Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques

Uzi Vishkin *

February 21, 2002

Contents

1. Preface - A Case for Studying Parallel Algorithms

1.1. Summary

We start with two kinds of justification, and proceed to a suggestion:

- *Basic Need.* Fundamental physical limitations have led computer designers to introduce an increasing amount of parallelism into essentially all the machines that they are building. It appears that one of the most effective and promising ways for taking advantage of machine parallelism is by programming them based on parallel algorithms. Since most computer science and engineering graduates work in various levels of computer programming, they will eventually have to be able to design parallel algorithms; even a more modest goal of expressing parallelism which is inherent in an existing algorithm requires non-trivial understanding. A computer science/engineering curriculum should attempt to prepare its graduates for for a life time career (which may span over more than forty years), and therefore must include some exposure to parallel algorithmics.
- *Methodology - the system of methods and principles is new.* Parallelism is a concern that is missing from "traditional" algorithmic design. Unfortunately, it turns out that most efficient serial algorithms become rather inefficient parallel algorithms. The design of parallel algorithms requires new paradigms and techniques. These notes attempt to provide a short guided tour of some of the new concepts at a level and scope which make it possible for inclusion as early as in an undergraduate curriculum in computer science and engineering.
- *Suggestion - where to teach this?* We suggest to augment the serial programming paradigm in the computer science and engineering basic curriculum (including data-structures and algorithms) to reflect this trend. Since parallel algorithmics involves "first principles" that cannot be derived from other areas, we further suggest to include it in the standard curriculum for a bachelor degree in computer science and engineering, perhaps as a component in one of the courses on algorithms and data-structures.

To clarify the above statements on the basic need, we articulate two notions: machine parallelism and algorithm parallelism.

Machine parallelism - Each possible state of a machine can be described by listing the contents of all its *data cells*, where data cells include memory cells and registers. For instance, pipelining with, say s, single cycle stages, may be described by associating a data cell with each stage; all s cells may change in a single cycle of the machine. More generally, a transition function may describe all possible changes of data cells that can

occur in a single cycle; the set of data cells that change in a cycle define the *machine parallelism* of the cycle; a machine is *literally serial* if the size of this set never exceeds one. Machine parallelism comes in such forms as: (1) processor parallelism (a machine with several processors); (2) pipelining; or (3) in connection with the Very-Long Instruction Word (VLIW) technology, to mention just a few.
We claim that literally serial machines hardly exist and that considerable increase in machine parallelism is to be expected.

Parallel algorithms - We will focus our attention on the design and analysis of efficient parallel algorithms within the **Data-Parallel (DP)** model of parallel computation. The main methodological goal of these notes is to cope with the ill-defined goal of educating the reader to **"think in parallel"**. For this purpose, we outline an informal model of computation, called **Informal Data-Parallel (IDP)**. The presentation reaches this important model of computation at a relatively late stage. There is no inconsistency between the centrality of the IDP and the focus on the DP, as explained next. The academic research community has developed a tradition of presenting (serial or parallel) algorithmic methods and paradigms in a rigorous manner. It is common to see an algorithms researcher cast his/her idea-driven research in a complexity-result-driven setting. The advantage has been the ability to succinctly demonstrate and discuss the relative effectiveness of various methods and paradigms. In this spirit, the IDP will be used for outlining ideas, while the DP will be used for the detailed presentation.

The following two interrelated contexts may explain why the IDP model may be more robust than any particular DP model.
(i) *Direct hardware implementation of some routines.* It may be possible to implement some routines, such as performing the sum or prefix-sum of n variables, within the same performance bounds as simply "reading these variables". A reasonable rule-of-thumb for selecting a programmer's model for parallel computation might be to start with some model, that includes primitives which are considered essential, and then augment it with useful primitives, as long as the cost of implementing them effectively does not increase the cost of implementing the original model.
(ii) *The ongoing evolution of programming languages.* Development of facilities for expressing parallelism is an important driving force there; popular programming languages (such as Fortran, C and Pascal) are being augmented with constructs for this purpose. Constructs offered by a language may affect the programmer's view on his/her model of computation, and are likely to fit better the more loosely defined IDP. See references to Fetch-and-Add and Scan constructs in the bibliographic notes.

1.2. More background and detail

A legacy of traditional computer science has been to try and develop appropriate levels of abstraction. But, why have abstractions worked? To what extent does the introduction of abstractions between the user and the machine reduce the available computational capability? Following the ingenious insights of Alan Turing, in 1936, where he showed the existence of a universal computing machine that can simulate any computing machine, we emphasize high-level computer languages. Such languages are much more convenient to human beings than are machine languages whose instructions consists of sequences of zeros and ones that machine can execute. Programs written in the high-level languages can be translated into machine languages to yield the desired results without sacrificing expressive power. Usually, the overheads involved are minimal and could be offset only by very sophisticated machine languages programmers, and even then only after an overwhelming investment in human time. So far for abstraction in computing.

We assert that Data-Parallelism has to be a standard programmer's model for any successful general-purpose parallel machine. In other words, our assertion implies that a general-purpose parallel machine cannot be successful unless it can be effectively programmed using the Data-Parallel programmer's model. This does not mean that there will not be others styles of programming, or models of parallel computation, which some, or all, of these computer systems will support. The author predicted respeatedly since the early 1980's that the strongest non parallel machine will continue in the future to outperform, as a general-purpose machine, any parallel machine that does not support Data-Parallelism. Indeed, currently there is no other parallel programming models which is a serious contender primarily since no other model enables to solve nearly as many problems as the Data-Parallel model.

However, a skeptical reader may wonder, *why should Data-Parallelism be a preferred programmer's model?* Before proceeding to answering this question, we note that Data-Parallelism can also be useful for alternative more involved models of computation in providing high-level descriptions of algorithms.

We base our answer to this question on experience. For over twenty years, numerous researchers have asked this very question, and quite a few alternative models of parallel computation have been suggested. At least several thousands research papers were published with algorithms for these models. This exciting research experience can be summarized as follows:

- *Unique knowledge-base.* The knowledge-base on Data-Parallel (or PRAM) algorithms exceeds in order of magnitude any knowledge-base of parallel algorithms within any other model. Paradigms and techniques that have been developed led to efficient and fast parallel algorithms for numerous problems. This applies to a diversity of areas, including data-structures, computational geometry, graph problems, pattern matching, arithmetic computations and comparison problems. This provides an overwhelming circumstantial evidence for the unique importance of Data-Parallel algorithms.

- *Simplicity.* A majority of the users of a future general-purpose parallel computer would like, or need, the convenience of a simple programmer's model, since they will not have the time to master advanced, complex computer science skills. Designing algorithms, and particularly parallel ones, is an intellectually demanding and time consuming job and therefore represents the most expensive component in using parallel machines, in the long term. The relative simplicity of the Data-Parallel model is one of the main reasons for its success. The Data-Parallel (or PRAM) model of computation strips away levels of algorithmic complexity concerning synchronization, reliability, data locality, machine connectivity, and communication contention and thereby allows the algorithm designer to focus on the fundamental computational difficulties of the problem at hand. Indeed, the result has been a substantial number of efficient algorithms designed in this model, and a growing number of design paradigms and utilities for designing such algorithms.

- *Reusability.* All generations of an evolutionary development of parallel machines must support a robust programmer's model, or otherwise risk failure, because of the following. Imagine a computer industry decision maker that considers whether to invest several human-years in writing code for a certain parallel programming language (or stick to his/her existing serial code). By the time the programming project will have been finished, the language is likely to become, or about to become, obsolete. The only reasonable business decision under this circumstances is simply not to do it. Machines that do not support a robust parallel programming language are likely to remain an academic exercise, since from the industry perspective, the test for successful parallel computers is their usability at the end. At the present time, "Data-Parallel-related" programming language is the only serious candidate for playing the role of such a robust programming language.

- *To get started.* Some sophisticated programmers of parallel machines are willing to tune an algorithm that they design to the specifics of a certain parallel machine. The following methodology has become common practice among such programmers: start with a Data-Parallel algorithm for the problem under consideration and advance from there.

- *Formal emulation.* Recent advances have shown Data-Parallel algorithms to be formally emulatable on high interconnect machines, and formal machine designs that support a large number of virtual processes can, in fact, give a speedup that approaches the number of processors for some sufficiently large problems. Some new machine designs are aimed at realizing idealizations that support pipelined, virtual unit time access Data-Parallelism.

Note that in the context of serial computation, which has of course been a tremendous success story (the whole computer era), all the above points can be attributed to the serial random-access-machine (RAM) model of serial computation, which is arguably, a "standard programmer's model" for a general-purpose serial machine. We finish with commenting on what I view as a common misconception:

- *Misconception: The Data-Parallel model, or the closely related PRAM model, are machine models.* These model are only meant to be convenient programmer's models; in other words, design your algorithms for the Data-Parallel, or the PRAM, model; the machine software will later take your code and translate it, to result in an effective use of the machine.

2. Introduction

We start with describing a model of computation which is called the parallel random-access machine (PRAM). Besides its historical role, as the model for which many parallel algorithms were originally written, it is easy to understand its assumption. We then proceed to describe the Data-Parallel (DP) model, which is essentially equivalent to the PRAM model. The DP model, which is more convenient for describing parallel algorithms, is the principal mode for presenting parallel algorithms in these notes.

2.1. The PRAM Model

We review the basics of the PRAM model. A PRAM employs p synchronous processors, all having unit time access to a shared memory. Each processor has also a local memory. See Figure 1.

At each time unit a processor can write into the shared memory (i.e., copy one of its local memory registers into a shared memory cell), read into shared memory (i.e., copy a shared memory cell into one of its local memory registers), or do some computation with respect to its local memory. We will avoid entering this level of detail in describing PRAM algorithms. That is, an instruction of the form:
processor i: $c := a + b$
where a, b and c are shared memory locations, will be a short form for instructing processor i to: first, copy location a into its local memory, second, copy location b into its local memory, third, add them, and, fourth, write the result into location c.

There are a variety of rules for resolving access conflicts to the shared memory. The most common are *exclusive-read exclusive-write (EREW), concurrent-read exclusive-write*

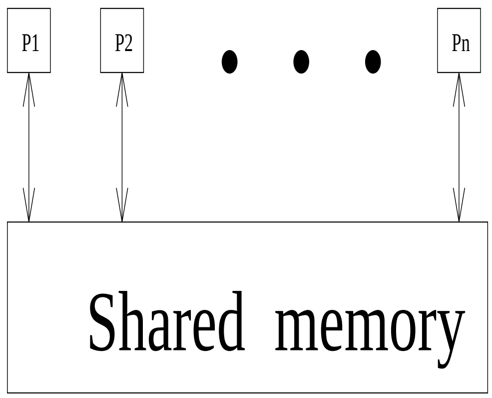


Figure 1: Processors and shared memory

(CREW), and concurrent-read concurrent-write (CRCW), giving rise to several PRAM models. An EREW PRAM does not allow simultaneous access by more than one processor to the same memory location for read or write purposes, while a CREW PRAM allows concurrent access for reads but not for writes, and a CRCW PRAM allows concurrent access for both reads and writes. We shall assume that in a concurrent-write model, the smallest numbered, among the processors attempting to write into a common memory location, actually succeeds. This is called the *Priority CRCW* rule. There are two alternative CRCW rules: (i) By the *Arbitrary CRCW* rule, an arbitrary processor among the processors attempting to write into a common memory location, succeeds. (ii) The *Common CRCW* rule allows concurrent writes only when all the processors attempting to write into a common memory location are trying to write the same value.

For concreteness, we proceed to an example of a PRAM algorithm. However, before doing this we present the **pardo** "programming construct" which is heavily used in these notes to express operations which are performed in parallel:

\- **for** P_i , $1 \leq i \leq n$ **pardo**

\- $A(i) := B(i)$

This means that the following n operations are performed concurrently: processor P_1 assigns $B(1)$ into $A(1)$, processor P_2 assigns $B(2)$ into $A(2)$, and so on.

2.1.1. Example of a PRAM algorithm .

The summation problem

Input: An array $A = A(1) \dots A(n)$ of n numbers.

The problem is to compute $A(1) + \dots + A(n)$.

The summation algorithm below works in rounds. In each round add, in parallel, pairs of elements as follows: add each odd-numbered element and its successive even-numbered element.

For example, assume that $n = 8$; then the outcome of the first round is

$$A(1) + A(2),\ A(3) + A(4),\ A(5) + A(6),\ A(7) + A(8)$$

the outcome of the second round is

$$A(1) + A(2) + A(3) + A(4),\ A(5) + A(6) + A(7) + A(8)$$

and the outcome of the third round is

$$A(1) + A(2) + A(3) + A(4) + A(5) + A(6) + A(7) + A(8)$$

which is the sum that we seek. A detailed PRAM description of this "pairwise summation" algorithm follows.

For simplicity, assume that: (i) we are given a two dimensional array B (whose entries are $B(h, i)$, $0 \leq h \leq \log n$[1] and $1 \leq i \leq n/2^h$) for storing all intermediate steps of the computation, and (ii) $n = 2^l$ for some integer l.

ALGORITHM 1 (Summation)

1. **for** P_i , $1 \leq i \leq n$ **pardo**
2. $\quad B(0, i) := A(i)$
3. $\quad$ **for** $h := 1$ **to** $\log\ n$ **do**
4. $\quad\quad$ **if** $i \leq n/2^h$
5. $\quad\quad$ **then** $B(h, i) := B(h-1, 2i-1) + B(h-1, 2i)$
6. $\quad\quad$ **else** stay idle
7. $\quad$ **for** $i = 1$: output $B(\log\ n, 1)$; **for** $i > 1$: stay idle

See Figure 2.

Algorithm 1 uses $p = n$ processors. Line 2 takes one round, line 3 defines a loop taking $\log n$ rounds, and line 7 takes one round. Since each round takes constant time, Algorithm 1 runs in $O(\log\ n)$ time.

So, an algorithm in the PRAM model

is presented in terms of a sequence of parallel time units (or "rounds", or "pulses"); we allow p instructions to be performed at each time unit one per perocessor; this means that a time unit consists of a sequence of exactly p instructions to be performed

[1] Unless otherwise stated the base of the logarithms is always assumed to be 2

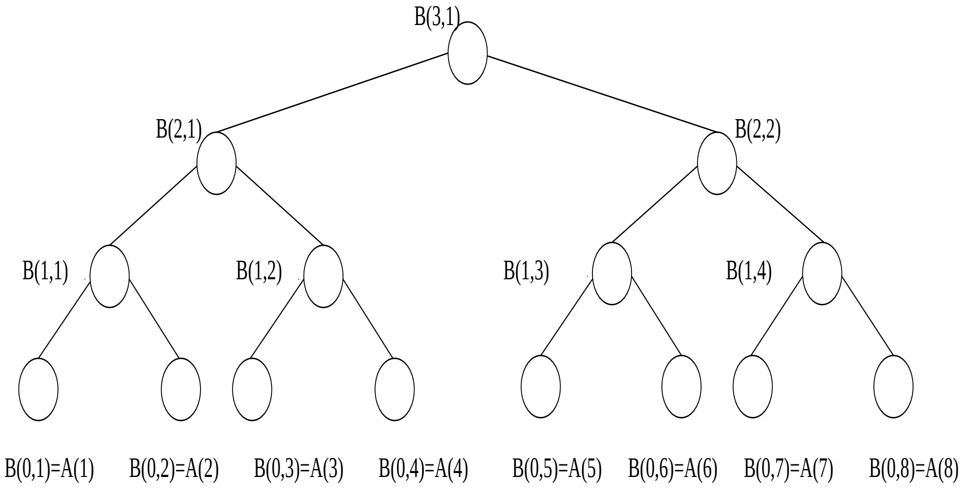


Figure 2: Summation on an $n = 8$ processor PRAM

concurrently.

We refer to such a presentation, as the *standard PRAM mode*.

The standard PRAM mode has a few drawbacks: (i) It does not reveal how the algorithm will run on PRAMs with different number of processors; specifically, it does not tell to what extent more processors will speed the computation, or fewer processors will slow it. (ii) Fully specifying the allocation of instructions to processors requires a level of detail which might be unnecessary (since a compiler can extract it automatically - see the DP-presentation sufficiency theorem below).

2.2. Data-Parallel presentation of algorithms

An alternative model which is actually an alternative presentation mode, called Data-Parallel, is outlined next. Data-Parallel algorithms are also presented *in terms of a sequence of parallel time units (or "rounds", or "pulses"); however, each time unit consists of a sequence of instructions to be performed concurrently; the sequence of instructions may include any number.*

Comment on rigor. Strictly speaking, DP actually defines a slightly different model of computation. Consider an instruction of the form

\- **for** i , $1 \leq i \leq \alpha$ **pardo**

\- $A(i) := B(C(i))$

where the time unit under consideration, consists of a sequence of α concurrent instructions, for some positive integer α. Models such as a common CRCW DP, an arbitrary

CRCW DP, or a priority CRCW DP, are defined as their PRAM respective counterparts with α processors. We explain below why these DP models are essentially equivalent to their PRAM counterpart and therefore treat the DP only as a separate presentation mode and suppress the fact that these are actually (slightly) different models. The only additional assumption which we make for proving these equivalences is that in cases where the same variable is accessed for both reads and write in the same time unit, then all the reads precede all the writes.

The summation example (cont'd). We give a DP presentation for a summation algorithm. It is interesting to note that Algorithm 2, below, enables to attribute **"greedy-parallelism"** to the summation algorithm in the following sense. At each point in time the summation algorithm seeks to break the problem into as many pairwise additions as possible, or, in other words, into the largest possible number of independent tasks that can performed concurrently. We make the same assumptions as for Algorithm 1, above.

ALGORITHM 2 (DP-Summation)
1. **for** i , $1 \leq i \leq n$ **pardo**
2. $\quad B(0,i) \ := \ A(i)$
3. **for** $h \ := \ 1$ **to** $\log \ n$
4. $\quad$ **for** i , $1 \leq i \leq n/2^h$ **pardo**
5. $\quad\quad B(h,i) \ := \ B(h-1, 2i-1) \ + \ B(h-1, 2i)$
6. **for** $i \ = \ 1$ **pardo** output $B(\log \ n, 1)$

The first round of the algorithm (lines 1 and 2) has n operations. The second round (lines 4 and 5 for $h = 1$) has $n/2$ operations. The third round (lines 4 and 5 for $h = 2$) has $n/4$ operations. In general, the kth round of the algorithm, $1 \leq k \leq \log n + 1$, has $n/2^{k-1}$ operations and round $\log n \ + 2$ (line 6) has one more operation (note that the use of a **pardo** instruction in line 6 is somewhat artificial). The total number of operations is $2n$ and the time is $\log n \ + 2$. We will use this information in the corollary below.

The next theorem demonstrates that the DP presentation mode does not suffer from the same drawbacks as the standard PRAM mode, and that every algorithm in the DP mode can be automatically translated into a PRAM algorithm.

The DP-presentation sufficiency Theorem. Consider an algorithm in the DP mode that takes a total of $x = x(n)$ elementary operations and $d = d(n)$ time. The algorithm can be implemented by any $p = p(n)$-processor within $O(x/p \ + \ d)$ time, using the same concurrent-write convention as in the DP presentation.
Note that the theorem actually represents five separate theorems for five respective concurrent-read and concurrent-write models: EREW, CREW, common CRCW, arbitrary CRCW and priority CRCW.

Proof of the Theorem. After explaining the notion of a round-robin simulation, we advance to proving the theorem. Given a sequence of α instructions $y_1, \ldots, y_\alpha$, a *round-robin* simulation of these instructions by p processors, $P_1 \ldots P_p$, means the following $\lceil \alpha/p \rceil$ rounds. In round 1, the first group of p instructions $y_1 \ldots y_p$ is performed in parallel, as follows. For each j, $1 \leq j \leq p$, processor P_j performs instruction y_j, respectively. In round 2, the second group of p instructions $y_1 \ldots y_p$ is performed in parallel, as follows. For each j, $1 \leq j \leq p$, processor P_j performs instruction y_{j+p}, respectively. And so on, where in round $\lceil y/p \rceil$ (the last round) some of the processors may stay idle if there are not enough instructions for all of them. The reader can also find a concrete demonstration of this important notion of round-robin simulation in Algorithm 2' below.

We are ready to proceed with the proof of the theorem. Let x_i denote the number of instructions to be performed by the algorithm at round i. Note that by the definition of x, $\sum_{i=1}^{d} x_i = x$. The p processors can "simulate" round i of the algorithm, in two stages. In the first stage, only the read instructions of round i are emulated; each read instruction causes storage into a temporary variable. This is done in $\lceil x_i/p \rceil \leq x_i/p + 1$ time units, using a round-robin simulation of the x_i reads of round i by p processors. In the second stage, all the other instructions of round i (including write instructions) are performed in additional $\lceil x_i/p \rceil \leq x_i/p + 1$ time units, again using a round-robin simulations by p processors. The theorem follows.

We leave it as an exercise to verify the above proof for the various concurrent-read and concurrent-write models. It can also be readily seen that a converse of the theorem holds true: simply consider a p-processor PRAM algorithm as being in the DP mode, with a sequence of p instructions at each time unit.

A Corollary (The summation example (cont'd)). As a corollary of the theorem, we conclude that the summation algorithm, Algorithm 2, would run in $O(n/p + \log n)$ time on a p-processor PRAM. For $p \leq n/\log n$, this implies a running time of $O(n/p)$, while for $p \geq n/\log n$, the implied running time is $O(\log n)$. Since the algorithm does not involve concurrent reads or writes, the p-processors algorithm can run on an EREW PRAM.

For concreteness, we demonstrate below the spirit of the above proof (of the DP-presentation sufficiency theorem) with respect to Algorithm 2 that was given in the DP-presentation mode. We show how to run it on a p-processor PRAM. (Note that, unlike the proof, we emulate the read instructions in the same time unit as other instructions.)

ALGORITHM 2' (Summation on a p-processor PRAM)

1. **for** P_i , $1 \leq i \leq p$ **pardo**
2.   **for** $j := 1$ **to** $\lceil n/p \rceil - 1$ **do**
-         $B(0, i + (j-1)p) := A(i + (j-1)p)$
3.   **for** i , $1 \leq i \leq n - (\lceil n/p \rceil - 1)p$

- $B(0, i + (\lceil n/p \rceil - 1)p) := A(i + (\lceil n/p \rceil - 1)p)$
- **for** i , $n - (\lceil n/p \rceil - 1)p \leq i \leq p$
- stay idle

4. **for** $h := 1$ **to** $\log n$
5. **for** $j := 1$ **to** $\lceil n/(2^h p) \rceil - 1$ **do** (*an instruction $j := 1$ **to** 0 **do** means: "do nothing"*)
- $B(h, i + (j-1)p) := B(h-1, 2(i + (j-1)p) - 1) + B(h-1, 2(i + (j-1)p))$

6. **for** i , $1 \leq i \leq n - (\lceil n/(2^h p) \rceil - 1)p$
- $B(h, i + (\lceil n/(2^h p) \rceil - 1)p) := B(h-1, 2(i + (\lceil n/(2^h p) \rceil - 1)p) - 1) +$
- $B(h-1, 2(i + (\lceil n/(2^h p) \rceil - 1)p))$
- **for** i , $n - (\lceil n/(2^h p) \rceil - 1)p \leq i \leq p$
- stay idle

7. **for** $i = 1$ output $B(\log n, 1)$; **for** $i > 1$ stay idle

Algorithm 2' simply emulates in a round-robin fashion each sequence of concurrent instruction of Algorithm 2. For instance, lines 1 and 2 in Algorithm 2 have a sequence of n instructions. These instructions are emulated in Algorithm 2' in $\lceil n/p \rceil - 1$ rounds by line 2, and in an additional round by line 3. The running time of Algorithm 2' is as follows. Line 2 take $\lceil n/p \rceil - 1$ rounds, and line 3 takes one round, for a total of $\lceil n/p \rceil$ rounds. The loop for $h = k$ of lines 5-6 takes a total $\lceil n/(2^h p) \rceil$ rounds. Line 7 takes one more round. Summing up the rounds gives:

$$\lceil n/p \rceil + \sum_{i=1}^{\log n} \lceil n/(2^h p) \rceil + 1 \leq \sum_{i=0}^{\log n} (n/(2^h p) + 1) + 1 = O(n/p + \log n)$$

as implied by directly applying the DP-presentation sufficiency theorem.

Measuring the performance of parallel algorithm.

Consider a problem whose input length is n, and a parallel algorithm in the DP mode for it. We can measure the performance of the algorithm in terms of worst-case running time, denoted $T(n)$, and total number of operations, denoted $W(n)$ (where W stands for work). The following are alternative ways for measuring the performance of a parallel algorithm. We have actually shown that they are all asymptotically equivalent.

1. $W(n)$ operations and $T(n)$ time.
2. $P(n) = W(n)/T(n)$ processors and $T(n)$ time (on a PRAM).
3. $W(n)/p$ time using any number of $p \leq W(n)/T(n)$ processors (on a PRAM).
4. $W(n)/p + T(n)$ time using any number of p processors (on a PRAM).

Exercise 1: *The above four ways for measuring performance of a parallel algorithms form six pairs. Prove the pairs are all asymptotically equivalent.*

2.3. Goals for Designers of Parallel Algorithms

Our main goal in designing parallel algorithms is efficiency. We intentionally refrain from giving a formal definition for when an algorithm is to be considered efficient, but give several informal guidelines.

Consider two parallel algorithm for the same problem. One performs a total of $W_1(n)$ operations in $T_1(n)$ time and the other performs a total of $W_2(n)$ operations in $T_2(n)$ time. Generally, we would like to consider the first algorithm *more efficient* than the second algorithm if $W_1(n) = o(W_2(n))$, regardless of their running times; and if $W_1(n)$ and $W_2(n)$ grow asymptotically the same, then the first algorithm is considered more efficient if $T_1(n) = o(T_2(n))$. A reason for not giving a categoric definition is the following example. Consider an extreme case where $W_1(n) = O(n)$ and $T_1(n) = O(n)$, and $W_2(n) = O(n \log n)$ and $T_2(n) = O(\log n)$. It is hard to decide which algorithm is more efficient since the first algorithm needs less work, but the second is much faster. In this case, both algorithms are probably interesting (and only implementing them on a specific target machine which is of interest to the user, can determine which one runs faster on that machine).

Consider a problem, and let $T(n)$ be the best known worst case time upper bound on a serial algorithm for an input of length n.

Assume further that we can prove that this upper bound cannot be asymptotically improved. (Using complexity theory terms, $T(n)$ is the *serial time complexity* of the problem.) Consider a parallel algorithm for the same problem that performs $W(n)$ operations in $T_{par}(n)$ time. The parallel algorithm is said to be **work-optimal**, if $W(n)$ grows asymptotically the same as $T(n)$. A work-optimal parallel algorithm is **work-time-optimal** if its running time $T(n)$ cannot be improved by another work-optimal algorithm.

