condition along with the above observation implies that for optimal parallel algorithms $w(n) = \Theta(T(n))$. This gives the possibility of creating new sequential algorithms from optimal parallel ones.

A parallel algorithm is *efficient* iff

$t(n) \in polylog(n)$ and

$w(n) \in T(n) \cdot polylog(n)$.

The idea behind this definition is that efficient algorithms achieve a high degree of parallelism while the work they perform is within a polylogarithmic factor of optimal speed-up.

For practical purposes, an optimal or efficient algorithm is much more useful than one that simply is in NC. However, membership in NC reveals a problem with inherent parallelism and has some theoretical interest as a problem with a promising parallel solution.

Of course, we cannot expect to find NC algorithms for $NP-complete$ problems, i.e. problems for which no polynomial-time sequential algorithm is known. A question that naturally arises is, can we find NC algorithms for all the problems in P (problems solvable by polynomial-time sequential algorithms)? This is the well known $P \overset{?}{=} NC$ question, and it is widely believed that it has a negative answer [Gol77].

One of the major goals in designing parallel algorithms is to minimize $t(n)$ as well as $w(n)$. It has been demonstrated that it is easier to design algorithms for the more powerful CRCW PRAM model than for the CREW or EREW PRAM models [KR90]. However, due to the existing simulations between the PRAM models, the notion of efficient parallel algorithms is *robust*, in the sense that any efficient algorithm for some model is still efficient in any weaker model. On the other hand, the notion of optimal algorithms is not robust in this sense.

Chapter 3

Parallel Algorithmic Techniques

3.1 Parallel vs Sequential Complexity

Probably the easiest approach to designing a parallel algorithm is to modify and parallelize an existing sequential one. If this approach were generally successful, the task of designing parallel algorithms would be much easier. Unfortunately, it seems that very few sequential algorithms are modifiable and even fewer have obvious or simple parallel modifications. Moreover, problems that have simple sequential solutions do not necessarily have a practical parallel solution, often do not have an efficient parallel solution at all!

The most interesting example of the latter case is depth-first search [Tar72], a technique which has applications in almost every area of computer science. It has been used as a basis for many problem solutions and, as is well known, it has a simple linear sequential implementation. However, despite considerable research efforts, an efficient parallel implementation for this technique has not yet been found. Aggarwal, Anderson and Kao [AAK89] have given the fastest (deterministic) parallel implementation which runs in $O(\lg^{11} n \cdot \sqrt{n})$ time using n^3 processors. Even the best randomized algorithm runs in $O(\lg^7 n)$ parallel time using almost n^4 processors. [1]

[1]The exact number of processors is $n \cdot MM(n)$, where $MM(n)$ is the sequential time required to

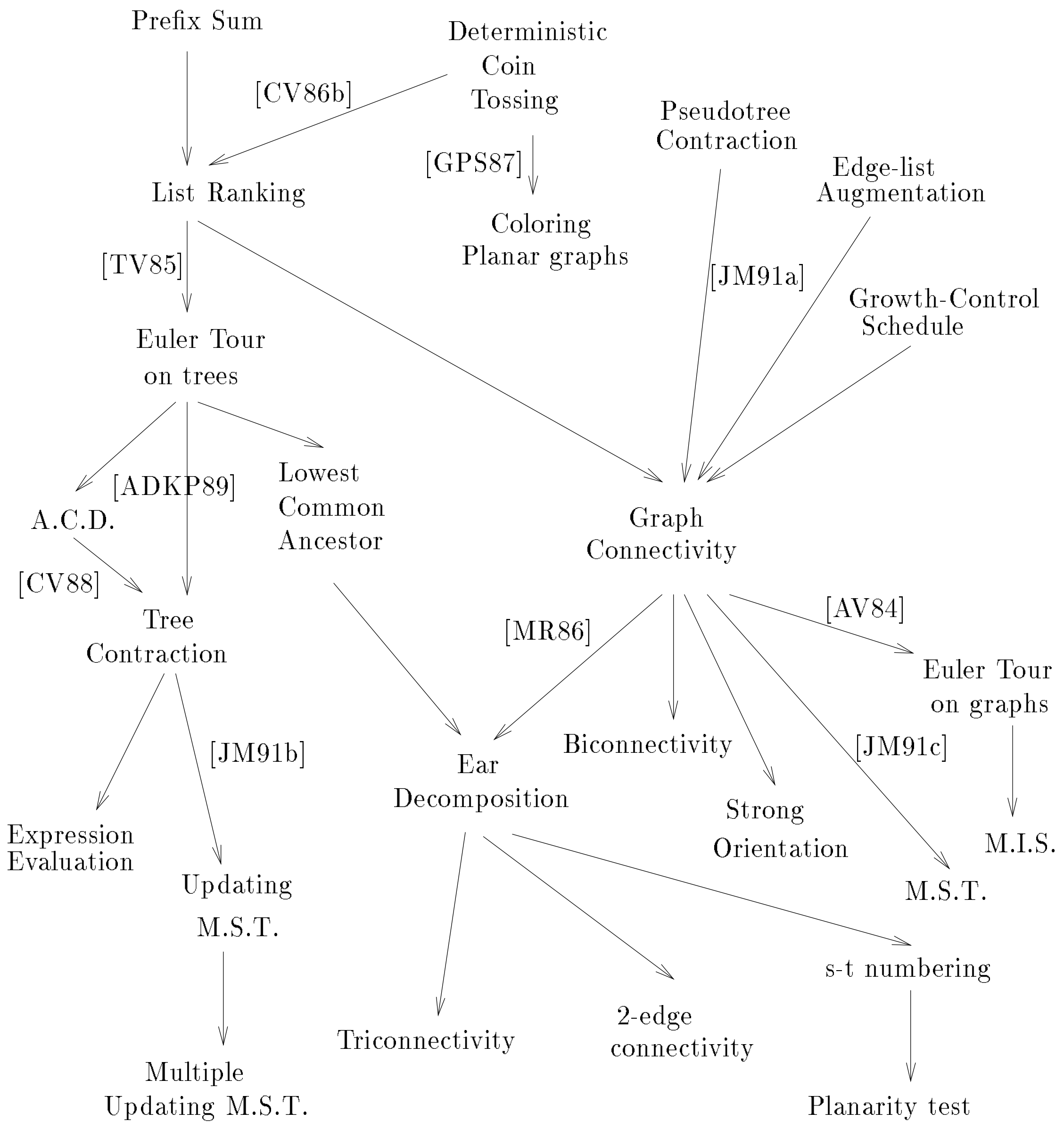

Figure 3.1: Part of the Parallel Graph Complexity. A solution to the problem appearing in the beginning of an arrow was used as a subroutine, crucial to a solution of the problem at the end of the arrow.

What is worse, there is some evidence that an efficient parallel algorithm may not even exist [Rei85].

In the past decade, a considerable body of PRAM algorithms has been developed for a variety of areas, including graph theory, computational geometry, pattern matching, etc. In Figure 3.1 we show a small part of the parallel complexity for graph problems.[2] As one can see, at the top of the figure are a number of techniques have been developed so far. These techniques are regarded as basic building blocks in the design of new parallel algorithms. Prefix sum, list ranking, and tree contraction are among them. In addition, our research has revealed two more useful techniques: the pseudotree contraction and the edge-plugging scheme. In the following sections of this chapter we give a brief overview of the existing techniques and then describe the new ones in some depth.

3.2 Prefix Sum

We start our discussion on parallel algorithmic techniques with the Prefix Sum, defined as follows:

Problem 1 *We are given an associative operator* $\circ$ *and* n *elements* $x_1, \ldots, x_n$ *drawn from some domain* D *placed in an array* M*. For* $i = 1, \cdots, n$ *we want to compute the* n *prefix sums*

$$x_1 \circ x_2 \circ \cdots \circ x_i$$

Let us use the $\sum_{1 \leq j \leq i} x_j$ notation to symbolize this sum without restricting the $\circ$ operation to addition. We will describe a recursive algorithm for this problem (Figure

multiply two $n \times n$ matrices in Strassen's model. Currently $MM(n) = n^{2.376}$.

[2]The paper by Vishkin [Vis91] contains more on the relations between the complexities of parallel problems.

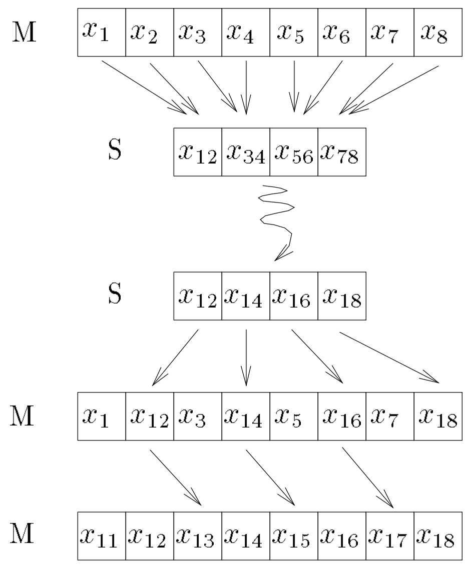

Figure 3.2: The prefix sum algorithm. Element x_{ab} denotes the sum $x_a \circ \cdots \circ x_b$.

3.2), adopted from the one given by Ladner and Fisher [LF80]. This algorithm works in $O(\lg n)$ time using n EREW PRAM processors.

Input: n elements $x_1, \ldots, x_n$ placed in memory cells $M[1..n]$, such that $M[i]$ contains element x_i.

Output: For $i = 1, \ldots, n$ each $M[i]$ contains the prefix sum $\sum_{1 \le j \le i} x_j$

Prefix Sum Algorithm:

```
Procedure Prefix_Sum(M, 1, n)
            for $i = 1$ to $n/2$ in parallel do
                let $S[i] \leftarrow M[2i-1] \circ M[2i]$
            call Prefix_Sum(S, 1, n/2)
            for $i = 1$ to $n$ in parallel do
                if $even(i)$ then $M[i] \leftarrow S[i/2]$
```

```
        **if** $odd(i)$ **then** $M[i] \leftarrow M[i-1] \circ M[i]$
```

The running time of this algorithm is given by the recursion $t(n) = t(n/2) + 4$ with $t(1) = 0$, therefore $t(n) = O(\lg n)$. Moreover, no concurrent access to memory cells occurs. We could easily implement the algorithm by using n processors, each one assigned to a different element. According to our definition of work this is not an optimal algorithm, since $w(n) = O(n \lg n)$.

However, the work actually performed is $w(n) = w(n/2) + O(n) = O(n)$. This discrepancy occurs because the naive implementation wastes a lot of processors. There actually exists a better implementation, based on Brent's *scheduling principle* [Bre74], leading to an optimal algorithm. The idea is as follows:

The Scheduling Principle.

Observe that the computation is done in $t = O(\lg n)$ parallel steps with y_j primitive operations in each step. Let $Y = \sum_{1 \leq j \leq t} y_j$. If we employed $\max_j\{y_j\} = n$ processors, we could execute the j'th step in constant time. On the other hand, if we had $p < \max_j\{y_j\}$ processors available, we could simulate the computation of the j'th parallel step in time $\lceil \frac{y_j}{p} \rceil \leq (\frac{y_j}{p}) + 1$. Therefore, the total parallel time to simulate the algorithm would be no more than

$$\sum_{1 \leq j \leq t} \lceil \frac{y_j}{p} \rceil = \sum_{1 \leq j \leq t} (\frac{y_j}{p} + 1) = \lceil \frac{\sum_j y_j}{p} \rceil + t = \lceil \frac{Y}{p} \rceil + t.$$

So, we have actually proven the following:

Theorem 1 *Suppose that an algorithm is composed of Y primitive computational operations executed in t parallel steps, and let $y_j, 1 \leq j \leq t$ be the operations executed at step j. Then the algorithm can be implemented in $O(\frac{Y}{p} + t)$ parallel steps using $p \leq \max_j\{y_j\}$ processors.*

$\square$

Brent's scheduling principle assumes that allocating processors to operations is not a problem. This is true when, for $1 \leq j \leq t$

(a) the y_j's can be computed and

(b) we can group the y_j operations in groups of cardinality p.

In fact, processor allocation is straightforward in the prefix sum algorithm. Consider having $p \leq n/\lg n$ processors available. Assign blocks of $\lg n$ consecutive array entries to each processor. The algorithm proceeds in three phases. In the first phase each processor computes the prefix sum of its own block sequentially. At the end of this phase, p values remain and their prefix sum is computed using the algorithm given above in $O(\lg p)$ time using p processors. In the last phase, the results found in the previous phase are distributed and used to compute the prefix sum of each element in the block sequentially.

3.3 Pointer Jumping and List Ranking

Consider having a linked list of n elements, each one having a pointer p pointing to the next element in the list, and the last element's pointer pointing to itself. Pointer-jumping is a technique, developed by Wyllie [Wyl81], that causes all the elements' pointers to point at the last element after $\lg n$ steps. (Figure 3.3)

Pointer Jumping Algorithm:

```
for $i = 1$ to $\lceil \lg n \rceil$ do
    for each node $v$ in parallel do
        $p(v) \leftarrow p(p(v))$
```

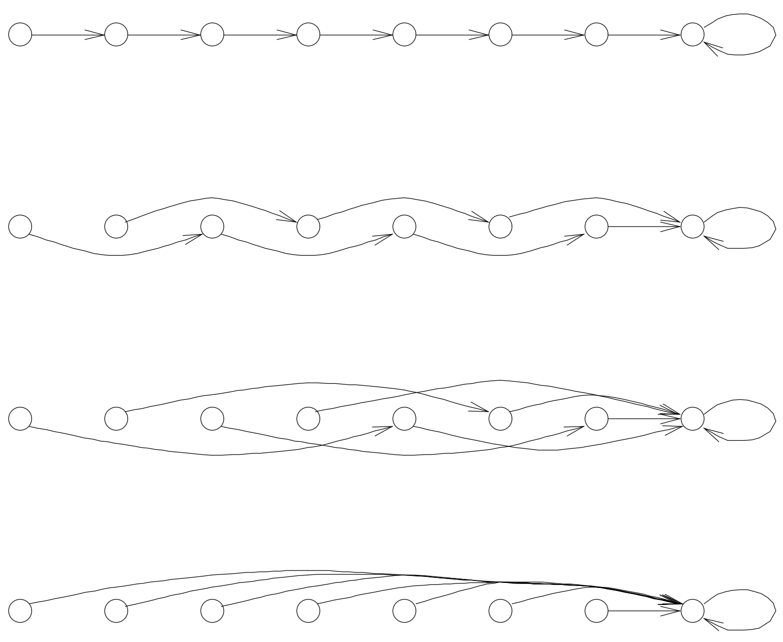
Figure 3.3: The pointer jumping technique.

The pointer jumping technique is used very often in parallel algorithms. It was first used in solving the list-ranking problem, a simple generalization of the prefix sum:

Problem 2 *We are given an associative operator* $\circ$ *and* n *elements* $x_1, \ldots, x_n$ *drawn from some domain* D*. The elements are being placed in the cells of a linked list. Each cell* v *in the list is assumed to have a field* $val(v)$ *containing* x_i *and a field* $p(v)$ *containing a pointer to the cell containing the next element* x_{i+1} *in the list. For* $i = 1, \ldots, n$ *we want to compute the* n *suffix sums*

$$x_i \circ x_{i+1} \circ \cdots \circ x_n = \sum_{i \leq j \leq n} x_j$$

A simple algorithm that solves the problem is given below:

Input: A linked list of n elements $\{x_i\}$.

Output: Each cell v in the list contains the suffix sum $\sum_{i \leq j \leq n} x_j$.

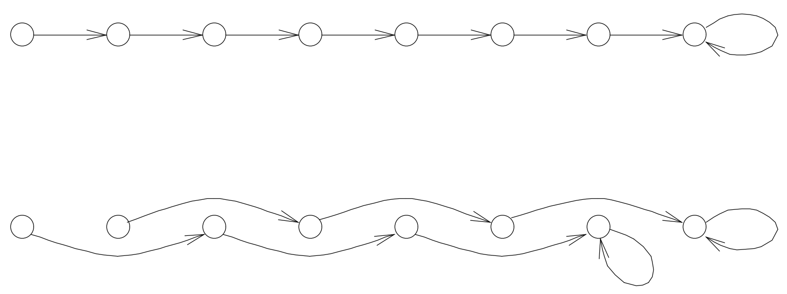

Figure 3.4: List ranking technique for the EREW PRAM.

List Ranking Algorithm:

```
Procedure List Ranking
            for $i = 1$ to $\lceil \lg n \rceil$ do
                for each cell $v$ in parallel do
                    $val(v) \leftarrow val(v) \circ val(p(v))$
                    $p(v) \leftarrow p(p(v))$
```

This simple algorithm works in $O(\lg n)$ using n CREW PRAMs. It can be easily transformed to run on an EREW PRAM by having each cell reaching the last element copy its contents, thus pretending it is the final node. (Figure 3.4).

However, $w(n) = \Theta(n \lg n)$, therefore, it is not optimal. Cole and Vishkin [CV86a] have given an optimal EREW PRAM algorithm using very elaborate techniques. The algorithm was later simplified somewhat by Anderson and Miller in [AM91]. The basic idea comes from the *deterministic coin tossing* technique developed in [CV86b].

3.4 Tree Contraction

Many problems have input represented as a rooted tree. Probably the best known of these is the *expression tree evaluation* problem: Given a tree whose external nodes are numbers and whose internal nodes are operations, compute the expression represented by the tree. One natural generalization of this problem is the *all subexpressions evaluation*, which asks to compute all the partial subexpressions defined by the internal nodes.

Both problems have a simple sequential solution based on the depth-first search technique. However, when working in parallel, things are not quite as simple. The obvious parallel solution which processes the leaves simultaneously performs well on a balanced tree but it is very slow when the tree is unbalanced. To get a better worst-case running time, one would have to process many tree nodes concurrently. A valid tree-contraction schedule achieves this objective.

Let's assume that we have defined a number of operations which process the nodes of the tree as follows: A *prune* operation removes the leaves of a tree, and a *shortcut* operation removes nodes of degree two from the tree. Of course, both operations may update the parent's and/or child's nodes. Moreover, individual prunings and shortcuttings take $O(1)$ time to be performed.

A *valid tree-contraction schedule* [JM91b] is one which schedules the nodes of the binary tree for pruning and shortcutting in such a way that:

(i) when a node is operated upon it has degree one or two, and

(ii) neighboring nodes are not operated upon simultaneously.

We first mention a lower bound for any valid tree-contraction schedule. Any valid tree-contraction schedule must have length $\Omega(\lg n)$ since at most $\lfloor n/2 \rfloor + 1$ nodes can be removed simultaneously.

There are, actually, several valid tree contraction schedules that can achieve this lower bound. First, Miller and Reif [MR85] proposed such a schedule, which was

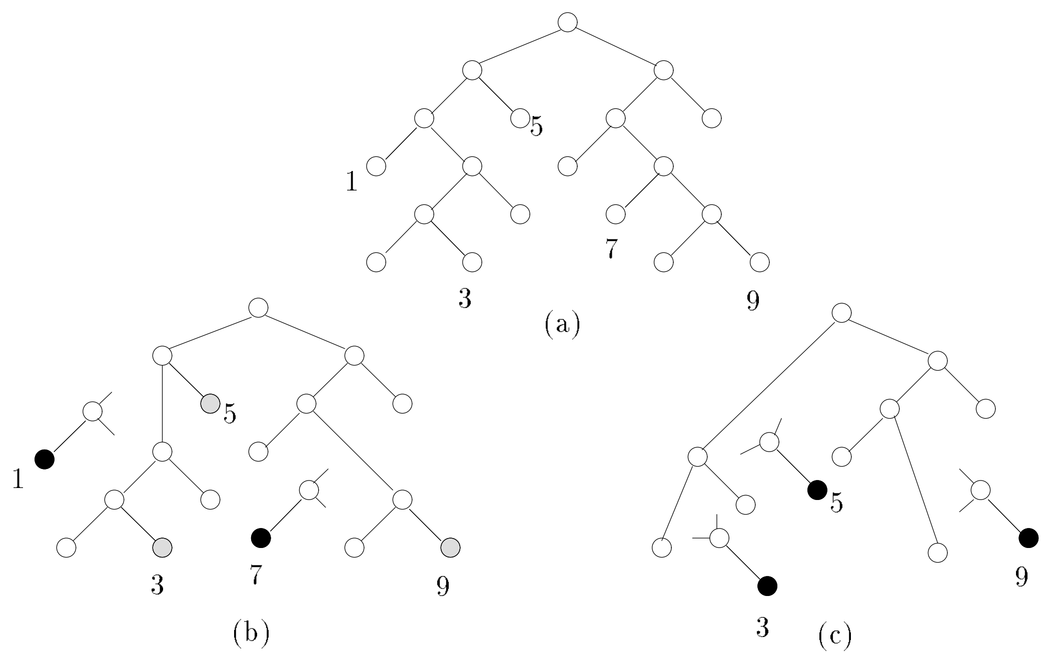

Figure 3.5: The Shunting Schedule: The first phase. (a) Numbering of the leaves. (b) Step 2: Shunting of odd numbered left children. (c) Step 3: Shunting of odd numbered right children.

constructed on the fly by an optimal randomized algorithm[3]. The problem and its applications drew the attention of researchers, and soon several optimal deterministic algorithms were presented [ADKP89, KD88, CV88, GR86, GMT88]. For a discussion of the history of the parallel tree-contraction algorithms see [KR90].