The problem with the definitions above is that often the serial complexity of problems is not known. We see a need to coin a term that will capture the sense of accomplishment for such cases, as well. Assume that we do not know whether $T(n)$ can be improved asymptotically, and consider a parallel algorithm for the same problem that performs $W(n)$ work in $T_{par}(n)$ time. The parallel algorithm is said to achieve **parallel speed-up**, if $W(n)$ grows asymptotically the same as $T(n)$. The problem with this definition is that it may not withstand the test of time: someday, an improved serial algorithm can change the best serial upper bound, and an algorithm which achieved parallel speed-up will no longer be so.

Perspective Earlier we used some analogies to serial computation in order to justify the choice of the Data-Parallel (or PRAM) model of computation. Still, it is crucial to recognize the following difference between parallel and serial computation in trying to draw further analogies. Serial computing opened the era of computing. To some extent the job was easy because there was no previous computing paradigm that had to be challenged. The situation with parallel computing is different. Serial computing is an enormous success. The only problem with this paradigm is that it may lead to a dead end when it come to building machines which are much stronger than currently available.

2.4. Some final Comments

2.4.1. The defualt assumption regarding shared assumption access resolution These notes will focus on the strongest model, the priority CRCW PRAM and its respective DP model. A partial explanation for this is provided by the following comments regarding the relative power of PRAM models: (1) The Priority CRCW PRAM is most powerful, then come the Arbitrary CRCW PRAM, the Common CRCW PRAM, the CREW PRAM, and finally the EREW PRAM. (2) Simulation results of the most powerful Priority CRCW PRAM of the least powerful EREW PRAM show that the models do not differ substantially. (3) Since all these PRAM models are only virtual models of real machines, a more practical perspective is to compare their emulations on a possible target machine. It turns out that the models differ even less from this perspective.

2.4.2. NC

A different, but related, efficiency goal In stating the goals for parallel computing, one has to remember that its primary objective is to challenge the supremacy of serial computing. It is interesting to mention that our definitions of optimality are more sensitive to relationship with the serial alternative than a widely recognized complexity theory approach which states that "a problem is amenable to parallel computing if it can be solved in polylogarithmic time using a polynomial number of processors (what is known in the literature as the class NC, or Nick's Class, in honor of Nick Pippenger)". Nevertheless, this complexity theory approach is of fundamental importance: for instance, if one shows that a problem is "log-space complete for P" then the problem is unlikely to be in the class NC - a very useful insight. Therefore, a next step, beyond these notes, for a reader who wants to know more about parallel algorithms, will have to include learning the basic concepts of this theory.

2.4.3. On selection of material for these notes The goal of these notes is to familiarize the reader with the elementary routines, techniques and paradigms in the current knowledge base on parallel algorithm. In planning these notes, we were guided by the following two principles.
The **first** principle is actually a compromise between two conflicting approaches: (1) **"Breadth-first search":** Present the theory as a "bag of tricks", describing one trick at a time. Assuming that the curriculum can allocate only a limited time to the topics covered in these notes, this will maximize the number of tricks learned at the expense of not exposing the reader to more involved parallel algorithms. Since, in general, designing parallel algorithms may be a rather involved task, an important aspect may be missed by this approach, which brings us to the second approach. (2) **"Depth-first search"**: Present in full a few relatively involved parallel algorithms.
The **second** principle applies a *"prefix rule of thumb"*. At each point, we tried to evaluate what would be the next most important thing that the reader should know assuming that he/she has time to learn only one more section. These two principles may explain to the more knowledgeable reader our selection of material.

2.4.4. Level of material I taught this material on several occasions to beginning graduate students; recently, undergraduates were allowed to take the course as an elective and constituted a considerable fraction of the class. The first ten sections (i.e., everything excluding the section on graph connectivity) were taught in 21 lecture hours, where a lecture hour is defined as 50 minutes net time of teaching. My impression is that this material is suitable for undergraduate computer science students in their junior or senior year, and can cover about a half of an undergraduate course. This can be combined into a full course with another topic for the remaining part of the course. Another option is to teach next the connectivity section, which took me 6-7 more hours. Together with some additional material (e.g., introduction to NP-Completeness) this can make a junior or senior level undergraduate course or a beginning graduate course.

3. Technique: Balanced Binary Trees; Problem: Prefix-Sums

Parallel prefix-sums might be the most heavily used routine in parallel algorithms. We already saw how balanced binary trees are used in a parallel summation algorithm. This section demonstrates a more involved use of balanced binary trees.

The prefix-sums problem
Input: An array A of size n of elements drawn from some set. Assume that a binary operation, denoted $*$, is defined on the set. Assume also that $*$ is *associative*; namely, for any elements a, b, c in the set, $a * (b * c) = (a * b) * c$.
(The operation $*$ is pronounced "star" and often referred to as "sum" because addition,

relative to real numbers, is a common example for $*$.)
The n prefix-sums of array A are defined as $\sum_{j=1}^{i} A(j)$, for $j,\ 1 \leq j \leq n$, or:
$A(1)$
$A(1) * A(2)$
$\dots$
$A(1) * A(2) * \dots * A(i)$
$\dots$
$A(1) * A(2) * \quad \dots \quad * A(n)$

The prefix sums algorithm below assumes that $n = 2^k$ for some integer k, and that the arrays $B(h,i)$ and $C(h,i)$ for $0 \leq h \leq \log n$ and $1 \leq i \leq n/2^h$, are given for storing intermediate steps of the computation. The algorithm works in two stages each taking logarithmic time. The first stage (lines 1-3 below) is similar to the summation algorithm of the previous section, where the computation advances from the leaves of a balanced binary tree to the root and computes, for each internal node $[h,i]$ of the tree, the sum of its descendant leaves into $B(h,i)$. The second stage (lines 4-7 below) advances in the opposite direction, from the root to the leaves. For each internal node $[h,i]$ a value is entered into $C(h,i)$. The main idea of the prefix-sums algorithm is that $C(h,i)$ is the prefix-sum $A(1) * A(2) * \dots * A(\alpha)$, where $[0,\alpha]$ is the rightmost descendant leaf of $[h,i]$. This immediately implies that $C(0,1), C(0,2), \dots, C(0,n)$ have the desired prefix-sums. Figure 3. describes a balanced binary tree, Figure 4. describes the B and C arrays, and

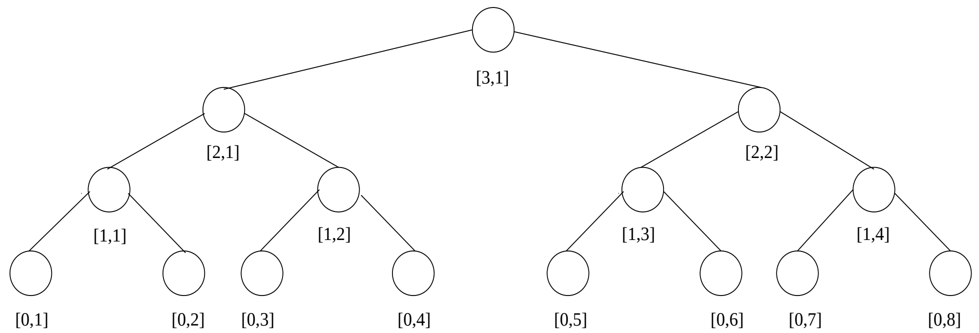


Figure 3: Balanced binary tree

the drilling example below.

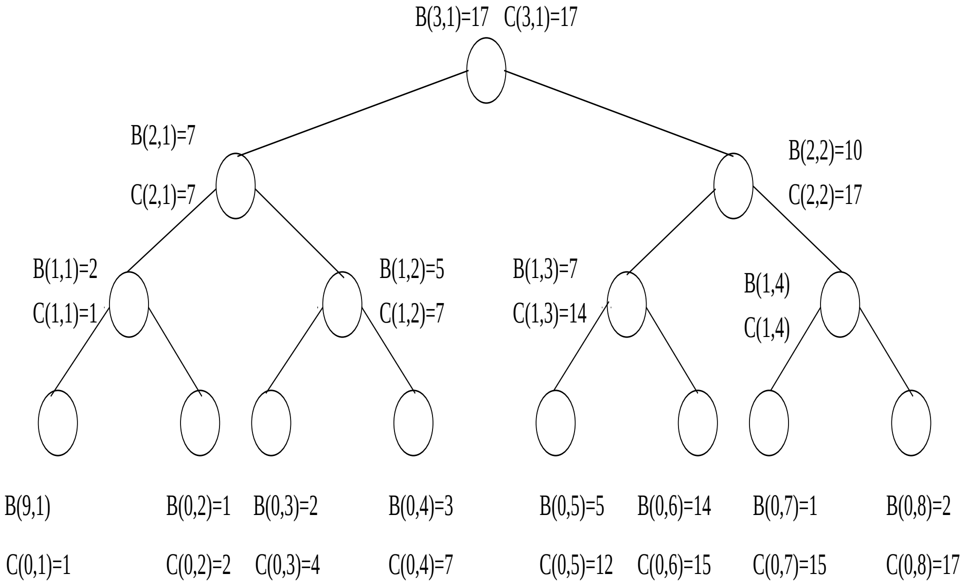


Figure 4: The prefix-sums algorithm

ALGORITHM 1 (Prefix-sums)

1. **for** i , $1 \leq i \leq n$ **pardo**
- $B(0, i) := A(i)$
2. **for** $h := 1$ **to** $\log n$
3. **for** i , $1 \leq i \leq n/2^h$ **pardo**
- $B(h, i) := B(h-1, 2i-1) * B(h-1, 2i)$
4. **for** $h := \log n$ **to** 0
5. **for** i even, $1 \leq i \leq n/2^h$ **pardo**
- $C(h, i) := C(h+1, i/2)$
6. **for** $i = 1$ **pardo**
- $C(h, 1) := B(h, 1)$
7. **for** i odd, $3 \leq i \leq n/2^h$ **pardo**
- $C(h, i) := C(h+1, (i-1)/2) * B(h, i)$
8. **for** i , $1 \leq i \leq n$ **pardo**
- Output $C(0, i)$

A drilling example. Let us run Algorithm 1 on the following instance. See also Figure 4. $n = 8$ and $A = 1, 1, 2, 3, 5, 2, 1, 2$ and $*$ is the $+$ operation. Lines 1 implies $B(0; 1 \ldots 8) = 1, 1, 2, 3, 5, 2, 1, 2$. Lines 2-3 imply: $B(1; , 1, 2, 3, 4) = 2, 5, 7, 3$ for $h = 1$, $B(2; , 1, 2) = 7, 10$ for $h = 2$, and $B(3, 1) = 17$ for $h = 3$. Lines 4-7 imply $C(3, 1) = 17$ for $h = 3$, $C(2; , 1, 2) = 7, 17$ for $h = 2$, $C(1; , 1, 2, 3, 4) = 2, 7, 14, 17$ for $h = 1$, and finally $C(0; 1 \ldots 8) = 1, 2, 4, 7, 12, 14, 15, 17$ for $h = 0$.

Complexity. The operations of the prefix-sums algorithm can be "charged" to nodes of the balanced binary tree, so that no node will be charged by more than a constant number of operations. This charges can be done as follows. For each assignment into either $B(h,i)$ or $C(h,i)$, the operations that led to is are charged to node $[i,j]$. Since the number of nodes of the tree is $2n-1$, $W(n)$, the total number of operations of the algorithm is $O(n)$. The time is $O(\log n)$ since it is dominated by the loops of lines 2 and 4, each requiring $\log n$ rounds.

Theorem 3.1: *The prefix-sums algorithm runs in $O(n)$ work and $O(\log n)$ time.*

3.1. Application - the Compaction Problem

We already mentioned that the prefix-sums routine is heavily used in parallel algorithm. One trivial application, which is needed later in these notes, follows:
Input. An array $A = A(1), \ldots, A(n)$ of (any kind) elements and another array $B = B(1), \ldots, B(n)$ of bits (each valued zero or one).
The **compaction** problem is to find a one-to-one mapping from the subset of elements of $A(i)$, for which $B(i) = 1$, $1 \leq i \leq n$, to the sequence $(1, 2, \ldots, s)$, where s is the (a priori unknown) numbers of ones in B. The mapping should be order preserving. That is, if $A(i)$ is mapped to k and $A(j)$ is mapped to $k+1$ then $i < j$.

For computing this mapping, simply compute all prefix sums with respect to array B; consider an entry $B(i) = 1$; if the prefix sum of i is j then map $A(i)$ into j. See also Figure 5.

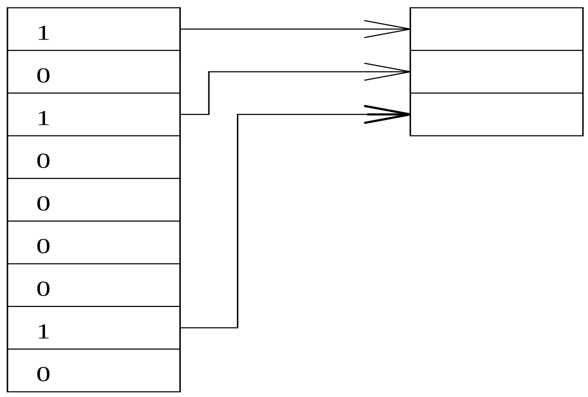


Figure 5: Application of prefix-sums for compaction

Theorem 3.2: *The compaction algorithm runs in $O(n)$ work and $O(\log n)$ time.*

Exercise 2: *Let A be a cell in the shared memory of a PRAM. All p processors of the PRAM need to "know" the value stored in A. Give a fast EREW algorithm for broadcasting the value of A to all p processors. How much time this will take?*

Exercise 3: *The minimum problem is defined as follows. Input: An array A of size n of elements drawn from some totally ordered set. The minimum problem is to find the smallest element in array A.*
(1) Give an EREW PRAM algorithm which runs in $O(n)$ work and $O(\log n)$ time for the problem.
(2) Suppose we are given only $p \leq n/\log n$ processors, which are numbered from 1 to p. Describe the algorithm of item (1) above to be executed by processor i, $1 \leq i \leq p$.
The prefix-min problem has the same input as for the minimum problem and we need to find for each i, $1 \leq i \leq n$, the smallest element among $A(1), A(2), \ldots, A(i)$.
(3) Give an EREW PRAM algorithm which runs in $O(n)$ work and $O(\log n)$ time for the problem.

Exercise 4: *The nearest-one problem is defined as follows. Input: An array A of size n of bit; that is the value of each entry of A is either 0 or 1. The nearest-one problem is to find for each i, $1 \leq i \leq n$, the largest index $j \leq i$, such that $A(j) = 1$.*
(1) Give an EREW PRAM algorithm which runs in $O(n)$ work and $O(\log n)$ time for the problem.
The input for the segmented prefix-sums problem, includes the same binary array A as above, and in addition and array B of size n of numbers. The segmented prefix-sums problem is to find for each i, $1 \leq i \leq n$, the sum $B(j) + B(j+1) + \ldots + B(i)$, where j is the nearest-one for i (if i has no nearest-one we define its nearest-one to be 1). (2) Give an EREW PRAM algorithm which runs in $O(n)$ work and $O(\log n)$ time for the problem.

3.2. Recursive Presentation of the Prefix-Sums Algorithm

Recursive presentations are useful for describing serial algorithms. They are also useful for describing parallel ones where sometimes they play a special role in articulating the technique being used.

The input for the recursive procedure is an array $(x_1, x_2, \ldots, x_m)$ and the output is given in another array $(u_1, u_2, \ldots, u_m)$. We assume that $\log_2 m$ is an non-negative integer.

PREFIX-SUMS$(x_1, x_2, \ldots, x_m; u_1, u_2, \ldots, u_m)$
1. **if** $m = 1$ **then** $u_1 := x_1$; **exit**

2. **for** i, $1 \leq i \leq m/2$ **pardo**
- $\quad y_i \ := \ x_{2i-1} * x_{2i}$
3. PREFIX-SUMS$(y_1, y_2, \ldots, y_{m/2}; v_1, v_2, \ldots, v_{m/2})$
4. **for** i even, $1 \leq i \leq m$ **pardo**
- $\quad u_i \ := \ v_{i/2}$
5. **for** $i = 1$ **pardo**
- $\quad u_1 \ := \ x_1$
6. **for** i odd, $3 \leq i \leq m$ **pardo**
- $\quad u_i \ := \ v_{(i-1)/2} * x_i$

The prefix-sums algorithm is started by the following routine call:
PREFIX-SUMS$(A(1), A(2), \ldots, A(n); C(0,1), C(0,2), \ldots, C(0,n))$.

We do not elaborate any more on this recursive presentation, but just take a second look at the complexity analysis which it implies, since it is concise and elegant. The time required by the routine PREFIX-SUMS, excluding the recursive call in instruction 3, is $O(1)$, or $\leq \alpha$, for some constant α. The number of operations required by the routine, excluding the recursive call in instruction 3, is $O(m)$, or $\leq \beta m$, for some constant β. Since the recursive call is for a problems of size $m/2$, the running time, $T(n)$, and the total number of operations, $W(n)$, satisfy the recurrences:

$$T(n) \leq T(n/2) \ + \ \alpha$$

$$W(n) \leq W(n/2) \ + \ \beta n$$

whose solutions are $T(n) \ = \ O(\log n)$, and $W(n) \ = \ O(n)$.

Exercise 5: *Multiplying two $n \times n$ matrices A and B results in another $n \times n$ matrix C, whose elements $c_{i,j}$ satisfy $c_{i,j} \ = \ \sum_{k=1}^{n} a_{i,k} b_{k,j}$.*
(1) Given two such matrices A and B, show how to compute matrix C in $O(\log n)$ time using n^3 processors.
(2) Suppose we are given only $p \leq n^3$ processors, which are numbered from 1 to p. Describe the algorithm of item (1) above to be executed by processor i, $1 \leq i \leq p$. (3) Show how to improve the work complexity in the algorithm of item (1) and get matrix C in $O(n^3)$ work and $O(\log n)$ time.
(4) Suppose we are given only $p \leq n^3/\log n$ processors, which are numbered from 1 to p. Describe the algorithm of item (3) above to be executed by processor i, $1 \leq i \leq p$.

4. The Simple Merging-Sorting Cluster

In this section we present two simple algorithmic paradigms called: *partitioning* and divide-and-conquer. Our main reason for considering the problems of merging and sorting in this section is for demonstrating these paradigms.

4.1. Technique: Partitioning; Problem: Merging

Input: Two arrays $A = A(1) \ldots A(n)$ and $B = B(1) \ldots B(m)$ consisting of elements drawn from a totally ordered domain S; we assume that each array is monotonically non-decreasing.
The **merging problem** is to map each of these elements into an array $C = C(1) \ldots C(n+m)$ which is also monotonic and non-decreasing.
For simplicity, we will assume that: (i)the elements of A and B are pairwise distinct, (ii) $n = m$, and (iii) both $\log n$ and $n / \log n$ are integers. Extending the algorithm to cases where some or all of these assumptions do not hold is left as an exercise.

It turns out that a very simple divide-and-conquer approach, which we call **partitioning**, enables to obtain an efficient parallel algorithm for the problem. An outline of the paradigm follows by rephrasing the merging problem as a ranking problem, for which three algorithms are presented. The first algorithm is parallel and is called a "surplus-log" algorithm. The second is a standard serial algorithm. The third algorithm is the parallel merging algorithm; Guided by the partitioning paradigm, the third and final algorithm uses the first and second algorithms as building blocks.

The partitioning paradigm Let n denote the size of the input for a problem. Informally, a partitioning paradigm for designing a parallel algorithm consists of the following two stages:

partitioning - partition the input into a large number, say p, of independent small jobs, so that the size of the largest small job is roughly n/p, and

actual work - do the small jobs concurrently, using a separate (possibly serial) algorithm for each.

The merging problem can be rephrased as a ranking problem, as follows. Let z be an element of S, which is not in A. Then $RANK(z, A)$ is defined to be i if the

$A(i) < z < A(i+1)$, for $1 \le i < n$; $RANK(z, A) = 0$ if $z < A(1)$ and $RANK(z, A) = n$ if $z > A(n)$. The **ranking problem**, denoted $RANK(A, b)$ is to compute:
(i) $RANK(A(i), B)$ (namely, to rank element $A(i)$ in B), for every $1 \le i \le n$, and (ii) $RANK(B(i), A)$ for every $1 \le i \le n$.

Claim. A solution for the ranking problem gives a solution for the merging problem. To see this, observe that element $A(i)$, $1 \le i \le n$ should appear in location $i + RANK(A(i), B)$ of the merged array C. That is, given a solution to the ranking problem, the following instructions extend it into a solution for the merging problem:

for $1 \le i \le n$ **pardo**
- $C(i + RANK(A(i), B)) := A(i)$

for $1 \le i \le n$ **pardo**
- $C(i + RANK(B(i), A)) := B(i)$

Therefore, the remainder of this section is devoted to the ranking problem.

Our first **"surplus-log" parallel algorithm** for the ranking problem works as follows.

for $1 \le i \le n$ **pardo**
- Compute $RANK(A(i), B)$ using the standard *binary search method.*
- Compute $RANK(B(i), A)$ using binary search

For completeness, we describe the **binary search method**. The goal is to recursively compute $RANK(A(i), B)$ where $B = B(1, \ldots, n)$. Element $B(x)$, where $x = \lceil n/2 \rceil$, is called the *middle* element of B.

if $n = 1$
then if $A(i) < B(1)$ **then** $RANK(A(i), B) := 0$ **else** $RANK(A(i), B) := 1$
else
- **if** $A(i) < B(x)$ **then** $RANK(A(i), B) := RANK(A(i), B(1, \ldots, x-1))$
- **if** $A(i) > B(x)$ **then** $RANK(A(i), B) := x + RANK(A(i), B(x+1, \ldots, n))$

Binary search takes $O(\log n)$ time. The above parallel algorithm takes a total of $O(n \log n)$ work and $O(\log n)$ time. Below, we refer to this ranking algorithm as a *surplus-log* algorithm, a name which highlights its use of $O(n \log n)$, instead of $O(n)$, work.

A **serial** routine for the **ranking** problem follows. Since it will be used later with $n \ne m$, we differentiate the two values in the routine below.

$ranking(A(1), A(2), \ldots, A(n); B(1), B(2), \ldots, B(m))$
$k := 0$ and $l := 0$; add two auxiliary element $A(n+1)$ and $B(m+1)$ which are each larger than both $A(n)$ and $B(m)$
while $k \le n$ or $l \le m$ **do**
- **if** $A(k+1) < B(l+1)$
- **then** $RANK(A(k+1), B) := l$; $k := k+1$
- **else** $RANK(B(l+1), A) := k$; $l := l+1$

In words, starting from $A(1)$ and $B(1)$, in each round one element from A is compared with one element of B and the rank of the smaller among them is determined. This takes $O(n+m)$ time.

We are ready to present the **parallel algorithm for ranking**.

Stage 1 (partitioning). We use essentially the surplus-log algorithm for the purpose of partitioning the original ranking problem into a number of problems which is proportional to $n/\log n$. Let $x = n/\log n$.

1.1 We select x elements of A, which partition A into x (roughly) equal blocks, into array A_SELECT. That is, $A\_SELECT = A(1), A(1 + \log n), A(1 + 2\log n) \ldots A(n + 1 - \log n)$. For each $A\_SELECT(i)$, $1 \leq i \leq x$ separately (and in parallel), compute $RANK(A\_SELECT(i), B)$ using binary search in $O(\log n)$ time.

1.2 Similarly, we select x elements of B, which partition B into p (roughly) equal blocks, into array B_SELECT. That is, $B\_SELECT = B(1), B(1 + \log n), B(1 + 2\log n) \ldots B(n+1-\log n)$. For each $B\_SELECT(i)$, $1 \leq i \leq x$ separately, compute $RANK(B\_SELECT(i), A)$ using binary search in $O(\log n)$ time.

The situation following the above computation is illustrated in Figure 6.

High-level description of Stage 2. The original ranking problem was partitioned in Stage 1 into $2n/\log n$ "small ranking problems" each of size $2\log n$. The ranking of each non-selected element of A and B is determined in Stage 2 by **exactly one** small problem.

Each element $A\_SELECT(i)$ and $B\_SELECT(i)$, $1 \leq i \leq n/\log n$, defines a small ranking problem of those elements in A and B which are just larger than itself. Specifically:

(a) Let $A\_SELECT(i)$, for some i, $1 \leq i \leq n/\log n$. The input of the ranking problem of $A\_SELECT(i)$ consists of two subarrays: (i) the successive subarray of A that spans between element $(i-1)\log n + 1$, denoted $a - start - a(i)$, and some element denoted $a - end - a(i)$, and (ii) the successive subarray of B that spans between element $a - start - b(i) = RANK(A\_SELECT(i), B) + 1$ and some element $a - end - b(i)$. We show how to determine the values of $a - end - a(i)$ and $a - end - b(i)$. Let $B\_SELECT(j)$ be the largest element in B_SELECT which is still smaller than $a - start - b(i)$ (the case where no such j exists is easy). Assume that both $A\_SELECT(i+1)$ and $B\_SELECT(j+1)$ exist (the cases where one or both don's exist are similar). There are two possibilities: (1) $A\_SELECT(i+1) < B\_SELECT(j+1)$; in this case $a - end - a(i)$ is $i \log n$ and $a - end - b(i)$ is $RANK(A\_SELECT(i+1), B)$; and (2) $A\_SELECT(i+1) > B\_SELECT(j+1)$; in this case $a - end - b(i)$ is $j \log n$ and $a - end - a(i)$ is $RANK(B\_SELECT(j+1), A)$.

(b) The input of the problems defined by an element $B\_SELECT(i)$, $1 \leq i \leq n/\log n$, consists of: (i) the successive subarray of B that spans between element

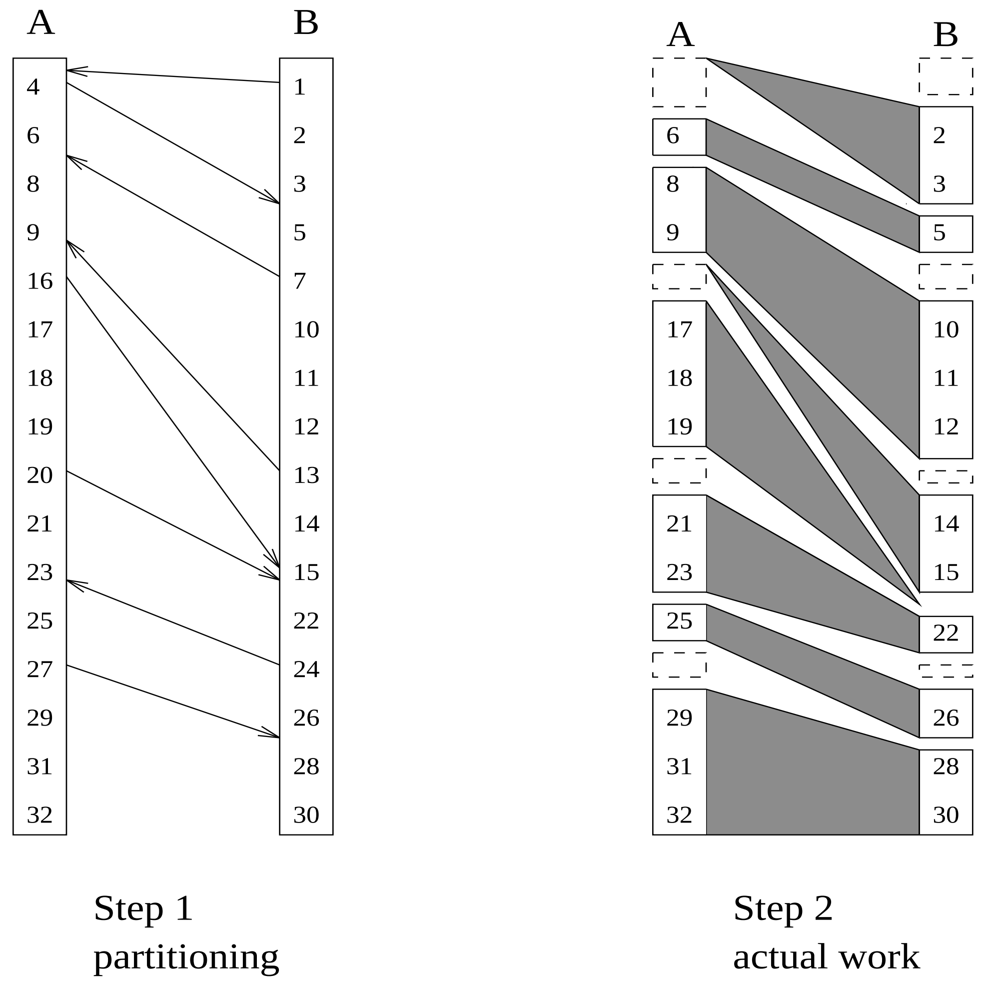


Figure 6: Main steps of the ranking/merging algorithm

$(i-1)\log n + 1$, denoted $b-start-a(i)$, and some element denoted $b-end-a(i)$, and (2) the successive subarray of B that spans between element $b-start-b(i) = RANK(A\_SELECT(i), B) + 1$ and some element $b-end-b(i)$. Determining the values of $b-end-a(i)$ and $b-end-b(i)$ is similar to case (a) above.

The ranking is done by simply calling the serial routine *ranking* for this input.

Stage 2 (actual ranking for the small problems of $A\_SELECT(1,\ldots,n/\log n)$.
for i, $1 \le i \le n/\log n$ **pardo**
- $a-start-a(i) := (i-1)\log n + 2$; $a-start-b(i) := RANK(A\_SELECT(i) + 1$
- $j := \lceil a-start-b(i) - 1/\log n \rceil$
- **if** $j = n/\log n$ **then** $a-end-a(i) := i\log n$; $a-end-b(i) := n$
- **if** $B\_SELECT(j+1) < A\_SELECT(i+1)$ and $j < n/\log n$
- **then** $a-end-b(i) := j\log n$; $a-end-a(i) := RANK(B\_SELECT(j+1), A)$

- **if** $B\_SELECT(j+1) \ > \ A\_SELECT(i+1)$ and $j < n/\log n$
- **then** $a-end-a(i) \ := \ i\log n$; $a-end-b(i) \ := \ RANK(A\_SELECT(i+1), B)$

for $i,\ 1 \le i \le n/\log n$ **pardo**
- $ranking(A(a-start-a(i), \ldots, a-end-a(i)); B(a-start-b(i), \ldots, a-end-b(i)))$

Stage 2 for the small problems of $B\_SELECT(1, \ldots, n/\log n)$ is similar.

Complexity. Stage 1 takes $O(x \log x)$ work and $O(\log x)$ time, which translates into $O(n)$ work and $O(\log n)$ time. Since the input for each of the $2n/\log n$ small ranking problems contains at most $\log n$ elements from array A and at most $\log n$ elements from array B, Stage 2 employs $O(n/\log n)$ serial algorithms, each takes $O(\log n)$ time. The total work is, therefore, $O(n)$ and the time is $O(\log n)$.

Theorem 4.1: *The merging/ranking algorithm runs in $O(n)$ work and $O(\log n)$ time.*

Exercise 6: *Consider the merging problem as above, and assume that the values of the input elements are not pairwise distinct. Adapt the merging algorithm for this problem, so that it will take the same work and the same running time.*

Exercise 7: *Consider the merging problem as above, and assume that the values of n and m are not equal. Adapt the merging algorithm for this problem. What are the new work and time complexities?*

4.2. Technique: Divide and Conquer; Problem: Sorting-by-merging

Divide and conquer is a useful paradigm for designing serial algorithms. Below, we show that it can also be useful for designing a parallel algorithm for the **sorting problem**.

Input: An array of n elements $A = A(1), \ldots, A(n)$, drawn from a totally ordered domain.

The sorting problem is to reorder (permute) the elements of A into an array $B = B(1), \ldots, B(n)$, such that $B(1) \le B(2) \le \ldots \le B(n)$.

Sorting-by-merging is a classic serial algorithm. This section demonstrates that sometimes a serial algorithm translates directly into a reasonably efficient parallel algorithm. A recursive description follows.

$MERGE-SORT(A(1), A(2), \ldots, A(n); B(1), B(2), \ldots, B(n))$

Assume that $n = 2^l$ for some integer $l \ge 0$

if $n = 1$
then return $B(1) := A(1)$
else call, in parallel, $MERGE-SORT(A(1), \ldots, A(n/2); C(1), \ldots C(n/2))$ and
- $MERGE-SORT(A(n/2+1), \ldots, A(n); C(n/2+1), \ldots, C(n))$
- Merge $C(1), \ldots C(n/2))$ and $C(n/2+1), \ldots, C(n))$ into $B(1), B(2), \ldots, B(n))$

Example Figure 7 shows how the recursion unfolds for an input of eight elements

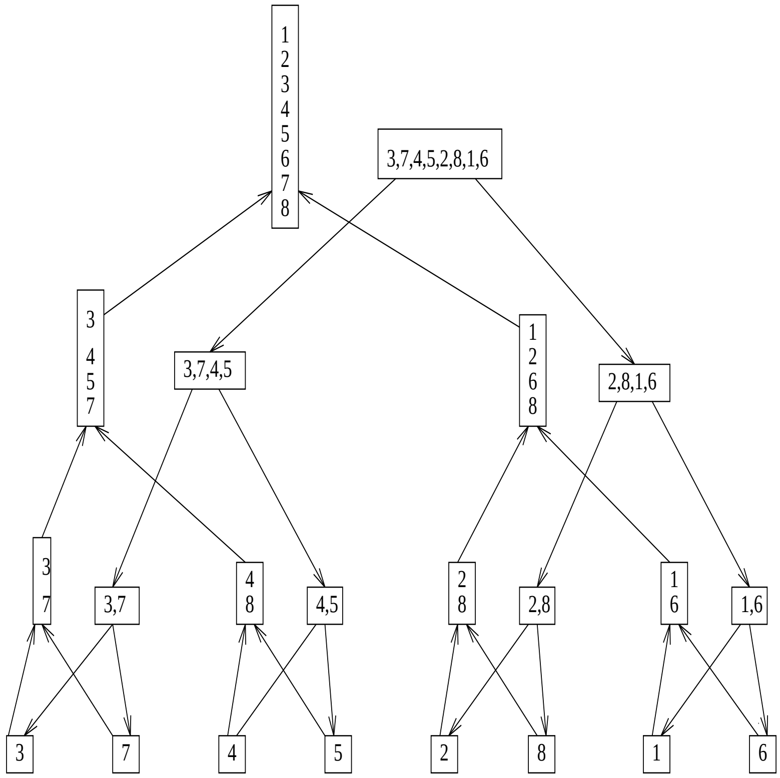


Figure 7: Example of the merge-sort algorithm

$A = (3, 7, 4, 5, 2, 8, 1, 6)$. The eight elements are repeatedly split into two sets till singleton sets are obtained; in the figure, move downwards as this process progresses. The single elements are then repeatedly pairwise merged till the original eight elements are sorted; in the figure, move upwards as this process progresses.

Complexity. The merging algorithm of the previous section runs in logarithmic time and linear work. Hence, the time and work of merge-sort satisfies the following recurrence inequalities:

$$T(n) \leq T(n/2) + \alpha \log n; \quad W(n) \leq 2W(n/2) + \beta n$$

where $\alpha, \beta > 0$ are constants. The solutions are $T(n) = O(\log^2 n)$ and

$W(n) \;=\; O(n \log n)$.

Theorem 4.2: *Merge-sort runs in $O(n \log n)$ work and $O(\log^2 n)$ time.*

Similar to the prefix-sums algorithm above, the merge-sort algorithm could have also been classified as a "balanced binary tree" algorithm. In order to see the connection, try to give a non-recursive description of merge-sort.

5. "Putting Things Together" - a First Example. Techniques: Informal Data-Parallelism (IDP), and Accelerating Cascades; Problem: Selection

One of the biggest challenges in designing parallel algorithms is overall planning ("macro-design"). In this section, we will learn two macro-techniques: (1) Accelerating Cascades provides a way for taking several parallel algorithms for a given problem and deriving out of them a parallel algorithm which is more efficient than any of them; and (2) Informal Data-Parallelism (IDP) - a description methodology for parallel algorithm, which uses a certain kind of informal approach to Data-Parallelism for high-level descriptions. In this section, we demonstrate these techniques for designing a fast parallel algorithm for the selection problem, that needs only $O(n)$ work.

The selection problem

Input: An array $A \;=\; A(1), A(2), \ldots, A(n)$ of n elements drawn from a totally ordered domain, and an integer k, $1 \le k \le n$.

An element $A(j)$ is a k-th smallest element of A if at most $k-1$ elements of A are smaller and at most $n-k$ elements are larger. Formally, $|\{i : A(i) < A(j) \; for \; 1 \le i \le n\}| \le k-1$ and $|\{i : A(i) > A(j) \; for \; 1 \le i \le n\}| \le n-k$.

The selection problem is to find a k-th smallest element of A.

Example. Let $A \;=\; 9, 7, 2, 3, 8, 5, 7, 4, 2, 3, 5, 6$ be an array of 12 integers, and let $k = 4$. Then, each among $A(4)$ and $A(10)$ (whose value is 3) is a 4-th smallest element of A. For $k = 5$, $A(8) = 4$ is the only 5-th smallest element.

The selection problem has several notable instances. Note, that if $k = 1$, the selection problem becomes the problem of finding the minimum element in A, and if $k = n$, the selection problem becomes the problem of finding the maximum element in A. For $k = \lceil n/2 \rceil$ the k-smallest element is called the *median* of A.

5.1. Accelerating Cascades

For devising the fast $O(n)$-work algorithm for the selection problem, we will use two building blocks. Each building block is itself an algorithm for the problem:
(1) Algorithm 1 works in $O(\log n)$ iterations, each takes an instance of the selection problem of size, say, m, and reduces it in $O(\log m)$ time and $O(m)$ work to an instance of the selection problem whose size is bounded by a fraction of m (specifically, $\leq 3m/4$). Concluding that the total running time of such an algorithm is $O(\log^2 n)$ and total work is $O(n)$ is easy.
(2) Algorithm 2 runs in $O(\log n)$ time and $O(n \log n)$ work.

The advantage of Algorithm 1 is that it needs only $O(n)$ work, while the advantage of Algorithm 2 is that it is faster. The accelerating cascades technique gives a way for combining these algorithms into a single algorithm which is fast and need $O(n)$ work. This is demonstrated next.

An ultra-high-level description of the selection algorithm
Step 1. Apply Algorithm 1 for $O(\log \log n)$ rounds to reduce an original instance for the selection problem into an instance of size $\leq n/\log n$.
Step 2. Apply Algorithm 2.

Complexity analysis. We first show that $r = O(\log \log n)$ rounds are enough to bring the size of the problem below $n/\log n$. We need that $(3/4)^r n \leq n/\log n$, so $(4/3)^r \geq \log n$. The smallest value of r for which this holds is $\log_{4/3} \log n$ which is indeed $O(\log \log n)$. So, Step 1 takes $O(\log n \log \log n)$ time. The number of operations is $\sum_{i=0}^{r-1} (3/4)^i n = O(n)$. Step 2 takes additional $O(\log n)$ time and $O(n)$ work. So, in total, we get $O(\log n \log \log n)$ time, and $O(n)$ work.

This has actually been an illustration of the accelerating cascades technique. The general technique follows.

The Accelerating Cascades Technique
Consider the following situation: for a given problem of size n we have two parallel algorithms. Algorithm A performs $W_1(n)$ operations in $T_1(n)$ time, and Algorithm B performs $W_2(n)$ operations in $T_2(n)$ time. Suppose that Algorithm A is more efficient ($W_1(n) < W_2(n)$), while Algorithm B is faster ($T_1(n) < T_2(n)$). Assume also that Algorithm A is a "reducing algorithm" in the following sense. Given a problem of size n, Algorithm A operates in phases where the output of each successive phase is a smaller instance of the problem. The accelerating cascades technique composes a new algorithm out of algorithms A and B, as follows. Start by applying Algorithm A. Once the output size of a phase of this algorithm is below some threshold, finish by switching to Algorithm B. It is easy to see from the ultra-high-level description of the selection algorithm that it indeed demonstrates the accelerating cascades technique. It is, of course, possible that

instead of the first algorithm there will be a chain of several reducing algorithms with increasing work and decreasing running times. The composed algorithm will start with the slowest (and most work-efficient algorithm), switch to the second-slowest (and second most work-efficient), and so on, until the fastest and least work-efficient one. Finally, it will switch to the last algorithm which need not be a reducing one.

Remark The accelerating cascades framework permits the following kind of budget considerations in the design of parallel algorithms and programs. Given a "budget" for the total number of operations (i.e., some upper bound on the total number of operations allowed), the framework enables to determine the fastest running time that one can achieve from given building blocks. It should be clear that such budgeting extends to *constant factors* and need not stop at the asymptotic level. Similarly, given a budget for running time, the framework enables to determine the most work-efficient implementation that one can achieve from given building blocks.

5.2. The Selection Algorithm (continued)

Algorithm 2 is actually a sorting algorithm. Observe that selection is an instance of the sorting problem, since a k-th smallest element for every $1 \leq k \leq n$ is found. Elsewhere in these notes, we reference sorting algorithms that run in time $O(\log n)$ using $O(n \log n)$ work.

The remainder of this section is devoted to presenting Algorithm 1. A high-level description of Algorithm 1 is given next. Some features of the high-level description of Algorithm 1 demonstrate the IDP technique which is explained. Some missing details for Algorithm 1 are finally filled in.

Input. A instance $A = A(1), \ldots, A(n)$ of size n and an integer k, $1 \leq k \leq n$, where we need to find the k-th smallest element in A.

The algorithm works in "reducing" iterations. The input to an iteration is a subset $B = \{B(1), \ldots, B(m)\}$ of size m of A and an integer k_0, $1 \leq k_0 \leq m$, and we need to find the k_0-th element in B. The main idea behind the reducing iteration is to find an element α of B which is guaranteed to be not too small (specifically, at least $m/4$ elements of B are smaller), and not too large (at least $m/4$ elements of B are larger) - see the Reducing Lemma below. An exact ranking of α in B enables to conclude that at least $m/4$ elements of B do not contain the k_0-th smallest element, and, therefore, can be discarded; the other alternative being that the k_0-th smallest element (which is also the k-th smallest element with respect to the original input) has been found - see Corollary 1.

ALGORITHM 1 - High-level description

(Assume below, for simplicity, that $\log m$ and $m/\log m$ are integers.)
1. **for** i, $1 \leq i \leq n$ **pardo**
- $B(i) := A(i)$
2. $k_0 := k$; $m := n$
3. **while** $m > 1$ **do**
3.1. Partition B into $m/\log m$ blocks, each of size $\log m$ as follows. Denote
- the blocks $B_1, \ldots, B_{m/\log m}$, where $B_1 = B[1, \ldots, \log m]$,
- $B_2 = B[\log m + 1, \ldots, 2\log m], \ldots, B_{m/\log m} = B[m - \log m + 1, \ldots, m]$.
3.2. **for** block $B_i, 1 \leq i \leq m/\log m$ **pardo**
- compute the median α_i of B_i, using a linear time serial selection algorithm
3.3. Apply a sorting algorithm to find α the median of the medians $(\alpha_1, \ldots, \alpha_{m/\log m})$.
3.4. Compute s_1, the number of elements in B which are smaller than α, s_2, the number of elements equal to α, and s_3, the number of elements larger than α.
3.5. There are three possibilities:
3.5.1 (i) $k_0 \leq s_1$: the new subset B (which is the input for the next iteration) consists
- of the elements in B, which are smaller than α ($m := s_1$), and k_0 remains the same.
3.5.2 (ii) $s_1 < k_0 \leq s_1 + s_2$: α itself is the k_0-th smallest element in B, and the
- whole selection algorithm is finished.
3.5.3 (iii) $k_0 > s_1 + s_2$: the new subset B consists of the elements in B, which
- are larger than α ($m := s_3$), and $k_0 := k_0 - (s_1 + s_2)$.
4. (*we can reach this instruction only with* $m = 1$ *and* $k_0 = 1$)
- $B(1)$ is the k_0-th element in B.

Reducing Lemma. At least $m/4$ elements of B are smaller than α, and at least $m/4$ are larger.

Proof of the Reducing Lemma. α is the median of the medians $(\alpha_1, \ldots, \alpha_{m/\log m})$. Therefore, at least half of the medians are $\leq \alpha$. Consider a median α_i which is $\leq \alpha$. At least half the elements in its block B_i are $\leq \alpha_i$, and therefore $\leq \alpha$. We conclude that at least a quarter of the elements in B are $\leq \alpha$. An illustrative explanation is given in Figure 8, where each row represents a block of size $\log m$; we assume (for the figure only) that each block is sorted, and its median is the middle element of the row and that the rows are sorted by increasing values of their medians; the upper left quadrant (whose lower right endpoint is α) contains $\geq m/4$ elements, whose values are at most α.
The proof that at least a quarter of the elements in B are $\geq \alpha$ is similar.

Corollary 1. Following an iteration of Algorithm 1 the value of m decreases so that the new value of m, denoted here m', is at most $(3/4)m$.

Proof If $k_0 \leq s_1$ (case (i) in Step 3.5), then $s_2 + s_3 \geq m/4$; therefore, the m' is $s_1 \leq (3/4)m$. If $k_0 \geq s_1 + s_2$ (case (iii) in Step 3.5), then $s_1 + s_2 \geq m/4$, and therefore m' is $s_3 \leq (3/4)m$.

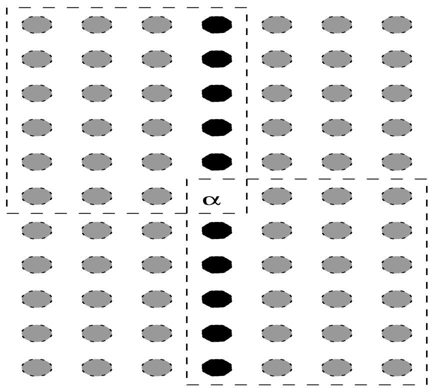


Figure 8: Proof that $\geq 1/4$ of the elements in B are $\geq \alpha$ and that $\geq 1/4$ are $\leq \alpha$

Our presentation of Algorithm 1 follows a general description methodology. We first present this methodology and then go back to wrap up the description of Algorithm 1.

5.3. A top-down description methodology for parallel algorithms

The methodology suggests to describe Data-Parallel (or PRAM) algorithms in an informal two-level top-down manner. The upper level (systematically) suppresses some specific details, which are then added in the low-level.

The Upper Level - an Informal Data-Parallel (IDP) description. Similar to Data-Parallelism, *the algorithm is presented in terms of a sequence of parallel time units (or "rounds", or "pulses" - which might represent a cluster of time units); however, at each time unit there is a* **set** *containing any number of instructions to be performed concurrently.*
The difference with respect to Data-Parallelism is that the instructions of a time unit are not required to be given as a sequence. Descriptions of the set of concurrent instructions can come in several flavors, and we will even tolerate implicit descriptions, where the number of instruction is not obvious. We call such descriptions Informal Data-Parallel (IDP).
An example for an IDP description is the high-level description of Algorithm 1 above. The input for each reducing iteration is given in the form of a set. We were also not specific on how to compute s_1, s_2 and s_3.
Important Comment. The main methodological issue which is addressed in these notes is how to train computer science and engineering professionals to **"to think in parallel"**. These notes suggest the following informal answer: *train yourself to provide*

IDP description of parallel algorithms. The rest is detail (although important).

The Lower Level. Present the algorithm as a Data-Parallel one. The job is to transform the (possibly implicit) set representation of the concurrent instructions, in each time unit, into a sequence. It is possible that a time unit in the upper level will require more than one time unit in the lower level. Numerous examples enable us to make the following circumstantial statement: *inserting later the details omitted by the upper level without increasing by "too much" the time or work requirements is often not very difficult.*

5.4. The Selection Algorithm (wrap-up)

To derive the lower level description of Algorithm 1, we simply apply the prefix-sums algorithm several times. We will assume that the input to an iteration includes the set B as an array, and show how to provide the output of an iteration in this form. A new set B can be generated in either instruction 3.5.1 or 3.5.3. In each of these cases, use the compaction algorithm (which was given above as an application of the prefix-sums algorithm) to get the output of an iteration in an array form. In instruction 3.4 it is enough to use a summation algorithm for computing s_1, s_2 and s_3. Each of these adds a time of $O(\log m)$ and $O(m)$ work to an iteration.

Complexity. Step 1 takes $O(n)$ work and $O(1)$ time. Step 2 takes $O(1)$ work. Each iteration of Step 3 runs in $O(\log m)$ time and $O(m)$ work: Step 3.1 take $O(1)$ time and $O(m)$ work; Step 3.2 takes $O(\log m)$ time and work, per block, and a total of $O(\log m)$ time and $O(m)$; using a $O(n \log n)$ work and $O(\log n)$ time sorting algorithm, Step 3.3 takes $O(m)$ work and $O(\log m)$ time; we already explained that the computation consuming parts of steps 3.4 and 3.5 apply summation, or prefix-sums algorithm and therefore, each takes $O(m)$ work and $O(\log m)$ time. This means the an upper bound on the total amount of work in Step 3 is proportional to $n + (3/4)n + (3/4)^2 n + \ldots$, which is $O(n)$. Since the number of iterations is $O(\log n)$, the total running time of Step 3 is $O(\log^2 n)$. Step 4 takes $O(1)$ work. So, the running time of Algorithm 1 for the selection problem is $O(\log^2 n)$ and the work is $O(n)$.

Theorem 5.1: *Algorithm 1 solves the selection problem is $O(\log^2 n)$ time and $O(n)$ work. The main selection algorithm of this chapter which is composed of algorithms 1 and 2 runs in $O(n)$ work and $O(\log n \log\log n)$ time.*

Exercise 8: *Consider the following sorting algorithm. Find the median element and then continue by sorting separately the elements larger than the median and the ones smaller than the median. Explain why this is indeed a sorting algorithm. What will be the time and work complexities of such algorithm?*

Summary of things learned in this chapter.
The accelerating cascades framework was presented and demonstrated by the selection algorithm. A top-down methodology for describing parallel algorithms was presented. Its upper level, called Informal Data-Parallelism (IDP), is being suggested as the essence of thinking in parallel; as an example, an IDP presentation is given for one of the building blocks for the selection algorithm.
Some literature on parallel algorithms refers to the Work-Time (WT) methodology for describing parallel algorithms. The difference with respect to our approach is rather small. Usually, the WT methodology suggests also the IDP for informal high level presentation of algorithms, but then aims directly at the PRAM for the lower level. The small difference here is that we systematically replaced the PRAM by Data-Parallelism (DP) early on, and our lower level aims at DP.

6. Integer Sorting

Input An array $A = A(1), \ldots, A(n)$ of integers from the range $[0, \ldots, r-1]$.
The *sorting problem* is to rank the elements from smallest to largest.
For simplicity assume that n is divisible by r. A typical value for r might be $n^{1/2}$, but other values are of course possible.

We mention two interesting things about the algorithm below. (i) Its performance depends on the value of r, and unlike other parallel algorithms we have seen, its running time may not be bounded by $O(\log^k n)$ for any constant k (such bound is sometimes called *poly-logarithmic*). It is a remarkable coincidence that the literature includes only very few work-efficient parallel algorithm whose running time is not poly-logarithmic. (ii) It lends itself to efficient implementation on several parallel machines; see a remark at the end of this section.

The algorithm works as follows.
Step 1 Partition array A into n/r subarrays of size r each, $B_1 = A[1, \ldots, r], B_2 = A[r+1, \ldots, 2r], \ldots, B_{n/r} = A[n-r+1, \ldots, n]$. Using a serial bucket sort (see also Exercise 10 below) algorithm, sort each subarray separately (and in parallel for all subarrays). Also compute:
(1) $number(v, s)$ - the number of elements whose value is v, in subarray B_s, for $0 \leq v \leq r-1$, and $1 \leq s \leq n/r$; and (2) $serial(i)$ - the number of elements $A(j)$ such that $A(j) = A(i)$ and precede element i in its subarray B_s (note that since element i is in subarray s, $serial(i)$ counts only $j < i$, where $\lceil j/r \rceil = \lceil i/r \rceil = s$), for $1 \leq i \leq n$.
Example Let n be some value and $r = 4$. Let $B_1 = (2, 3, 2, 2)$. Then, $number(2, 1) = 3$, since the value of three out of the four elements of B_1 is 2, and $serial(3) = 1$ since there is one element, that is $A(1)$, in B_1 which is equal to $A(3)$.

Step 2. Separately (and in parallel) for each value v, $0 \leq v \leq r-1$, compute the prefix-sums of $number(v,1), number(v,2), \ldots, number(v,n/r)$ into $ps(v,1), ps(v,2), \ldots, ps(v,n/r)$, and their sum, which reflects the number of elements whose value is v, into $cardinality(v)$.

Step 3. Compute the prefix sums of $cardinality(0), cardinality(1), \ldots, cardinality(r-1)$ into $global-ps(0), global-ps(1), \ldots, global-ps(r-1)$, respectively.