We should note here that several researchers who have given solutions to other tree contraction problems [4] have used a variety of names to denote the "removal of a leaf" and "removal of a node with degree two" operations. *Rake* has been used as a synonym for prune, *compress* and *by-pass* as synonyms for shortcut. Finally, *shunt*

[3]A newer version of their paper was published in [MR89]

[4]We mention some of these tree problems: Minimum vertex cover [CV88], maximum independent set, maximum matching [He86], algebraic tree computation, problems on cographs [ADKP89], partitioning weighted tree [KD88], etc.

and *rake* have been used to denote the application of a prune followed by a shortcut.

We will briefly describe here the simplest of these schedules, called the *shunting* algorithm, which was proposed independently by [ADKP89] and [KD88]. The algorithm is composed of a number of phases, each containing the following steps (Figure 3.5):

1. Number the leaves of the tree from left-to-right. Here, the input is supposed to be a *regular* binary tree, i.e. a binary tree in which every internal node has exactly two children. The numbering can be done using the eulerian tour technique developed by Tarjan and Vishkin [TV85] within the desired bounds.
2. Shunt the odd-numbered leaves that are the left children of their parent.
3. Shunt the odd-numbered leaves that are the right children of their parent.
4. Shift out the last bit of the remaining leaves and repeat steps 2 to 4 until the whole tree has been contracted.

Theorem 2 ([ADKP89]) *The shunting algorithm computes a valid tree-contraction schedule which has length $O(\lg n)$.*

$\square$

If we had one processor assigned to each node, we could contract the tree in the desired time using n processors. But the time in which each leaf is processed can be computed beforehand and placed in an array of length $\lceil n/2 \rceil + 1$. The array is filled with pointers to leaves having numbers 1,3,5,7,..., then to leaves having numbers 2,6,10,14,..., etc. In general there are $O(\lg n)$ phases numbered $0 \leq i \leq \lceil \lg(n-2) \rceil$, and in each of them, leaves numbered $2^i, 3 \cdot 2^i, 5 \cdot 2^i, 7 \cdot 2^i, \ldots$ are shunted. Thus, having the array, it takes time $O(n/p)$ using $p \leq n/\lg n$ processors to do the contraction. Optimality is achieved for $p = n/\lg n$.

Another valid tree-contraction schedule is computed by the so-called ACD technique. The accelerated centroid decomposition (ACD) technique was proposed by Cole and Vishkin [CV88]. Though the ACD method is rather complex and lengthy to fully describe, we will give here a brief outline of it.

We first define the centroid decomposition of a tree T. Let $size(v)$ of a node v be the number of nodes that are contained in the subtree rooted at v, and let *centroid level* of node v be $\lceil \lg size(v) \rceil$. Note that there are at most $\lceil \lg n \rceil$ centroid levels in any binary tree with n-nodes. The *centroid path* of v is the longest path which passes through v, and is composed of nodes having the same centroid level as v along with tree edges connecting them. The direction of the path is from child to parent. (Figure 3.6.)

Centroid decomposition of a tree T is the partition of the tree nodes into centroid paths. Such a partition is always possible, since each node can have the same centroid level with at most one of its children. A node v is the *tail* of its path, if it has no child with the same centroid level as itself.

According to the ACD technique, some node v is scheduled for removal as follows: Let $cp(v)$ be the centroid parent of v, that is, the parent of v if it belongs to the same centroid path, and let A_v be the sum of the sizes of the non-centroid children of all centroid ancestors of v in its centroid path. Define K_v to be the index of the most significant bit in which the bit representation of A_v and $A_{cp(v)}$ differ.

Schedule. If v is the tail of a centroid path with level i, it is pruned at time $2i + 1 = 2K_v + 1$; otherwise it is shortcutted at time $2K_v + 2$.

The idea behind this schedule is that, if a node is shortcutted at time t, its non-

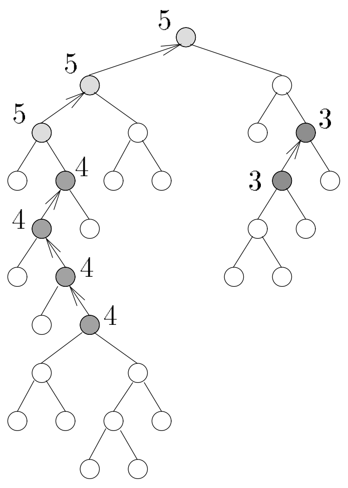

Figure 3.6: A tree and its centroid paths.

centroid child was pruned at time $t' < t$.

Finally, we should note that other valid tree-contraction schedules can be derived by the algorithms in [GR86] and [GMT88].

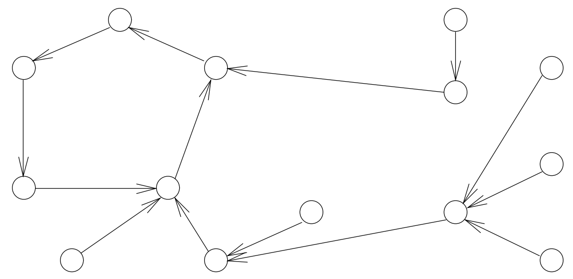

Figure 3.7: A pseudotree.

3.5 Pseudotree Contraction

A *pseudotree* $P = (C, D)$ is a maximal connected directed subgraph with $|C| = n$ vertices and $|D| = n$ arcs for some n, for which each vertex has outdegree one (Figure 3.7). An immediate consequence of the outdegree constraint is that every pseudotree has exactly one simple directed cycle (which may be a loop). We call the number of arcs in the cycle of a pseudotree P its *circumference*, $circ(P)$.

A *rooted tree* is a pseudotree whose cycle is a loop on some vertex r called the *root*. So, it has circumference one. A *rooted star* R with root r, is a rooted tree whose arcs are of the form (x, r) with $x \in R$, i.e. all of whose arcs point to r.

A *pseudoforest* $F = (V, A)$ is a collection of pseudotrees. An equivalent definition of a pseudoforest [Ber85] is a *functional graph* $F = (V, f)$, the graph of a finite function f on a set of vertices V. We can think of f as being implemented by a set of pointers p, so we will also call F a *pointer graph* (V, p). We will refer to pseudoforests using any of the three equivalent definitions.

We define the pseudotree contraction problem as follows:

Problem 3 *Given a pseudotree $P = (C, D)$, create a rooted star $R = (C, D')$ having root some vertex $r \in C$ such that for each $v \in C$, then $(v, r) \in D'$.*

We will show how to solve the pseudotree contraction problem in $O(\lg |C|)$ parallel time using $|C|$ CREW PRAM processors.

Pseudoforests are especially interesting in parallel computation, since they arise often in parallel graph algorithms when vertices of a graph draw simultaneously a pointer to a neighbor, an operation called *hooking*. Connectivity, minimum spanning tree, maximal independent set and tree-coloring, [JM91a, NM82, SV82, HCS79, AS87, GPS87, JM91c] are among the problems that deal with creation and manipulation of pseudotrees.

However, the presence of the cycle complicates the parallel contraction of the pseudotrees. The reason that the well known pointer-doubling technique [Wyl81] does not work on a cycle is that it will not terminate (Figure 3.8). Even if one modifies this technique to recognize a cycle by keeping track of all the vertices pointed to by some pointer, pointer-doubling performs poorly as it may run in time linear in the circumference of the cycle.

We introduce here a set of pointer-jumping rules called *cycle-reducing (CR) short-cutting rules* [JM91a]. These rules are used to reduce a pseudotree to a rooted star, without concurrent writing by the processors involved, in time $\lceil \log_{3/2} h \rceil$ where h is the longest simple directed path of the pseudotree. (Figure 3.9.)

Let $G = (V, E)$ be a graph. We assume that each vertex $v \in V$ has been assigned an $id(v) \in Id$, which is a distinct integer, for example the id-number of the processor which is responsible for the vertex. Let $F = (V, p)$ with $F \subseteq G$ be a given pseudoforest defined on the vertices of G. We would like to contract each of F's pseudotrees to a rooted star. We will do that using the CR shortcutting rules (Figure 3.10). These rules assign the vertex r having the smallest id among all the vertices in the cycle to be the root of the future rooted tree. The idea behind the CR rules is that they

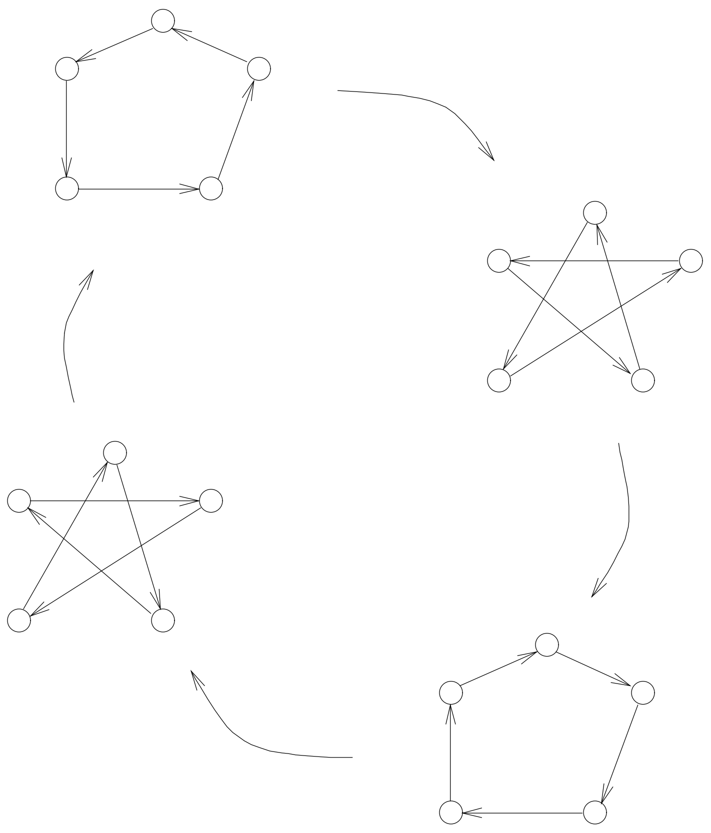

Figure 3.8: The usual pointer jumping technique cannot deal with cycles.

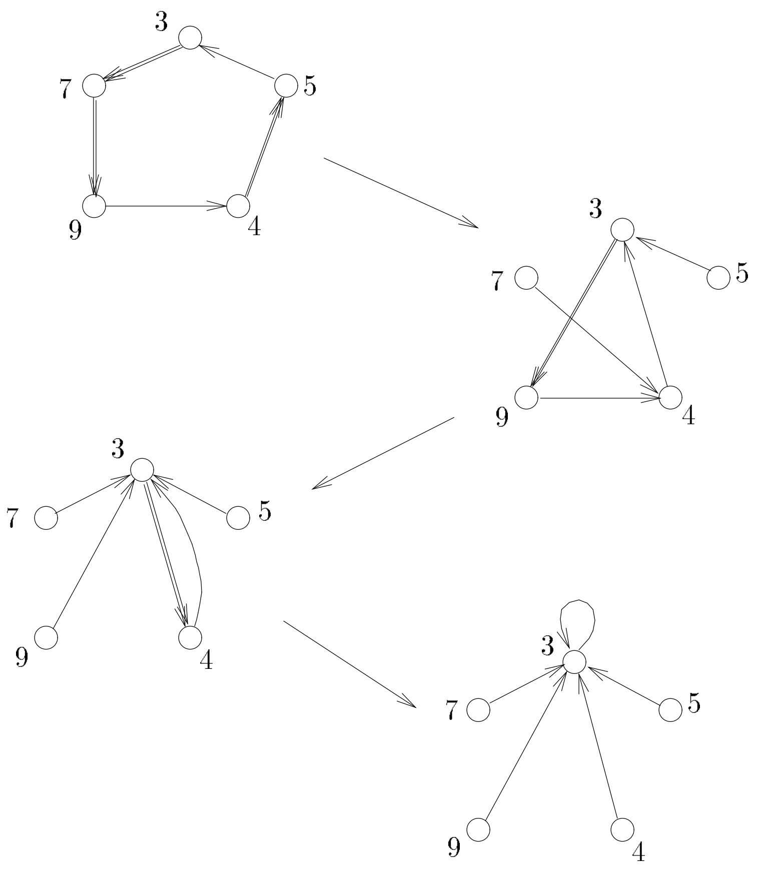

Figure 3.9: Using the CR shortcutting rules the cycle can be reduced to a rooted tree in logarithmic number of steps.

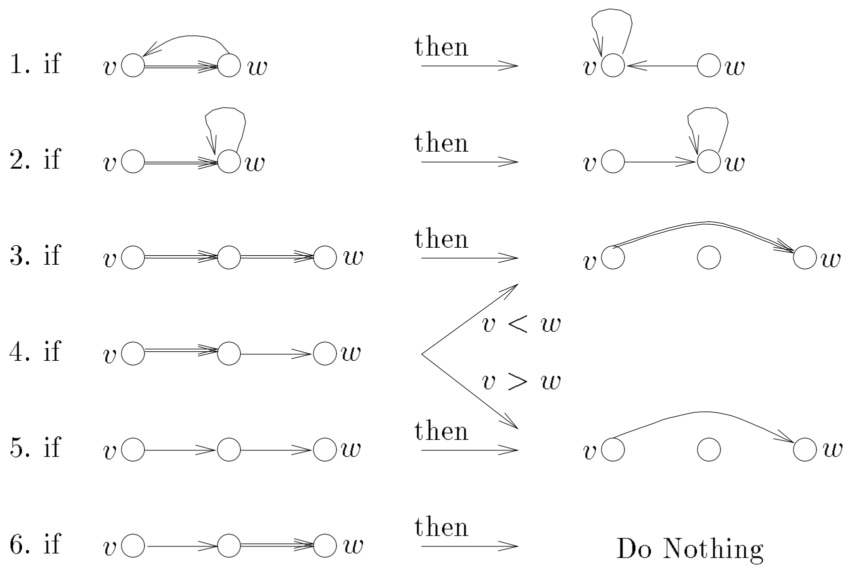

Figure 3.10: Cycle-Reducing rules. (1) and (2) Terminating rules, (3) Bold-bold rule, (4) Bold-light rule, (5) Light-light rule, (6) Light-Bold rule. Comparison between vertices means comparison between their corresponding id's.

will not let any of the vertices of the pseudotree shortcut over the future root, even though at the outset the root is not known.

To use the CR rules we will create two kinds of p pointers: *bold* and *light*. Bold pointers belong only to possible roots of a pseudotree. All other vertices have light pointers. Each vertex v of the pseudotree will first execute the *rule enabling* statement:

```
**for** each vertex $v \in C$ **in parallel do**
        $bold(v) \leftarrow id(v) < id(p(v))$
```

To contract a pseudotree, its vertices repeatedly execute the *CR procedure* given below. The rules of this procedure are also graphically described in Figure 3.10.

```
**if** $bold(v)$ **and** $p(p(v)) = v$
        **then** $bold(v) \leftarrow false$
```

```
                      $p(v) \leftarrow v$
else if $bold(v)$ and $p(p(v)) = p(v)$ then $bold(v) \leftarrow false$
else if $bold(v)$ and $bold(p(v))$ then $p(v) \leftarrow p(p(v))$
else if $bold(v)$ and $not(bold(p(v)))$
                      then if $id(v) > id(p(v))$ then $bold(v) \leftarrow false$
                           $p(v) \leftarrow p(p(v))$
else if $not(bold(v))$ and $not(bold(p(v)))$ then $p(v) \leftarrow p(p(v))$
{ else if $not(bold(v))$ and $bold(p(v))$ then Do-Nothing }
```

We will refer to the first two rules as *terminating rules*, and to the remaining four rules as *bold-bold*, *bold-light*, *light-light* and *light-bold*, respectively. The names are given according to the v and $p(v)$ pointers. Shortcutting occurs in all but the light-bold rule, in which case the vertex attempting the jump might shortcut over the smallest numbered vertex of a cycle.

We now show that the CR rules will, in fact, contract a pseudotree to a rooted tree. Then we will prove that this contraction takes time logarithmic in the number of vertices in the pseudotree.

Lemma 1 *Let P be a pseudotree with cycle c. When the CR rules apply on P for a long enough period of time, they contract it to a rooted tree. The root of this tree is the smallest numbered vertex r that appears on c.*

Proof. We say that vertex v has *reached* vertex u if $p(v) = u$. According to the rule enabling statement and the CR rules, vertex r will have a bold pointer for as long as there is a nontrivial cycle in the pseudotree. At the same time, any vertex that reaches r must do so with a light pointer. So, no vertex will shortcut over r because the light-bold rule applies. At the same time, r will be continually shortcutting over the other pointers in the cycle since the bold-bold and bold-light rules permit it. Eventually, the first terminating rule will apply and r will reach itself, effectively becoming the root of a rooted tree. $\square$

Let P be a pseudotree composed of n vertices and n pointers (arcs) and let r be its root, if P is a rooted tree, or its *future root* (the smallest numbered vertex on P's cycle). We define *distance* d_v of a vertex $v \in C$ to be the number of pointers on the simple directed path from v to r. Also, we define *k'th distance* d_v^k of vertex v to denote d_v after k applications of the CR rules on P. We can write d_v as d_v^0. We will show that each application of the CR rules on P decreases the distances d_v of the vertices $v \in C$ by roughly a factor of two-thirds. More specifically, we will show that:

Lemma 2

$$d_v^k \leq \lceil \frac{2d_v^{k-1}}{3} \rceil$$

Proof: The proof is by induction on the distance d_v^k. The base case holds trivially, since for all v and for all k such that $d_v^{k-1} \leq 2$ the lemma is true.

For a given vertex v we assume that for all the vertices having distance smaller than d_v^k and for all the $k-1$ previous applications of the CR rules the hypothesis holds. That is, we assume that for all vertices u and for all k satisfying $1 \leq d_u^{k-1} < d_v^{k-1}$ the following holds:

$$d_u^k \leq \left\lceil \frac{2d_u^{k-1}}{3} \right\rceil$$

With this hypothesis we now prove:

$$d_v^k \leq \left\lceil \frac{2d_v^{k-1}}{3} \right\rceil$$

We will consider two cases, one accounting for application of the bold-bold, bold-light and light-light rules on v (that is, when v's pointer is shortcutting), and one for the light-bold rule (when v's pointer is stuck, but its parent's pointer is shortcutting).

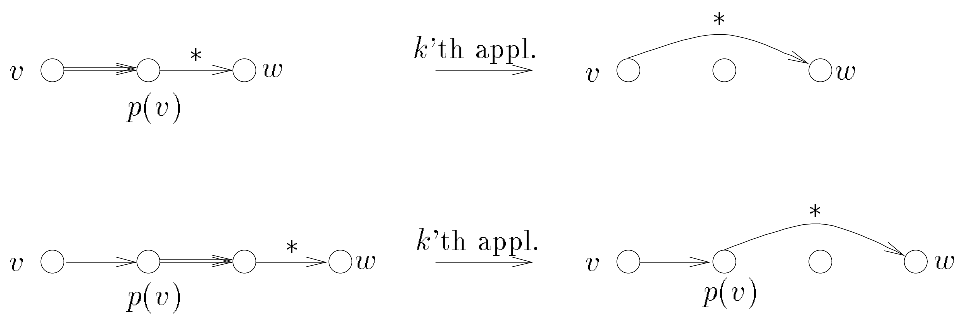

Figure 3.11: Top figure: Case 1 of the Lemma. Bottom Figure: Case 2 of the Lemma. An asterisk (*) over a pointer means that this pointer could be either bold or light.

Case 1: Let's assume that v's pointer is bold or $p(v)$'s pointer is light. Since in all cases v's pointer will shortcut (Figure 3.11, top), the analysis is identical.

Let $w = p(p(v))$. We have that $d_v^{k-1} - 2 = d_w^{k-1}$, and we assume that $d_w^k \leq \lceil \frac{2d_w^{k-1}}{3} \rceil$. We want to prove that $d_v^k \leq \lceil \frac{2d_v^{k-1}}{3} \rceil$.

At the k'th application of the CR rules v will reach w, so we have:

$$\begin{aligned}
d_v^k &\leq 1 + d_w^k \\
&\leq 1 + \left\lceil \frac{2d_w^{k-1}}{3} \right\rceil \\
&= 1 + \left\lceil \frac{2(d_v^{k-1} - 2)}{3} \right\rceil \\
&= 1 + \left\lceil \frac{2d_v^{k-1}}{3} - \frac{4}{3} \right\rceil \\
&\leq 1 + \left\lceil \frac{2d_v^{k-1}}{3} \right\rceil - 1 \\
&= \left\lceil \frac{2d_v^{k-1}}{3} \right\rceil .
\end{aligned}$$

Case 2: This is the case where v's pointer is light (Figure 3.11, bottom) and $p(v)$'s pointer is bold. The pointer of v's grandparent is either light or bold, but in any case $p(v)$ will shortcut. So, let $w = p(p(p(v)))$. We have that $d_v^{k-1} - 3 = d_w^{k-1}$, and we assume that: $d_w^k \leq \lceil \frac{2d_w^{k-1}}{3} \rceil$. We want to prove that: $d_v^k \leq \lceil \frac{2d_v^{k-1}}{3} \rceil$.