Step 4. For every element i, $1 \leq i \leq n$ do the following in parallel. Let $v = A(i)$ and B_s be the subarray of element i ($s = \lceil i/r \rceil$). The rank of element i for the sorting problem is

$$1 \ + \ serial(i) \ + \ ps(v, s-1) \ + \ global - ps(v-1)$$

(where $ps(0,s)$ and $global-ps(0)$ are defined as 0, each). See also Figure 9, where the

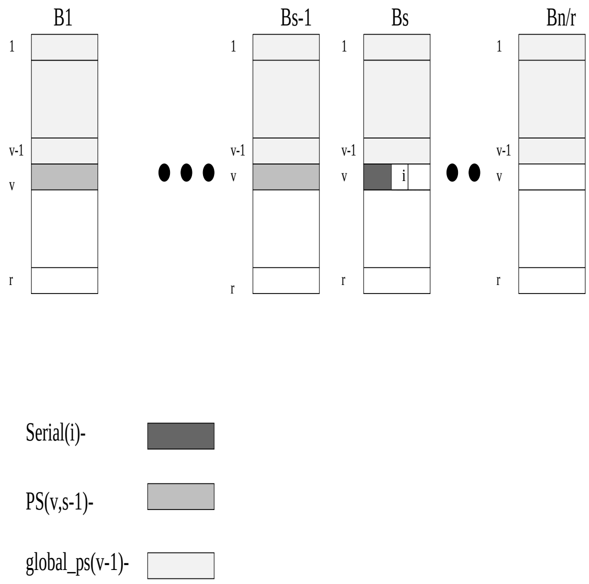


Figure 9: Output of Step 1

output of Step 1 is sorted into each box. The elements of B_1 are in the leftmost box: those whose value is 0 appear highest (within each row they appear in the order of their original indices), followed by those with value 1, and finally the ones with value $r-1$. The other boxes represent subarrays $B_2, \ldots, B_{n/r}$. The elements counted in $serial(i)$ are marked by diagonals which increase from left to right. Elements counted in $ps(v, s-1)$ are shaded. Elements counted in $global-ps(v-1)$ are marked by diagonals which decrease from left to right.

Exercise 9: *Describe the integer sorting algorithm in a "parallel program", similar to the pseudo-code that we usually give.*

Complexity Step 1 takes $O(r)$ time and $O(r)$ work per subarray and a total of $O(r)$ time and $O(n)$ work. In Step 2, there are r computations, each taking $O(\log(n/r))$, which is $O(\log n)$, time and $O(n/r)$ work, or a total of $O(\log n)$ time and $O(n)$ work. Step 3, $O(\log r)$ time and $O(r)$ work and Step 4, $O(1)$ time and $O(n)$ work. In total, we get $O(r + \log n)$ time and $O(n)$ work.

Theorem 6.1: *(1) The integer algorithm runs in $O(r + \log n)$ time and $O(n)$ work. (2) The integer sorting algorithm can be applied to run in time $O(k(r^{(1/k)} + \log n))$ and $O(kn)$ work for any positive integer k.*

Item (1) has been shown above. Item (2) is based on applying the integer sorting algorithm in the same way that radix sort uses bucket sort. A sorting algorithm is *stable* if for every pair of two equal input elements $A(i) = A(j)$ where $1 \leq i < j \leq n$, it outputs a smaller rank for element i than for element j. Observe that the integer sort algorithm is stable. Instead of giving a formal proof of item (2) in Theorem 6.1 we describe a two-step algorithm for an integer sort problem with $r = n$, which runs in $O(\sqrt{n})$ time and $O(n)$ work. Assume that $\sqrt{n}$ is an integer.

Step 1. Apply the integer sorting algorithm to sort array A using $A(1) \pmod{\sqrt{n}}$, $A(2) \pmod{\sqrt{n}}, \ldots,$ $A(n) \pmod{\sqrt{n}}$ as keys. If the computed rank of an element i is j then set $B(j) := A(i)$.

Step 2. Apply again the integer sorting algorithm this time to sort array B using $\lfloor B(1)/\sqrt{n} \rfloor, \lfloor B(2)/\sqrt{n} \rfloor, \ldots, \lfloor B(n)/\sqrt{n} \rfloor$ as keys.

Example Let $A = 10, 12, 9, 2, 3, 11, 10, 12, 4, 5, 9, 4, 3, 7, 15, 1$ with $n = 16$ and $r = 16$. The keys for the first step will be the values of A modulo 4. That is, $2, 0, 1, 2, 3, 3, 2, 0, 0, 1, 1, 0, 3, 3, 3, 1$. The sorting and assignment into array B will lead to $12, 12, 4, 4, 9, 5, 9, 1, 10, 2, 10, 3, 11, 3, 15$. In Step 2, the keys to be sorted are $\lfloor v/4 \rfloor$, where v is the value of an element of B. For instance $\lfloor 9/4 \rfloor = 2$. So, the values will be $3, 3, 1, 1, 2, 1, 2, 0, 2, 0, 2, 0, 2, 0, 3$ and the result relative to the original values of A is $1, 2, 3, 3, 4, 5, 7, 9, 9, 10, 10, 11, 12, 12, 15$.

Remark This simple integer sorting algorithm has led to efficient implementation on parallel machines such as some Cray machines and the Connection Machine (CM-2). See [BLM$^+$91] and [ZB91], who report that it gave competitive performance on the machines that they examined. Besides it simplicity, the algorithm enables to break the input into the local memories of (physical) processors, and compute Step 1, without any interprocessor communication. In steps 2 and 3, communication is used for applying the prefix-sums routine; however, some machines have even special constructs that enable very fast implementation of such a routine.

Exercise 10: *(This exercise is redundant for those students, who remember the serial bucket-sort algorithm).*
The serial bucket-sort (called also bin-sort) algorithm works as follows. Input An array $A = A(1), \ldots, A(n)$ *of integers from the range* $[0, \ldots, n-1]$*. For each value* v, $0 \leq v \leq n-1$*, the algorithm forms a linked of all elements* $A(i) = v$, $0 \leq i \leq n-1$*. Initially, all lists are empty. Then, at step* i, $0 \leq i \leq n-1$*, element* $A(i)$ *is inserted to the linked list of value* v*, where* $v = A(i)$*. Finally, the linked list are traversed from value 0 to value* $n-1$*, and all the input elements are ranked. (1) Describe this serial bucket-sort algorithm in pseudo-code using a "structured programming style". Make sure that the version you describe provides stables sorting. (2) Show that the time complexity is* $O(n)$*.*

6.1. An orthogonal-trees algorithm

Consider the integer sorting problem where the integers are in the range $[1 \ldots n]$. An alternative which uses "orthogonal trees" also works for this case. For each possible integer value v, $1 \leq v \leq n$, we keep a separate balanced binary tree with n leaves. Assign processor i, $1 \leq i \leq n$ to input element $A(i) = v_i$. Starting with leaf number i in tree v_i advance in $\log n$ steps towards the root of tree v_i, computing in the process the number of elements whose value is v_i.

Exercise 11: *Show how to finish this sorting algorithm so that it will run in* $O(\log n)$ *time,* $O(n \log n)$ *work, and* $O(n^2)$ *space.*

7. 2-3 trees; Technique: Pipelining

Pipelining is an important paradigm in computer science. By way of example, consider a sequence of stages like in a car assembly line. Upon finishing a stage the car advances to the next stage. Different cars can be at different stages at the same time, and assume that there are s stages each taking the same time (to be called "unit" time). The pipelining paradigm resembles the following. At the beginning the first car enters the first stage. Upon finishing the first stage, the first car advances to the second stage, while a second car enters the first stage. At the time in which the first car enters stage s, the second enters stage $s-1$ and car s enters the first stage. Figure 10 illustrates a four stage assembly line. It is not hard to see that t cars will be done in $s+t-1$ time,

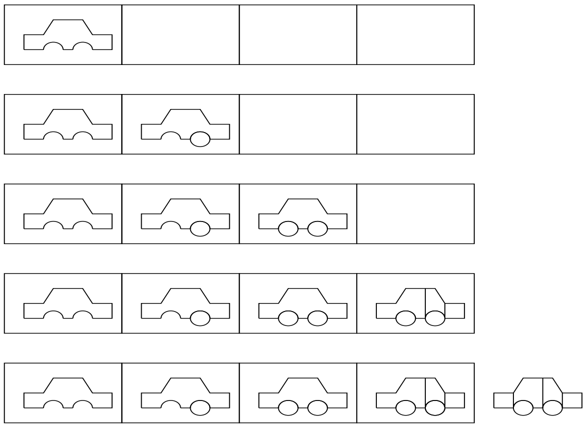


Figure 10: 4-stage assembly line with pipelining

had we waited to finish with one car before entering the second this would have taken st time. We demonstrate how to use this paradigm for the design of parallel algorithms.

A **2-3 tree** is a rooted tree with the following two properties: (1) each internal node of the tree has two (called left and right) or three (left, middle and right) ordered children; and (2) given any internal node of the 2-3 tree, the length of every directed path from this node to any leaf in its subtree is the same; this length, counting edges is called the *height* of the tree; the height of a 2-3 tree with n leaves is not less than $\log_3 n$ and not more than $\log_2 n$.

2-3 trees are used as data-structures for representing sets of elements from a totally ordered domain, as follows. Consider a monotonically increasing sequence of n elements $a_1 < a_2 < \ldots < a_n$. A 2-3 tree with n leaves will represent the set $S = \{a_1, a_2, \ldots, a_n\}$, as follows. Leaf i of the tree stores element a_i for i, $1 \leq i \leq n$, respectively. Each internal node v stores the maximum (i.e., value of the largest element) in the subtree rooted at the left child of v in L(v); the maximum in the subtree of the right child in $R(v)$; and if v has a middle child, the maximum in its subtree rooted in $M(v)$. See Figure 11. for an example.

In serial computation, the 2-3 tree data-structure supports **several kinds of queries**:

search(a) - "is $a \in S$?"; specifically, determine whether $a \in S$; if yes find its leaf, and if not, find the smallest $b > a$ such that $b \in S$, if such b exists.

insert(a) - "$S := S \cup \{a\}$"; if a is not in S, add a leaf whose value is a.

delete(a) - "$S := S - \{a\}$"; if $a \in S$, delete the leaf whose value is a.

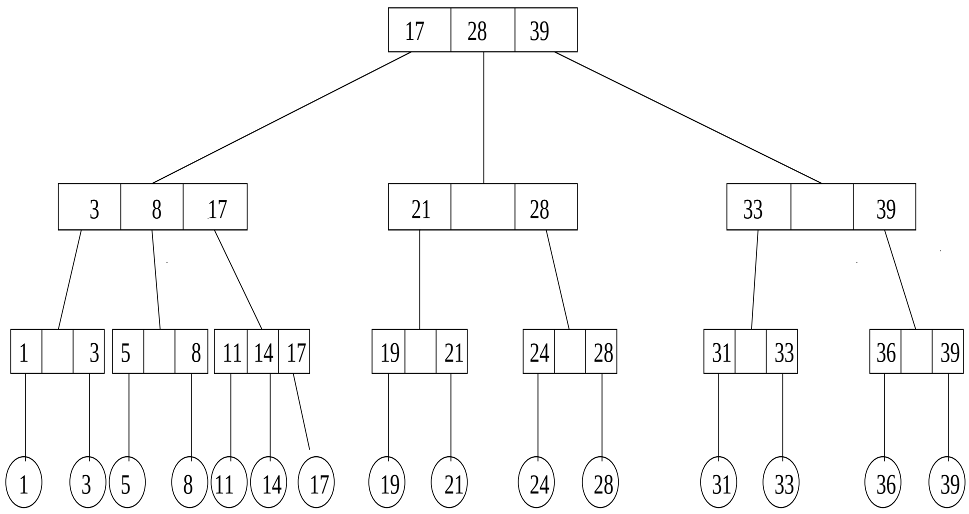


Figure 11: A 2-3 Tree

A data-structure which supports these three queries is called a **dictionary**. Interestingly, the serial algorithms presented for processing each of these queries are simpler than in some textbooks on serial algorithms, perhaps because those textbooks do not maintain $R(v)$.

7.1. Search

$search(a)$ - start from the root and advance down the tree deciding at each internal node which among its children is the one that might have the value a in one of its leaves (this search is similar to binary search).
Complexity The time is proportional to the height of the tree, which is $O(\log n)$. Allowing concurrent-reads, we can process k insert queries in $O(\log n)$ time using k processors.

7.1.1. Parallel search Suppose that we are given k search queries $search(a_1), search(a_2), \ldots, search(a_k)$, denoted also $search(a_1, a_2, \ldots, a_k)$, with a processor standing by each of them. Then, we can simply let each processor perform the serial algorithm separately in parallel.

7.2. Insert

We start with a serial procedure for processing $insert(a)$. The serial procedure is then enhanced to deal with a restricted version of the parallel problem. Finally, the restricted algorithm is used by means of pipelining for solving the general problem.

7.2.1. Serial processing The processing of $insert(a)$ begins with processing $search(a)$. If a leaf whose value is a is found, stop. Otherwise, assume (for now) that the largest leaf value is larger than a, and let b be the smallest leaf value such that $b > a$. The changes in the 2-3 tree are done using a routine called $absorb(a, b)$. In order to prepare the grounds for the parallel algorithms we assume that a single processor is standing by and does the computation. Initially the processor is standing by node b. It might be helpful to think about the routine as a "recursive process", since it may generate a call to itself with different parameters while advancing towards the root of the tree , however no recursive backtracking is involved. The same processor will stand by the new call as well. We describe only the structural changes to the 2-3 tree and leave the updates to the L, M and R fields as an exercise.

Verbal description of $absorb(a, b)$. If node b is the root of the 2-3 tree then $absorb(a, b)$ creates a new node, to be called $root$, which becomes the root of the tree. Node b is its right child and node a is its left child. Otherwise, let $parent$ be the parent of b. Routine $absorb(a, b)$ makes node a a new child of node $parent$ placing it as an immediate left sibling of b. If $parent$ has now three children, no more structural changes are needed, and routine $absorb$ halts. If $parent$ has four children, then add a new node $left-parent$. The two left children of $parent$ become the children of $left-parent$, while the two right children of $parent$ remain so. The processor advances to node $parent$ and a call to $absorb(left-parent, parent)$ will take care of adding $left-parent$ to the tree. See also an example in Figure 14.

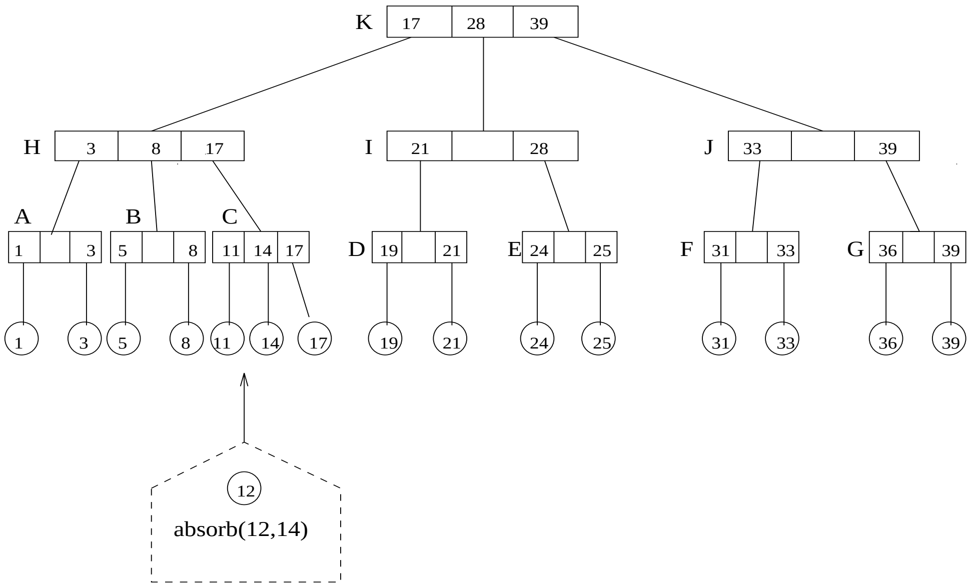


Figure 12: begin $insert(12)$

$absorb(a, b)$

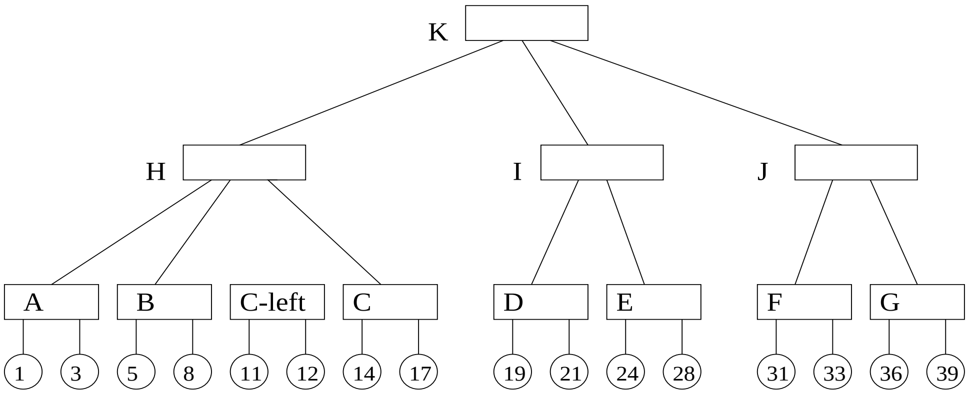


Figure 13: $absorb(C - LEFT, C)$

if b is the root of the 2-3 tree
then create a new node $root$, the new root of the tree; leaf b is
- the right child of $root$ and leaf a is its left child

else (let $parent$ be the parent of b) node a becomes a new child to node $parent$ as
- an immediate left sibling to node b.
- **if** $parent$ has more than three (i.e., four) children
- **then** add a new node $left - parent$; the two left children of $parent$ become the
- children of $left - parent$, while the two right children of $parent$ remain so;
- call $absorb(left - parent, parent)$.

Correctness. We need to show that if prior to the insertion the 2-3 tree represented the set S then after the insertion, we get a 2-3 tree which represents the set $S \cup \{a\}$. Following each application of routine $absorb$ two invariants are maintained: (1) each node is either a leaf or has two or three children; and (2) for any given node the length of every directed path from it to any leaf in its subtree is the same. The only possible anomaly is that the algorithm generates node which are not connected to a parent. However, prior to quitting the processor connects the unconnected node to a parent. Since the algorithm terminates, we eventually get a 2-3 tree. This tree represents the set $S \cup \{a\}$.

Complexity The time is $O(\log n)$.

It remains to discuss the case where a is larger than the largest element in the tree. Processing $insert(a)$ in this case is similar. Let c denote the largest element in the 2-3 tree. The only difference is that node a is added as a right (instead of left) sibling to node c with the same complexity.

7.2.2. Parallel processing Next, we proceed to parallel processing of several $insert$ queries.

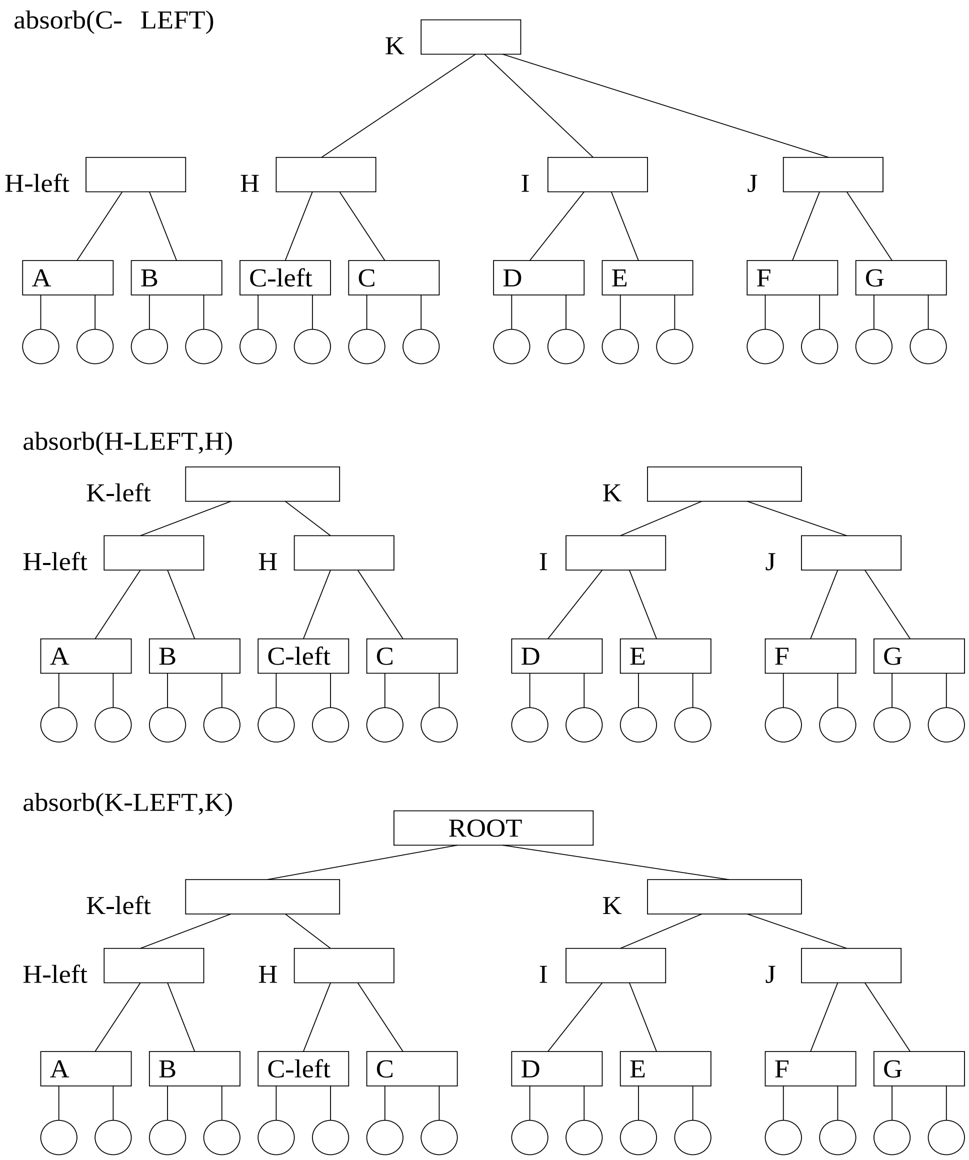


Figure 14: end $insert(12)$

Earlier, we used the attribute **"greedy-parallelism"**, for referring to parallel algorithms that at each point in time seek to break the problem at hand into as many independent tasks as possible. The algorithm below is based on: (1) a subtler characterization notion of "independent tasks" (i.e., tasks that can be performed concurrently); (2) applying pipelining.

Assume that k (sorted) elements $c_1 < \ldots < c_k$ are given. A query of the form $insert(c_1, \ldots, c_k)$ means that we want to insert these k elements into the 2-3 tree. We will assume that processor P_i is standing by element c_i, for each i, $1 \leq i \leq k$, respectively.

If the elements arrive unsorted, then they can be sorted within the time bounds of the parallel insert algorithm using available parallel sorting algorithms.

Restricted parallel problem

We first solve a restricted problem and then advance to the general one. Let b_i be the smallest leaf value which satisfies $b_i \geq c_i$, for $1 \leq i \leq k$. For the **restricted problem** we assume that all these b_i values exist and are pairwise distinct. The insertion algorithm works as follows. Later, we explain how to update the *absorb* routine so that it will fit the parallel setting.

$insert(c_1, \ldots, c_k)$

for P_i, $1 \leq i \leq k$ **pardo**

\- perform $search(c_i)$

\- **if** c_i is in the 2-3 tree

\- **then** quit

\- **else** call $absorb(c_i, b_i)$ (b_i is computed by $search(c_i)$)

The key observation is that all k routines $absorb(c_1, b_1)$, $absorb(c_2, b_2), \ldots, absorb(c_k, b_k)$ can begin at the same time at a respective subset (leaves) $b_1, \ldots, b_k$. Let $parent_1, \ldots, parent_k$ be the parents of nodes $b_1, \ldots, b_k$, respectively. Note that two *absorb* processes $absorb(c_i, b_i)$ and $absorb(c_j, b_j)$ "interact" only if $parent_i = parent_j$, and that at most three *absorb* processes can hit the same *parent* node (since at most one process can come from each child of *parent* in the 2-3 tree). If several different processes hit the same *parent*, only the one with the smallest serial number continues, while the others quit.

Processor P_i performing $absorb(c_i, b_i)$

if b_i is the root of the 2-3 tree

then create a new node *root*, the new root of the tree; leaf b_i is

\- the right child of *root* and leaf c_i is its left child

else (let $parent_i$ be the parent of b_i)

\- node c_i becomes a new child to node $parent_i$ as

\- an immediate left sibling to node b_i

\- **if** $parent_i$ has $s > 3$ children (up to 6 children are possible)

\- **then if** $parent_i = parent_j$ only for $j > i$

\- **then** add a new node $left-parent_i$; the $\lceil s/2 \rceil$ left children of *parent* become the

\- children of $left-parent_i$, while the $\lfloor s/2 \rfloor$ right children of *parent* remain so;

\- call $absorb(left-parent_i, parent_i)$

\- **else** (i.e., $parent_i = parent_j$ for some $j < i$) quit

Complexity The time is $O(\log n)$, using k processors.

General parallel problem

We are ready to consider the most general case where a query $insert(c_1, \ldots, c_k)$ needs to be processed. Given are k (sorted) elements $c_1 < \ldots < c_k$. and a processor P_i is standing by element c_i, for each i, $1 \leq i \leq k$, respectively. We show how to apply the above algorithm for the restricted problem. We first insert the largest element c_k. This guarantee that each element among $c_1, \ldots, c_{k-1}$ has a larger element in the 2-3 tree. However, we also want that each c_i, $1 \leq i \leq k-1$, will have a different smallest larger value $b_i > c_i$, $1 \leq i \leq k-1$. The idea is to recursively insert middle elements and thereby recursively bisect the chain $c_1, \ldots, c_{k-1}$.

Example. For $k = 8$, the order of insertion will be (1) c_4; (2) c_2 and c_6; and finally (3) c_1, c_3, c_5, c_7. For example, we show that when c_2 and c_6 are inserted they have different "smallest larger values": c_4 is already in the tree, and since c_4 is smaller than c_6 and larger than c_2, we conclude that c_2 and c_6 indeed have different "smallest larger values".

$insert(c_1, \ldots, c_k)$

\- Processor P_k : $insert(c_k)$

for P_i, $1 \leq i \leq k-1$ **pardo**

\- perform $search(c_i)$

\- **for** $t = 1$ to $\log k$ **do**

\- **if** i is divisible by $k/2^t$ but is not divisible by $k/2^{t-1}$

\- **then if** c_i is in the 2-3 tree

\- **then** quit

\- **else** call $absorb(c_i, b_i)$ (b_i is computed by $search(c_i)$)

Complexity The running time so far is $O(\log n \log k)$, using k processors. Below, we conclude the insertion algorithm by showing how to improve the running time using pipelining.

Pipelining The above procedure enters $\log k$ "waves" of the $absorb$ procedure into the 2-3 tree. Once the first wave reaches nodes of the tree which are at height 3, there is no impediment to initiating the second wave, and then the third wave and so on.

Complexity The first wave is finished in $O(\log n)$ time, the second in additional $O(1)$ time and then each wave takes $O(1)$ more time than its previous one. So the total running time is $O(\log n + \log k)$ using k processors.

We have shown,

Theorem 7.1: *Any k sorted elements can be inserted into a 2-3 tree in time $O(\log n)$ using k processors.*

Exercise 12: *Consider a 2-3 tree with nine leaves. The tree is a complete ternary tree;*

that is, the root has three children and each child has itself three children. The value of the nine leaves are $10, 20, 30, 40, 50, 60, 70, 80, 90$. *In this drilling question you are asked to show how the algorithm will process the query* $insert(41, 42, 43, 44, 45, 46, 47)$

7.3. Delete

We follow the footsteps of the insertion algorithm. For simplicity, we will assume throughout the delete section that we never reach a situation where following a deletion the 2-3 tree represents a singleton set.