At the k'th application of the CR rules $p(v)$ will reach w, therefore we have:

$$\begin{aligned}
d_v^k &\leq 2 + d_w^k \\
&\leq 2 + \left\lceil \frac{2d_w^{k-1}}{3} \right\rceil \\
&= 2 + \left\lceil \frac{2(d_v^{k-1} - 3)}{3} \right\rceil \\
&= 2 + \left\lceil \frac{2d_v^{k-1}}{3} - \frac{6}{3} \right\rceil \\
&= 2 + \left\lceil \frac{2d_v^{k-1}}{3} \right\rceil - 2 \\
&= \left\lceil \frac{2d_v^{k-1}}{3} \right\rceil .
\end{aligned}$$

□

Now we prove the following lemmas that will be useful in the connectivity algorithm. For increased efficiency, we will assume that the rules are applied in two steps. In the first, the two terminating rules apply, and in the second step, the remaining rules apply. Let $\alpha = 1.71 > 1/(\lg \frac{3}{2})$.

Lemma 3 *When the cycle-reducing rules are applied $\lceil \alpha k \rceil$ times to a rooted tree, any vertex within distance 2^k from the root reaches the root of the tree.*

Proof. From Lemma 2 and the two terminating rules we derive that when the cycle-reducing rules are applied k times to a rooted tree, any vertex within distance $(\frac{3}{2})^k - 2$ from the root reaches the root of the tree. The Lemma follows by observing that $\lceil \alpha k \rceil > k/(\lg \frac{3}{2})$, therefore $(\frac{3}{2})^{\lceil \alpha k \rceil} > 2^k$. □

Lemma 4 *When the cycle-reducing rules are applied $\lceil \alpha k \rceil$ times to a pseudotree P whose cycle has circumference no larger than 2^k, they contract it to a rooted tree with root the vertex r having the smallest id among the vertices in the cycle. Moreover, any vertex within distance 2^k from r in the original pseudotree has reached r.*

Proof. We can view r as a vertex at distance $circ(P)$ from itself. If $circ(P) < 2^k$, then r's pointer will reach itself in $\lceil \alpha k \rceil$ steps (Lemma 3) and r will become the root of a rooted tree. On the other hand, any vertex at distance 2^k from r will reach r after $\lceil \alpha k \rceil$ applications of the CR rules. $\square$

So we have proved the following theorem for the CR rules:

Theorem 3 *A pseudotree P whose longest simple path is $\max_{v \in P}\{d_v\} = h$ will be contracted to a rooted star R after $\lceil \lg_{3/2} h \rceil$ applications of the CR rules. The root of R is the smallest numbered vertex on P's cycle.*

$\square$

3.6 Edge-Plugging Scheme

A common representation of a graph $G = (V, E)$ is the *adjacency list*: The graph is represented as an array of $|V|$ vertices, and each vertex v is equipped with a pointer to its *edge-list* $L(v)$, a linked list of all the edges that connect v to other vertices of the graph. The pointer $next(e)$ points to the edge appearing after edge e in the edge-list where e is contained.

Contraction is one of the basic operations defined on graphs [Wil85]. Under this operation, two vertices v and w connected with an edge (v, w) are identified as a new vertex vw (Figure 3.12). We can generalize slightly this operation to be performed on a subset of tree-connected vertices of the graph, rather than just on the vertices of one edge. Again, this subset is identified with a new vertex.[5] In practice, one of the vertices in the subset, called the *representative*, plays the role of the new vertex. To keep the representation of the graph consistent, one needs to include all the edges

[5]Sometimes this new vertex is called a *supervertex*.

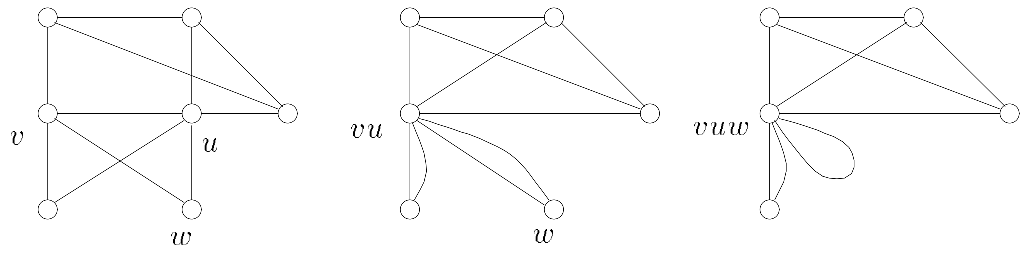

Figure 3.12: Contraction of the vertices v, u, w.

formerly belonging to the edge-lists of each vertex in the set, into the edge-list of the newly formed vertex.

As we discussed in the previous section, the vertex subsets that appear naturally in parallel computation are pseudotrees. Without loss of generality, we can think of the representative r as being the vertex assigned as the root by the CR rules. So, the following *edge-list augmentation* problem naturally arises:

Problem 4 *Given a pseudotree $P = (C, D) \subseteq G = (V, E)$, i.e. $C \subseteq V$ and $D \subseteq E$, augment the edge-list of one of C's vertices, say the representative r, with the edges that are included in the edge-lists of all the vertices $v \in C$.*

We will show how to solve this problem in constant time without memory access conflicts once the representative of the pseudotree is known.

Let the *first* and *last* functions defined on $L(v)$ give the first and last edges, respectively, appearing in $L(v)$. For implementation reasons it is convenient to assume that there is a fake edge at the end of each edge-list. All these functions are easily implemented with pointers in the straightforward way.

We represent each edge (v, w) by two *twin* copies (v, w) and (w, v). The former is included in $L(v)$ and the latter in $L(w)$. The two copies are interconnected via a

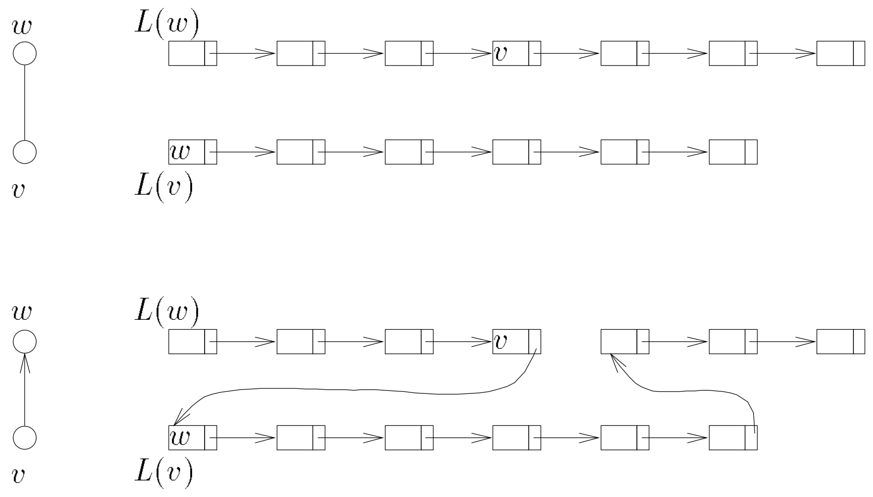

Figure 3.13: Edge lists of nodes v and w before (top) and after the edge plugging step (bottom).

function $twin(e)$ which gives the address of the twin copy of edge e.

The edge-list augmentation problem can be solved with the *edge-plugging* scheme we present here. Let $v \in C$, $v \neq r$ be a vertex in the pseudotree and $(v, w) \in D$ be its outgoing arc of the pseudotree P. According to this scheme, v will *plug* its edge-list $L(v)$ into w's edge-list by redirecting a couple of pointers. The exact place that $L(v)$ is plugged is after the twin edge (w, v) contained in $L(w)$ (Figure 3.13). This ensures exclusive writing. The edge-plugging is done by having each vertex $v \in C - \{r\}$ execute the *plugging step*:

```
**for each** vertex $v \in C - \{r\}$ **in parallel do**
    **let** $(v, w) \in D$
    **let** $(w, v) = twin(v, w)$
    $next(last(L(v))) \leftarrow next(w, v)$
    $next(w, v) \leftarrow first(L(v))$
```

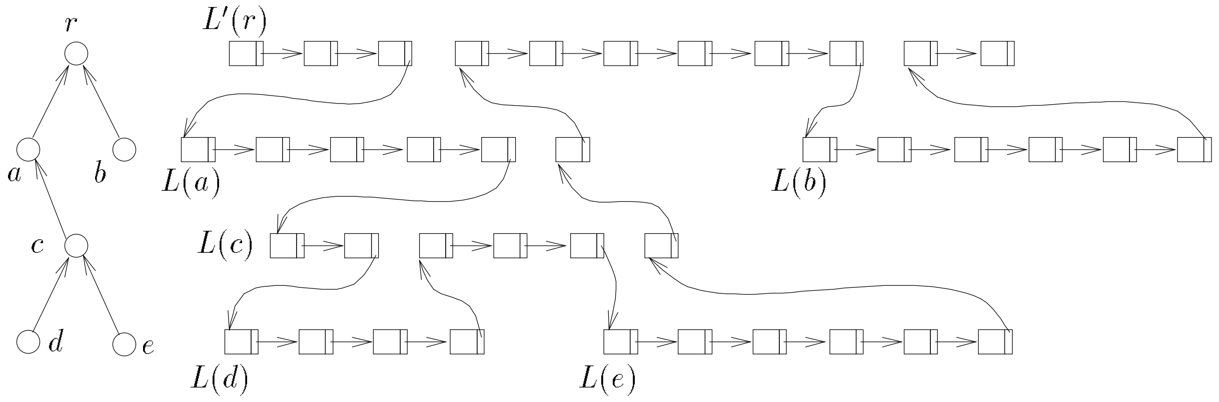

Figure 3.14: The effect of the plugging step execution by all vertices of a pseudotree but the representative r. On the left is $P = (C, D)$. On the right is $L'(r)$ after the execution of the plugging step.

We can see that the effect of having all $v \in C - \{r\}$ perform the plugging step simultaneously is to place all the edges in their edge-lists into r's updated edge-list $L'(r)$ (Figure 3.14). In particular we can prove the following lemma.

Lemma 5 *If there is a directed path p from v to r in $P = (C, D)$, then after the execution of the plugging step by all the vertices in p but r,*

$$(v, w) \in L(v) \Rightarrow (v, w) \in L'(r).$$

No writing conflict occurs during this operation.

Proof. The path p in P composed of edges in D corresponds to a unique sequence of *next* pointers in $L'(r)$, through which any edge in $L(v)$ is accessible. Moreover, the only processor that will access pointer $next(twin(v, w))$ is the processor assigned to vertex v. Therefore there is no writing conflict. $\Box$

So, we have shown that the edge-list augmentation problem can be solved in constant time once the representative of the pseudotree has been determined.

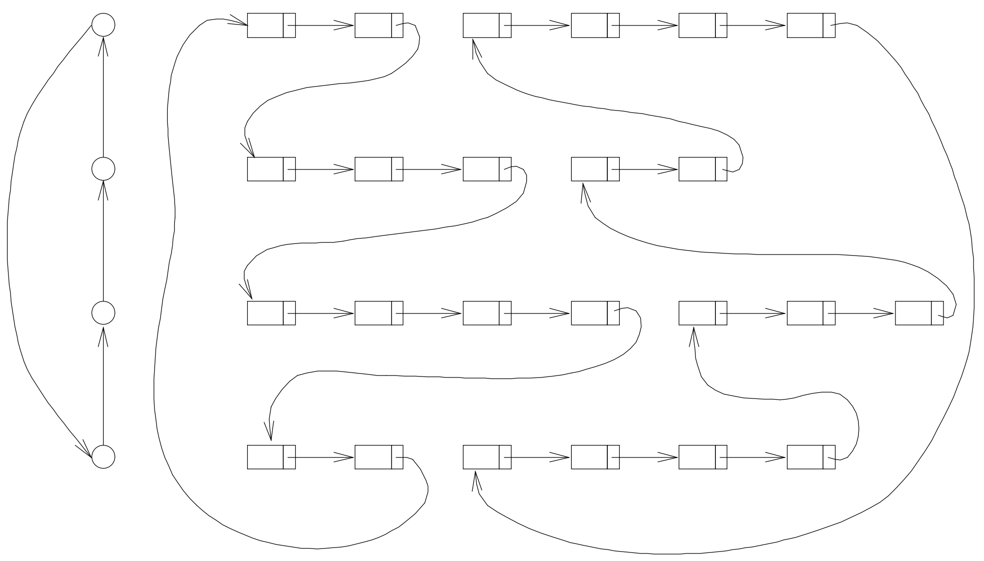

Figure 3.15: When all the vertices in a cycle execute the plugging step, their edge-lists are connected with two rings of edges. Observe that the *first* pointers of the vertices in the cycle end up in the first ring while the *last* in the second. This will enable the future root of the cycle to reverse the effect of its own edge-plugging, thus rejoining the two rings into a single edge-list.

One may ask what the effect is of having the edge-plugging step being executed by the vertices of the pseudotree *before* determining the representative. (In particular, the connectivity algorithm we will present in Section 4.4 may execute the plugging step before the contraction of a pseudotree to a rooted tree.) In other words, what if the representative also executes the edge-plugging step.

The effect of this is to place the edges of the pseudotree's vertices into two linked rings (Figure 3.15). However, this is a recoverable situation since, in general, the representative can later reverse the effects of its own edge-plugging, thus joining the two rings into a single linked list. Lemma 6 in Section 4.4.3 explains how this can be accompliced.

Chapter 4

Algorithms for Graph Problems

4.1 Finding Connected Components

4.1.1 Definition

Computing the connected components of an undirected graph is a fundamental computational problem. We will define the problem and then discuss some of the sequential and parallel algorithms that solve it.

Let $G = (V, E)$ be an undirected graph on $|V| = n$ vertices and $|E| = m$ edges. A *path* p of length k is a sequence of edges $(e_1, \cdots, e_i, \cdots, e_k)$ such that $e_i \in E$ for $i = 1, \cdots, k$, and e_i's terminal endpoint is e_{i+1}'s initial endpoint for $i = 1, \ldots, k-1$. We say that two vertices belong to the same *connected component* if and only if there is a path connecting them.

The problem of finding connected components of a graph $G = (V, E)$ is to divide the vertex set V into equivalence classes, each one containing vertices that belong to the same connected component. These classes are usually expressed by a set of pointers p such that, vertices v and w are in the came class if and only if $p(v) = p(w)$ (Figure 4.1).

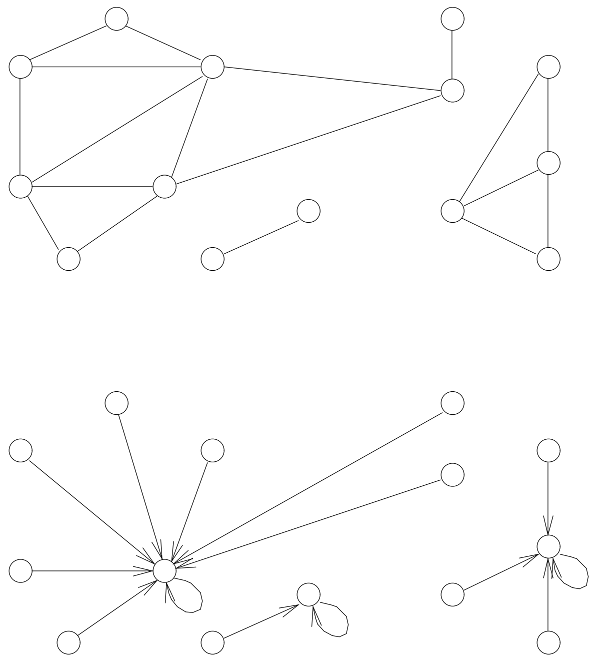

Figure 4.1: A graph G with three connected components (top) and the pointers p at the end of the computation (bottom).

4.1.2 History

It is well known that the problem has a simple linear sequential solution using depth-first search [Tar72]. In brief, the algorithm finds the connected components one-by-one by repeating the following step until all vertices have been visited:

Step. *Find a vertex that has not been visited yet. Vertices reachable from this initial vertex belong to a connected component.*

Unfortunately, efficient implementation of depth-first search in parallel seems very difficult [Rei85]. As we have stated, no polylogarithmic-time deterministic parallel algorithm is known and the best randomized algorithm [AAK89] runs in $O(\log^7 n)$ using almost n^4 processors.

So, the designers of parallel algorithms have followed a different approach which has roots in the early work of the Czechoslovakian engineer Borůvka [Bor26] on minimum spanning trees. [1]

Hirschberg, Chandra and Sarwate [HCS79] gave the first efficient parallel connectivity algorithm. It has parallel running time $O(\lg^2 n)$ using $n^2/\lg n$ CREW PRAM processors.

This algorithm was later improved by Chin, Lam and Chen [CLC82] who reduced the number of processors by a factor of $\lg n$, while Nath and Maheshwari [NM82] showed how to implement it on the EREW PRAM model when n^2 processors are available.

For the CRCW PRAM model in which concurrent writing is permitted, Shiloach and Vishkin [SV82] gave an algorithm which has running time $O(\lg n)$ using $n + m$ processors. This algorithm was later improved by Cole and Vishkin [CV86a] to use only $(n + m)/\alpha(n, m) \cdot \lg n$ ARBITRARY CRCW PRAM processors, where $\alpha(n, m)$

[1] See the article by Graham and Hell [GH85] on this algorithm's interesting history.

is the inverse Ackerman function. There is also an optimal randomized algorithm [Gaz91] for the latter model with $O(\lg n)$ time complexity. Other algorithms with minor improvements are reported in [AS87, LM86, Phi89].

If we were to implement a connectivity algorithm on a PRAM, the implementation bound would be $O(\lg n)$ time using $n+m$ CRCW PRAM processors, or equivalently, $O(\lg^2 n)$ time using the same number of CREW PRAM processors. The reason is that the [CV86a] algorithm uses very elaborate techniques. This fact was noted by Karp and Ramachandran in [KR90], where a connectivity algorithm with running time $o(\lg^2 n)$ for the CREW PRAM is included among the challenges that had thus far eluded researchers.

4.1.3 On the Existing Algorithms

We mentioned that all the existing parallel connectivity algorithms could be viewed as implementations of Borůvka's minimum spanning tree algorithm. This algorithm maintains a minimum spanning forest by repeating the following step (adapted from Tarjan [Tar83, page 73] who also makes the observation that Borůvka's method is well suited for parallel computation):

Step. *For every tree T in the forest, select a min-cost edge incident to T. Construct the new forest by augmenting the trees with the selected edges.*

Initially, each vertex is trivially a tree. At the end we are left with a minimum spanning tree (assuming the input graph was connected).

Basically, there are two parallel implementations of this idea: The [HCS79] algorithm and the [SV82] algorithm. They mainly differ on the selection of the data structure used to represent the set of edges and on the model of parallel computation they use.

The algorithms deal with *components*, sets of vertices organized as rooted trees or rooted stars. A component C_2 is *neighboring* to another component C_1 if and only if there is an edge (v, w) between vertices $v \in C_1$ and $w \in C_2$. We briefly describe the algorithms here.

The HCS Algorithm.

In the [HCS79] algorithm, the input graph $G = (V, E)$ is given by an array D of the $|V|$ vertices and an adjacent matrix A of size $|V| \times |V|$ for the edges of E.

At the end of the algorithm, $D(v) = D(w) = x$ if and only if vertices v and w belong to the same connected component and vertex x is the representative of this component.

In the beginning, each vertex v is the representative of a component containing only itself. The algorithm proceeds as follows:

for $\lg n$ times **in parallel do:**

1. The representative of each component finds the neighboring component with the smallest id-number using the adjacency matrix, and hooks to it by drawing a pointer to it. The pseudotrees with circumference 2 generated by this step easily become rooted trees.

2. The trees are contracted to rooted stars using pointer-jumping. The roots of these stars are the representatives of the newly formed components.

3. The entries of D and A are updated to reflect the new situation.

Each of the steps of the algorithm takes $O(\lg n)$ time, so the total running time is $O(\lg^2 n)$. Moreover, no write conflict occurs, and careful allocation of processors

[CLC82] implements the algorithm employing only $n^2/\lg^2 n$ of them. Note that the algorithm actually computes a minimum spanning tree of a graph at no extra cost if hooking is done over least cost edges.

The SV Algorithm.

On the other hand, the [SV82] algorithm keeps the edges of its input graph $G = (V, E)$ in an array of $|E|$ edges, not organized in any particular way. (Even though in their description the edge vector is lexicographically sorted, this is not essential.) The vertices are placed in the array D and at the end the output is given, as in the previous algorithm, by the entries of the D array.

The description of the algorithm in their paper is a little complicated, so we present a simpler, less formal, description of it. Recall that the algorithm is for a model which supports concurrent write, which is a feature used extensively in the algorithm.

for $\lg n$ times **in parallel do:**

1. The representative of each component tries to hook to some neighboring component by pointing to it. This is achieved by having all *good* edges of the component competing to be selected for hooking by writing to some memory location. By "good" edges, we mean edges that know they are not internal to the component and also that their selection will not create a cycle.