7.3.1. Serial processing Perform $search(a)$. If no leaf whose value is a is found, stop. The changes in the 2-3 tree are done using a routine called $discard(a)$. Assume that a single processor is standing by and does the computation. Initially the processor is standing by node a. We describe only the structural changes to the 2-3 tree. Since we assumeed that following a delete the 2-3 tree represents a set of at least two elements, node a cannot be the root of the 2-3 tree. Let *parent* be the parent of a. Routine $discard(a)$ deletes the connection between node a and *parent*. If *parent* has now two children, no more structural changes are needed, and routine *discard* halts. If *parent* has one child (denoted b), then consider the total number of children of the siblings of *parent* plus 1 (for node b). If this number is at least 4, then reallocate the children to their parents at the level of node *parent* so that each parent has at least 2 children (possibly deleting one of the parents) and quit. If this number is 3, then allocate node b to the sibling of *parent*; if node *parent* is the root then declare node b the new root, and, otherwise, call *discard*(*parent*). See Figure 16.

    discard(a)
(let *parent* be the parent of a) delete the connection between node a and *parent*
if *parent* has less than two (i.e., one) children
then compute *count* := *the total number of children of the siblings of node*
-     *parent plus* 1
-     **if** *count* ≥ 4
-     **then** reallocate the children to their parents at the level of node *parent* so that
-         each parent has at least 2 children (possibly deleting a parent); quit
-       **else if** node *parent* is the root
-           **then** node b becomes the new root of the tree
-           **else** allocate node b to the sibling of *parent*; call *discard*(*parent*)

Correctness. We need to show that if prior to the insertion the 2-3 tree represented the set S then after the insertion, it became a 2-3 tree which represents the set $S - \{a\}$.

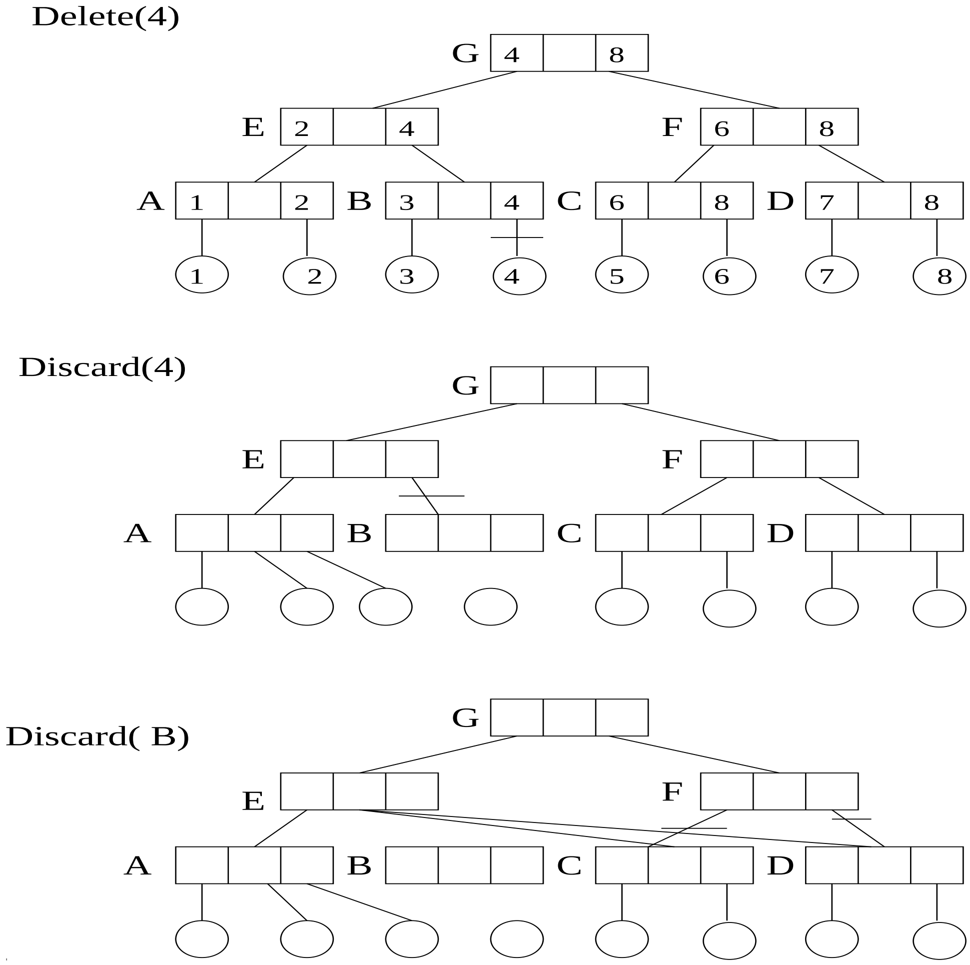


Figure 15: Processing *delete*(4): first *discard*(4), second *discard*(B)

Following each application of routine *discard* the following invariants are maintained: (1) there is at most one node (denoted v) which is not at the leaf level and has no children. (1) each other node is either a leaf or has two or three children; and (2) for any given node (other than v) the length of every directed path from it to any leaf in its subtree is the same. Prior to quitting the processor deletes the "childless node". Since the algorithm terminates, we eventually get a 2-3 tree. This tree represents the set $S - \{a\}$.

Complexity The time is $O(\log n)$.

7.3.2. Parallel processing Next, we proceed to parallel processing of several *delete* queries.

Assume that k (sorted) elements $c_1 < \ldots < c_k$ are given. A query of the form

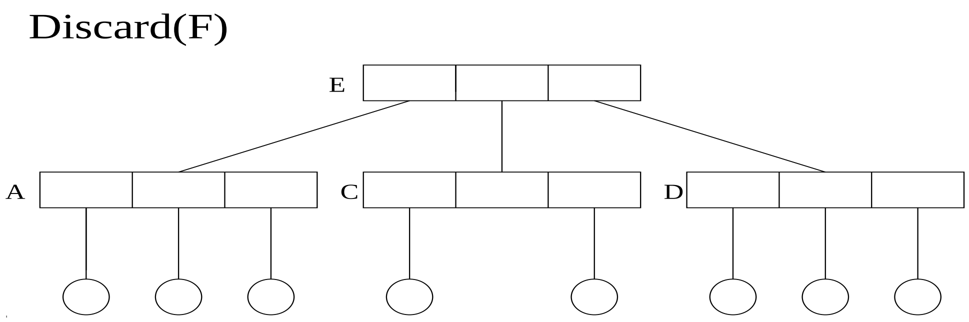


Figure 16: finally $discard(F)$

$delete(c_1, \ldots, c_k)$ means that we want to delete these k elements from the 2-3 tree. We will assume that processor P_i is standing by element c_i, for each i, $1 \leq i \leq k$, respectively. For simplicity, we assume that each c_i appears in the tree, since otherwise we can search for them and renumber the one that do. If the elements arrive unsorted, then they can be sorted within the time bounds of the parallel delete algorithm using available parallel sorting algorithms.

Restricted parallel problem

We first solve a restricted problem and then advance to the general one. For the **restricted problem** we assume that the parent of each c_i has at least one leaf which is not one of the c_i values. The deletion algorithm works as follows. Later, we explain how to update the *discard* routine so that it will fit the parallel setting.

$delete(c_1, \ldots, c_k)$
for P_i, $1 \leq i \leq k$ **pardo**
\- call $discard(c_i)$

The key observation is that all k routines $discard(c_1)$, $discard(c_2), \ldots, discard(c_k)$ can begin at the same time. Let $parent_1, \ldots, parent_k$ be the parents of nodes $c_1, \ldots, c_k$, respectively. Observe that: (i) two *discard* processes $discard(c_i)$ and $discard(c_j)$ "interact" only if $parent_i = parent_j$; (ii) at most two *discard* processes can hit the same *parent* node; and (iii) at least one child of a parent node is not to be discarded (this holds true initially because of the assumption of the restricted problem and also later on). If several different processes hit the same *parent*, only the one with the smallest serial number continues, while the others quit.

Processor P_i performing $discard(c_i)$
(let $parent_i$ be the parent of c_i and $grandparent_i$ the parent of $parent_i$, unless $parent_i$ is the root) delete the connection between node c_i and $parent_i$
if $grandparent_i$ exists
then if $grandparent_i = grandparent_j$ only for $j > i$

```
-      then compute count_i := the total number of grandchildren of grandparent_i
-         if count_i ≥ 4
-         then reallocate the grandchildren to their parents (at the level of node parent)
-            so that each parent has at least 2 children (possibly deleting a parent); quit
-         else (count_i = 2 or 3) allocate the 2 or 3 grandchildren to one of the nodes
-            at the parents level; pick another parent into b_i; delete additional parents;
-            call discard(b_i)
-      else quit
else (parent_i is the root)
-     if parent_i = parent_j only for j > i
-      then compute count_i := the total number of children of parent_i
-         if count_i ≥ 2
-         then quit
-         else (count_i = 1) delete parent_i and the (single) remaining sibling
-            of c_i becomes the new root; quit
-      else quit
```

Complexity The time is $O(\log n)$, using k processors.

General parallel problem

We are ready to consider the most general case where a query $delete(c_1, \ldots, c_k)$ needs to be processed. For simplicity though, we maintain the assumption that every c_i is in the tree. Given are k (sorted) elements $c_1 < \ldots < c_k$. and a processor P_i is standing by element c_i, for each i, $1 \leq i \leq k$, respectively. We show how to apply the above algorithm for the restricted problem. For this we want that the parent of each c_i, $1 \leq i \leq k-1$, will have at least one child which is not a candidate for deletion. The idea is to recursively delete odd numbered elements.

Example. For $k = 8$, the order of deletion will be (1) c_1, c_3, c_5 and c_7; (2) c_2 and c_6; (3) c_4; and finally (4) c_8. For example, we show that when c_2 and c_6 are deleted their parent must have at least one other node; this is since c_4 is still in the tree.

```
    delete(c_1, ..., c_k)
for P_i, 1 ≤ i ≤ k pardo
-     for t = 0 to log k do
-        if i (mod 2^(t+1)) = 2^t
-           then call discard(c_i)
```

Complexity The time is $O(\log n \log k)$, using k processors.

Pipelining The above procedure sends $\log k$ "waves" of the *discard* procedure into the 2-3 tree. Once the first wave reaches nodes of the tree which are at height 3, there is no impediment to initiating the second wave, and then the third wave and so on.

Complexity The first wave is finished in $O(\log n)$ time, the second in additional $O(1)$ time and then each wave takes $O(1)$ more time than its previous one. So the total running time is $O(\log n + \log k)$ using k processors.

We have shown,

Theorem 7.2: *Any k sorted elements can be deleted from a 2-3 tree in time $O(\log n)$ using k processors.*

Exercise 13: *Consider a 2-3 tree with sixteen leaves. The tree is a complete binary tree. The value of the sixteen leaves are the the integers $1, 2, \ldots, 16$. In this drilling question you are asked to show how the algorithm will process the query $delete(4, 5, 6, 7, 8)$.*

Exercise 14: *All algorithms in this section assume the CREW PRAM. (1) Show how to implement a parallel search on the EREW PRAM within the same complexity bounds. (2) Show how to do this for parallel insert, and parallel delete, as well.*

8. Maximum Finding

Very fast algorithms for the following problem are presented.

Input: An array $A = A(1), \ldots, A(n)$ of n elements drawn from a totally ordered domain.
The **maximum finding** problem is to find a largest element in A.

The summation algorithm, given earlier, implies an $O(\log n)$ time algorithm with $O(n)$ work; to see this simply substitute each addition operation by a binary operations that picks the maximum value of its two operands.

We give two algorithms which are even faster. The first is deterministic and runs in $O(\log \log n)$ (or *"doubly logarithmic"*) time and $O(n)$ work. The second is randomized and runs in $O(1)$ time and $O(n)$ work with high probability.

8.1. A doubly-logarithmic Paradigm

The presentation has three stages. (1) An algorithm that runs in $O(1)$ time and $O(n^2)$ work. (2) An algorithm that runs in $O(\log logn)$ time and $O(n \log \log n)$ work. (3)

Finally, we reduce the work to $O(n)$. The first two stages are described first for the informal data-parallel (IDP) level, and then for the data-parallel level.

IDP description of the first two stages:
A constant-time algorithm with $O(n^2)$ work Compare every pair of elements in $A[1 \ldots n]$, and conclude which element is the largest. **Complexity:** There are $n(n-1)/2 = O(n^2)$ pairs and they are compared simultaneously; this takes $O(n^2)$ operations and $O(1)$ time.
An $O \log\log n)$ time and $O(n \log\log n)$ work algorithm
For simplicity assume that $n = 2^{2^h}$ for some integer $h > 0$, which means that $h = \log\log n$. Let $A_1, A_2, \ldots, A_{\sqrt{n}}$ denote subarrays $A[1 \ldots \sqrt{n}], A[\sqrt{n}+1 \ldots 2\sqrt{n}], \ldots, A[n - \sqrt{n} + 1 \ldots n]$, respectively. Suppose that the maximum value for each subarray A_i, $1 \leq i \leq sqrtn$, has been computed. Then in one more round and $O(n)$ operations the maximum value among these $\sqrt{n}$ maxima is computed using the above constant time algorithm.
Complexity: The following two recursive inequalities hold true,

$$T(n) \leq T(\sqrt{n} + c_1); \quad W(n) \leq \sqrt{n} W(\sqrt{n}) + c_2 n$$

and imply, $T(n) = O(\log\log n)$, and $W(n) = O(n \log\log n)$.

DP descriptions: An auxiliary array B is initialized to zero. For simplicity, The second **for** statement below implies a sequence of n^2 concurrent operations (for every pair of values $1 \leq k, l \leq n$ the **for** statement implies that both variables i and j will get the values k and l implying redundant comparisons unlike the IDP description above; we do this for simplicity.) An element $A(k)$ is not the maximum if there are one or more other elements $A(l)$, such that $A(l) > A(k)$ or $A(l) = A(k)$ and $l < k$. In each of these cases the value 1 is written into $B(k)$; several write attempts into the same memory locations are possible, and the Common-CRCW convention is used. There will be exactly one element i for which $B(i)$ remains 0, and this element has the largest value in A.

for i, $1 \leq i \leq n$ **pardo**
\- $B(i) := 0$
for i and j, where $1 \leq i, j \leq n$ **pardo**
\- **if** either $A(i) < A(j)$, or $A(i) = A(j)$ and $i < j$
\- **then** $B(i) := 1$
\- **else** $B(j) := 1$
for i, $1 \leq i \leq n$ **pardo**
\- **if** $B(i) := 0$
\- **then** $A(i)$ is a maximum in A

This show that the maximum among n elements can indeed be found in $O(n^2)$ work and $O(1)$ time.

A balanced binary tree whose height is logarithmic was used to guide the computation in the summation (and prefix sums) algorithms above. For the DP presentation, we introduce another kind of a rooted tree whose height is doubly-logarithmic (i.e., $\log\log n$). The doubly-logarithmic tree is particularly helpful for more involved doubly-logarithmic time parallel algorithms. No such algorithm is given in these notes, though.

Doubly-logarithmic trees

The *level* of a node in the tree is the length, counting edges, of the path from the root to the node. If the level of a node of the tree is $s \leq \log\log n - 1$ then that node has $2^{2^{h-s-1}}$ children. However, if $s = \log\log n$ then the node will have two children which are leaves. For establishing the connection between the above IDP description and the doubly-logarithmic tree, observe that the rooted subtree of a node with $s \leq \log\log n - 1$ has $2^{2^{h-s}}$ leaves, and the number of its children is the square root of the number of its leaves (since $2^{2^{h-s-1}} = \sqrt{2^{2^{h-s}}}$). An example of a doubly-logarithmic tree for $n = 16$ is given in Figure Figure 17.

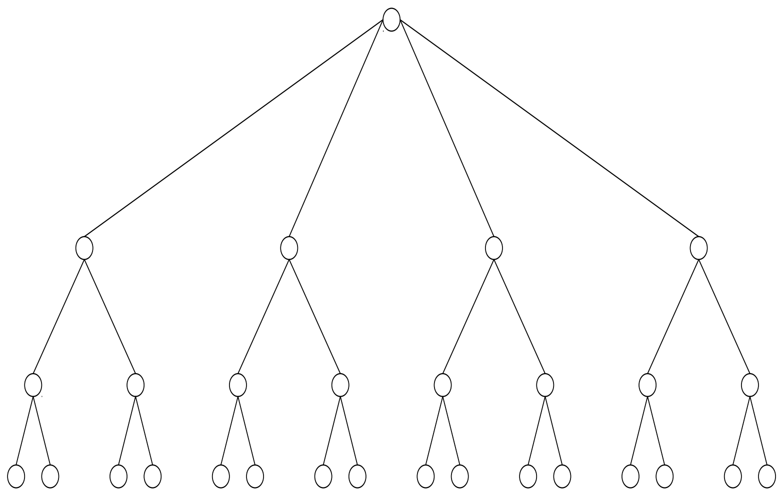


Figure 17: A doubly logarithmic tree

Guided by the doubly-logarithmic tree, the algorithm advances from the leaves to the root, one level at a time. Our description will focus on the advancement from level s to level $s-1$ for s, $1 \leq s \leq \log\log n - 1$. We start with $s = 1$ as an example. Inductively, each of the $2^{2^{h-1}}$ children of the root will have the maximum over the $2^{2^{h-1}}$ leaves in its subtree. Since, $2^{2^{h-1}} = \sqrt{n}$ the above constant-time algorithm is applied to finish the computation in $O(1)$ time using $O(n)$ work. For a general such s, each node v at level $s-1$ has $2^{2^{h-s-2}}$ children, where each child inductively has the maximum among its own $2^{2^{h-s-2}}$ leaves. So, we can find the maximum among the children of node v in constant time using $O((2^{2^{h-s-2}})^2) = O(2^{2^{h-s-1}})$ work. Since $2^{2^{h-s-1}}$ is the number of leaves in the subtree rooted at v, the total amount of work for each value of s is proportional to the total number of leaves, which is $O(n)$. To sum up, we just showed that one can advance one level in the tree in constant time and $O(n)$ work, or a total of $O(\log\log n)$ time and $O(n \log\log n)$ work.

An $O(\log\log n)$ time and $O(n)$ work algorithm
The algorithm works in two steps which follow the accelerating cascades paradigm.
Step 1 Partition the input array A into blocks of size $\log\log n$, as follows: $A[1 \ldots \log\log n], A[\log\log n + 1 \ldots 2\log\log n], \ldots, A[n - \log\log n + 1 \ldots n]$. To each block separately apply a linear time serial algorithm for finding its maximum. This gives $n/\log\log n$ maxima.
Step 2 Apply the above doubly-logarithmic time algorithm to the $n/\log\log n$ maxima.

Complexity Step 1 takes $O(\log\log n)$ time and $O(n)$ work and so does Step 2, and the whole algorithm.

Theorem 8.1: *The maximum finding problem can be solved in $O(\log\log n)$ time and optimal work on a Common CRCW PRAM.*

Exercise 15: *The nearest-one problem is restated. Input: An array A of size n of bit; that is the value of each entry of A is either 0 or 1. The nearest-one problem is to find for each i, $1 \le i \le n$, the largest index $j \le i$, such that $A(j) = 1$.*
(1) Give a parallel algorithm which runs in $O(n)$ work and $O(\log\log n)$ time for the problem. It might be helpful to first design a parallel algorithm that achieves this running time and $O(n \log\log n)$ work.
The input for the segmented prefix-min problem, includes the same binary array A as above, and in addition and array B of size n of numbers. The segmented prefix-min problem is to find for each i, $1 \le i \le n$, the minimum value among $A(j), A(j+1), \ldots, A(i)$, where j is the nearest-one for i (if i has no nearest-one we define its nearest-one to be 1).
(2) Give a parallel algorithm which runs in $O(n)$ work and $O(\log\log n)$ time for the problem. It might again be helpful to first design a parallel algorithm that achieves this running time and $O(n \log\log n)$ work.

Exercise 16: *Given is an array A of size n. Each $A(i)$ is either 0 or 1.*
(a) Give an algorithm that finds for each i in the array the largest $j < i$ for which $A(j) = 1$ (or the value 0 in no such j exists).
(Example: If $A(1,2,3,4,5,6,7,8,9,10,11) = (0,1,1,0,0,0,1,1,0,0,1)$ then $(0,0,2,3,3,3,3,7,8,8,8)$.)
Your algorithm should be as efficient as possible. In other words, it should take linear work and be as fast as you can.
(b) What is the work and time complexity of your algorithm? Explain.

8.2. Random Sampling

We show how to compute the maximum among n elements in $O(1)$ time and $O(n)$ work with very high probability, on an Arbitrary CRCW PRAM.

Step 1 Using an auxiliary array B of size $n^{7/8}$ we do the following.
Independently for each entry of B, pick at random (with equal probability) one of the input elements (of A).

Step 2 Find the maximum m in array B in $O(1)$ time and $O(n)$ work deterministically. Use a variant of the deterministic algorithm which works in three pulses. (The exercise below explains why these are actually the last three pulses of the recursive doubly-logarithmic time algorithm, as per the second stage in the description of the deterministic algorithm for finding the maximum above. The first pulse partitions B into $n^{3/4}$ blocks of size $n^{1/8}$ each and finds the maximum in each block in $O(1)$ time and $O(n^{1/4})$ work, or a total of $O(n)$ work. The second pulse partitions the $n^{3/4}$ maxima into $n^{1/2}$ block of size $n^{1/4}$ each and finds the maximum in each block in $O(1)$ time and $O(n^{1/2})$ work, or a total of $O(n)$ work. The third pulse finds the maximum among the $n^{1/2}$ maxima found in the second pulse in $O(1)$ time and $O(n)$ work.

Exercise 17: *Give the fastest deterministic algorithm you can for finding the maximum among n^{ϵ} elements in $O(n)$ operations. What is the running time of your algorithm as a function of $\epsilon \leq 1$?*

Step 3
While there is an element larger than m **do**
- For each element in A which is larger than m, throw it into a random place in
- a new array of size $n^{7/8}$
- Compute the maximum in the array of size $n^{7/8}$ deterministically into m.
- (Handling entries of the array in which no value is written is straightforward.)

Complexity (preview). Step 1 takes $O(1)$ time and $O(n^{7/8})$ work. Step 2 takes $O(1)$ time and $O(n)$ work. Each iteration of Step 3 takes $O(1)$ time and $O(n)$ work. The proof of the theorem below implies that with very high probability only one iteration of Step 3 is needed, and then the total work is $O(n)$ and total time is $O(1)$.

Theorem 8.2: *The algorithm finds the maximum among n elements. With very high probability it runs in $O(1)$ time and $O(n)$ work. The probability of not finishing within this time and work complexity is $O(1/n^c)$ for some positive constant c.*

Proof What is the probability that the elements of B do not include one of the $n^{1/4}$ largest elements of A? The probability that each element of B separately is not one of these is $\frac{n - n^{1/4}}{n} = 1 - 1/n^{3/4}$. The probability that no element of B is one of the $n^{1/4}$ largest elements of A is $(1 - 1/n^{3/4})^{n^{7/8}}$ which is $((1 - 1/n^{3/4})^{n^{3/4}})^{n^{1/8}} \leq c_1^{n^{1/8}})$ for some positive constant $c_1 < 1$. So, following Step 2 we have at most $n^{1/4}$ elements which are larger than m with probability $\geq 1 - c_1^{n^{1/8}})$. Next, we derive the theorem by properly upper bounding the probability that no two of these elements hit the same location in the array of size $n^{7/8}$ at the first iteration of Step 3. (The theorem will follow since if at most $n^{1/4}$ elements of A are larger than m in Step 2, and all these larger elements appear in the new auxiliary array at the first iteration of Step 3, then the maximum over A must be found in that iteration.) Consider serializing the process of throwing the elements. The first one will occupy a new cell of the array. The second element will collide with it with probability $1 - 1/n^{7/8}$. Suppose that in the first i steps no collision occurred. The probability that the $i+1$'st element does not collide is $1 - i/n^{7/8}$. So the probability that no collision occurred is $\geq (1-1/n^{7/8})(1-2/n^{7/8})\ldots(1-n^{1/4}/n^{7/8}) \geq (1-n^{1/4}/n^{7/8})(1-n^{1/4}/n^{7/8})\ldots(1-n^{1/4}/n^{7/8}) = (1-(1/n^{5/8}))^{n^{1/4}} = ((1-(1/n^{5/8}))^{n^{5/8}})^{1/n^{3/8}} \geq (1/3)^{n^{-3/8}}$, for large enough n. To finish the proof of the theorem we seek to upper bound the gap, denoted $f(n)$, between this number and 1. Namely, $1 - f(n) = (1/3)^{n^{-3/8}}$, or $(1-f(n))^{n^{3/8}} = 1/3$. So, for large enough n, $f(n) = O(1/n^{c_2})$ for some constant $c_2 > 0$. To sum up, the probability of failure in either Step 1 or Step 3 is $O(1/n^c)$ for some constant $c > 0$. Otherwise, the algorithm finds the maximum among n elements in $O(1)$ time and $O(n)$ work.

Exercise 18: *The problem of selecting the k-th largest out of n elements was considered in Section 5. Replace Algorithm 2 in that section by a randomized algorithm so that the same complexity results as there will be achieved (but this time only on the average). Explain your answer. (Hint: Algorithm 2 was based on finding an element which is not smaller than a quarter of the elements and not larger than a quarter of the elements. Pick a random element r instead; then partition the elements according to whether they are larger than, equal to, or smaller than r; remain with one of these three groups.)*

Exercise 19: *Consider the problems of sorting n elements. Try to extend the following approach into a sorting algorithm. Similar to the hint in Exercise 18 pick a random element r; then partition the elements according to whether they are larger than, equal to, or smaller than r. What is the time and work complexity of your algorithm? Explain your answer. Note that this algorithm is actually closely related to the sorting method called "Quicksort".*

9. The List Ranking Cluster; Techniques: Euler tours; pointer jumping; randomized and deterministic symmetry breaking

We start this section with a "toy problem", the tree rooting problem, which motivates the presentation. The Euler tour technique provides a constant time optimal-work reduction of tree rooting (and other tree problems) to the list ranking problem. This section can be viewed as an extensive top-down description of an algorithm for any of these tree problems, since the list ranking algorithms that follow are also described in a top-down manner. Such top-down structure of problems and techniques from involved to elementary has become a "trade mark" of the theory of parallel algorithms, as reviewed in [Vis91], since it highlights the *elegance* of this theory. Such fine structure (actually much finer) exist in some classical fields of Mathematics, but is rather unique for Combinatorics-related theories. Figure 18 illustrates this structure.

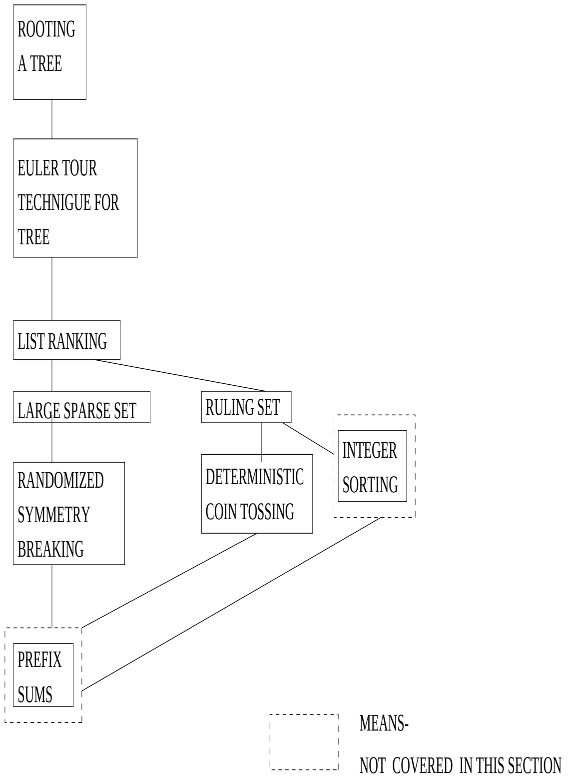


Figure 18: Structural overview of the chapter

9.1. The Euler Tour Technique for Trees

By way of motivation, consider the following **tree rooting** problem.
Input A tree $T(V, E)$, and some specified vertex $r \in V$. V is the set of vertices and E, the set of edges, contains unordered pairs of vertices.
The problem is to select a direction for each edge in E, so that the resulting directed graph is a (directed) rooted tree whose root is vertex r. This is equivalent to finding for every edge $e = (u, v)$ in E which among its two endpoints is closer to r.

The tree is given in an adjacency list representation. Let $V = 1, 2, \ldots, n$ and assume the edges are stored in an array. The edges adjacent on vertex 1, are followed by the edges adjacent on vertex 2, and so on. Each edge appears twice: once for each of its endpoints; each of the two copies of an edge has a pointer to the other copy. See Figure 19.

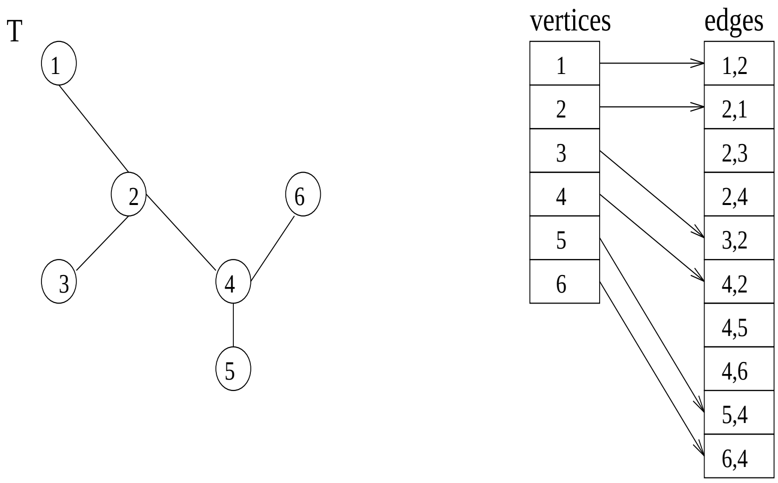


Figure 19: Tree T and its input representation

The Euler tour technique reduces the tree rooting problem to a **list ranking problem** in three steps, as follows. In Step 1, each edge of T is replaced by two anti-parallel edges. So that the rooted tree problem amounts to selecting one among each pair of anti-parallel edges. For each edge e of the resulting directed graph, Step 2 sets a pointer $next(e)$ to another edge. See Figure 20. Claim 1 below establishes that a path in the directed graph which traces these pointers is actually a circuit which visits each of its directed edges exactly once prior to returning to a starting edge. Such a circuit is called an *Euler tour*. Step 3 disconnects the pointer of some edge entering r. This gives a linked list and we assume that we know to find for each edge of the list, its distance (or rank) from the end of the list (the so-called *list ranking problem*, which is studied later). By Claim 2 below, for each pair of anti-parallel edges it is enough to compare their ranks to determine which anti-parallel edge leads towards r. See also Figure 20.

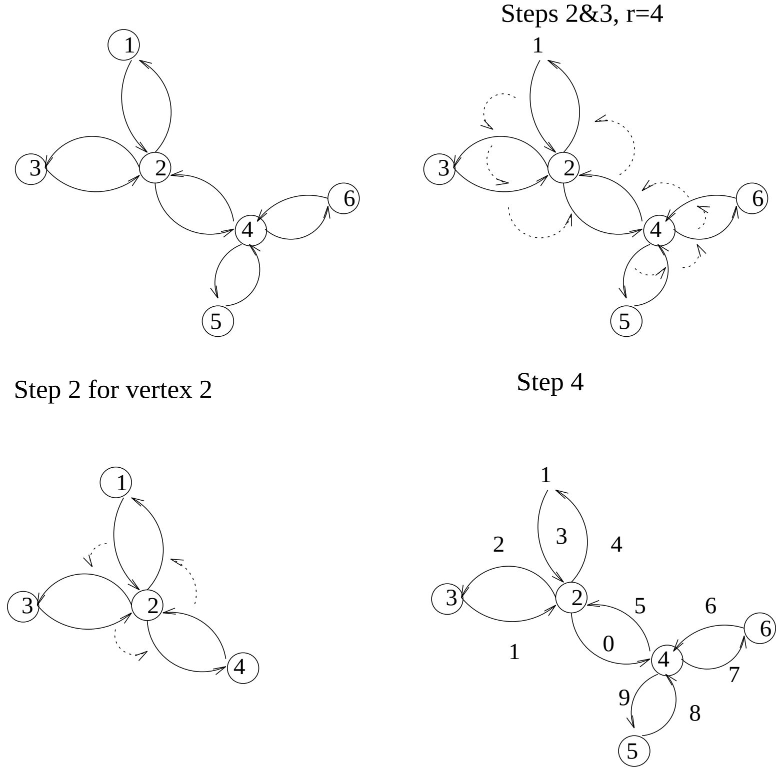


Figure 20: The Euler tour technique

Step 1 Get a new directed graph, denoted T_D, from T
for every edge (u, v) in T **pardo**
- Replace (u, v) by two anti-parallel edges: $u \to v$ and $v \to u$.

Given a vertex $v \in V$, we assume that its adjacent edges in T are denoted $(v, u_{v,1}), (v, u_{v,2}), \ldots, (v, u_{v,deg(v)})$, where the number of edges adjacent on v (the *degree* of v) is denoted $deg(v)$.
Step 2 For every directed edge $u \to v$ in T_D set a pointer $next(u \to v)$ to another directed edge in T_D, as follows.
for every vertex v in V **pardo**
- **for** every i, $1 \le i \le deg(v)$ **pardo**
- $next(u_{v,i} \to v) \ := \ v \to u_{v,i+1} \pmod{deg(v)}$

Step 3 $next(u_{r,1} \to r) \ := \ NIL$

Step 4 Apply a list ranking algorithm computing $rank(u \to v)$ for every edge $u \to v$ in T_D.

Step 5
for every edge (u, v) in T **pardo**

- **if** $rank(u \rightarrow v) < rank(v \rightarrow u)$
- **then** vertex v is closer to the root r than vertex u
- **else** vertex u is closer to the root r than vertex v

Claim 1 The path defined by Step 2 of the algorithm is an Euler circuit.

Proof By induction on n, the number of vertices in T. For $n = 2$ the claim readily holds. Assuming that the claim holds for every tree T with $n-1$ vertices, we show that it holds for trees with n vertices. Let T be a tree with n vertices. T must have at least one leaf (i.e., a vertex whose degree is one) v. Delete v and its adjacent edge from T. We get another tree S with only $n-1$ vertices. By the inductive hypothesis Step 2 finds an Euler tour in S_D. The directed graph T_D has vertex v, as well as the pair of anti-parallel edges $u \rightarrow v$ and $v \rightarrow u$, in addition to S_D. These two anti-parallel edges will be visited by the following detour with respect to the Euler tour in S_D. There is some edge $w_1 \rightarrow u$ for which $next(w_1 \rightarrow u) = (u \rightarrow w_2)$ in S_D, while in T_D we have $next(w_1 \rightarrow u) = (u \rightarrow v)$. The detour continues by following $next(u \rightarrow v) = (v \rightarrow u)$, and finally $next(v \rightarrow u) = (u \rightarrow w_2)$. So the Euler tour of S_D extends into an Euler tour of T_D.

Claim 2 In case $rank(u \rightarrow v) < rank(v \rightarrow u)$ the directed edge $u \rightarrow v$ (and not $v \rightarrow u$) leads towards the root r.

Complexity The reduction into list ranking (i.e., Step 1,2,3 and 5) take $O(1)$ time and $O(n)$ work. List ranking algorithms are discussed later.

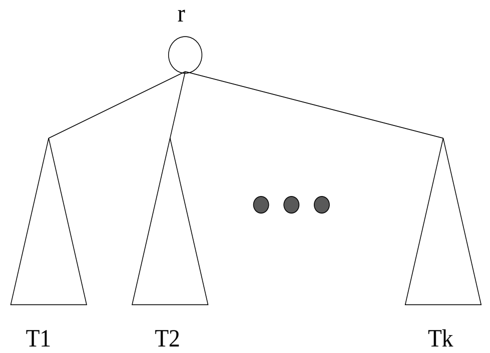
9.1.1. More tree problems In the tree rooting algorithm above, we actually assumed that the original length of each directed edge e in T_D is 1, and initialize the distance between each directed edge e in T_D and its successor in the list, $next(e)$ to 1. All problems below use essentially the above reduction into list ranking. However, the initial distances will be different, and we assume that the input tree is already rooted at some vertex r.

Preorder numbering

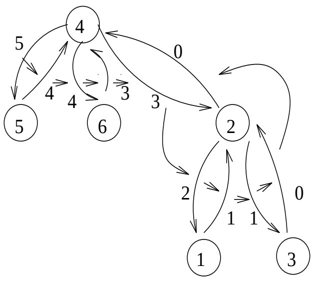
Input A (directed) tree $T = (V, E)$ rooted at a vertex r. The tree is given in an adjacency list representation. Let $V = 1, 2, \ldots, n$ and assume the edges are stored in an array. The incoming edge of vertex 1 (only the root r does not have one) is followed by its outgoing edges (if exist). This is followed by the incoming edge of vertex 2 followed by its outgoing edges. Each edge appears twice: once for each of its endpoints.
The *preorder listing* of the vertices of the tree is defined recursively as follows: (1) If the number of vertices in the tree is one ($n = 1$) then that node itself is the preorder listing of T. (2) Suppose $n > 1$. Then r, the root of T, has k outgoing edges to subtrees $T_1, T_2, \ldots, T_k$, as suggested by Figure 21; the preorder listing of T consists of the vertex

r followed by the preorder listing of tree T_1, followed by this of T_2, and so on up to the preorder listing of T_k.
Our problem is to compute the numbering of the vertices as they occur in the preorder listing.

We use the same Euler tour as we saw above, which visits each edge of T and its backwards counterpart. The preorder number of a vertex v of T is one plus the number of vertices visited prior to v. Since v itself and the vertices visited after it are counted into $distance(u \rightarrow v)$, the preorder number of v is $n - distance(u \rightarrow v) + 1$. See also Figure 21.



Tree T for definition of preorder listing



Distances computed in preorder algorithn (same graph as for Euler tour technique)

preorder(1)=n-distance(2-> 1)+1=6-2+1=5

Figure 21: Preorder algorithm

Step 1 (*initialization*)
for every edge $e \in E$ **pardo**
- **if** e is a tree edge **then** $distance(e) := 1$ **else** $distance(e) := 0$

Step 2
list ranking

Step 3
$preorder(r) := 1$
for every tree edge $e = u \rightarrow v$ **pardo**
- $preorder(v) := n - distance(e) + 1$

Exercise 20: *The postorder listing of the vertices of the tree is defined recursively similar to the preorder listing with one difference. Consider the case $n > 1$. Then, the postorder listing of T consists of the postorder listing of tree T_1, followed by this of T_2, and so on up to the postorder listing of T_k, and finally the vertex r.*
Show how to compute the numbering of the vertices as they occur in the postorder listing.

Size of subtrees

Consider the subtree rooted at some vertex v. The *size of the subtree*, denoted $size(v)$, is the number of its vertices. We show how to find the sizes of all rooted subtrees. Assume that v is not the root and let $u \rightarrow v$ be the incoming tree edge of v. Then, the part of the Euler tour which spans between the edge $u \rightarrow v$ and its anti-parallel edge $v \rightarrow u$ actually forms an Euler tour of the subtree of v. The number of tree edges in this part is $distance(u \rightarrow v) - distance(v \rightarrow u) - 1$, which is the number of vertices in the subtree of v excluding v itself. So, perform steps 1 and 2 as in the preorder numbering algorithm and then replace Step 3 by the following:

New Step 3
$size(r) := n$
for every tree edge $e = u \rightarrow v$ **pardo**
- $size(v) := distance(u \rightarrow v) - distance(v \rightarrow u)$

Level of vertices

The level of a vertex v, denoted $level(v)$, is the length of the directed path from r to v counting edges. The Euler tour consists of single steps each either going down one level or going up one level in the tree. The initialization in Step 1 implies a charge of one for each downgoing step and of minus one for each upgoing step. The suffix of the Euler tour which begins at a tree edge $e = u \rightarrow v$ includes exactly $level(u)$ upgoing steps that do not have matching downgoing steps; and $level(v)$ is $level(u) + 1$.

Step 1 (*initialization*)
for every edge $e \in E$ **pardo**
- **if** e is a tree edge **then** $distance(e) := -1$ **else** $distance(e) := 1$

Step 2
list ranking

Step 3

for every tree edge $e = u \rightarrow v$ **pardo**

- $level(v) := distance(e) + 1$

Exercise 21: *Consider a rooted tree, whose root is vertex r. A node i is an ancestor of node j if it is on the path from r to j.*
(a) Prove that node i is an ancestor of node j if and only if $preorder(i) < preorder(j)$ and $postorder(i) > postorder(j)$.
(b) Explain how to preprocess the tree, efficiently in parallel, so that a query of the form "is node i an ancestor of node j?" can be processed in constant time using a single processor.

Exercise 22: *Let T be an undirected tree with n nodes. We would like to find a node whose removal disconnects the graph into connected components so that no connected component contains more than $n/2$ vertices.*
(a) Give an efficient algorithm for the problem. Your algorithm should be as efficient as possible.
(b) What is its work and time complexity? Explain.

9.2. A first list ranking algorithm: Technique: Pointer Jumping

The **list ranking problem** is defined as follows.
Input: A linked list of n elements. The elements are stored in an array A of size n. See Figure 22. Each element, except one (to be called $last$), has a pointer to its successor in

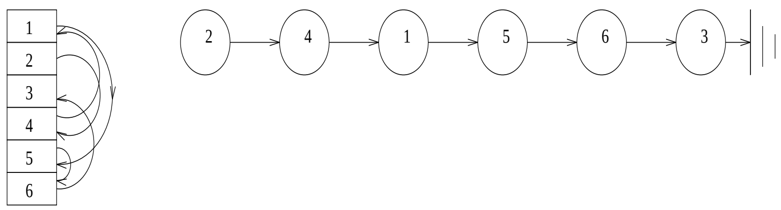


Figure 22: Input for list ranking problem

the list; also, each element, except one (to be called $first$), is the successor of exactly one element in the list. Formally, element i, $1 \leq i \leq n$, has a pointer field $next(i)$ which contains the array index of its successor. We define $rank(i)$ as follows: for each i,

$rank(i) = 0$ if $next(i) = NIL$ (or "end-of-list") and $rank(i) = rank(next(i)) + 1$ otherwise.
The list ranking problem is to compute $rank(i)$ for every i, $1 \leq i \leq n$. The list ranking problem has a straightforward linear time serial algorithm. (1) Find for every element i, its predecessor, $pre(i)$, in the list, as follows: for every element i, $1 \leq i \leq n$, unless $next(i) = NIL$, set $pre(next(i)) := i$. (2) Trace the list by following the pre pointers and thereby compute $rank(i)$ for every element i.

Before proceeding to parallel algorithms for list ranking we note that step (1) of the above serial algorithm can be implemented in $O(1)$ time and $O(n)$ work.

We assume that $\log n$ is an integer. A simple parallel list ranking algorithm follows. See also Figure 23.

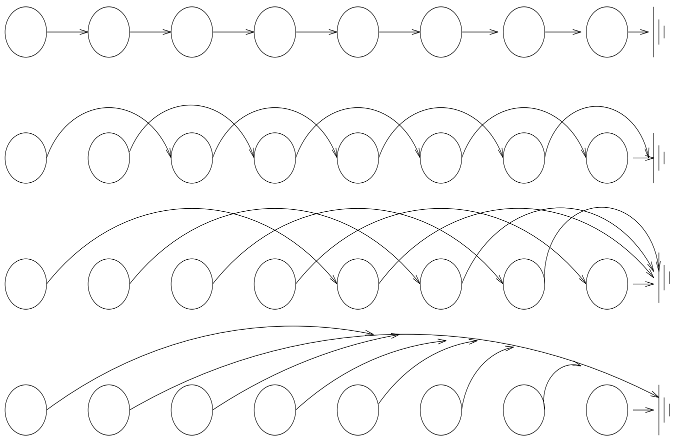


Figure 23: Basic parallel pointer jumping algorithm

ALGORITHM 1 (Basic pointer jumping)
for i, $1 \leq i \leq n$ **pardo**
\- **if** $next(i) = NIL$ **then** $distance(i) := 0$ **else** $distance(i) := 1$
\- **for** $k := 1$ **to** $\log n$
\- $distance(i) := distance(i) + distance(next(i))$
\- $next(i) := next(next(i))$

Correctness claim Upon termination of the basic pointer jumping algorithm, $distance(i) = rank(i)$ for every i, $1 \leq i \leq n$.

Proof of claim After iteration k, for every element i, $1 \leq i \leq n$, the following holds: (a) the number of edges between element i and $next(i)$ is $distance(i)$, not counting the edge leading from the first element of the list to the end-of-list sign; and (b) if $next(i) \neq NIL$ then $distance(i) = 2^k$. To see this, apply a simple induction on k. This implies that after iteration $\log n$, $next(i) = NIL$ for every element i, $1 \leq i \leq n$. The claim follows.

Theorem 9.1: *The basic pointer jumping algorithm runs in $O(\log n)$ time and $O(n \log n)$ work.*

Exercise 23: *Tune up the basic pointer jumping algorithm so that the above theorem will hold for the EREW PRAM within the same work and time complexities.*

We note that: (1) Even if we revise he algorithm to avoid repeated pointer jumping for elements whose $next$ pointer already reached the end of the list (namely, is NIL), the work complexity will remain $\Omega(n \log n)$, while the work complexity of the work-optimal algorithms below is $O(n)$. (2) The same method could have been applied to a rooted tree, instead of a linked list, to find the distance of node from the root.

Preview of a symmetry breaking challenge. By way of motivation, we present a hypothetical approach towards designing a work-optimal list ranking algorithm. Consider an instance of the list ranking problem, where the list is in an array order (i.e., $next(i) = i + 1$ for $1 \leq i < n$, and $next(n) = NIL$). Selecting (or marking) all the elements of the list whose rank is an odd number in $O(1)$ time and $O(n)$ work is simple. Next consider advancing to a new linked list which includes only the even ranked elements by shortcutting over all odd ranked ones. The new list includes $n/2$ elements. Our hypothetical list ranking algorithm will do this link-halving recursively till $n = 1$. It is not hard to see the strong resemblance between such a recursive algorithm and the recursive prefix-sums algorithm, given earlier. Unfortunately, we do not know how to select all odd-ranked elements in general, without already ranking the whole list. A known theorem actually implies that using a polynomial number of processors this should take $\Omega(\log n / \log \log n)$ time. This may explain, why we consider such selection an *ideal break of symmetry*. For the sequel, we note two properties of such selection: (1) *Uniform sparsity*: For every chain of two (successive) elements in the list at least one is not selected. (2) *Uniform density*: For every chain of two (successive) elements in the list at least one is selected. We finish this preview by saying that the gist of the next few sections are ways for approximating this ideal symmetry breaking.

9.3. A Framework for Work-optimal List Ranking Algorithms

This section actually follows the same accelerating cascades framework as in the selection algorithm. For devising fast $O(n)$-work algorithm for list ranking, we will use two building blocks. Each building block is itself a list ranking algorithm:
(1) Algorithm 1 runs in $O(\log^2 n)$ time and $O(n)$ work. It works in $O(\log n)$ iterations, each takes an instance of the list ranking problem of size, say m, and reduces it in $O(\log m)$ time and $O(m)$ work to an instance of the list ranking problem whose size is bounded by cm where $0 < c < 1$ is some constant fraction. It will be easy to use elements from the analysis of algorithms later in this section, to figure out why this indeed leads to a total of $O(\log^2 n)$ time and $O(n)$ work.
(2) Algorithm 2 runs in $O(\log n)$ time and $O(n \log n)$ work. Actually, the basic pointer jumping algorithm of the previous section will serve as algorithm 2.

An ultra-high-level description of a fast work-optimal list ranking algorithm
Step 1. Repeat the reducing iterations of Algorithm 1 to reduce an original instance for the selection problem into an instance of size $\leq n/\log n$.
Step 2. Apply Algorithm 2.

Complexity analysis. Step 1 takes $r = O(\log\log n)$ reducing iterations and a total of $O(\log n \log\log n)$ time. The number of operations is $\sum_{i=0}^{r-1} nc^i = O(n)$, where c is the constant fraction of Algorithm 1. Step 2 takes additional $O(\log n)$ time and $O(n)$ work. So, in total we get $O(\log n \log\log n)$ time, and $O(n)$ work. However, in case Algorithm 1 is randomized then the whole list ranking algorithm becomes randomized as well; see Theorem 9.2 for its complexity bounds.

In the subsequent sections, we will present two alternatives for Algorithm 1. For completeness, we describe below in detail some features which are shared by both alternatives.

Detailed description of some recursive work-optimal list ranking algorithms The key step involves finding a "large sparse" subset S which contains at least a fraction of the elements in the linked list (*no uniform density requirement*), but no two successive elements (*a uniform sparsity requirement*). Every element of S is detoured, and thereby omitted from the list, by advancing the *next* pointer of its predecessor to its successor. We compact the elements of $A-S$ into a smaller array B using the compaction application of prefix-sums, and "relate" the *next* pointers to the elements of B (this last bookkeeping operation is suppressed from the pseudo-code below). We recursively solve the list ranking problem with respect to array B, finding the final ranking for all elements in B. Finally, this ranking is extended to elements of S. An example is given in Figure 24.

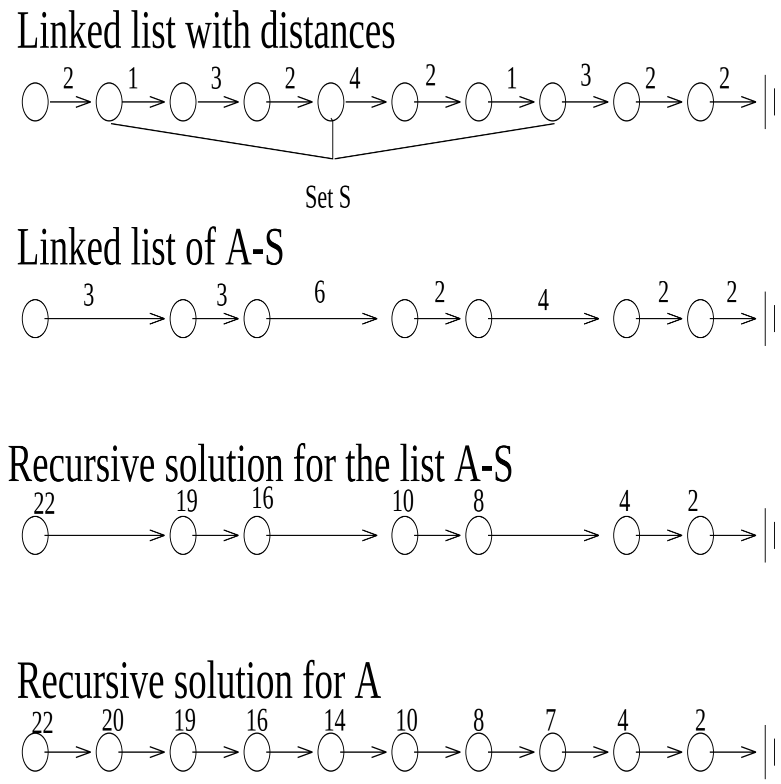


Figure 24: Recursive work-optimal list ranking

List_Rank($A(1),\ldots,A(n); next(1),\ldots,next(n); distance(1),\ldots,distance(n)$)

1. **if** $n = 1$
2. **then** Return
3. **else** Find a (large sparse) subset S of A such that:
- (1) $|S| > bn$, where b is some constant fraction, $0 < b \leq 1/2$; and
- (2) if some element i is in S then $next(i)$ is not in S.
4. **for** i, $1 \leq i \leq n$ **pardo**
- **if** $i \in S$
- **then** $distance(pre(i)) := distance(pre(i)) + distance(i)$
- $next(pre(i)) := next(i)$
5. Use a prefix-sums routine to compact the elements of $A - S$ into a smaller
- array $B = B(1),\ldots,B(m)$, where $m = |A - S|$.
6. Call List_Rank($B(1),\ldots,B(m); next(1),\ldots,next(m); distance(1),\ldots,distance(m)$)
7. **for** i, $1 \leq i \leq n$ **pardo**
- **if** $i \in S$
- **then** $distance(i) := distance(i) + distance((next(i))$

Complexity Analysis Finding a large sparse subset, as per Step 3, is deferred to the next sections. Step 4 takes $O(n)$ work and $O(1)$ time. Step 5: $O(n)$ work, $O(\log n)$ time. Let $c = 1 - b$. Step 6: $T(nc)$ time, $W(nc)$ work. Step 7: $O(n)$ work and $O(1)$ time. So, excluding Step 3, the recurrences that have to be satisfied are $T(n) \leq T(nc) + O(\log n)$ and $W(n) \leq W(nc) + O(n)$. The solutions are $T(n) = O(\log^2 n)$ and $W(n) = O(n)$. If we use only $r(n) = O(\log \log n)$ rounds of reducing iterations, the running time becomes $T(n) = O(\log n \log \log n)$ and the work remains $W(n) = O(n)$.

In the next two section, we give algorithms for the large sparse subset problem. The first method is randomized. The second method is deterministic and actually solves a more difficult problem (namely, where a uniform density requirement is also satisfied).

9.4. Randomized Symmetry Breaking

For simplicity, we will define the large sparse set problem for a linked ring; a linked ring can be obtained from a linked list as follows. Define the first element to be the successor of the last element; formally $next(last)$ is $first$ where $first$ is the index of the first element and $last$ of the last element.