2. The hooking process in the previous step was "conservative" in selecting neighboring components, to make sure that no cycles were introduced. This conservatism could result in halting the growth of some components if no good

edge was found. In this step, these latter components hook to *any* neighboring component they can find.

3. The trees created in the previous step are contracted a little bit using the pointer-jumping technique for a constant number of times.

The lack of simplicity in the description of the algorithm comes from the effort to avoid creating directed cycles during the hooking process. We remark that, if we let cycles occur and then use the CR rules [JM91a] to deal with them, a much simpler algorithm results. See Appendix 6.1 for details on this.

4.2 Designing a Connectivity Algorithm

4.2.1 A General Idea and its Difficulties

Let $G = (V, E)$ be the input graph with $n = |V|$ vertices and $m = |E|$ edges. We will assume that there is one processor *Proc(i)* assigned to each vertex $i \in V$ and one processor *Proc(i,j)* assigned to each edge $(i, j) \in E$. Each component is equipped with an edge-list, a linked list of the edges that connect it to other components.

Typically the algorithm will deal with *components*, which are sets of vertices found to belong to the same connected component of G. Initially each component is a single vertex. The algorithm proceeds as follows (See Figure 4.2):

repeat until there are no edges left:

1. Each component picks, if possible, the first edge from its edge-list leading to a neighboring component (called *the mate*), and *hooks* by pointing to it. The hooking process creates pseudotrees. If a component has an empty edge list, it hooks to itself.

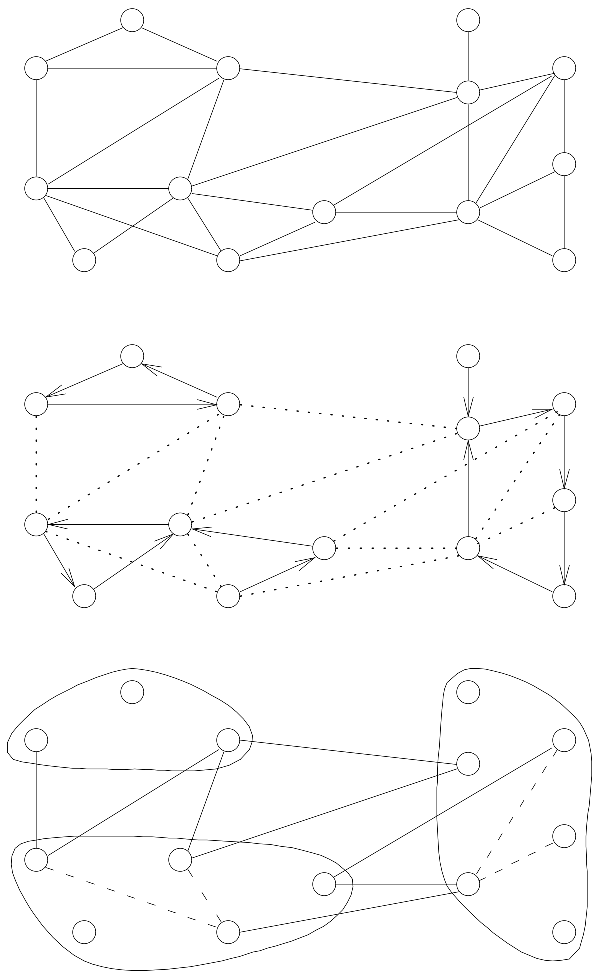

Figure 4.2: (Top) the input graph G. (Middle) Vertices have picked their mates. Dotted are those edges they were not picked by any vertex. An arc points from a vertex to its mate. Three pseudotrees are shown in this figure. Note that each pseudotree contains a cycle. (Bottom) The new components have been identified. Dashed edges are internal edges that will not help the component grow.

2. Each pseudotree is identified as a new component with one of its vertices as its representative. Each representative receives into its edge-list all the edges contained in the edge-lists of its pseudotree.

3. Edges internal to components are removed.

There are three problems we have to deal with in order for the algorithm to run fast without concurrent writing.

The existence of cycles. The parallel hooking in the first step of the algorithm above creates pseudotrees which need to be contracted. The usual pointer-doubling technique does not work on cycles when exclusive writing is required As we have said, the HCS and SV algorithms put some effort in dealing with them: The first spends $O(\lg n)$ time to create trivial pseudotrees while the second uses the power of concurrent write to avoid their creation.

We can solve this problem using the cycle-reducing shortcutting technique we introduced in a previous section. Recall that this technique, when applied to a pseudotree, contracts it to a rooted tree in time logarithmic in the length of its cycle, and when applied to a rooted tree, contracts it to a rooted star in time logarithmic in the length of its longest path.

The edge-list of a new component. Computing the set of the edges of all the components in a pseudotree without concurrent writing may be time consuming: There is possibly a large number of components that hook together in the first step and therefore a large number of components that are ready to give their edge-lists simultaneously to the new component's edge-list. Again, SV uses the power of concurrent writing to overcome this problem, while HCS uses the adjacency matrix and $O(n^2)$ processors to solve it in $O(\lg n)$ time.

The edge-plugging scheme we introduced in a previous section achieves the objective in constant time without concurrent writing, whether or not the component is contracted to a rooted star.

Finding a mate component. Having a component pick a mate may also be time consuming: There may be a large number of edges internal to the component, and this number grows every time components hook. None of these internal edges can be used to find a mate. Therefore, a component may attempt to find a mate many times without success if it picks an internal edge. On the other hand, removing all the internal edges before picking an edge may also take a lot of time.

This problem is solved by the *growth-control schedule* we will introduce in the next section. Components grow in size in a uniform way that controls their minimum sizes as long as continued growth is possible. At the same time internal edges are identified and removed periodically to make hooking more efficient.

We should note that, even though both the cycle-reducing technique and the edge-plugging scheme provide valuable tools for the algorithm, it is the growth-control schedule that achieves the $o(\lg^2 n)$ running time. We have succeeded not only in defining an optimal rate for components to grow (so as to control the overhead attendant on redundant edge removal) but in enabling the algorithm to recognize when necessary whether a component is growing too fast and therefore can be ignored.

4.3 Growth-Control Schedule

First, let us illustrate the need of such a schedule. We argued that having a component pick a mate may also be time consuming. We will make this statement more precise.

The cycle-reducing rules and the edge-plugging scheme provide the elements of a greedy-like connectivity algorithm that works correctly on the CREW PRAM model of parallel computation. Let $T(n)$ be some number that we will compute shortly. The algorithm is as follows:

Algorithm 1.

for $T(n)$ phases **in parallel do**

1. Representatives of the components perform the hooking step if they can find a mate.

2. Try to contract each of the resulting pseudotrees by applying the CR rules for a constant number of times.

3. Identify the roots of any of the resulting rooted trees as new vertices.

4. Perform the plugging step on all the vertices but the representative of each component.

5. Identify internal edges and remove them using pointer jumping for a constant number of times.

It is not difficult to see that this algorithm correctly computes the connected components of a graph in $T(n)$ phases. Moreover, we observe that, if we are sure that each component can hook in each phase, then only $\lg n$ phases are needed.

However, we cannot be sure that every component will hook in every phase. The reason is that every time two components C_1 and C_2 hook together, the number of internal edges grows. This growth is by a factor of $|C_1| \times |C_2|$ in the worst case, and the time needed to remove them using pointer jumping is $\lg(|C_1| \times |C_2|)$.

As a result of this, some component may attempt to hook many times before it can find a neighboring component. In particular, when components grow at the slowest rate, that is by just pairing up in every hook, the number of internal edges added in the edge-lists in the worst case follows the sequence:

$$1^2, 2^2, 4^2, 8^2, 16^2, \ldots, (\frac{n}{2})^2 = 2^0, 2^2, 2^4, 2^6, 2^8, \ldots, 2^{2\lg(n/2)}$$

So, the time to remove them is:

$$\sum_{0 \leq i \leq \lg(n/2)} \lg 2^{2i} = 2 \cdot \sum_{0 \leq i \leq \lg(n/2)} i = O(\lg^2 n)$$

Therefore, the number of phases $T(n)$ in this particular case would be $O(\lg^2 n)$. So, the crucial observation is the following:

In the beginning, components grow very fast. Later, when they have grown and there are lots of internal edges, there is a slowdown on the growth rate.

This observation leads to the need for controlling the components' minimum sizes. We introduce the *growth-control schedule* which lowers the running time by a factor of $\sqrt{\lg n}$ without increasing the number of processors involved. We give a brief description of it here and in the next section go into the details.

The growth-control schedule allows the size of the connected components of a graph to increase in a uniform way. To implement the schedule, the algorithm is divided into a number of phases. Phase i takes as input a graph $G_i = (V_i, E_i)$. Each vertex $v^i \in V_i$ represents a component formed in the previous phase $i-1$ containing

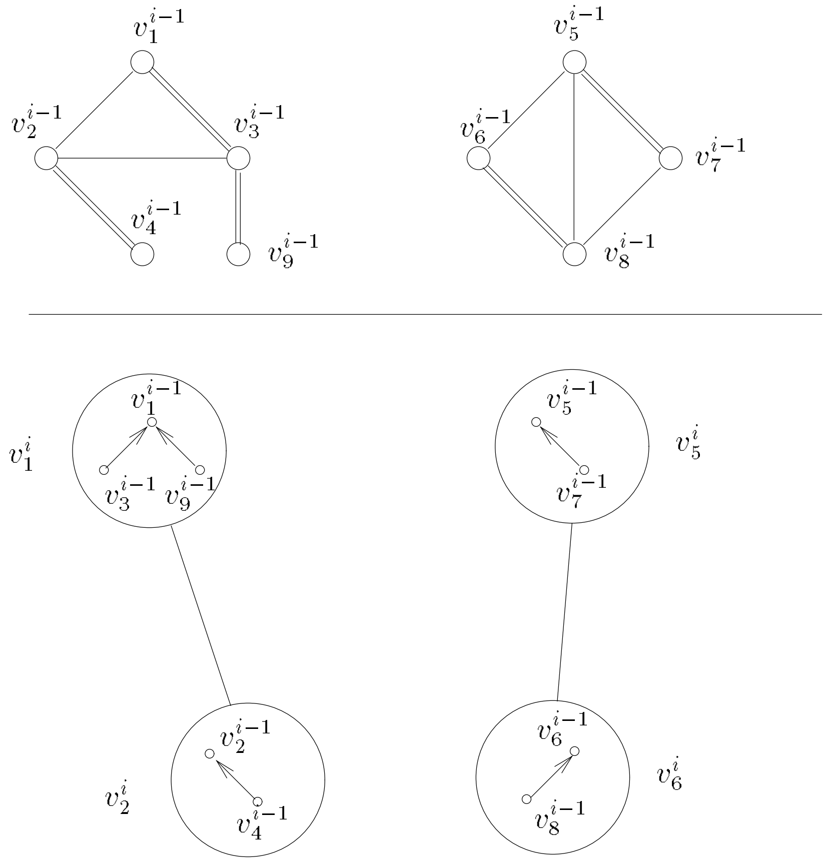

Figure 4.3: (Top) Graph G_{i-1} in the beginning of phase $i-1$. To simplify the figure, we assume that only the doubly marked edges will be used during phase $i-1$ for hooking. (Bottom) Graph G_i in the beginning of phase i. Vertex v_1^i represents the component $\{v_1^{i-1}, v_3^{i-1}, v_9^{i-1}\}$. Edge (v_1^i, v_2^i) represents the set of edges $\{(v_1^{i-1}, v_2^{i-1}), (v_2^{i-1}, v_3^{i-1})\}$.

several vertices $v_l^{i-1} \in V_{i-1}$, where $l \in Id$. We will write $v_l^{i-1} \in v^i$ to denote the membership of a component found in this way. Each edge $(v^i, w^i) \in E_i$ represents the set

$$\{(v_l^{i-1}, w_k^{i-1}) | (v_l^{i-1}, w_k^{i-1}) \in E_{i-1}, \text{ where } v_l^{i-1} \in v^i \text{ and } w_k^{i-1} \in w^i, \text{ with } l, k \in Id\}$$

of edges between distinct components v_l^{i-1} and w_k^{i-1}, which were found to belong to components v^i and w^i respectively of G^i during phase $i-1$. (See Figure 4.3.)

The purpose of each phase i is to allow just enough time for the components constituting the input graph G_i to grow by a factor of at least $2^{\sqrt{\lg n}}$. Therefore only $\lceil \sqrt{\lg n} \rceil$ phases are needed. When this growth has been achieved, the components are contracted to rooted stars using the CR rules, while useless edges (internal to the components and multiple among components) are removed from the graph. Then, a new graph G_{i+1} is created, induced by the components formed during the i'th phase. The next phase will operate on this graph.

A phase is further divided into stages. Stage j contains one hooking step followed by a number c_1 of pseudotree contraction steps and a number c_2 of useless-edges removal steps. The constants c_1 and c_2 are chosen so that components that have grown enough to enter the next phase can be recognized immediately. The growth factor has been chosen so that each phase has $O(\lg n)$ running time.

4.4 An $O(\lg^{3/2} n)$ Connectivity Algorithm

4.4.1 Outline of the Algorithm

Definitions

Let *critical bound* B be the quantity $2^{\sqrt{\lg n}}$. Some component C is in *category* i if $\lfloor B^i \rfloor \leq |C| < \lfloor B^{i+1} \rfloor$, $i = 0, 1, \ldots$, where $|C|$ denotes the number of vertices in C. No component may be in a category $i > i_{max} = \lceil \sqrt{\lg n} \rceil$.

The algorithm executes a number of *phases*, requiring that each component entering phase i is at least in category i. Therefore at most i_{max} phases are needed.

We say that some component has been *promoted* to phase i, if its size is at least $\lfloor B^i \rfloor$. We should note that the notion of promotion is needed mainly for the analysis; a component may or may not know whether it has been promoted during a phase.

In the beginning of the algorithm all components (which are simply single vertices), are in category 0. Each phase is divided into *stages*. In each stage j, components grow in size by hooking to other components. The purpose of a phase can be seen as allowing just enough time for hookings between components, so that all the components have either been promoted to the next phase or they cannot grow any more. If some component cannot grow any more, it is because it is not connected to any other component. In this case it is called *done*, else it is called *active*.

We identify edges that components do not need to keep: *Internal edges* are edges between vertices within the same component. These edges are useless and may be removed. In general, it is difficult to recognize these edges immediately. When an internal edge is recognized as such, it is declared *null*. Since each component will contain many vertices, there may be *multiple edges* between two components. For each component pair, only one such edge needs to be kept in order to hook the two components. It is called the *useful edge*. The remaining edges are called *redundant edges*. When redundant edges are recognized as such, they are also declared null and can be removed along with the null internal edges. Of course, it does not matter which of the multiple edges is kept as useful. Any one will do.

The Stages of a Phase

As we stated in the previous section, phase i takes as input a graph $G_i = (V_i, E_i)$. Each component $C \in V_i$ contains at least[2] B^i vertices of the original graph G organized in a rooted star, with *representative* the root of the star denoted by v^i_C. These vertices were found to belong to C in previous phases, and they will not play any role in phase i or in later phases. So we may assume that in the beginning of a phase each component C is a single vertex v^i_C, the representative. In the remainder of the section, when referring to a fixed phase i, we will use the term "component" to refer both to the set C of vertices in the component and to the representative v^i_C of the component. It should be clear from the context which definition applies.

Each component is equipped with an edge-list. We will keep the following invariant that will be used to prove correctness:

Invariant 1 *In the beginning of a phase i there is at most one edge between any two distinct components v^i and w^i. In particular,*

$$(v^i, w^i) \in E_i \text{ \textit{iff there was an edge} } (v, w) \in E \text{ \textit{and} } v \in v^i, w \in w^i$$

During each phase i, components continually hook to form bigger components. As we have described in previous sections, the hooking is done by having each component C pick, if possible, the first edge (v, w), $v \in C$ from its edge-list and point to its mate

[2]If a component has fewer than B^i vertices, then it is as big as it will get.

w. This operation is carried out by the representative of C. If there is no edge left in C's edge-list, then C is not connected to any other component, and it is done. At the end of the last phase all components are done.

The hooking operation creates a pseudoforest having bold and light pointers as needed by the cycle-reducing technique. Any component will perform $O(\sqrt{\lg n}\,)$ hookings per phase, each one in a different stage.

The pseudoforest is then contracted using the cycle-reducing shortcutting technique for $O(\sqrt{\lg n}\,)$ steps. The objective is the following: After $O(\sqrt{\lg n}\,)$ steps, components with fewer than B vertices have become rooted stars and are ready to hook in subsequent stages to keep growing. The exact number of CR rule applications that achieve this objective is $\lceil \alpha\sqrt{\lg n}\,\rceil$, for $\alpha = 1.71$.

Components that are rooted trees at the end of a stage are called *ready.* Those that still do not have a root are called *busy.* If some pseudotree C is busy at the end of a stage, its cycle had circumference greater that B, and therefore it has been promoted to the next phase. C will not hook in subsequent stages of this phase. At the end of the phase it will be given enough time to become contracted to a star and to prepare for the next phase.

Next, the vertices of the newly formed rooted trees are recognized. Then, all the vertices v but the roots of the contracted trees will execute the plugging step described in Section 3.6. Let r be the root of a rooted tree and v be a vertex executing the plugging step. This will place the edges of v's edge-list into r's edge-list. However, if v is a vertex of a busy component, this will not work, since there is no root in v's component. Fortunately, a busy component will be detected in a later stage and this problem will be fixed.

Edges (x, y) can now recognize their new endpoints $p(x)$ and $p(y)$ and are renamed

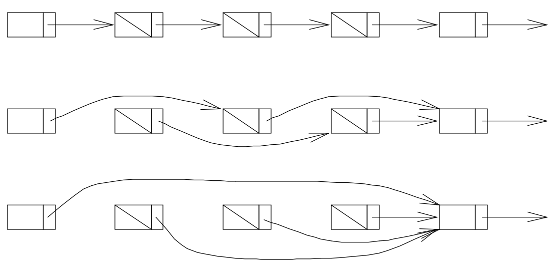

Figure 4.4: Example of two null-edge removal steps. The three null edges in the middle are being removed from the edge-list.

accordingly. Those having both of their endpoints pointing at the same root are apparently internal and *nullify* themselves. Then, the edge-list of the root is cleared of null edges by $O(\sqrt{\lg n}\,)$ *null-edge removal* steps. This is a simple application of the pointer-doubling technique (Figure 4.4):

```
for all edges $e$ do in parallel
    if $null(next(e))$ then $next(e) \leftarrow next(next(e))$
```

The exact number of null-edge removal steps is $2\lceil\sqrt{\lg n}\,\rceil + 1$, and is chosen so that any component having fewer than B vertices (and therefore fewer than B^2 null edges) can remove all its internal edges, thus it can find a non-null edge in the next stage. This ends a stage.

In the next stage, the roots of ready components will try to hook again. We say that a vertex (root) v had a *successful hooking*, if its mate w belongs to a different component. Observe that it is possible for a promoted component not to become a rooted star at the end of some stage j, because it contained a path longer than B. As a consequence, some internal edge e may not be nullified at the end of j. Moreover, the root of the component may pick e for hooking in a later stage. This is called *internal hooking.*

A root having an internal hooking may or may not detect it. For example, some un-nullified internal edge (x, y) may be recognized by the root r at the time of the hooking by checking if $r \neq x$: If this is the case, x was at a distance greater than B from r in the tree, and so x did not have the time to reach r and rename (x, y) to (r, y). In this case r will not hook, since its component is known to be promoted.

An (undetected) internal hooking will only create a pseudotree, and the cycle-reduction rules will be called again to deal with it. So, before the application of the CR rules, components have to execute the rule enabling statement, described in section 3.5.

The idea behind the $O(\sqrt{\lg n}\,)$ stages per phase is that after that many successful hookings a component has been promoted in any case. On the other hand, one internal hooking means that a component has already been promoted.

Finally, there is another case we should address. Consider a component C having more than B vertices, but whose height is less than B. The CR shortcutting process will contract it to a rooted star during the stage of its formation. However, C may have more than B^2 internal edges, and the edge-removal process may not remove them all. This component may be *unable* to hook if it picks one of the remaining unremoved null edges in the next stage. So, failure to hook means that the component has been promoted.

Each stage takes $O(\sqrt{\lg n}\,)$ steps and there are at most $O(\sqrt{\lg n}\,)$ stages, summing up to a total of $O(\lg n)$ steps per phase. We can prove that after $O(\sqrt{\lg n})$ stages all components have been promoted. They may not have been contracted, though. In the final part of the phase:

- All components are contracted to rooted stars, and representatives are identified.

- Edges are renamed by their new endpoints.

- All internal edges are identified and nullified.

- All multiple edges are identified. One of them is kept as useful while the rest are nullified as redundant. This is done as follows: First, we sort the edge list of each component in lexicographical order. We note that there are optimal sorting algorithms for both the CREW PRAM model [Col88, AKS83, Hag87] and the CREW PPM model [GK89] that run in $O(\lg n)$ time. Then, blocks of redundant edges are identified and nullified. This step takes $O(\lg n)$ time using m processors.

- The edge-list of each component is prepared by deleting all the null edges.

Each of these steps take $O(\lg n)$ parallel running time. So, the total running time of the algorithm is $O(\lg^{3/2} n)$ using at most $n + m$ CREW PRAM processors.

4.4.2 The Algorithm

The important ideas have been presented in the previous section. We now present the algorithm in detail.

In the beginning, each component contains a single vertex $v \in V$ of the input graph $G = (V, E)$, so we initialize by setting $root(v)$ to true for each $v \in V$. We also form the edge-list of each component. Then, we perform the procedure `phase` $i_{max} = \lceil \sqrt{\lg n} \rceil$ times. Finally, we execute procedure `component identification` after which, the vertex set V has been divided into a number of equivalence classes

$$CC_k = \{v | v \in V \text{ and } p(v) = v_k\},\ k \in Id$$

containing the connected components.

```
Procedure phase
1. { Initialization }

   for each vertex $v$ do in parallel
   if $root(v)$ then $not(promoted(v))$
                 $stage(v) \leftarrow 0$
                 $mate(v) \leftarrow v$

2. { Component Promotion }

   for $i \leftarrow 1$ to $\lceil \sqrt{\lg n}\,\rceil$ do
       execute stage(i).
```

COMMENT: This step tries to promote the components to the next phase. At the end of this step, each component is either promoted or done. Each stage takes time $O(\sqrt{\lg n}\,)$.

```
3. { Contract the pseudoforest to rooted stars }

   for each vertex $v$ such that $not(root(v))$ do in parallel
           $bold(v) \leftarrow id(v) < id(p(v))$

   for $i \leftarrow 1$ to $\lceil \alpha \cdot \lg n \rceil$ do
       for each vertex $v$ in parallel do
               $v$ executes the appropriate CR rule
```

COMMENT: At the end of the last stage components were rooted stars, rooted trees or pseudotrees. This step gives enough time to the last two categories to become rooted stars before they enter the next phase. First we execute the rule enabling step and then apply the CR rules.

```
4. { Rename edges and identify internal edges }

   for each edge $(v, w)$ in parallel do
         rename $(v, w)$ to $(p(v), p(w))$

   for each edge $(v, v)$ in parallel do
         $null(v, v)$
```

COMMENT: Internal edges of rooted stars are easily recognized and nullified.

```
5. { Identify redundant edges }
```

Run list-ranking [Wyl81] on the edge-list of each component to find the distance of each edge from the end of its list.

Copy each edge-list in an array using as index the results of the list ranking.

Sort the array [Col88, GK89] and use the results to form a sorted linked list.

COMMENT: The sorting places all multiple edges in blocks of consecutive identically named edges.

```
   for each edge $(v, w)$ in parallel do
         if $next(v, w) = (v, w)$ then $null(v, w)$
```

COMMENT: The last edge in a block of consecutive edges having identical (v, w) names is kept as useful. The rest nullify themselves as redundant. Since the useful edge (v, w) found in $L(v)$ may, in general, differ from the useful edge (w, v) found in $L(w)$, some care must be taken for the *twin* function to be recomputed correctly. Step 7 below takes care of it.

```
6. { Remove internal and redundant edges. }
```

```
for $j \leftarrow 1$ to $2\lceil \lg n \rceil$ do
    for each edge $e$ do in parallel
        if $null(next(e))$
          then $next(e) \leftarrow next(next(e))$
```

COMMENT: This step tries to remove blocks containing up to B^2 consecutive null edges. However, if the first edge, say e_v on some list $L(v)$ was null, it has not been removed. This is so because no pointer jumped e_v.

```
for each vertex $v$ such that $root(v)$ in parallel do
          if $null(first(L(v)))$
            then $first(L(v)) \leftarrow next(first(L(v)))$
```

COMMENT: This step explicitly removes any null e_v from $first(L(v))$. The remaining edges satisfy Invariant 1.

7. { `Recomputation of the` *twin* `function.` }

```
for all useful edges $(v, w)$ in parallel do
    let $(v', w') = next(v, w)$
    $(v, w)$ writes its address to $prev(v', w')$
    $twin(v, w) \leftarrow prev(next(twin(w, v)))$
```

COMMENT: This final step recomputes the *twin* function of the useful edges (v, w) in constant time as follows: First, observe that, after removing redundant edges from an edge-list, all edges named (v, w), useful and redundant, point at the same location. This location is the edge (v', w') that comes lexicographically after (v, w). The useful edge (v, w) passes its address to a field

$prev(v', w')$. From there, the useful edge (w, v) reads it, by following pointer $next(twin(w, v))$.

`Procedure stage(i)`

```
1. { The hooking step }

   for each active vertex $v$ such that $root(v)$ and $not(promoted(v))$
      do in parallel
            let $(x, y) \leftarrow first(L(v))$
            if $(x, y) = nil$ then $done(v)$
            else if $x \neq v$ then $promoted(v)$
            else if $(x, y) = (v, w)$ and $null(v, w)$
                  then $promoted(v)$
            else $mate(v) \leftarrow w$
                  $p(v) \leftarrow w$
                  $not(root(v))$
```

COMMENT: Roots of still active and possibly unpromoted components try to pick an edge from their edge-list. If there is no edge in $L(v)$, i.e. $first(L(v)) = nil$, its component is not connected to any other component and it is done. If $x \neq v$ then $p(x) \neq v$, then $d_x^0 > B$. This indicates that v was the root of a tree, not a star. If the edge found was null, v's component had more than B^2 null edges and therefore more than B vertices. In the last two cases, r's component is promoted. Otherwise, v can hook to its mate vertex w.

2. { `Pseudotree contraction` }

```
for each vertex $v$ such that $not(root(v))$ do in parallel
        $bold(v) \leftarrow id(v) < id(p(v))$

for $j \leftarrow 1$ to $\lceil \alpha\sqrt{\lg n}\,\rceil$ do
    for each vertex $v$ do in parallel
            $v$ executes the appropriate CR rule
```

COMMENT: First we execute the rule enabling statement and then apply the CR rules for $\lceil \alpha\sqrt{\lg n}\,\rceil$ times, which forces all components with fewer than B members to become rooted stars. Observe that after this step components with more than B members may become rooted trees or non-rooted pseudotrees. This step takes $O(\sqrt{\lg n}\,)$ time.

```
3. { Root recognition step }

   for each vertex $v$ such that $p(v) = v$ do in parallel
       $root(v)$
       $mate(v) \leftarrow v$
       if $stage(v) = i - 1$
             then $stage(v) \leftarrow i$
             else $promoted(v)$
                  $next(twin(first(L(r)))) \leftarrow next(last(L(r)))$
                  $next(last(L(r))) \leftarrow nil$
```

COMMENT: The new roots of the newly formed trees or stars identify themselves. In root v was also root in the previous stage, its component may still be unpromoted. But, if there was at least one stage $j : stage(v) < j < i$ during which v did not hook, then during stage j vertex v belonged to either a busy component or a rooted tree with height more than B that had an internal hooking. In either case the component was promoted. Note that v performed the edge-plugging step during stage $stage(v)$. Lemma 6 of the next subsection explains why the effect of v's plugging step can be reversed by the last two statements.

4. { `The edge-plugging step` }

```
for each vertex $v$ such that $not(root(v))$ and $stage(v) = i - 1$
    do in parallel
        $next(last(L(v))) \leftarrow next(twin(v, w))$
        $next(twin(v, w)) \leftarrow first(L(v))$
```

COMMENT: Non-root vertices that were roots in the previous stage and therefore hold an edge-list plug it into their mate's edge-list. At this point each unpromoted star has all the edges of its component members contained in its root's edge-list.

5. { `Edge renaming and identification of internal edges` }

```
for each edge $(v, w)$ do in parallel
        if $p(v) = r$ and $root(r)$
            then rename $(v, w)$ to $(r, w)$
        if $p(w) = r$ and $root(r)$
            then rename $(v, w)$ to $(v, r)$


for each edge $(r, r)$ do in parallel
```

```
            $null(r, r)$

    for each vertex $v$ such that $not(root(v))$ and $root(p(v))$ in parallel do
            $null(last(L(v)))$
```

COMMENT: Edges identify their new endpoints. Those having both endpoints on the same root are apparently internal and so nullify themselves. The $root(r) = true$ condition assures that lists of non-rooted pseudotrees are not altered. Finally, the last statement explicitly nullifies the unnecessary fake edges at the end of the edge-lists.

6. { `Null-edge removal` }

```
    for $j \leftarrow 1$ to $2\lceil\sqrt{\lg n}\,\rceil + 1$ do
        for each edge $e$ do in parallel
            if $null(next(e))$
                then $next(e) \leftarrow next(next(e))$

    for each vertex $v$ such that $root(v)$ do
            if $null(first(L(v)))$
              then $first(L(v)) \leftarrow next(first(L(v)))$
```

COMMENT: Blocks composed of up to B^2 consecutive null edges are removed. Unpromoted stars now contain no null edges. This ensures that they will have a successful hooking at the next stage.

`Procedure component identification`

```
for $i \leftarrow 1$ to $\lceil \lg \sqrt{\lg n}\, \rceil$ do
    for each vertex $v \in V$ in parallel do
            $p(v) \leftarrow p(p(v))$
```

COMMENT: Let's assume that the input graph was composed of k connected components. The execution of the $\sqrt{\lg n}$ phases has created a pseudoforest $F = (V, p)$ composed of k rooted trees, one for each connected component. The depth of these trees is at most $\sqrt{\lg n}$: At the deepest level lie the vertices hooked in the first phase; at the next level lie the vertices hooked in the second phase, etc.

The execution of this procedure contracts each of these trees to rooted stars. After this step, vertices v and w are in the same connected component CC_k if and only if $p(v) = p(w) = v_k$.

4.4.3 Correctness and Time Bounds

Theorem 4 *The algorithm correctly computes the connected components of a graph in $O(\lg^{3/2} n)$ parallel running time without concurrent writing.*

Proof. Correctness follows from Lemma 13 below. The running time comes from the fact that there are $\lceil \sqrt{\lg n}\, \rceil$ phases, each taking $O(\lg n)$ parallel time. $\square$

We will prove that in the beginning of each stage j, the root r of each rooted tree P holds in $L(r)$ all the edges (v, w) which in the beginning of phase i belonged to the edge-lists $L(v)$ of vertices $v \in P$ and were not deleted as internal in previous stages.

Let $G_i = (V_i, E_i)$ be the input graph of phase i. We define $M_j = (V_i, mate)$ to be the pointer graph composed of the *mate* pointers of V_i at the beginning of stage j. Note that each of the M_j's is a pseudoforest.

We say that the root r of a rooted tree $P = (C, mate)$ *satisfies invariant* (2) if

$$\forall (v, w) \in E_i \text{ such that } v \in C \text{ and } not(null(v, w)) \Rightarrow (v, w) \in L(r) \tag{2}$$

Lemma 6 *At the beginning of stage j each root r of a rooted tree $P = (C, mate) \in M_j$ satisfies (2).*

Proof. We will prove the lemma by induction on j. In the beginning of the first stage M_1 is composed of $|V_i|$ vertices, and the lemma holds true.

We assume that the lemma is true at the beginning of stage j, that is, every root of M_j satisfies (2). We will show that the invariant holds true at the beginning of stage $j+1$.

During stage j the unpromoted roots of M_j hook to form larger components (step 1). Then, in step 3, some roots will recognize themselves as the roots of M_{j+1}. We have to prove that these roots satisfy invariant (2) at the beginning of stage $j+1$.

We will distinguish two cases:

(1) Root r was also a root in the beginning of stage j. Then, for every vertex v that belonged to a tree which during the hooking step hooked on r's tree, there is a path of mate pointers from v to r. So, after the plugging step (step 4) Lemma 5 applies. Moreover note that step 6 removes only null edges. Therefore, at the beginning of stage $j+1$, root r satisfies (2)

(2) Root r was not a root in the beginning of stage j, therefore r was a part of a promoted component. We can see that the effect of having all vertices in a cycle execute the plugging step (Figure 4.5) is to break the edge-lists in two rings. To reverse the effect of plugging, re-join these two rings of edges into a chain. This can be done by r in stage j.

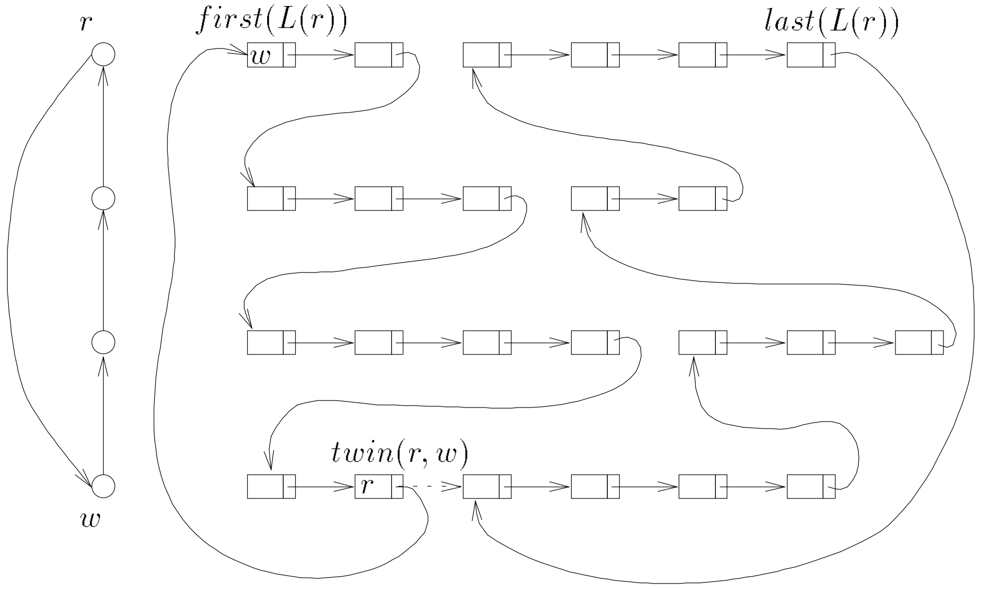

Figure 4.5: Assume that, at a later stage j, vertex r becomes the root of the pseudotree. Then, r can easily reverse its edge-plugging. In the figure, dotted lines denote the pointers that must be changed.

This operation is actually simple: r can reverse the steps of its own edge-plugging by executing the following statements:

$next(twin(first(L(r)))) \leftarrow next(last(L(r)))$

$next(last(L(r))) \leftarrow nil$

To prove that the above statements correctly re-join the rings, we have to show that (a) $first(L(r))$ still points to edge (r, w), the edge that r chose during its most recent hooking stage $j' < j$, (b) $twin(r, w) = (w, r)$, and (c) no edge shortcutted over (r, w), (w, r), or $last(L(r))$ during stages j' to j.

The $first(L(r))$ pointer is only altered by a hooking step, so (a) is true. Twin functions are only computed at the end of a phase, not during stages, so (b) is also true. Finally, as one can see by examining step 5 of procedure `stage`, these three edges were never nullified; so, (c) is also true. $\square$

Remark. Since only the edge-list of an already promoted component will ever be divided into two rings, one can actually postpone dealing with them until the end of the phase. Then one can construct the edge-lists of the components from scratch. This takes $O(\lg n)$ time, so it can be done at no extra cost. The reason that we chose to describe the rejoining steps as we did in Lemma 6 instead, was to provide the details for an implementation of Algorithm 1 (Section 4.3).

Lemma 7 *If at the end of stage j some component is busy, it has been promoted.*

Proof. By definition, a component is busy if at the end of a stage it is still a pseudotree. Of course, such a component will not pick a mate in the beginning of the next stage because it has no root to do the operation. Procedure `stage` contracts the components for $\lceil \alpha \lg B \rceil$ steps. So, according to Lemma 4, pseudotrees with circumference less than or equal to B will be rooted trees at the end of stage j and therefore not busy. Thus, a busy component had more than B members, and so it has been promoted to the next phase. $\square$

Lemma 8 *Let C be a component which in stage j has an internal hooking. Then C has been promoted.*

Proof. Recall that internal hooking happens when C picks as a mate an internal edge (without knowing it). Also note that such a hooking cannot happen in stage 1 because all components enter the first stage without internal edges. So, $j > 1$.

At the end of stage $j-1$ all components within distance B from the root have reached the root (Lemmas 3 and 4) and have nullified the appropriate entries in the root's edge list. So, an internal edge must be connecting the root of the component to some vertex v in the tree, which was at a distance more than B from the root, since it did not have enough time to reach the root. Thus, there are at least B components that reached the root (namely those in the path from v to the root), and so C has been promoted. $\square$

Lemma 9 *If a component C cannot find a mate at some stage j, then either it has been promoted or it is not connected to any other component.*

Proof. Let C be a component that cannot find a mate at some stage j of phase i. We will distinguish two cases: (a) C found no edges in its edge list, and (b) C found a null edge in its edge list.

(a) According to Lemma 6, in the beginning of each stage the root of a component C holds all the edges of its members that have not been removed as null. So, if an edge of C was not null, it would be in C's edge list. Therefore, C is not connected to any other component in the graph.

(b) First we observe that $j > 1$. Note that at the end of stage $j-1$ the algorithm performed the null edge removal step for $2\lceil \lg B \rceil + 1$ times. This removes any block containing up to B^2 null edges from the edge list of the component's root. Next we observe that any component with fewer than B members cannot have more than $B(B-1)/2$ internal edges. So, a root may find a null edge in its edge list only if its component is bigger than B and therefore is promoted. $\square$

Lemma 10 *Every active, non-promoted component at stage j will have a successful hooking at stage $j+1$.*

Proof. Let C be a component that is not promoted at the end of stage j. C is a rooted star because $|C| < B$. Also, by Lemma 6, its root holds all the edges that belonged to the edge-lists of its vertices and were not deleted in previous stages. Moreover, $L(r)$ contains no internal edges because they were all identified and deleted. So, if $L(r)$ contain any edges, r will have a successful hooking at the next stage. $\square$

Lemma 11 *After $\lceil \lg B \rceil$ successful hookings in some phase, a component has been promoted.*

Proof. First we show that if a root r is not promoted after performing k successful hookings, it was continuously hooking to components having successful hookings. For, if one of these components had an internal hooking, it was promoted; therefore, r's component was part of a promoted component.

Next, we can easily prove by induction that, after each successful hooking at stage j, components have sizes at least 2^j. Therefore after $\lceil \lg B \rceil$ successful hookings, r is the root of a component of size B and thus has been promoted. $\square$

Lemma 12 *At the end of phase i each component is either promoted or not connected to any other components.*

Proof. Each phase is composed of $\lceil \lg B \rceil$ stages. In the beginning of a stage each component is either ready or busy. A busy component cannot pick a mate, but, according to Lemma 7, it is a promoted pseudotree. On the other hand, a ready component is a rooted tree which can pick a mate from its edge list that contains all the edges of its members (Lemma 6). So, the reason for which a ready component may not be able to find a mate (according to Lemma 9), is that the component is promoted or done. Otherwise the component will find a mate.

A hooking may either be successful or internal. An internal hooking, according to Lemma 8, can only happen to an already promoted component. So, we only have to follow components which have successful hookings for $\lceil \lg B \rceil$ stages. But these components (Lemma 11) have been promoted at the end of the last stage. $\square$

Lemma 13 *In the beginning and at the end of each phase i (a) The components are rooted stars. (b) The size of each active component is at least $\lfloor B^i \rfloor$. (c) Invariant 1 is preserved. (d) There are no internal edges.*

Proof. (a) This is obviously true in the first phase, where components are composed of a single vertex, the root. During the stages, these components are hooked to form a pseudoforest. Then, at step 2 of procedure `phase`, the pseudoforest is transformed to a set of rooted stars. The remaining steps do not affect the structure of the components, and so, in the beginning of the next phase, the components are stars.

(b) This is immediate from Lemma 12.

(c) Again, this is true for the first phase. For the remaining phases, step 4 of procedure `phase` uses bucket sort to identify multiple edges and the step 5 removes them.

(d) Internal edges are nullified in step 3 and are removed in step 5 of the procedure `phase`. □

4.5 Updating a Minimum Spanning Tree

4.5.1 Introduction

The vertex updating problem for a minimum spanning tree is the problem of updating a given MST of a graph G when a new vertex z is introduced with weighted edges to the vertices of G.

This problem was first addressed by Spira and Pan in [SP75], where a $O(n)$ sequential algorithm was presented. Another solution using depth-first-search and having the same time complexity was later given by Chin and Houck in [CH78].

Pawagi and Ramakrishnan [PR86] gave the first parallel solution to the problem. Their algorithm, which runs in $O(\lg n)$ time using n^2 CREW PRAM processors, precomputes all maximum weight edges on paths between any two nodes in the tree, and then breaks the $\binom{n}{2}$ cycles simultaneously in constant time. Varman and Doshi [VD86] presented an efficient solution that works in the same parallel time, but uses only n CREW PRAMs. Their solution is a "divide and conquer" algorithm which, using the separator theorem, breaks the input tree in $\sqrt{n}$ subtrees of roughly the same size by removing $\sqrt{n}-1$ edges. Then, it solves the problem recursively in all subtrees and merges the results. Even though their idea is simple, the implementation details make the algorithm rather complex. Lately, Jung and Mehlhorn [JM88] gave an optimal solution for the more powerful CRCW PRAM model. However, simulating this algorithm without concurrent writing slows down its performance by a factor of $\lg n$. Their approach reduces the problem to an expression evaluation problem by defining a function on the nodes of the MST which, when evaluated effectively, computes the new MST.