The large sparse set problem
Input: A linked ring of n elements. The elements are stored in an array A of size n. Each element has a pointer to its successor in the ring; also, each element is the successor of exactly one element in the ring. Formally, element i, $1 \leq i \leq n$, has a pointer field $next(i)$ which contains the array index of its successor.
The problem is to find a subset S of A such that: (1) $|S| > bn$, where b is some constant fraction, $0 < b \leq 1/2$; and (2) if some element $A(i)$ is in S then $next(i)$ is not in S. Formally, the output is given in an array $S = S(1), \ldots, S(n))$. If $S(i) = 0$ then i is in the sparse set S and otherwise $S(i) = 1$, for i, $1 \leq i \leq n$.

The algorithm below has a randomized step. An independent coin is flipped for each element i, $1 \leq i \leq n$, choosing the values $HEAD$ and $TAIL$ with equal probabilities; the value is written into $R(i)$, $1 \leq i \leq n$. All elements which get $HEAD$ and whose successor in the ring get $TAIL$ are selected into S.

Large_sparse_set
for i, $1 \leq i \leq n$ **pardo**
- With equal probabilities write the values $HEAD$ and $TAIL$ into $R(i)$
- **if** $R(i) = HEAD$ and $R(next(i)) = TAIL$
- **then** $S(i) := 0$
- **else** $S(i) := 1$

Complexity The algorithm runs in $O(1)$ time and $O(n)$ work.

The set S is indeed a large sparse set, as explained below. First, it satisfies uniform sparsity (item (1)), since an element i is selected only if $R(next(i)) = TAIL$, and therefore its successor $next(i)$ cannot be selected. The number of elements in S is a random variable, and the following Lemma establishes that with high probability S is large enough.

Lemma Assume that n is even.
(1) The expected size of S is $n/4$.
(2) The probability that $|S| \leq n/16$ is exponentially small; formally, it is $O(c^{\Omega(n)})$, where c is a constant $0 < c < 1$.
Extensions to odd values of n here and in the corollaries of the Lemma make an easy exercise.

Proof of Lemma (1) Each element gets $HEAD$ with probability 1/2; independently, its successors gets $TAIL$ with probability 1/2. So, the probability that an element is in S is 1/4 and item (1) of the Lemma follows. (2) For the proof we quote Corollary 6.7 from page 125 in [CLR90], for bounding the right tail of a binomial distribution (henceforth called the **Right Tail theorem**):
Consider a sequence of n Bernoulli trials, where in each trial success occurs with probability p and failure occurs with probability $q = 1 - p$. Let X be the (random variable counting the) number of successes. Then for $r > 0$

$$Pr(X - np \geq r) \leq (\frac{npq}{r})^r$$

Going back to our algorithm, pick into a set B, some element i and then the other $n/2 - 1$ elements of the ring whose distance from i is even. We will derive item (2) by analyzing the probabilities only for these elements of A which are also in B. We saw above that each element of B is *not selected* into S with probability 3/4. These probabilities are pairwise independent. (To be on the safe side, we do not make any assumptions about the elements in $A - B$.) We have a sequence of $n/2$ Bernoulli trials, where in each trial success occurs with probability $p = 3/4$. By the Right Tail theorem, the probability that at least 7/8 of the $n/2$ elements of B are not selected (and therefore $r = (7/8)(n/2) = 7n/16$) is at most

$$(\frac{(n/2)(3/4)(1/4)}{(7n/16)})^{(7n/16)} = (3/14)^{(7n/16)}$$

Item 2 of the Lemma follows.

Each iteration of the recursive work-optimal list ranking algorithm of the previous section employs the Large_sparse_set routine. The corollary below allows to wrap up the complexity analysis for the work-optimal fast list ranking algorithms.

Corollary Suppose that we apply the recursive work-optimal algorithm till a list of size $\leq n/\log n$ is reached. We consider the probability that we reach a list of this size in several iterations so that in each iteration $|S| \leq 15|A|/16$; the probability that does not happen is exponentially small. Formally, there is some integer N such that for every $n \geq N$, the probability is $O(\alpha^{\Omega(n/\log n)})$, where α is a constant $0 < \alpha < 1$.

Proof of corollary The size of A in the last application of the recursive work-optimal algorithm is at least $n/\log n$. The probability for $|S| \leq 15|A|/16$ in the last application is at least

$$1 \;-\; (3/14)^{\frac{7n}{16\log n}}$$

The probability for $|S| \leq 15|A|/16$ in each of the preceding applications is at least this much and we have at most $\log_{16/15}\log n$ applications in case all succeeded. Therefore the probability for $|S| \leq 15|A|/16$ in all applications before we reach a list of size $n/\log n$ is at least

$$1 \;-\; (3/14)^{\frac{7n}{16\log n}}\log_{16/15}\log n$$

Deriving the corollary is now straightforward.

We conclude,

Theorem 9.2: *Using the randomized large sparse set routine, the work-optimal fast list ranking algorithm runs in time $O(\log n \log\log n)$ and $O(n)$ work with probability of $1-x$, where x is exponentially small.*

It is also possible to argue that using the randomized large sparse set routine, the average running time of the recursive work-optimal algorithm is $O(\log^2 n)$ and the average work is $O(n)$. However, this will not be covered here, since our interest in the recursive work-optimal algorithm is primarily in the context of the work-optimal fast list ranking algorithm.

Exercise 24: *(List ranking wrap-up). Describe the full randomized fast work-optimal list ranking algorithm in a "parallel program", similar to pseudo-code that we usually give. Review briefly the explanation of why it runs in $O(\log n \log\log n)$ time and $O(n)$ work, with high-probability.*

9.5. Deterministic Symmetry Breaking

We actually solve the following problem.

The r-ruling set problem

Input: A linked ring of n elements. The elements are stored in an array A of size n, where each element i, $1 \leq i \leq n$, has one successor $next(i)$ and one predecessor. So far, this is as in the large sparse set problem. The input also includes an integer $r \geq 2$.
A subset S of the elements is *r-ruling* if the following two conditions hold true: (1) (*uniform density*) if some element i is not in S, then at least one of its r successors is in S (for example, if $r = 2$ then element $next(i)$ or element $next(next(i))$ must be in S; and (2) (*uniform sparsity*) if some element i is in S then $next(i)$ is not in S.
The problem is to find an r-ruling set. As before, the output is given in an array $S = S(1), \ldots, S(n))$; if $S(i) = 0$ then i is in the r-ruling set S and otherwise $S(i) = 1$, for i, $1 \leq i \leq n$.
We will also assume for convenience that $\log n$ is an integer.

Item (1) implies that any chain of $r + 1$ elements in the linked list must include at least one element in S. So an r-ruling set must contain a least $\lceil \frac{n}{r+1} \rceil$ elements. Item (2) is as in the large sparse set problem, and implies that an r-ruling set contains at most $n/2$ elements.

The randomized algorithm singled out those elements which had a $HEAD$ tag followed by a $TAIL$ tag. An intriguing question is how to achieve a similar effect in a deterministic algorithm.

Setting an intermediate objective A first observation is that the *input array itself provides each element in the ring with an index in the range* $[0, \ldots, n-1]$. For each element i in the ring, we use the index as its tag, and note that the tags satisfy the following *local asymmetry* property: $tag(i) \neq tag(next(i))$. We state an **intermediate objective** for guiding the presentation. *Replace each $tag(i)$ value by an alternative $newtag(i)$ value, such that, the range $[0, \ldots, x-1]$ of the $newtag$ values is much smaller (i.e., $x \ll n$), but maintain the local asymmetry property.* We first show that the intermediate objective is helpful and then explain how to accomplish it.

Why is the intermediate objective relevant for the r-ruling set problem? Mark each element i whose $newtag$ value is a local maximum (i.e., $newtag(pre(i)) < newtag(i)$ and $newtag(next(i)) < newtag(i)$); mark also each element i whose $newtag$ value is a local minimum (i.e., $newtag(pre(i)) > newtag(i)$ and $newtag(next(i)) > newtag(i)$), if neither its successor nor its predecessor is marked.
Observation. The marked elements form an $(x-1)$-ruling set.
We first show that the distance (i.e., number of edges) between one local maximum, and its subsequent local minimum in the ring, can not exceed $x - 1$. To see this, note the following (see also See Figure 25. the tag of a local maximum is $\leq x - 1$, its immediate successors form a monotonically decreasing sequence with at most $x-1$ elements, leading to a local minimum, whose tag is ≥ 0. For similar reasons, the distance between one local

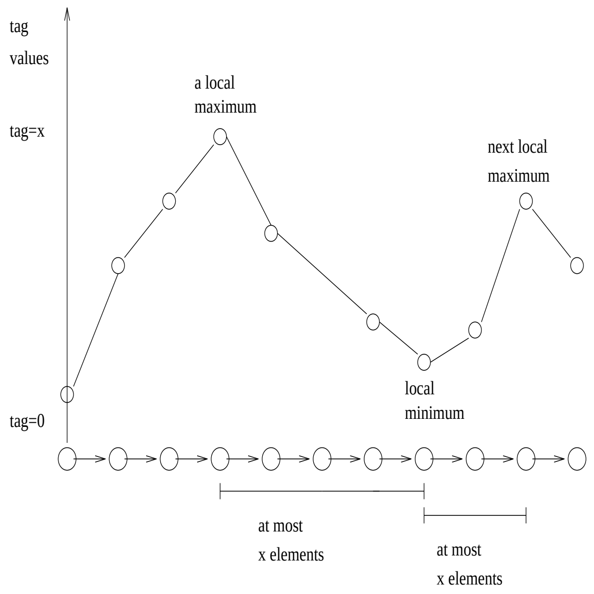


Figure 25: There are at most $2x-1$ elements between two successive local maxima

minimum, and its subsequent local maximum in the ring, can not exceed $x-1$. Consider an element i which is not marked. The distance to its subsequent local extremum (i.e., maximum or minimum) is $x-2$. The only case where such an extremum is not marked is where it is a local minimum whose successor is marked (being a local maximum). In any case, the distance from element i to its subsequent marked element is at most $x-1$. The observation follows.

The basic idea

The **deterministic coin tossing** method accomplishes the intermediate objective, by considering the binary representation of the tag values, as follows:

for i, $1 \leq i \leq n$ **pardo**

- find $\alpha(i)$, the rightmost bit position where bit $\alpha(i)$ in $tag(i)$ and bit $\alpha(i))$
- in $tag(next(i))$ differ;
- set $newtag(i)$:= the ordered pair ($\alpha(i)$, bit $\alpha(i)$ of i)

Remark We will alternate between the above definition of $newtag(i)$ as an ordered pair (convenient for the proof of Claim 1 below) and the following alternative definition $newtag(i)$:= $2\alpha(i)$ plus bit $\alpha(i)$ of i (convenient for Claim 2 below). No confusion will arise.

Example 1 Let $i = 20$, or $0\ldots010100$ in binary, and $next(i) = 24$, or $0\ldots011000$ in binary. Both i and $next(i)$ have 0 in bit position 0, and 0 in bit position 1. They differ in bit position 2 where i has 1 and $next(i)$ has 0. So, $newtag(i) = (2,1)$.

Claim 1 The *newtag* values maintain the local asymmetry property. That is, for every element i, $newtag(i) \neq newtag(next(i))$.

Proof Assume in contradiction that $newtag(i) = newtag(next(i))$. This implies that $\alpha(i)$, the first component in the pair of $newtag(i)$, is equal to $\alpha(next(i))$, the first component in the pair of $newtag(next(i))$. However, the definition of $\alpha(i)$ implies that bits $alpha(i)$ in $tag(i)$ and $tag(next(i))$ are different, and therefore their second component must be different; a contradiction.

Claim 2 An alternative definition of the *newtag* values maps them to integers in the range $0, \ldots, 2\log n - 1$, as follows. Set
$newtag(i) := 2\alpha(i)$ plus bit $\alpha(i)$ of i.

Example 1 (continued) Using the alternative definition $newtag(i)$ will be $2{\cdot}2 + 1 = 5$.

Remark In our complexity analysis we will refer to the computation of $newtag(i)$ as taking a constant time using a single processor, for each i. This needs further justification. Without loss of generality assume that $i > next(i)$ (otherwise interchange the two numbers). Set $h = i - next(i)$, and $k = h - 1$, and observe that h has a 1 bit $\alpha(i)$ and a 0 for all bits of lesser significance, while k has a 0 bit $\alpha(i)$ and a 1 for all bits of lesser significance; also all bits of higher significance in h and k are equal. Compute l, the exclusive-or of the bit representation of h and k. Observe that l is the unary representation of $\alpha(i)+1$,and it remains to convert this value from unary to binary (and do some additional standard arithmetic operations) to get $newtag(i)$. So we just demonstrated that if a conversion from unary to binary can be done in constant time than so does the computation of $newtag(i)$.

Theorem 9.3: *The deterministic coin tossing technique computes an r-ruling set, where $r = 2\log n - 1$, in $O(1)$ time using $O(n)$ work.*

Proof of Theorem The proof follows from claims 1 and 2, above.

Exercise 25: *Consider a linked list whose nodes, in their order in the list, are $9, 5, 1, 13, 0, 8, 4, 12, 14, 15, 11, 7, 3, 2, 6, 10$. In this drilling question, with $n = 16$, show how to find an r-ruling set as per Theorem 9.3 above.*

Exercise 26: *Iterating deterministic coin tossing*
By claims 1 and 2, after applying the deterministic coin tossing technique, each element in the ring has a tag in the range $[0, \ldots, 2\log n - 1]$; and the tags satisfy the local

asymmetry property. Reapply the deterministic coin tossing to replace each $tag(i)$ value by an alternative $newtag(i)$ value, whose range is even smaller; specifically, the smaller range will be $[0,\ldots,x-1]$, where $x = lglgn$ where lg denotes the logarithm function with base $\sqrt{2}$ (i.e., $\log_{\sqrt{2}}$).
Let $\log^* n$ be the function of n which represents the number of applications of the $\log$ function needed to bring n down to at most 2. Formally, denote $\log^{(1)} n = \log n$ and $\log^{(i)} n = \log\log^{(i-1)} n$; $\log^* n$ is now the minimum i for which $\log^{(i)} n \leq 2$.
(1) Show that after $O(\log^* n)$ iterations the range of tags will be $[0,1,2,3,4,5]$. Explain why it takes a total of $O(\log^* n)$ time and $O(n\log^* n)$ work.
(2) Show how to derive a 2-ruling set from these tags in additional $O(1)$ time, and $O(n)$ work. (3) Show how to use this 2-ruling set algorithm for a list ranking algorithm which runs in $O(\log n)$ time and $O(n\log^* n)$.

The local asymmetry property guarantees that the tags are always a *coloring*; that is, no two adjacent nodes have the same tag value. Clearly a 2-ruling set implies a coloring of a ring by 3 colors, which is called a 3-coloring.

Exercise 27: *Let T be a rooted tree where each vertex, except the root, has a pointer to its parent in the tree. Assume that the set of vertices is numbered by integers from 1 to n. Give an algorithm for 3-coloring T in $O(\log^* n)$ time and $O(n\log^* n)$ work.*

9.6. An Optimal-Work 2-Ruling set Algorithm

We present a 2-ruling set algorithm that runs in $O(\log n)$ time and $O(n)$ work, using the WT presentation methodology.

Input: A linked ring of n elements. The elements are stored in an array A of size n; $next(i)$ is a pointer to the successor of i in the list, and $pre(i)$ is a pointer to its predecessor, for every i, $1 \leq i \leq n$. Our problem is to find a subset S of A which is 2-ruling. That is: (1) if some element $A(i)$ is not in S then either $next(i)$ or $next(next(i))$ is in S; (2) if some element $A(i)$ is in S then $next(i)$ is not in S. Formally, the output is given in an array $S = S(1),\ldots,S(n))$; if $S(i) = 0$ then i is in the ruling set S and otherwise $S(i) = 1$, for i, $1 \leq i \leq n$.

Using deterministic coin tossing, a tag in the range $[0,\ldots,2\log n - 1]$ is computed for every element i. The main loop of the algorithm has $2\log n$ iterations. Consider an element i such that $tag(i) = k$. Iteration k visits element i, and if neither its predecessor nor its successor are in the ruling set, selects element i into the ruling set.

The algorithm uses the WT methodology in specifying the elements that participate in an iteration of the main loop, by means of a set.

ALGORITHM 1 (2-ruling set; WT upper level)

1. Apply one iteration of deterministic coin tossing to get for every $1 \leq i \leq n$, a value $newtag(i)$ in the range $[0, \ldots, 2\log n - 1]$.
2. **for** i, $1 \leq i \leq n$ **pardo**
- $S(i) := 1$ (S is initially empty)
3. **for** $k := 0$ to $2\log n - 1$ **do**
4. **for** i, $1 \leq i \leq n$, such that $newtag(i) = k$ **pardo**
- **if** $S(pre(i)) = S(next(i)) = 1$
- **then** $S(i) := 0$

Claim The algorithm finds a 2-ruling set.

Proof (1) We show that if an element i is in S then $next(i)$ is not in S. Element i was selected into S at iteration $newtag(i)$. Element $next(i)$ was not selected in a previous iteration, since by the selection rule element i would not have been selected. Since $newtag(next(i)) \neq newtag(i)$, element $next(i)$ is not even looked at iteration $newtag(i)$. Later iterations will not select element $next(i)$ since i is already in S. (2) We show that if element i was not selected into S then at least one among $pre(i)$ and $next(i)$ was. If neither $pre(i)$ nor $next(i)$ were in S by the beginning of iteration $newtag(i)$, then element i is selected. This implies that if element i is not in S then either $next(i)$ or $next(next(i))$ is.

Complexity (preview) Step 1 takes $O(1)$ time and $O(n)$ work. So does Step 2. Each element participates in one iteration of Step 3 and needs $O(1)$ work. So, if properly sequenced for the lower level of the WT methodology, the total number of operations can be $O(n)$. The number of iterations is $O(2\log n)$ and therefore the time is $O(\log n)$. This gives a total of $O(\log n)$ time and $O(n)$ work.

Exercise 28: *(WT lower level.) Add a new step after Step 1 above: apply the integer sorting algorithm, given earlier, to sort the newtag values. Explain why the new step takes $O(\log n)$ time and $O(n)$ work. Use the output of the sorting algorithm in order to get a lower-level Data-Parallel description of the 2-ruling set algorithm. Show that the complexity bounds of the above preview analysis still apply.*

Exercise 29: *(List ranking wrap-up.) Describe the full fast work-optimal list ranking algorithm in a "parallel program", similar to pseudo code that we usually give. Review briefly the explanation of why it runs in $O(\log n \log\log n)$ time and $O(n)$ work.*

10. Tree Contraction

Consider a rooted binary tree T where each node has either two children - a left child and a right child - or is a leaf, and let x be a leaf of T. Assume that the root has two children, and at least one of them has two children. We will define the rake operation for leaves whose parent is not the root. Let x be a leaf, let y be its parent (also called the *stem* of leaf x) and u be the parent of y, and let the other child of y (which is also the sibling of x) be z. Applying the **rake** operation to leaf x means the following: *delete x and y and have z become a child of u instead of y.* See Figure 26.

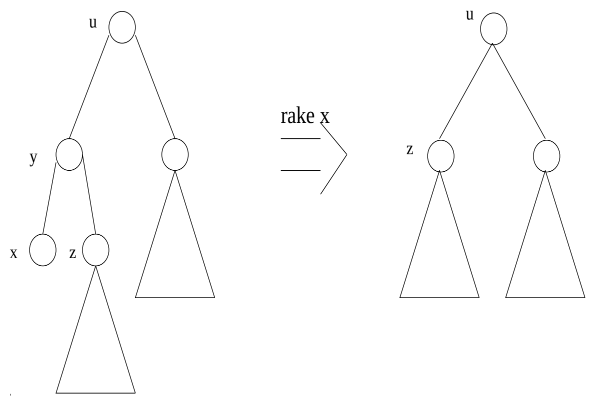


Figure 26: A rake operation

Observation 1. Applying the rake operation to a leaf x of the binary tree T, whose parent is not the root yields a tree which is still binary.

A **serial tree contraction scheme** applies rakes in several steps until the tree T becomes a 3-node binary tree (which consists of a root having two leaves as children).

Parallel rake. Applying the rake operation in parallel to several leaves is considered *legal* as long as the following two conditions hold: (i) no two of the raked leaves have the same parent (stem); and (ii) the parent of a stem cannot be a stem of a leaf which is currently being raked.

Observation 2. Applying a legal parallel rake operation to leaves of the binary tree T, yields a tree which is still binary.

A **parallel tree contraction scheme** applies rounds of legal parallel rakes until the tree T becomes a 3-node binary tree. The notion of parallel tree contraction scheme serves as an important paradigm in parallel algorithms. Specifying the exact order of rake operations gives a **parallel tree contraction algorithm**. Next, we present a tree contraction algorithm whose number of rounds is logarithmic in the number of leaves.

Suppose T has n leaves.

Step 1 Number the leaves from 1 to n in the same order in which they are visited in a depth-first search of T. Use the Euler tour technique to do this in $O(\log n)$ time and $O(n)$ operations assuming that the input tree is given in a proper way.

Observation 3. Let L be the subset of the leaves consisting of the odd-numbered leaves of T. Each leaf in L has a stem which is a node of T; let S be the set consisting of these stems. Then, for each node in S there is at most one other node in S which is adjacent to it. In other words, consider the subgraph of T which is induced by S; then each of its connected components consist of either two node connected by an edge or a singleton node.

Proof of Observation 3. The proof follows by a simple case analysis. We will examine cases of possible paths in T which contain three nodes; for each of these case, we argue that it is not possible that all three nodes are in S. See Figure 27. (1) The

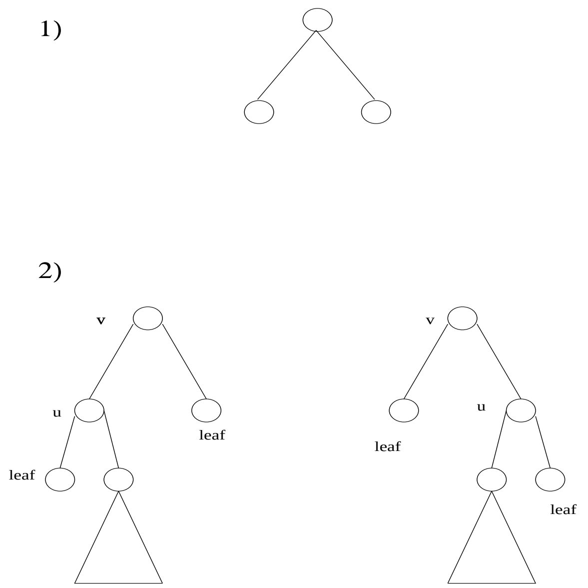


Figure 27: Case analysis for tree contraction observation

path consist of a node and its two children; if the childrens are stems in S, they are not leaves and the node cannot be a stem in S. (2) Let u and its parent v be two nodes in S. Consider the case where u is a left child of v. Then for both of them to be stems, the right child of v and the left child of u must be leaves. It is important to notice that these are the leftmost and rightmost leaves in the subtree rooted in v. We argue that w, the parent of v, cannot be a stem in S. For w to be a stem, its other child must be a leaf in L. In the serial order of leaves this leaf-child of w is next to a leaf of L, and therefore cannot be in L! Other cases are similar.

The main step of the tree contraction algorithm follows:

Step 2 Pick L, the subset of the leaves consisting of the odd-numbered leaves of the binary tree, and let S be the set of their stems. Let S_1 be the subset of those stems in S whose parent is not in S and let L_1 be the subset of L whose stems are in S_1.
Step 2.1 Apply parallel rakes to all leaves in L_1, with the exception of a leaf whose parent is the root.
Step 2.2 Apply parallel rakes to all leaves in $L - L_1$, with the exception of a leaf whose parent is the root.
We iterate Step 2 until a 3-node binary tree is reached.

We note the following.

It is easy to get a renumbering of the leaves after discarding the leaves of L in a round of Step 2 in $O(1)$ time (divide each even serial number of a leaf by 2). Each parallel rake is legal because of Observation 3.

Complexity Step 2 is iterated takes $\log n$ rounds, and since each leaf is not raked more than once, this takes a total of $O(n)$ operations and $O(\log n)$ time.

Note that the repeated selection of set L somewhat resembles the parallel delete algorithm for 2-3 trees.

We bring one application of parallel tree contraction.

10.1. Evaluating an arithmetic expression

Consider an arithmetic expression which include only two kinds of operations: addition (+) and multiplication ($\times$). The expression can be represented by a binary tree.
Example. A binary tree for the expression $(q \times (r + (s \times t))) \times p$ is given in Figure 28.

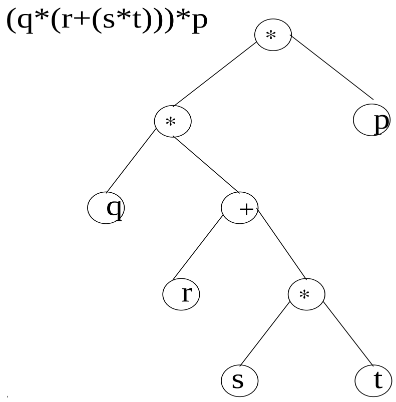


Figure 28: An arithmetic expression and its binary tree

We attach a quadruple of numbers (a, b, c, d) to each internal node of the tree to mean the following. Let the operator of the internal node be ϕ (where ϕ is either $+$ or $\times$), and assume that the value of the left child is α and the value of the right child is β. Then, the value of the node is $(a \times \alpha + b)\phi(c \times \beta + d)$. Initially, all quadruples are $(1, 0, 1, 0)$.

Exercise 30: *(1) Show that if prior to a rake operation, a quadruple of numbers per internal node represents an arithmetic expression, then the same applies after the rake operation. (Hint: Figure 29. describes one possible way in which a quadruple is updated*

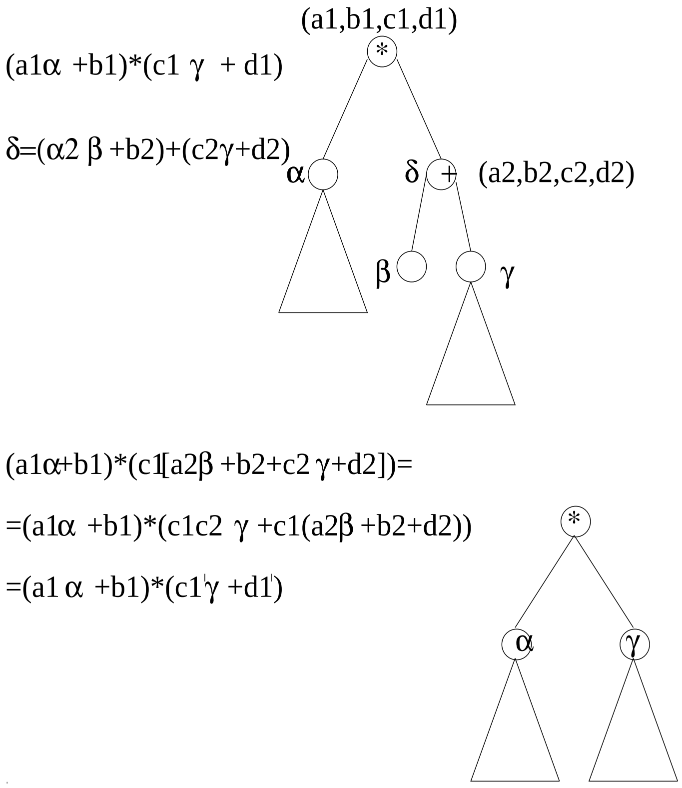


Figure 29: A rake operation on a tree of an arithmetic expression

following a rake operation; recall that the value of a leaf is constant.) (2) Show that this extends to a parallel rake of non-successive leaves, as well.