We present an optimal yet simple solution for the EREW PRAM model which works in $O(\lg n)$ parallel time using $n/\lg n$ processors. Our solution needs a valid tree-contraction schedule. Several methods that provide such schedules have been proposed in the past few years, such as the ones reported in [MR85, ADKP89, KD88, CV88, GR86, GMT88]. All these methods give a schedule for contracting a tree of n nodes into a single node in $O(\lg n)$ parallel time using $n/\lg n$ processors, provided that the *prune* (remove leaves) and the *shortcut* (remove nodes of degree two) operations have been defined.

4.5.2 Definition of the Problem

We are given a weighted graph $G = (V, E_G)$, along with a minimum spanning tree (MST) $T = (V, E)$ and a new vertex z with n weighted edges connecting z to every vertex in V. (Note: If, in an instance of the problem, z is not connected to some vertex x, we can assume an edge (z, x) having maximum weight.) We want to compute a new MST $T' = (V \cup \{z\}, E')$.

One parallel solution would be to compute the MST from scratch, but this requires time $O(\lg^2 n)$ with $n^2/\lg^2 n$ CREW PRAMs [CLC82]. Sequential algorithms for computing the MST from scratch on a sparse graph take $O(m \lg\lg n)$ time, where m is the number of edges in the graph [Yao75, CT76]. The fact that the number of cycles in the input graph is small (there are $O(n^2)$ cycles versus an exponential number of cycles in general graphs) enables us to compute the new MST faster, by breaking the cycles.

Upon introducing the new vertex z along with n weighted edges, $\binom{n}{2}$ cycles are created. If we break these cycles by deleting the maximum weight edge (MaxWE) that appears in each cycle, the resulting tree will be the new MST. It is easy to see that at most n of these $2n - 1$ weighted edges will be included in the new MST.

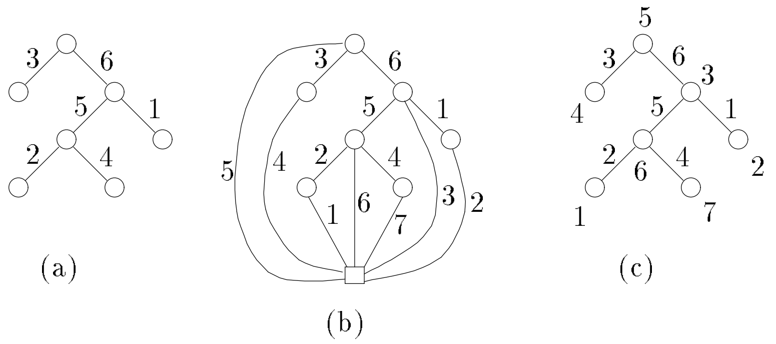

Figure 4.6: (a) The initial (given) MST, (b) the implicit graph after introducing the new vertex z along with its weighted edges and (c) the corresponding weighted tree.

No non-tree edge of the old graph can be included because all of them are already MaxWEs on some existing cycle in the original graph; so we need not consider any such edge. Moreover, any sequential algorithm that solves the problem must take time $\Omega(n)$: Consider the case when an existing MST forms a path and vertex z is connected to the two ends of the path. Then, any of the $n+1$ edges could be the MaxWE, and a sequential algorithm must examine them all.

4.5.3 Representation

In light of this discussion, we may take the input to be a tree T with $n-1$ weighted edges (corresponding to the given MST) and n weighted nodes (corresponding to weights of the newly introduced edges to z). We will call this object a *weighted tree* (Figure 4.6). Note that *a path between two weighted nodes in T corresponds to a cycle in the graph augmented with z.* Such a graph is shown in Figure 4.6b and is implied in Figure 4.6c. Thus we call this object the *implicit graph.*

In the discussion that follows, reference to the weight of a node will mean reference to the corresponding edge in the implicit graph, unless noted otherwise.

4.5.4 Breaking the Cycles

Outline of the Algorithm

As we have said, we are given the input in the form of a weighted rooted tree. We assume that each vertex has a pointer to a circular linked list of its children, and the linked lists are stored in an array. This representation of the input is not crucial, since it can be derived in $O(\lg n)$ time using $n/\lg n$ processors from any reasonable representation (see [CV88] for a discussion of representation).

The algorithm consists of a number of phases. During each phase, nodes of degree 1 (i.e. leaves) and nodes of degree 2 (internal nodes having one child) of the weighted tree are being *processed*. Each tree-node is processed once in the entire course of the algorithm. The order in which the nodes are processed in parallel is dictated by a tree-contraction schedule. Two such schedulings – the Shunting and the ACD – were described in Section 3.4. Processing a node means *examining* the edges composing *small cycles* (cycles of length 3 or 4) that the node is part of, and *breaking* these cycles by removing the MaxWE that appears in them, effectively computing the MST of the subgraph induced by the examined edges. This is done by a set of rules which also update neighboring nodes, so that the size of the unprocessed part of the tree decreases without losing any information about the MaxWEs of larger cycles.

A sequential algorithm needs only to apply the appropriate rule while visiting the nodes of the tree. Thus a depth-first-search visit of the nodes suffices. When working in parallel, though, the rules can apply to many nodes at once, provided that no confusion arizes from the updating of neighboring nodes. A valid tree contraction schedule suffices to assure that neighboring nodes are not processed at the same time. When all edges have been examined (i.e. after processing all the nodes), the MST of

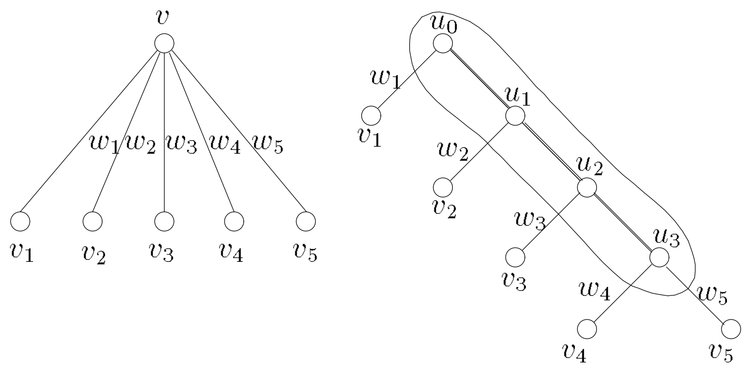

Figure 4.7: Binarization: A node with more than two children is represented by a right path of unremovable edges.

the implicit graph has been computed.

The rules assume a binary tree as input, so some preprocessing is needed to transform the weighted tree to a *binary weighted tree*. This can be achieved in $O(\lg n)$ time using $n/\lg n$ EREW PRAM processors, and is needed only for ordering the processing of a node's children; only the parallel algorithms need this transformation. This transformation is performed by the procedure *binarize*, which we briefly describe here. For the details we refer to Section 4.5.6. Each node $v = u_0$ with $k > 2$ children is augmented with $k-2$ fake nodes u_i (see Figure 4.7) to form a right path. Each of the children v_j of node v is attached as a left child of node u_{j-1}, while u_{k-2} has a right child as well. The weights of the edges connecting the u_i's have weights $-\infty$ which makes them unremovable by the MST algorithm. The fake nodes do not have weights. They are introduced only to facilitate the order of processing of v's children. At the end of the algorithm the right path is always included in the MST of the binarized problem, giving a unique obvious solution to the original problem. The binary weighted tree has the same number of cycles, but may have height much larger than the input tree and may contain twice as many nodes. This, though, does not affect the running time of the algorithm, which is logarithmic in the number of tree nodes, regardless of the height of the tree.

Invariants and Rules

The rules are divided into two categories: Rules that are applied to nodes of degree 1 (pruning rules), and rules that are applied to nodes of degree 2 (shortcutting rules). For simplicity we will assume that the former is a leaf and the latter is an internal node with one child. This will not always be the case but the treatment is essentially the same.

Each node is examined once for rule application. Then, its incident edges are identified as being either (unconditionally) *included* into the new MST, *excluded* from it, or *conditionally included* in it. A conditionally included edge is one which will be included in the MST unless it is the MaxWE of another cycle which has not been yet considered. In the figures of this section, a conditionally included edge is marked with a star (*).

It is useful to observe that the edge with minimum weight incident to some node will always be included in the MST. Actually, many sequential and parallel algorithms are based on this observation (i.e. [Pri57, SP75, CLC82]). Edge inclusion makes use of this observation.

Another useful observation is that whenever some edge is found to correspond to the MaxWE of some cycle it can be removed from the tree without affecting the computation of the remaining graph. (Kruskal's MST algorithm makes use of this fact.) Edge exclusion is based on this observation.

Let $w : V \cup E \to R$ give the weights of the nodes and the edges. In the beginning of the algorithm the weight of a node v, if it exists, is the weight of the edge connecting v to z. Some nodes (fake or not) may not have an assigned weight. For the purposes of the algorithm, a weight of $+\infty$ is assumed on each of them. To resolve ties, we adopt the convention that the currently processed node has weight larger than its equally weighted neighbor.

Let us define the *provisional graph* to be the graph induced by the edges examined since the beginning of the algorithm. Note that whenever a leaf v is processed, three edges are examined: $(z, v), (v, p(v))$ and $(p(v), z)$, where $p(v)$ represents the parent of v. Whenever an internal node is processed, five edges are examined: $(z, v), (v, p(v)), (p(v), z), (v, u)$ and (u, z), where u represents v's only child. Each application of a rule on some node preserves the following three invariants:

Invariant 2 *The minimum spanning tree of the provisional graph has been computed.*

This will be true, because we will consider and break all cycles of the provisional graph by removing the MaxWE appearing in them. Furthermore, the rules will assure that the non-excluded edges of the provisional graph will constitute a MST. So, it will be the MST of the provisional graph because (i) connectivity is preserved, (ii) no cycles remain and (iii) the cycles have been broken by removing their MaxWE.

Invariant 3 *The weight $w[v]$, if it is defined, of some unprocessed node v corresponds to the max weight edge (MaxWE) on the unique path from z to v either via edge (z, v) or through edges contained in the provisional graph.*

As we said, application of a rule on some node v will examine and break all the small cycles that v is on, by removing their MaxWE. Some larger cycle sharing the MaxWE with a broken small cycle will also be broken. The next invariant describes how other larger cycles containing v are affected.

Invariant 4 *Let c be a larger cycle containing node v, and let e be its MaxWE. After the application of a rule on v, then either e has been deleted or there remains another cycle in which e is the MaxWE.*

Initially, since there are no processed nodes and the provisional graph is empty, the invariants hold. We now show how these invariants can be preserved under tree-contraction operations.

Pruning Rules

Consider a cycle involving leaf v, $p(v)$ and z. Let $w[(v, p(v))] = a$, $w[v] = b$ and $w[p(v)] = c$ (see Figure 4.8). The small cycle of length 3 they form can be broken in such a way that the invariants are preserved by removing one of the three edges involved. We consider the following cases:

$a = \max\{a, b, c\}$: Then, edge $(v, p(v))$ must be excluded from the new MST. When it is removed, the tree is broken into two subtrees (which are connected in the implicit graph via z). Moreover, the edge that corresponds to b has to be included in the MST, since this is the only way that v can be connected with the graph.

$b = \max\{a, b, c\}$: Then, node v can be included in the MST only via $(v, p(v))$. Therefore, this edge is added in the MST while edge (z, v) is excluded from it.

$c = \max\{a, b, c\}$: In this case the best way to include $p(v)$ is *not* via $(z, p(v))$, so this edge is excluded. It could be best to include $p(v)$ via v, but if so, we do not know it yet. But in this case the MaxWE that connects $p(v)$ with the provisional tree will have weight $\max\{a, b\}$. The edge corresponding to $\min\{a, b\}$ has to

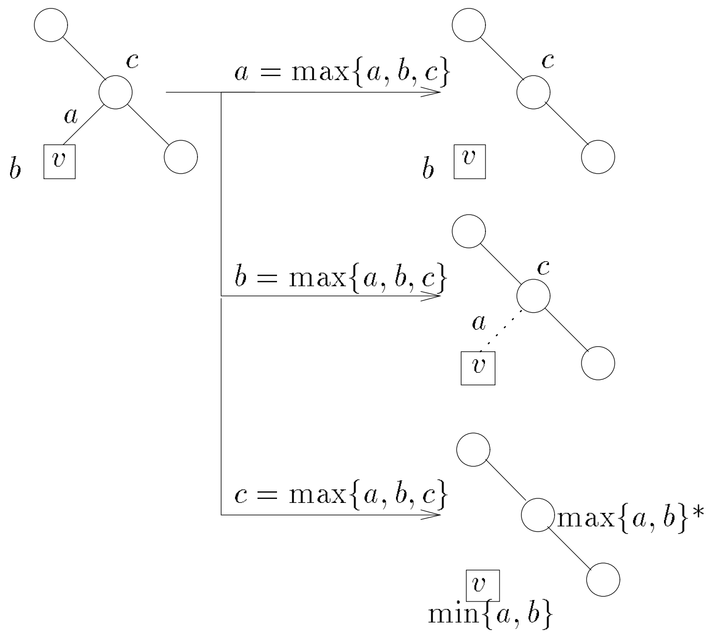

Figure 4.8: Pruning Rules for a Leaf. In this and in following figures, a conditionally included edge is marked with a star (*), an (unconditionally) included edge is dotted and we keep its weight letter. We erase an excluded edge along with its weight letter.

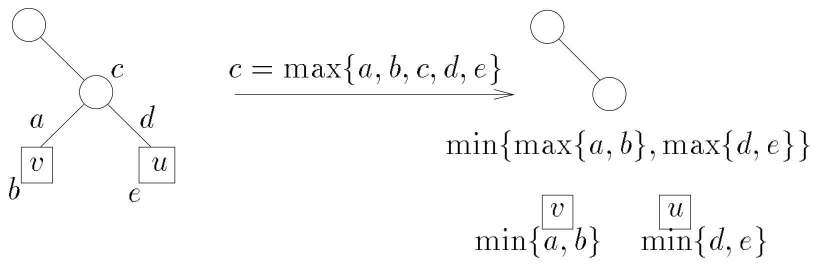
c
a
d
v
u
b
e
$c = \max\{a, b, c, d, e\}$
$\min\{\max\{a, b\}, \max\{d, e\}\}$
v
u
$\min\{a, b\}$
$\min\{d, e\}$

Figure 4.9: When v's sibling is a leaf.

be included in the MST. In the figure, this is indicated by labeling v with $\min\{a, b\}$. The edge corresponding to $\max\{a, b\}$ is conditionally included. We update $w[p(v)] := \max\{a, b\}$ to preserve Invariant 3. The effect of the update is the following: If $w[p(v)]$ is found to be a MaxWE of some other cycle later on, the edge corresponding to $\max\{a, b\}$ will not be included in the MST. Otherwise, if upon termination of the algorithm no rule has excluded $w[p(v)]$, it will be included.

Some special care must be taken in the case that v's sibling, say u, is also a leaf (Figure 4.9) and is processed at the same time. This, for example, can happen when the ACD scheduling method is used, which schedules v and u to be processed at the same time. Let $w[u] = e$ and $w[(u, p(v))] = d$. If $c = \max\{a, b, c, d, e\}$, then $(z, p(v))$ must be excluded and some precaution taken to avoid simultaneous reading or writing on $p(v)$ by the two processors. Moreover this cycle of length 4 $(z, v, p(v), u, z)$ must be broken, by excluding one of the four edges. Also $w[p(v)]$ must be updated to either $\max\{a, b\}$ or $\max\{d, e\}$, depending on which child was connected to the excluded edge. In particular, if $\max\{a, b\} > \max\{d, e\}$ then $\max\{a, b\}$ is excluded and $w[p(v)] := \max\{d, e\}$. Else if $\max\{d, e\} > \max\{a, b\}$ then $\max\{d, e\}$ is

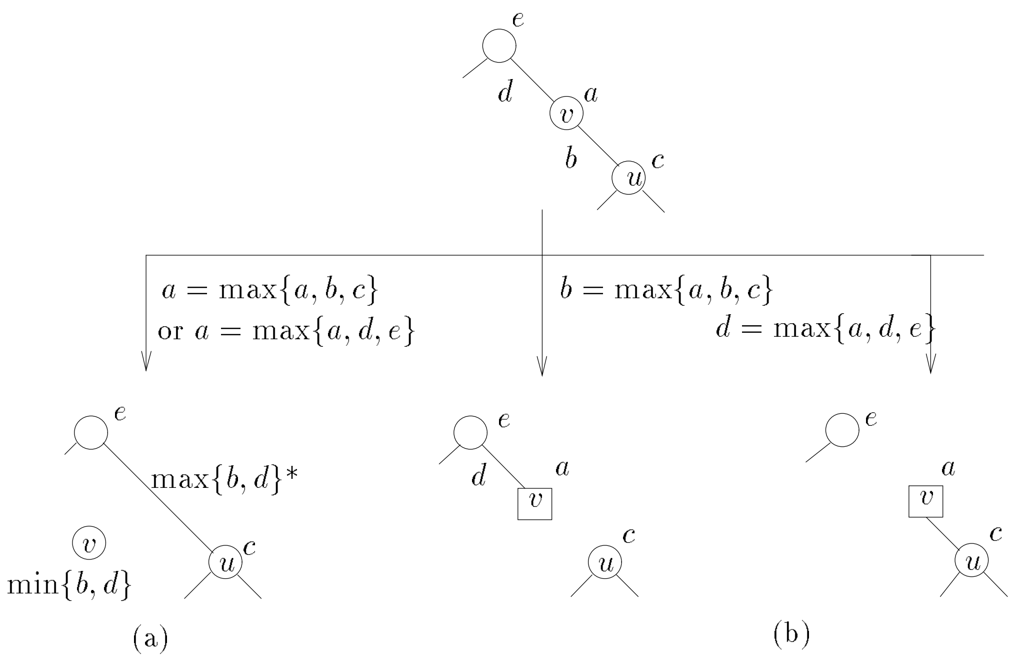

Figure 4.10: Shortcutting Rules: The first two cases.

excluded and $w[p(v)] := \max\{a, b\}$.

Shortcutting Rules

Let's consider a situation where v has only one child u. There are two possible small cycles involving v: A "lower" one (z, v, u, z) and an "upper" one $(z, v, p(v), z)$. We will describe how to break these cycles in such a way that the invariants are preserved. Let $w[v] = a$, $w[(v, u)] = b$, $w[u] = c$, $w[(v, p(v))] = d$ and $w[p(v)] = e$ (see Figure 4.10).

$a = \max\{a, b, c\}$ or $a = \max\{a, d, e\}$: Then, the best way to include v in the MST is either via $(v, p(v))$ or via (v, u). Thus, the edge with weight $\min\{b, d\}$ must be included in the MST. The other, equal to $\max\{b, d\}$ will be conditionally included, i.e. it will be included if and only if it is not the MaxWE in another cycle involving $z, u, p(v)$, and possibly other nodes. Therefore we can preserve the MaxWE-related information about these cycles by introducing a new shortcutting edge $(u, p(v))$ corresponding to the edge with weight $\max\{b, d\}$.

$b = \max\{a, b, c\}$ **or** $d = \max\{a, d, e\}$**:** In the first case edge (v, u) must be excluded from the new MST and can be deleted. In the second case edge $(v, p(v))$ must be excluded. In either case v becomes a "leaf", and it then can be processed using the rules for pruning. (Of course, in the second case we use the term "leaf" loosely: Node v is the parent of u but, nevertheless, it can be pruned as a leaf because it has degree one.)

$c = \max\{a, b, c\}$ **and** $e = \max\{a, d, e\}$**:** Then, (z, u) and $(z, p(v))$ must be excluded from the MST, and the edge corresponding to $\min\{a, b, d\}$ must be included. We have to update $p(v)$ and u so that they reflect this fact, preserving the invariants. In Figure 4.11 this case is presented, with the further assumption that $b > d$ for a simplified picture. We observe that there may exist (i) cycles involving edges (z, v), $(v, p(v))$ and edges in the upper part of the tree, (ii) cycles involving (z, v), (v, u) and edges in the lower part of the tree, and finally (iii) cycles going through $u, v, p(v)$ that do not involve (z, v). The update should preserve information about these cycles as dictated by Invariant 4. We update $w[p(v)] := \max\{a, d\}$ to account for cycles of the first type, and $w[u] := \max\{a, b\}$ to account for cycles of the second type, both corresponding to conditionally included edges. Finally, we introduce a conditionally included shortcutting edge $(p(v), u)$ with weight $\max\{b, d\}$ to account for cycles of the third type.

It is worth noting that in the last shortcutting rule not all three weight-updates are needed for the correctness of the algorithm. One of them can be omitted as

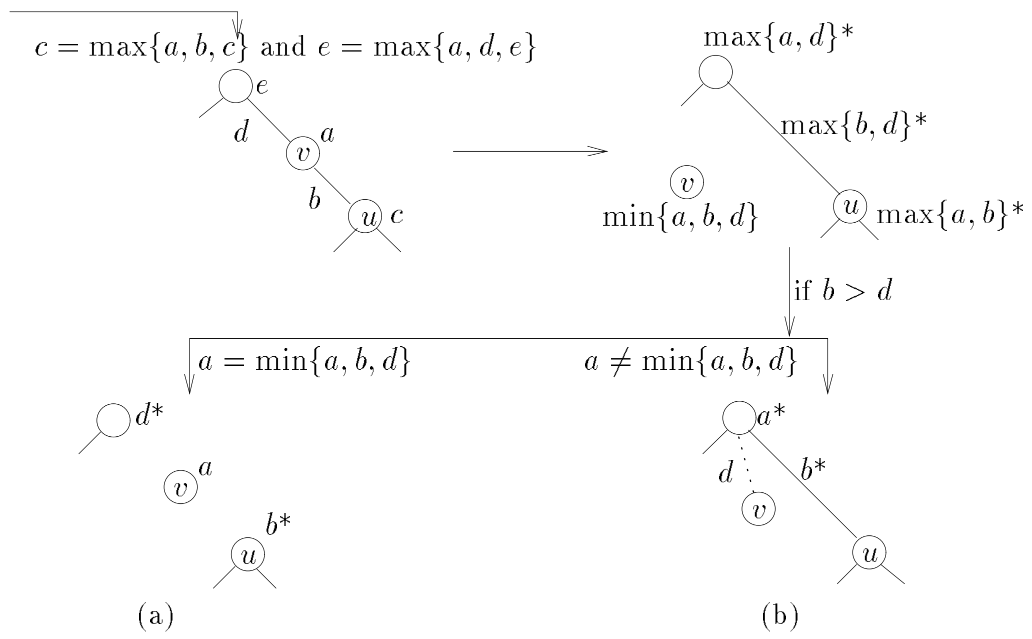

Figure 4.11: Shortcutting Rules: The third case. In (a) and (b) we show the simplified updating assuming $b > d$.

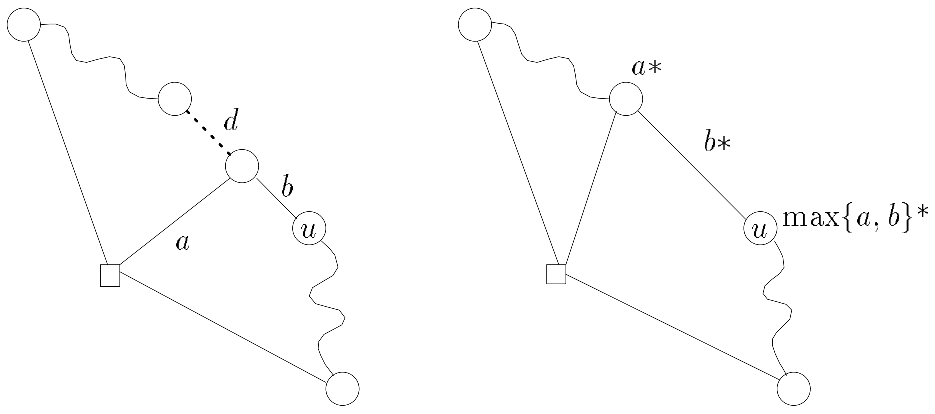

Figure 4.12: Assuming $d = \min\{a, d, b\}$: By omitting u's updating we have still preserved the MaxWE's of all three kinds of cycles.

redundant (Figure 4.11): If $a = \min\{a, b, d\}$, then we don't need the shortcutting edge because some MaxWE occurring in cycles of the third type will also be MaxWE in some other cycle of the first or the second type. So, updating $w[p(v)] := d^*$ and $w[u] := b^*$ is enough. Otherwise, if $a \neq \min\{a, b, d\}$, and say $b > d$, then updating of u is not needed, because information about cycles of all three types is preserved (see Figure 4.12). This observation can lead to a simpler implementation of the algorithm. Again, a weightless node is treated as if it where holding the maximum value.

Memory Access Conflicts

The fact that shortcutting may update both parent and child nodes creates a possibility of a write conflict. Consider the following situation: Let (see Figure 4.13a) nodes x, v, y satisfy $p(y) = v$ and $p(v) = x$, and assume that nodes x and y are to be processed simultaneously. Moreover, let's assume that shortcutting node x calls for updating child node v with $w[v] := a$, and shortcutting node y calls for updating parent node v with $w[v] := b$.

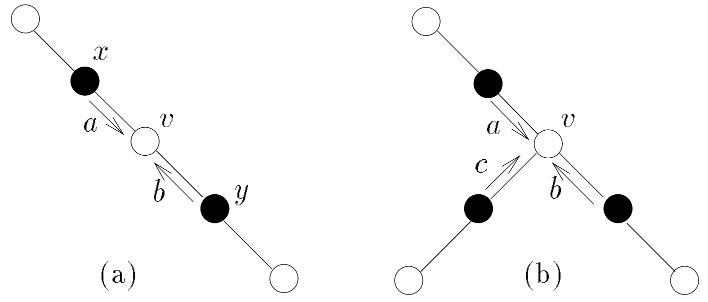

Figure 4.13: Memory Access Conflicts and how to resolve them. Two (a) and three (b) processors attempt to update v.

Recall that, according to Invariant 3, the weight of a node represents the MaxWE on the path from this node to z via processed neighbors. The write conflict actually represents a cycle involving nodes z, x, v, y, and possibly x's parent and/or y's child. Since just two processors may try to write on the same memory cell, the conflict can be avoided and the cycle behind it broken by removing the edge $\max\{a, b\}$ while updating $w[v] := \min\{a, b\}$.

This is the only kind of access conflict that can be created by the shunting schedule we have described. There are other schedules, though, which create a slightly more complicated conflict when updating a node with three values (see Figure 4.13b), say a, b, and c. Again, this can be resolved by breaking the MaxWEs of the three cycles behind the conflict. Therefore, in this case the algorithm should update $w[v] := \min\{a, b, c\}$ and should exclude the edges that correspond to the other two values.[3]

The above discussion verifies that application of the rules preserve the invariants, proving the following Lemma:

[3]The write conflict that occurs when two sibling leaves are pruned simultaneously (Figure 4.9) can be viewed as a special case of this conflict.

Lemma 14 *Application of a pruning or a shortcutting rule on some node v of the weighted graph preserves the invariants.*

$\square$

Correctness Lemma

We have described a set of rules that define a *prune* operation which removes the leaves of a tree, and a *shortcut* operation which removes nodes of degree two from the tree. Note that individual prunings and shortcuttings take $O(1)$ time to be performed.

Lemma 15 *When the rules are applied on the nodes of a binary weighted tree at times given by any valid tree contraction scheduling, they correctly produce the updated minimum spanning tree.*

Proof. As we said, a valid contraction schedule defines processing times on nodes having degree 1 or 2. So, only the prune and shortcut operations are needed, and they are provided by the rules. Moreover neighboring nodes are not scheduled at the same time, and any node may be accessed for updating its weight simultaneously by at most three processors. This conflict can be resolved easily as mentioned in the previous subsection. Therefore, the shortcutting and pruning operations can be done without confusion, and their application, according to Lemma 14, preserves the invariants. Thus, at the end of the schedule, the MST of the provisional graph, and therefore the updated MST of the implicit graph, has been computed. $\square$

4.5.5 The Algorithms

As we said earlier, the vertex updating problem has a lower bound of $\Omega(n)$ sequential time. A sequential algorithm for some problem of size n is *optimal* if it runs in time that matches a lower bound for the problem to within a constant factor. The sequential and parallel algorithms we present here are all optimal. [4] We discuss now the sequential and parallel algorithms that can be derived using the rules.

Theorem 5 *There are optimal sequential and parallel algorithms that solve the MST vertex updating problem based on the rules presented above. The sequential algorithms run in $O(n)$ time and the parallel algorithms run on a binary weighted tree in $O(n/p)$ time using $p \le n/\lg n$ EREW PRAM processors.*

Proof. A. THE OPTIMAL PARALLEL ALGORITHMS. In Section 3.4 we presented two valid tree-contraction schedulings that can be used to contract a binary tree in logarithmic time. Using the pruning and shortcutting rules with the schedulings we arrive at two parallel algorithms that solve the MST vertex updating problem. Since, in either scheduling, at most three processors may attempt to operate on the same data item at a certain time, we can schedule the operations in such a way that no read or write conflicts occur. This observation along with Lemma 15 completes the proof for the parallel case.

B. THE OPTIMAL SEQUENTIAL ALGORITHMS. The rules we presented do not depend on the particular order in which the nodes of the tree are removed. So, different removal sequences of the nodes yield different algorithms and, in particular, we can derive sequential algorithms from the parallel ones. The running time of these algorithms differ only by a constant. We present here some of these algorithms:

Remove on the fly. Use depth first search to visit the nodes of the tree. Every time a node of degree 1 or 2 is encountered, process it using pruning or shortcutting rule, respectively. Each node will be visited at most twice (on the way down the tree and

[4] Optimality of parallel algorithms is defined in Section 2.4.

on the way up the tree), so the algorithm runs in $O(n)$.

Postorder. Another algorithm simpler to implement but with the same time-bound is the following: Visit the nodes of the weighted tree in postorder (using, for example, depth-first-search). A node is processed after all its children have been processed, so only the pruning rules are needed. Since each node is processed at most once, we have an $O(n)$ sequential algorithm. [5]

Shunting. A third algorithm follows from the parallel scheduling of [ADKP89]. Number the leaves of the tree in a left-to-right manner and place their addresses in an array of length $\lfloor n/2 \rfloor + 1$ as dictated by the schedule. Then visit the array performing shunts on the nodes of the array.

ACD. Visit the leaves of the tree computing their centroid levels and performing centroid decomposition [CV88]. Bucket-sort the scheduled numbers; place them in an array of length n and then process the nodes of the array. □

Remark 1. This list does not exhaust the possible sequential and parallel algorithms that can be based on the rules presented, but includes only the simpler ones. We should note that other tree-contraction techniques (like the ones presented in [GR86] and [GMT88]) lead to different algorithms with the same bounds. (The shunting scheduling is shown in Figure 4.14).

Remark 2. The shunting method described in [ADKP89] requires that the root of the tree not be shunted until the end. This is needed mainly for the expression tree evaluation algorithm. In our case, shunting the root is permitted since there is no top-to-bottom information to be preserved.

[5] As a matter of fact, this algorithm is similar to the one given in [CH78], but the pruning rules make it easier to describe and understand.

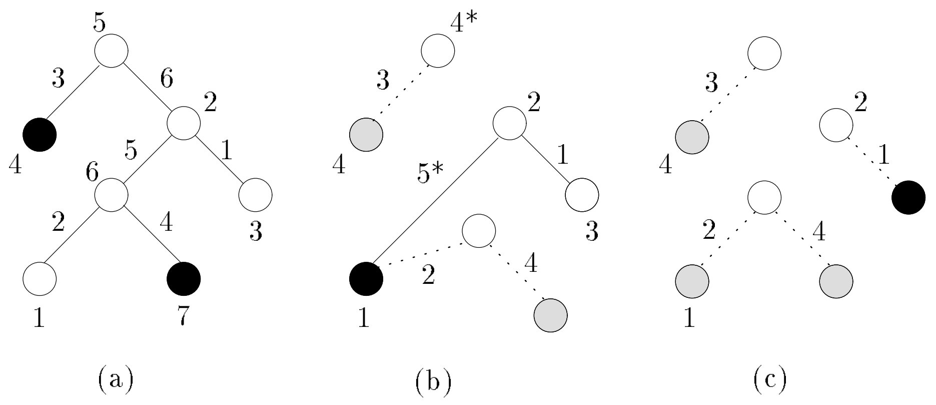

Figure 4.14: Run of the Parallel Algorithm using SHUNTING on the tree of Figure 1. Dotted edges are included in the MST, deleted edges are excluded from it. (a) In black are the first two processed vertices. (b) Third step. (c) Fourth and final step.

REMARK 3. Tree contraction algorithms often require a second "expansion" phase after the contraction phase. The algorithms we present here do not need a second phase because of Invariant 2: At the end of the tree contraction phase the new MST has been computed.

4.5.6 Binarization

We now discuss how a general tree can be transformed into a binary tree using the procedure *binarize*:

Each node v having k children $v_1, \cdots, v_k$ is represented by a *right path* (Figure 4.7) composed of $v = u_0$ and $k-2$ fake nodes $u_1, \cdots, u_{k-2}$, so that node u_j is the right child of u_{j-1} and the parent of u_{j+1}. Node v_i, $i = 1, \cdots, k-1$ becomes the left child of node u_{i-1} and v_k the right child of u_{k-2}. We assign weights of $-\infty$ to the edges of the right path, so they can not be excluded by the rules. The fake nodes have no assigned weight (which is treated as the maximum value weight by the algorithm) because they are only introduced to facilitate the processing order of v's children. Of course, the real nodes v and the v_i's keep their weights.

Recall that the shunting scheduling accepts as input a regular binary tree. When using this technique, the binarization should be extended to handle nodes v with only one child in the input tree. For each of them, a second child v' is introduced, for which $w[v'] = \infty$ and $w[(v, v')] = -\infty$.

Theorem 6 *There are logarithmic-time optimal parallel algorithms solving the MST vertex updating problem on a rooted tree.*

Proof: The binarized graph has exactly the same number of cycles as the given graph, and at the end of the algorithm the edges composing the right path are always included into the new MST. Therefore, the solution of the binarized problem shows a corresponding unique and unambiguous solution to the general problem. □

Similar binarization techniques to the one described have been used in [VD86, CV88, ADKP89]. Another technique [JM88] "plants" a balanced binary tree over the v_i's with v as the root. The internal nodes and the internal edges have weights as those in the right path in the previously described technique. Both constructions require the *list ranking* algorithm [CV86a] which runs within the desired bounds. (Actually, in [VD86] the Eulerian tour technique is used which has the list ranking procedure as a subroutine.)

4.6 On the Edge Updating Problem

The edge updating problem of a MST can be defined as follows: We are given a graph $G = (V, E)$ and its MST $T = (V, E')$ as input along with the weight function $w : E \rightarrow R$. Also we know that one of the edges e changes weight, and we want to compute the new MST T'. A sequential algorithm for this problem is given in [Fre83] with $O(\sqrt{m})$ running time.

It is obvious that if e is a tree edge and decreases its value, or if e is not a tree edge and increases its value, then $T' = T$. In the other two cases we have:

A TREE EDGE INCREASES. A simple solution is to consider the two connected components that e divides the graph into, and to find the minimum weight edge connecting them. So, a simple algorithm for it, is:

1. Preorder the tree using the Eulerian tour technique [TV85].
2. Using the preorder numbering, divide V into the two subsets V_1 and V_2 created by removing e from T.
3. Put all edges connecting some vertex in V_1 to some vertex in V_2 in an array of length $m = |E|$ and compute the minimum.

The running time of this algorithm is $O(\lg n)$ using $m/\lg n$ EREW PRAMs.

A NONTREE EDGE DECREASES. Consider the cycle it creates in the tree when this edge is added, and remove the MaxWE in it using a variation of the list ranking algorithm in which the rank of a list element l is defined to be the maximum of the values appearing in the elements following l in the list. The running time is $O(\lg n)$ using $n/\lg n$ EREW PRAMs [CV86a].

This problem is a special case of the *edge insertion in a MST* problem for which [CH78] have shown a simple reduction to the vertex insertion problem. Our algorithm can be used to solve this problem in parallel as well.

4.7 On the Multiple Vertex Updates Problem

4.7.1 Introduction

We define the problem of *multiple vertex updates* of a MST as follows: Let $G = (V_G, E_G)$ be a weighted graph on n vertices and m weighted edges and $T = (V_G, E_T)$ be its MST. Suppose G is augmented with $k = |V_k|$ new vertices that are connected to V_G by $kn = |E_k|$ new weighted edges, but they are not connected among themselves. We are asked to compute the new MST T'.

We will prove the following Theorem:

Theorem 7 *The multiple updates MST problem can be solved in parallel in time $O(\lg n \lg k)$ using $nk / \lg n \lg k$ EREW PRAM processors.*

The problem of multiple vertex updates was considered in [Paw89] where a parallel algorithm is presented. It runs in $O(\lg n \lg k)$ time using nk CREW PRAM processors. We will show how our solution for the (single) vertex update problem can be used to achieve a better solution for the multiple updates problem.

4.7.2 Optimality

A sequential algorithm can solve the problem in time $O(kn)$, by solving k single update problems sequentially. Another approach would be to compute the MST of the augmented graph $G' = (V_G \cup V_k, E_T \cup E_k)$ from scratch. Again, the edges of G that are not in the given MST do not need to be considered.

The augmented graph G' can be sparse or dense depending on the value of k. The best algorithms to compute the MST of a graph $G = (V, E)$ sequentially run in time $O(m \lg \lg n)$ for sparse graphs [Yao75, CT76] and in time $O(n^2)$ for dense graphs (using Prim's well known algorithm [Tar83, page 75]). Assuming that the number of edges connecting the new vertices to the tree is $O(kn)$, the k single updates solution is preferable, since it is simpler and, for sparse graphs, asymptotically faster.

The solution we present solves the problem in $O(\lg n \lg k)$ time using $nk/\lg n \lg k$ EREW PRAM processors. It is therefore optimal for graphs having $O(kn)$ edges and also uses a weaker model of parallel computation than the one used in [Paw89].

4.7.3 The algorithm

Our solution follows in general the solution presented in [Paw89] but in certain parts uses different implementation techniques to achieve the tighter time and processor bounds.

The algorithm consists essentially of three parts.

1. Make k copies of T, and solve k update MST problems in parallel.
2. Combine the MSTs of the k solutions into a new graph G_z. This graph may contain cycles. Transform it to an equivalent bipartite graph G_b.
3. Solve the bipartite MST problem on the graph G_b.

We will show that each of these parts can be implemented within the desired time-processor bounds.

Solving k Updating Problems

Making k copies of T can be done efficiently as an application of Brent's scheduling principle, which we explained in Section 3.2: Making k copies of T requires $O(kn)$ operations and can be done in constant time if kn processors are available. Therefore, it can be done in $O(\lg n \lg k)$ time using $kn/(\lg n \lg k)$ processors.

According to Theorem 1, a single updating of a MST can be done in $O(n/p)$ time when p processors are available. Here we have k problems to solve, each of size n. Allocating $n/(\lg n \lg k)$ processors per problem, it takes $O(\lg n \lg k)$ time to solve each problem in parallel.

Creating the Bipartite Graph

Next, we have to combine the k solutions found in the first part into a new graph G_z which in turn is transformed to an equivalent bipartite graph G_b. By *equivalent* here, we mean that *there is a cycle in G_b if and only if there is a cycle in G_z*. Graph G_z will never be explicitly created. It is only defined for the sake of description.

Consider the MST of some solution $T_i = (V \cup \{z_i\}, E_i)$. Then, E_i consists of edges (v, w) that belonged to the old MST T, along with new edges of the form (z_i, v). An important observation is that *if some edge (v, w) of the old MST does not appear in all k solutions, it will not be included in the final MST.* This is so, because edge (v, w) was the MaxWE of some cycle in one of the subproblems and thus must be excluded. So, $G_z = (V \cup \{z_1, \cdots, z_k\}, E_z)$ is composed of original edges (v, w) that appear in *all k* solutions, along with edges of the form (z_i, v), $\forall i \in \{1, \cdots, k\}$. It is easy to see that the formation of G_z can be done within the desired bounds, because there are at most $n - 1$ such edges per solution to examine.

We can view G_z as a collection of subtrees C_j of the old MST that are held together by the z_i's (see Figure 4.15). Every cycle in G_z can be viewed as starting at some z_i, then entering subtree C_j at a node v_e and visiting some of its nodes, then exiting through a node v'_e and visiting z_l, etc, until returning back to z_i (Figure 4.16). The nodes v_e that are adjacent to some z_i are called *e-nodes*.

The transformed graph $G_b = (V_b, \{z_1, \cdots z_k\}, E_b)$ has a set of vertices V_b which

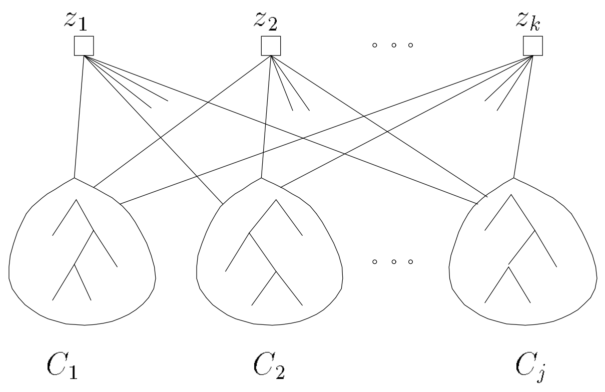

Figure 4.15: The graph G_z results from putting together the k solutions. The figure points out G_z's bipartite nature.

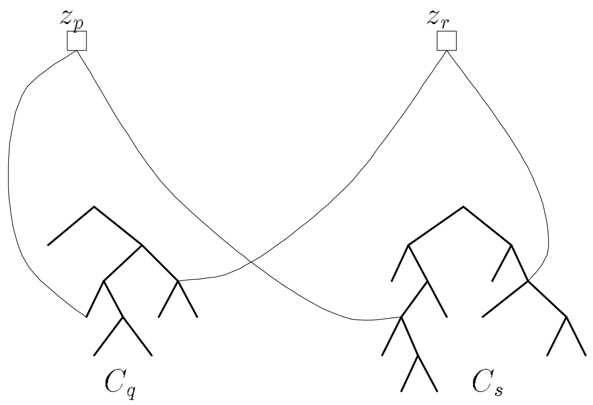

Figure 4.16: Cycles in G_z. They are composed of alternative visits to z_i's and to tree components. Vertices that are connected to z_i's are called *e-vertices*. In this picture a cycle of length 4 is shown.

contains one vertex v for each e-node v_e of G_z. Consider a path from z_i to some e-node v_e, and let x be the MaxWE on this path. Then edge $(z_i, v) \in E_b$ corresponds to this path and has cost equal to x's cost. The algorithm needs, therefore, to compute all MaxWE on all paths from z_i to e-nodes v_e. This is done as follows:

First, all e-nodes v_e are recognized, and then we root each of the T_i's at z_i. Both of these operations can be done within the desired bounds. Now we have to compute the MaxWEs on the paths between the z_i's and the e-nodes. So, we have to solve k instances of the following problem:

All Distances to Root (ADR). Given a (regular binary rooted) tree with n nodes having weights associated with its edges, find the MaxWE for each path from a node to the root in $O(n/p)$ time using $p \leq n/\lg n$ EREW PRAM processors.

A sequential algorithm for the problem uses depth first search and runs in linear time. The parallel algorithm reduces the problem to tree-contraction as follows: Associate a function $mwe(v)$ for each node v, where $mwe(v) = \max\{(v, p(v)), mwe(p(v))\}$ if v is not the root and $mwe(root) = \emptyset$. Use tree contraction to contract the tree and then tree expansion[6] to compute $mwe(v)$ for all v. We can come up with simple rules which are given in Figure 4.17.

The multiple ADR problem. Given k (regular binary rooted) weighted trees each one with n nodes, compute the ADR problem on each of them in time $O(\lg n \lg k)$ using $nk/\lg n \lg k$ processors.

The solution is a simple application of the previous algorithm. We associate $n/\lg n \lg k$ processors per tree and compute the problem in time $O(\lg n \lg k)$.

[6]The expansion phase is needed because the definition of the function is top-down.

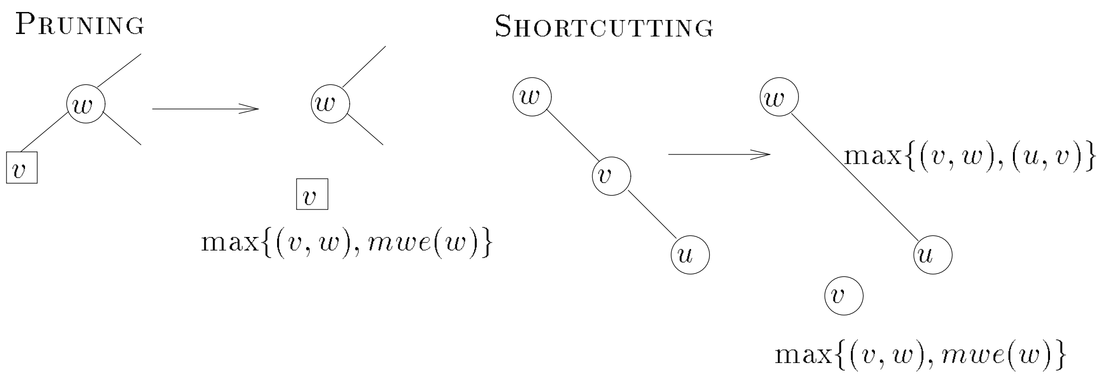

Figure 4.17: Prunning and Shortcutting Rules for the CADR problem.

The Bipartite MST problem

For the third part of the algorithm we need the following definition of the bipartite-MST problem: Let $G = (V_k, V_n, E)$ be a weighted bipartite graph, where $|V_k| = k$ and $|V_n| = n$, $k \leq n$. We want to compute its MST.

Lemma 16 *The bipartite-MST problem can be solved in $O(\lg n \lg k)$ parallel time using $kn/(\lg n \lg k)$ processors.*

Proof. The algorithm that we use is a well-known algorithm whose main idea is attributed to Borůvka [Tar83, page 73] and was described in its parallel form in [CLC82]. The analysis, though, and the time-processors bounds for the bipartite-MST problem, are new.

First, let us give some definitions. A *pseudotree* is a digraph in which each node has outdegree one. A pseudotree has, at most, one (simple) cycle. A *pseudoforest* is a graph whose components are pseudotrees. The algorithm consists of a number of stages. In each stage, each vertex v selects the minimum weight edge (v, w) incident to it. This creates a pseudoforest of vertices connected via the selected edges. In this case the cycle of each pseudotree involves only two vertices, so it can be easily transformed into a tree. Next, each tree is contracted to a star using pointer-doubling, and vertices in the same component are identified with the root of the star.

A crucial observation here is that, after the first stage, there will be no more than k vertices in the resulting graph and the problem can be solved in $O(\lg^2 k)$ time using $k^2/\lg^2 k$ processors. So, we only have to show that the first stage can be performed within the desired bounds.

As we said, the first stage consists of finding the minima of $O(n)$ sets of vertices, each with cardinality $O(k)$ and then to reduce the $O(k)$ resulting pseudotrees of height $O(n)$ to stars. For the first part, Brent's technique applies. For the second part, we use the optimal list-ranking technique of [CV86a].

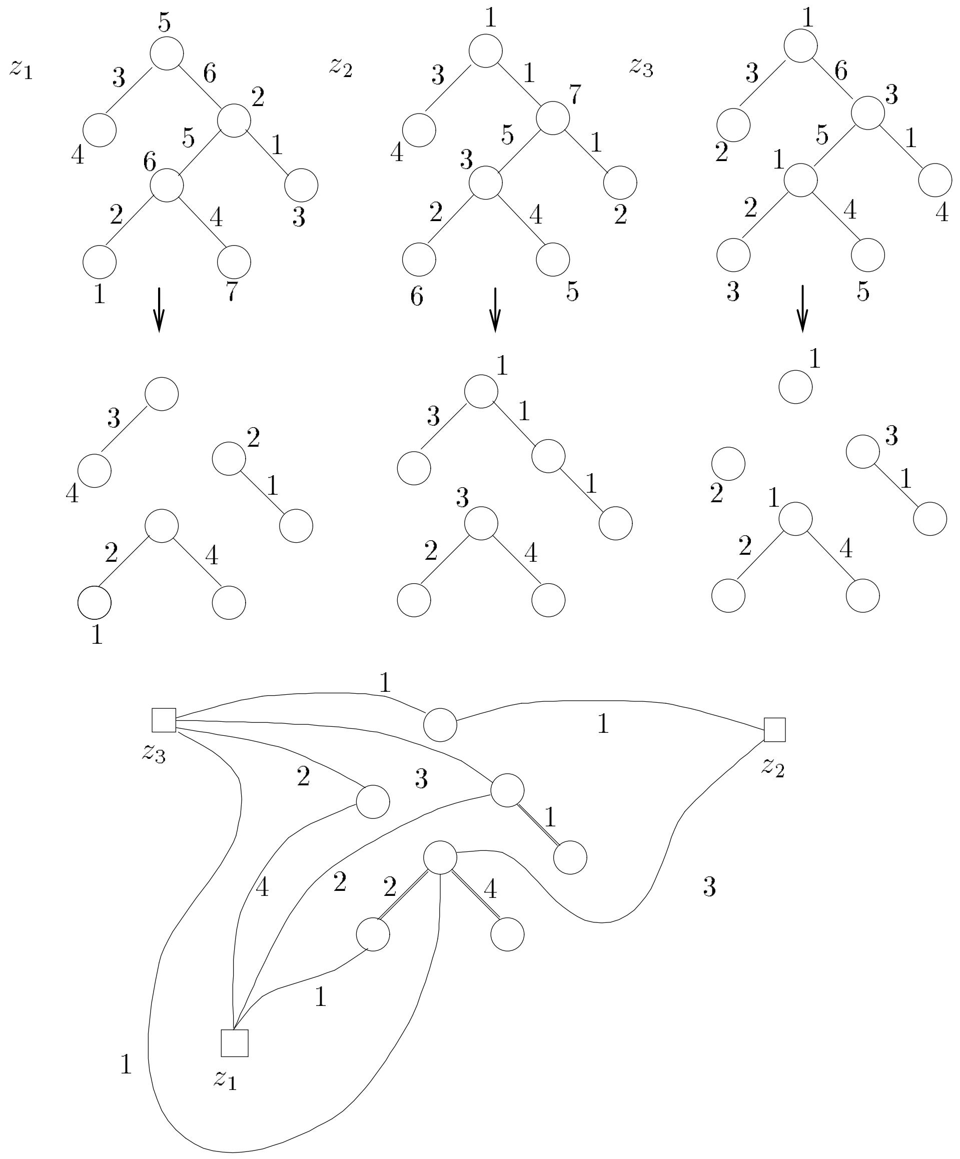

Figure 4.18: Updating of a MST with 3 new vertices. In the lower part of the figure the graph G_z is shown.

Chapter 5

Conclusion and Open Problems

5.1 Conclusions

We have presented new parallel algorithmic techniques for the pseudotree contraction and the edge-list augmentation problems. We also discussed some of the existing parallel connectivity algorithms and argued that what they are missing is a scheduling that carefully balances the work-performed versus progress-made ratio. Then we presented the growth-control schedule that achieves this objective. These techniques have been used to design new parallel algorithms for the minimum spanning tree problem [JM91c].

The major contribution of this thesis is a simple connectivity algorithm for the CREW PRAM model of parallel computation. This algorithm works in $O(\lg^{3/2} n)$ time and narrows the gap of the performance between several CREW and CRCW PRAM graph algorithms by a factor of $\lg^{1/2} n$.

Other algorithms having running times that depend on a connectivity algorithm include the Euler tour on graphs [AV84, AIS84], biconnectivity [TV85], ear decomposition [MSV86, MR86] and its applications on 2-edge connectivity, triconnectivity, strong orientation, s-t numbering etc. The surveys by Karp and Ramachandran [KR90], Eppstein and Galil [EG88], and Vishkin [Vis91] include more details on these problems.

We should also mention that, with a minor modification, the algorithm works on the weaker CREW PPM (Parallel Pointer Machine) model [GK89]. The modification is to substitute the sorting routine we use at the end of each phase by the optimal sorting algorithm of Goodrich and Kosaraju [GK89]. In the PPM model, the memory can be viewed as a directed graph whose vertices correspond to memory cells, each having a constant number of fields.

Other contributions of the thesis are the optimal parallel and sequential algorithms for the vertex updating of a minimum spanning tree problem and the algorithm for the multiple updates of an MST problem. The algorithms for the former problem run optimally in $O(\lg |V|)$ time using $\frac{|V|}{\lg |V|}$ EREW PRAM processors, also leading to optimal sequential algorithms. The algorithm for the latter problem runs in $O(\lg k \cdot \lg |V|)$ parallel time using $\frac{k \cdot |V|}{\lg k \cdot \lg |V|}$ EREW PRAM processors.

5.2 Open Problems

We will now discuss some of the open questions that arise from our work.

The connectivity algorithm is an efficient but not optimal algorithm. Two natural problems arising are how to reduce the number of processors employed by the algorithm to $(n+m)/\lg n$ and/or how to implement it using the weaker EREW PRAM model. For both problems, the CR rules must be modified. Since these are essentially pointer jumping rules, techniques analogous to those used in [AM91, CV89] and in [NM82] may apply.

The $O(\lg^{3/2} n)$ time bound in the algorithm is one that does not arise often in parallel algorithms, and so it looks possible that it can be reduced. We remark that Wyllie [Wyl81] and Shiloach and Vishkin [SV82] conjecture that a $O(\lg n)$ bound can not be achieved by the CREW PRAM model. Even if this is true, a $O(\lg^{1+\epsilon} n)$ bound may be possible. One way to achieve this result is to remove efficiently the null edges that are generated when the components grow in size. A bucket-sorting algorithm with running time $o(\lg n)$ would suffice to improve the time bound, but it does not seem possible. Probably the best way to go about it, is to create a better design of the duration of phases and stages and/or to apply growth-control arguments to null edge removal procedures.

Another interesting open problem is to examine the average case running time for Algorithm 1: It is natural to assume that null edges will not always be found in the beginning of some component's edge-list, therefore some component may be able to make progress even if it has not removed all of its null edges.

We should also mention that the techniques presented may apply to directed graphs. Most of the directed graph algorithms use matrix multiplication techniques which requires almost n^3 processors, making them highly impractical.

The vertex updating for a minimum spanning tree showed an application of tree contraction. It is interesting to find other tree problems for which the tree contraction technique applies. Moreover, a relevant question is the following: Can one come up with a general way of defining pruning and shortcutting rules for tree-contraction in parallel, for those tree problems that are solved sequentially using depth-first-search? Finally, developing simple and fast SCAN algorithms for tree contraction would lead to implementation of several PRAM algorithms on machines which are described by the SCAN model.

Chapter 6

Appendix

6.1 On The SV Algorithm

Consider the following connectivity algorithm:

for $\lg n$ phases **in parallel do**

1. The representatives of the components hook, if they can. The hooking is done by having all edges (r, x) that are adjacent to some component representative r compete to be chosen by r.

2. The pseudotrees are contracted for a while by a constant number of CR rules application.

3. Representatives of new components are recognized. Edges rename their endpoints according to the representative's names. Internal edges become idle for the remaining of the run of the algorithm.

As one can see, this algorithm solves correctly the connectivity problem in $O(\lg n)$ steps using $n + m$ ARBITRARY CRCW PRAM processors. The proof is similar to the one given in [SV82].

Bibliography

[AAK89] A. Aggarwal, R.J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. In *Proc. of the 21st Annual ACM Symposium on the Theory of Computing*, pages 297–308, May 15-17 1989.

[ADKP89] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

[AIS84] B. Awerbuch, A. Israeli, and Y. Shiloach. Finding Euler circuits in logarithmic parallel time. In *Proc. 16th Annual ACM Symposium on Theory of Computing*, pages 249–257, 1984.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[AM91] R.J. Anderson and G.L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–868, 1991. Also: Proc. 3rd AWOC 1988.

[AS87] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36:1258–1263, 1987.

[AV84] M Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29:330–337, 1984.

[Ber85] C. Berger. *Graphs.* North-Holland, Amsterdam, The Netherlands, 2nd edition, 1985.

[Bor26] Otakar Borůvka. O jistém problému minimálním. *Práce Mor. Přírodověd. Spol. v Brně (Acta Societ. Scient. Natur. Moravicae)*, 3:37–58, 1926. (In Czech.) Translation of the main parts of this paper appears in [GH85].

[Bre74] R. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[CDR86] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal of Computing*, 15(1):87–97, February 1986.

[CH78] F. Chin and D. Houck. Algorithms for updating minimal spanning trees. *Journal of Computer and System Sciences*, 16(12):333–344, 1978.

[CLC82] F.Y. Chin, J. Lam, and I-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 25(9):659–665, September 1982.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, August 1988.

[CR73] S.A. Cook and R.A. Reckhow. Time-bounded random access machines. *Journal of Computing Systems Science*, 7:354–375, 1973.

[CT76] D. Cheriton and R.E. Tarjan. Finding minimum spanning trees. *SIAM Journal of Computing*, 5:724–742, 1976.

[CV86a] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[CV86b] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms. In *18th Annual ACM Symposium on Theory of Computing*, 1986.

[CV88] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[CV89] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[Den86] P.J. Denning. Parallel computing and its evolution. *Communications of the ACM*, 29(12):1163–1167, December 1986.

[Eck77] D.M. Eckstein. Simultaneous memory accesses. Technical Report TR-79-6, Computer Science Dept, Iowa State Univ., Ames, IA, 1977.

[EG88] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Annual Review of Computer Science*, 3:233–283, 1988.

[Fre83] G. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proc. of the 15th Annual ACM Symp. on Theory of Comput.*, pages 252–257, 1983.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. of the 10th Ann. ACM Symposium on the Theory of Computing*, pages 114–118. ACM, 1978.

[Gaz91] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, pages 1046–1067, 1991. Also: Proc. 27th FOCS 1986.

[GH85] R.L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, January 1985.

[GK89] M.T. Goodrich and S.R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. In *Proc. 30th IEEE Symposium on Foundations of Computer Science*, pages 190–195, 1989.

[GMT88] H. Gazit, G. L. Miller, and S. H. Teng. Optimal tree contraction in the EREW model. In *Concurrent Computations: Algorithms, Architecture, and Technology*, New York, 1988. Tewksbury and Dickinson and Schwartz (editors), Plenum Press.

[Gol77] L. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2):25–29, Summer 1977.

[GPS87] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparce graphs. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[GR86] A. Gibbons and W. Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. In *Proc. of Symposium on*

Foundations of Software Technology and Theoretical Computer Science, volume 6, pages 453–469. Springer Verlag, 1986.

[Hag87] T. Hagerup. Towards optimal parallel bucket sorting. *Informaton and Computation*, 75:39–51, 1987.

[HCS79] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate. Computing connected components on parallel computers. *Communications of ACM*, 22(8):461–464, August 1979.

[He86] Xin He. Efficient parallel algorithms for solving some tree problems. *24th Allerton Conference on Communication, Control and Computing*, pages 777–786, 1986.

[JM88] H. Jung and K. Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Information Processing Letters*, 27(5):227–236, April, 28 1988.

[JM91a] D.B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for the CREW PRAM. In *Proc. of 32nd IEEE Symposium on the Foundations of Computer Science (FOCS'91)*, October 1991.

[JM91b] D.B. Johnson and P. Metaxas. Optimal parallel and sequential algorithms for the vertex updating problem of a MST. Technical Report PCS-TR91-159, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, 1991.

[JM91c] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. Technical Report PCS-TR91-166, Department of

Mathematics and Computer Science, Dartmouth College, Hanover, NH, November 1991.

[KD88] S. Rao Kosaraju and Arthur L. Delcher. Optimal parallel evaluation of tree-structured computations by raking. *Aegean Workshop on Computing*, pages 101–110, 1988.

[Knu80] D.E. Knuth. *The art of Computer Programming: Vol. 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1980.

[KR90] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook for Theoretical Computer Science*, 1:869–941, 1990.

[Kuc82] L. Kucera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14(2):93–96, 1982.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of ACM*, 27:831–838, 1980.

[LM86] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel graph algorithms. In *International Conference on Parallel Processing*, pages 861–868, 1986.

[MR85] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th IEEE Symposium on Foundations of Computer Science*, pages 478–488, 1985.

[MR86] G.L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Technical report, MSRI, Berkeley, CA, 1986.

[MR89] G. L. Miller and J. H. Reif. Parallel tree contraction. Part 1: Fundamentals. *Advances in Computing Research*, 5:47–72, 1989. Part in 26th FOCS 1985.

[MSV86] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and s-t numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[NM82] D. Nath and S.N. Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Information Processing Letters*, 14(1):7–11, March 1982.

[Paw89] S. Pawagi. A parallel algorithm for multiple updates of minimum spanning trees. In *International Conference on Parallel Processing*, volume III, pages 9–15, 1989.

[Phi89] C.A. Phillips. Parallel graph contraction. In *1st Symposium on Parallel Algorithms and Architectures*, pages 148–157, 1989.

[PR86] S. Pawagi and I.V. Ramakrishnan. An $O(\log n)$ algorithm for parallel update of minimum spanning trees. *Information Processing Letters*, 22(5):223–229, April, 28 1986.

[Pri57] R.C. Prim. Shortest connection networks and some generalizations. *Tech. Journal, Bell Labs*, 36:1389–1401, 1957.

[Rei85] J. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.

[SP75] P.M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing*, 4(3):375–380, September 1975.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[Tar72] R.E Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 1972.

[Tar83] R.E. Tarjan. *Data Structures and Network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania 19103, 1983.

[TV85] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[Val90] L.G. Valiant. General purpose parallel architectures. *Handbook of Theoretical Computer Science*, 1:945–971, 1990. Edited by J. van Leewen, Elsevier Science Publishers B.V.

[VD86] P. Varman and K. Doshi. A parallel vertex insertion algorithm for minimum spanning trees. In *13th ICALP, Lecture Notes*, volume 226, pages 424–433, 1986.

[Vis83] U. Vishkin. Implementation of simultaneous memory address access in models that forbid it. *Journal of Algorithms*, 4:45–50, 1983.

[Vis91] U. Vishkin. Structural parallel algorithms. Technical Report UMIACS-TR-91-53, CS-TR-2652, University of Maryland, College Park, Maryland 20742, April 1991.

[Wil85] R.J. Wilson. *Introduction to Graph Theory.* Longman, Inc., New York, 3rd edition, 1985.

[Wyl81] J.C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, August 1981.

[Yao75] A. Yao. An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters*, 4:21–23, 1975.