Complexity. The tree contraction paradigm runs in $O(\log n)$ time and $O(n)$ for a binary tree with n leaves and so does the above application for the evaluation of an arithmetic expression.

Exercise 31: *Fill in the details to prove the complexity result above.*

Exercise 32: *A vertex cover of a graph is a subset of the vertices such that at least one endpoint of each edge of the graph is incident on a vertex in the subset. The vertex cover problem is the find a vertex cover whose cardinality is the smallest possible. Give an efficient parallel algorithm which solves the vertex cover problem for a binary tree. What is the time and work complexity of your algorithm?*

11. Graph connectivity

The material and exercises in this section are more difficult than up to now. Our problem is to compute the connected components of a graph $G = (V, E)$, where $V = \{1, ..., n\}$ and $|E| = m$. We will consider two alternative input forms.
(i) **Incidence lists**. The edges are given in a vector of length $2m$. The vector contains first all the edges incident on vertex 1, then all the edges incident on vertex 2, and so on. Each edge appears twice in this vector.
(i) **Adjacency matrix**. An adjacency matrix A is an n by n matrix where $A(i, j) = 1$ if $(i, j) \in E$, and $A(i, j) = 0$ otherwise.
Both input forms are standard for serial algorithms. The incidence list representation has the advantage of using only $O(n + m)$ space, but processing a query of the form "are vertices i and j connected by an edge in G?" may need more than constant time; this holds true even if the edges incident on a certain vertex appear in sorted order of its other endpoint. While an adjacency matrix enables constant-time processing of such a query, the space requirement is proportional to n^2.

For describing the output form, we need the following definition: a **rooted star** is a rooted tree in which the path from each vertex to the root comprises (at most) one edge.
Output form. The vertices of each connected component comprise a rooted star.
As a result, a single processor can answer a query of the form "do vertices v and w belong to the same connected component?" in constant time. See Figure 30.

Exercise 33: *An alternative output form requires that the number of vertices in each connected component is given and that all the vertices that belong to the same component are grouped together in a separate list. Give an efficient parallel algorithm for deriving this alternative output form from the above output from. What is the time and work complexity of your algorithm.*

The following two exercises show what can be done with two standard search methods. We later proceed to considerably faster parallel graph algorithms.

Exercise 34: *(Breadth-first search in parallel) Given a connected undirected graph $G(V, E)$, and some node $v \in V$ the breadth-first search (BFS) method visits its ver-*

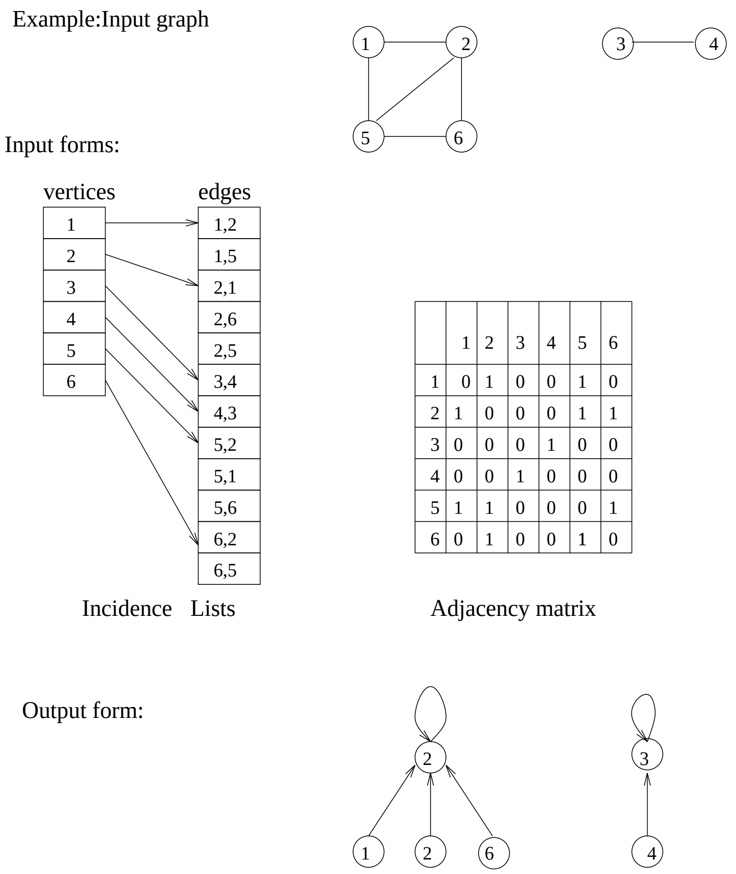


Figure 30: Two input forms and an output form for connectivity algorithms

tices in the following order. First visit v, then visit (in some order) all the vertices $u \in V$, where the edge (u, v) is in E; denote these vertices by V_1, and the singleton set consisting of v by V_0; in general, V_i is the subset of vertices of V which are adjacent on a vertex in V_{i-1} and have not been visited before (i.e., they are not in any of the sets $V_0, V_1, \ldots, V_{i-1}$). Each set V_i is called a layer of G and let h denote the number of layers in G. Serially BFS takes $O(n+m)$, or linear, time where the input is represented by incidence lists. Show how to run BFS on a PRAM in $O(h \log n)$ time and $O(n + m)$ work.

Exercise 35: *(Depth-first search in parallel) Given a connected undirected graph $G(V, E)$, and some node $v \in V$ the depth-first search (DFS) method visits its vertices in the following recursive order. For some vertex u, performing $DFS(u)$ consists of the following: if vertex u was visited before then call $DFS(u)$ ends; otherwise, visit vertex u and then call $DFS(w)$ in sequence for every vertex w and is adjacent on v. Depth-first search begins by calling $DFS(v)$. Show how to run DFS on a PRAM in $O(n)$ time and $O(n + m)$ work. (Hint. When DFS enters a vertex u for the first time, cancel all edges* *connecting another vertex to v, so that all remaining directed copies of edges always lead to yet unvisited vertices.)*

Preview

We present two connectivity algorithms. The following definitions are used in both algorithms, and illustrated in Figure 32.

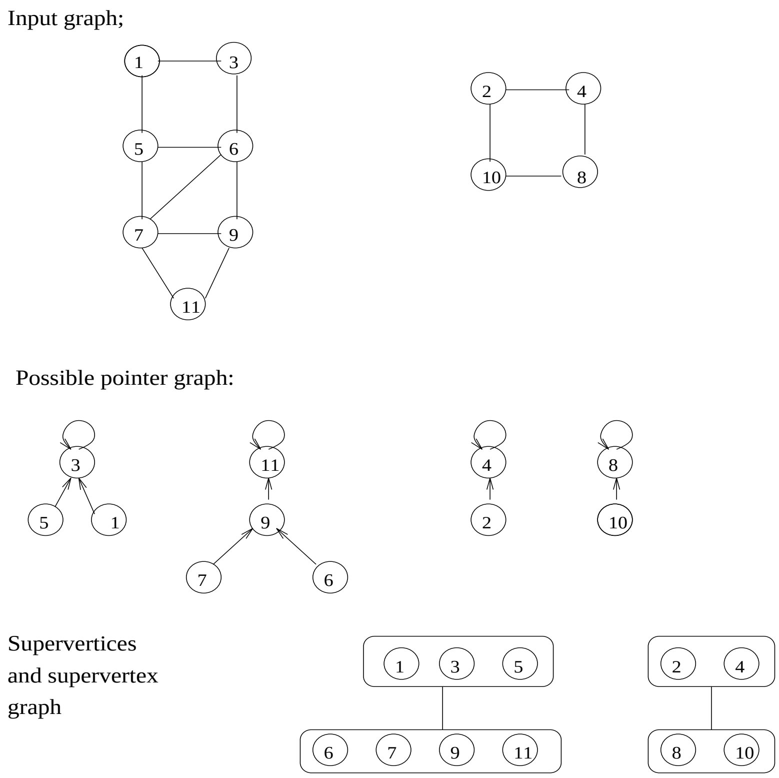


Figure 31: Definitions for the connectivity algorithms

- *The pointer graph.* At each step during the algorithms each vertex v has a pointer field D through which it points to another vertex or to itself. One can regard the directed edge $(v,\ D(v))$ as a directed edge in an auxiliary graph, called the **pointer graph**, with one exception: edges of the form $(v, D(v))$ where $D(v) = v$ are not considered part of the pointer graph. Initially, for each vertex v, $D(v) = v$; and the pointer graph consists of all vertices, but no edges. The pointer graph keeps changing during the course of the algorithms; throughout each of the connectivity algorithms the pointer graph consists of rooted trees; at the end, the pointer graph gives the output.

- *Supervertices.* We refer to the set of vertices comprising a tree in the pointer graph

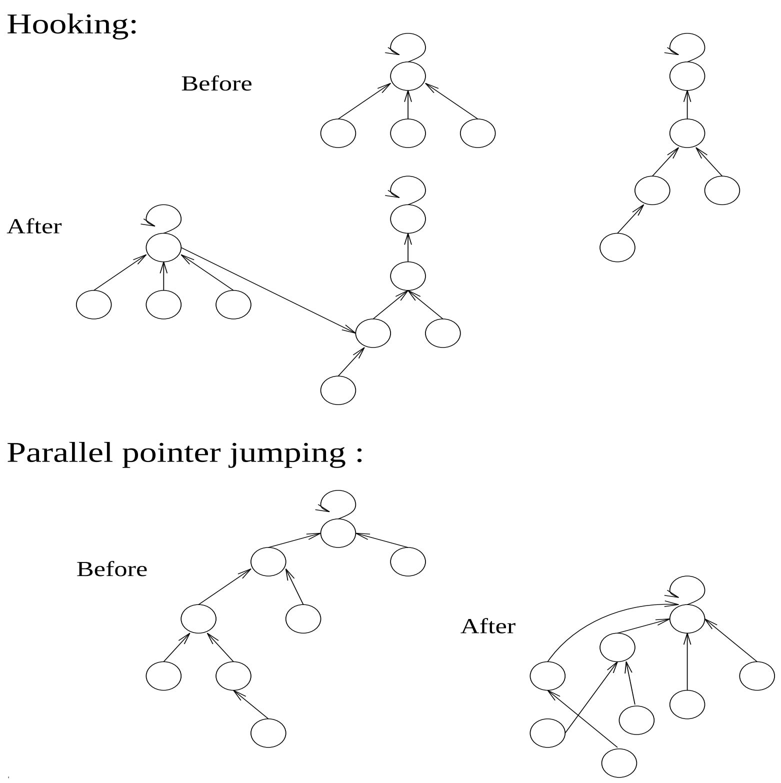


Figure 32: More definitions for the connectivity algorithms

as a **supervertex**. Sometimes, we identify a supervertex with the root of its tree. No confusion will arise.

- *The supervertex graph.* The algorithms also use the following graph. Each edge (u, v) in the input graph induces an edge connecting the supervertex containing u with the supervertex containing v. The graph whose vertices are the supervertices and whose edges are these induced edges is called the **supervertex graph**. At the end of each of the connectivity algorithms, the vertices of each connected component form a rooted star in the pointer graph, or a supervertex with no adjacent edges in the supervertex graph.

- *Hookings.* As the algorithms proceed, the number of trees (supervertices) decreases. This is caused by (possibly simultaneous) **hooking** operations. In each hooking a root r of a star is "hooked" onto a vertex v of another tree (that is, $D(r) := v$). Simultaneous hookings are performed in the connectivity algorithms in such a way that no cycles are introduced into the pointer graph.

- *Parallel pointer jumping.* The trees are also subject to a parallel **pointer jumping** operation. That is,
  for every vertex v of the tree **pardo**

- **if** $D(D(v))$ is some vertex (as opposed to no vertex)
- **then** $D(v) := D(D(v))$

Parallel pointer jumping (approximately) halves the height of a tree. Parallel pointer jumping does not introduce cycles into the pointer graph, as can be readily verified.

11.1. A first connectivity algorithm

Overview of the first connectivity algorithm
The algorithm works in iterations. Upon starting an iteration, each supervertex is represented by a rooted star in the pointer graph. An iteration has two steps:

1. *hookings*; Each root hooks itself onto a minimal root (i.e., a root whose number is smallest) among the roots which are adjacent to it in the supervertex graph. In case two roots are hooked on one another, we cancel the hooking of the smaller (numbered) root. As a result several rooted stars form a rooted tree, whose root is the root of one of these rooted stars. If a root does not have an adjacent root, then the rooted star together with all its vertices *quit* the algorithm, since they form a complete connected component.

2. *parallel pointer jumping*; To transform every rooted tree into a rooted star, an iteration finishes with $\log n$ rounds of parallel pointer jumping.

The algorithm ends if all its vertices quit, or proceeds into the next iteration otherwise. By Theorem 11.2, there are at most $\log n$ iterations. An example is illustrated in Figures 33, and 34.

Theorem 11.1: *The pointer graph always consists of rooted trees.*

Proof This trivially holds at the beginning of the algorithm. Inductively, a round of parallel pointer jumpings does not introduce cycles into the pointer graph. We show that a round of hookings does not introduce cycles into the pointer graph, as well. Consider the stars prior to a round of hookings, and let r be the root of such star. If $r = D(r)$ (r is a new root) or $D(r) = D(D(r))$ ($D(r)$ is a new root) after the hookings round then r cannot be on a cycle. Otherwise, we claim that $D^2(r) = D(D(r)) < r$ (in words, D^2 is *monotonically decreasing*). To see this, observe that root $D(r)$ selected to hook itself on root $D^2(r)$ which was the smallest among its adjacent roots in the supervertex graph, and therefore $D^2(r) \leq r$. This implies that $D^2(r) < r$, since if $D^2(r) = r$ we would have

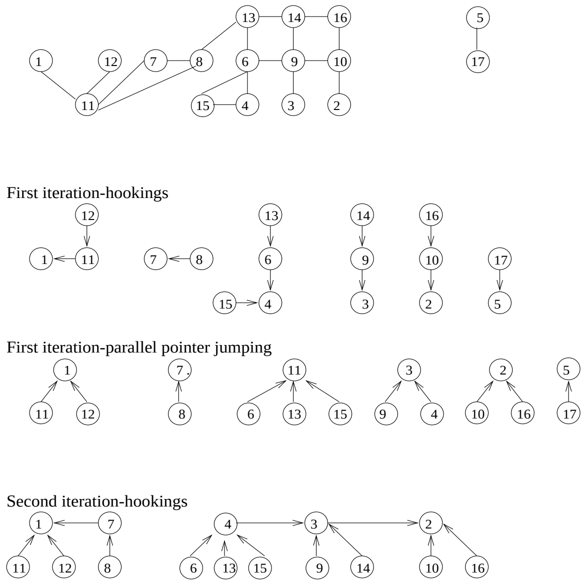


Figure 33: First connectivity algorithm: input and few first rounds

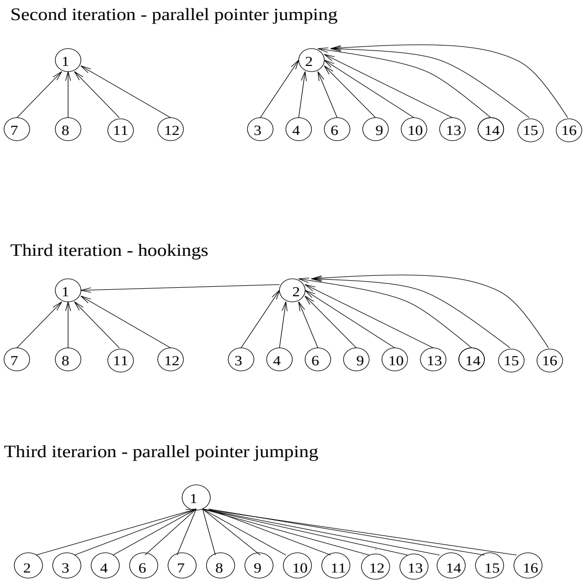


Figure 34: First connectivity algorithm continued

canceled either $D(r)$ or $D^2(r)$ (depending whether r or $D(r)$ is smaller), contradicting that neither $r = D(r)$ nor $D(r) = D(D(r))$. Consider the path which starts at a root r and keeps advancing from a vertex v to vertex $D^2(v)$. Since the only way to avoid an infinite D^2 sequence which is monotonically decreasing is by hitting a new root (where $D(v) = v$ and therefore $D^2(v) = v$), we conclude that r was not on a cycle.

Theorem 11.2: *Following at most $\log n$ iterations, all vertices of the graphs are contained in rooted stars that quit. Also, each connected component of the input graph G is represented by a single rooted star.*

Proof Consider a connected component in G at some iteration. If the number of supervertices in the component is at least two then this number decreases by a factor of at least two in the iteration. The reason being that in the round of hookings, each rooted star is either hooked on another rooted star in its connected component, or being hooked upon by such rooted star. If all vertices of the component already form a single supervertex, then the rooted star with all its vertices quits. The theorem follows.

11.1.1. Detailed description The algorithm as presented has at most $\log n$ iterations each taking $O(\log n)$ time or a total of $O(\log^2 n)$ time. This is the target time complexity of our detailed description. We will assume that the edges are given in an adjacency matrix A, since this will make it easier for us to define some of the exercises later.

while there exists a vertex which did not quit **do**
\- **for** every root v **pardo**
1. **if** $A(u_1, v_1) = 0$ for every vertex u_1 whose root is not v and every vertex v_1 whose root is v
2. **then** root v quits
3. **else** $D(v) := Min\{D(u_1) : A(u_1, v_1) = 1, D(v_1) = v; D(u_1) \neq v\}$;
4. **if** $v = D(D(v))$
5. **then** $D(v) := v$ if $v < D(v)$
6. perform $\log n$ iterations of parallel pointer jumping to reduce each rooted tree
\- to a rooted star; as a result $D(v)$ has the root of the current star of v
7. Use sorting to group together all vertices v, such that $D(v)$ is the same

Instruction 3 is implemented in two steps as follows:
3.1. For every vertex w separately, we find $Min\{D(u) : A(u, w) = 1, D(u) \neq D(w)\}$, by an algorithm for finding the minimum among n elements.
This takes $O(n)$ work and $O(\log n)$ time using a variant of the (balanced binary tree) summation algorithm, and a total of $O(n^2)$ work and $O(\log n)$ time over all vertices w.
3.2. Since all vertices with the same $D(w)$ are grouped together, another application of the algorithm for finding the minimum for "segmented" subarrays, as per Exercise 4, finishes computing $D(v)$ for every root v, in additional $O(n)$ work and $O(\log n)$ time.

Instruction 7 needs sorting n integers and can be done in $O(\log n)$ time and $O(n \log n)$ work, using a general parallel sorting algorithm. However, since the integers are in the range $[1 \ldots n]$, an alternative algorithm which uses "orthogonal trees" and was described in the integer sorting section would also work.

Theorem 11.3: *The first connectivity algorithm runs in* $O(\log^2 n)$ *time and* $O(n^2 \log n)$ *work using an adjacency matrix input representation on a CREW PRAM. Using incidence lists, it takes* $O(\log^2 n)$ *time and* $O(n \log^2 n \;+\; m \log n)$ *work.*

Proof The algorithm needs at most $\log n$ iterations each taking $O(\log n)$ time or a total of $O(\log^2 n)$ time. The work for an adjacency matrix representation is explained above. For incidence lists use $O(n + m)$ operations in each round of Step 3, for a total of $O((n + m) \log n)$ operations. In the other steps use a processor per vertex, for a total of n processors, and $O(n \log^2 n)$ operations. The theorem follows.

Exercise 36: *Consider the following revision to an iteration of the above parallel connected components algorithm. Instead of hooking the root* r *of a star on the smallest adjacent star, the hooking is performed only if the root of the adjacent star is numbered lower than* r.
(1) Show that $\Omega(\log n)$ *iterations are necessary.*
(2) How many iterations are sufficient? note that the root of a star may neither be hooked nor be hooked upon in a given iteration.

Exercise 37: *The goal of this exercise is to enhance the above algorithm into using only* $O(n^2)$ *work within the same* $O(\log^2 n)$ *time bound on a CRCW. In order to facilitate the reduction in the number of operations the algorithm begins, as before, with the original adjacency matrix* $A_0 = A$ *whose size is* $n \times n$*; but then after one iteration advances to* A_1*, the adjacency matrix of the supervertex graph, whose size is at most* $n/2 \times n/2$*, and so on; this enables the operation count to decrease geometrically. Below is a draft for such an algorithm. Fill in the missing steps, and prove that the required work and time complexity indeed hold.*

1. $n_0 := n$; $k := 0$
\- **for** *all* $i, j,\ 1 \le i, j \le n$ **pardo**
\- $A_0(i, j) := A(i, j)$
2. **while** $n_k > 0$ **do**
\- $k := k + 1$
\- **for** *every root* v **pardo**
\- **if** $A_{k-1}(u, v) = 0$ *for every other root* u
\- **then** *root* v *quits*
\- **else** $D(v) := Min\{u : A_{k-1}(u, v) = 1,\ u \ne v\}$;
\- **if** $v := D(D(v))$
\- **then** $D(v) := v$ *if* $v < D(v)$
\- *perform* $\log n$ *iterations of parallel pointer jumping to reduce each*
\- *rooted tree to a rooted star; as a result* $D(v)$ *has the root of the current star*
\- *of* v

- *Using parallel compaction compute n_k, the number of current roots, and for each*
- *current root v, compute $N_k(v)$, its serial number relative to the other current roots*
- **for** *every u, v that were roots at the beginning of this iteration* **pardo**
- $A_k(N_k(D(u)), N_k(D(v))) := A_{k-1}(u, v)$

Comment The following things may happen to a vertex in iteration i: (1) it may quit if it represents an connected component in the input graph; (2) it may be "absorbed" into a supervertex, represented by another vertex; (3) it may be the "representative" of its new supervertex, and then it will appear renumbered as implied by the parallel compaction algorithm. The draft of the algorithm above also suppresses the issue of how each vertex in the input graph (that might have been first renumbered and then quitted) will eventually find the connected component to which it belongs. The reader should be guided by the above draft in filling in the details.

Exercise 38: *Proceed to this exercise only after solving the Exercise 37 Get an alternative connectivity algorithm which runs also in $O(n^2)$ work and $O(\log^2 n)$ time but on a CREW. Suggested approach: Use sorting to group together all vertices which belong to the same rooted star. This will enable deriving A_k from A_{k-1} without concurrent writes within the performance bounds of the whole iteration.*

11.2. A second connectivity algorithm

The first connectivity algorithm repeatedly applies one round of hooking followed by $\log n$ rounds of parallel pointer jumping. The hooking round may produce one, or more, rooted trees whose leaves are all relatively close to the root. Such a tree will collapse to a rooted star after only a few rounds of parallel pointer jumping. Once a tree has collapsed, rounds of parallel pointer jumping do not change it and are redundant. The second connectivity algorithm obtains its improved running time by "pumping in" another round of hookings for those rooted trees in the pointer graph that recently collapsed into rooted stars.

Overview of the second connectivity algorithm

The second algorithm also works in $O(\log n)$ iterations. Unlike the first algorithm: (a) An iteration takes constant time, and (b) The pointer graph at the beginning of an iteration is a collection of rooted trees (which may not be stars). An iteration consists of the following steps.

1. *Probe quitting*; each rooted star whose supervertex is not adjacent to any other supervertex quits.
2. *Hooking on smaller*; each rooted star is hooked onto a smaller vertex based on an edge connecting a vertex of its supervertex to a vertex of another supervertex (if

such vertex exists); a vertex which is being hooked upon is never a leaf (where a vertex v is a *leaf* in the pointer graph if no other vertex points to v; namely there is no vertex u such that $D(u) = v$).

3. *Hooking non-hooked-upon*; every rooted star, which was not hooked upon in step (2), is hooked based on an edge connecting a vertex of its supervertex to a vertex of another supervertex if such vertex exists, and if not the rooted star quits; again, a vertex which is being hooked upon is never a leaf.

4. *Parallel pointer jumping*; an iteration finishes with one round of parallel pointer jumping.

Remark: The fact that Step (3) refers to (present) rooted stars excludes rooted stars that were hooked on others in Step (2).

A few missing details in the above description of the algorithm are added later.

Theorem 11.4: *The pointer graph always consists of rooted trees.*

Proof We prove the theorem by showing that the pointers graph can never contain a cycle. We define an "invariant" and Claim 1 establishes that the invariant holds (almost) throughout the algorithm.

- *Invariant*: For every vertex which is not a leaf in the pointer graph $D(v) \leq v$.

For proving Theorem 11.4 we first show the following.
Claim 1 The invariant holds after steps 1,2, and 4 of every iteration.
Proof of Claim 1 By induction on the steps (over iterations). Initially, each vertex forms a singleton rooted star and the invariant holds. If the invariant holds prior to Step 1 then it trivially holds after Step 1. Suppose that the invariant holds prior to Step 2. Changes introduced into the pointer graph in Step 2, may cause some vertices that were roots to point towards a lower numbered vertex. That is, if $D(v)$ changes in Step 2 then following Step 2 $D(v) < v$, and the invariant still holds. Suppose that the invariant holds prior to Step 3. We show that the invariant will hold true again following Step 4. In Step 3 the root r of a star may be hooked onto a vertex v (i.e., $D(r) = r$ changes into $D(r) = v$), where $v > r$, and the invariant may not hold; however, this will be rectified soon, as explained next.
Claim 2. All the vertices that point to such root r following Step 3 are leaves (formally, if $D(u) = r$ then u is a leaf) in the pointer graph.
We first show that Claim 1 follows from Claim 2 and then prove Claim 2. Consider a leaf u such that $D(u) = r$. The parallel pointer jumping in Step 4 advances all pointers of

such leaves into $D(u) = v$, and since vertex r becomes a leaf, the invariant holds again.
Proof of Claim 2. We claim that such a root r or any of the leaves which point to it, cannot be hooked upon in Step 3. To see this recall that Step 3 applies only to rooted stars that were not hooked upon in Step 2. Consider two such rooted stars (with roots r_1 and r_2, where $r_1 < r_2$) and their respective two supervertices. The two supervertices cannot be adjacent, since if they were then root r_2 should have been hooked in Step 2 (either on r_1 or on another vertex).

Proof of Theorem 11.4. Whenever the invariant holds true, the pointer graph consists of rooted trees. To see this, start a path in the pointers graph from any vertex which is not a leaf. The path advances through monotonically decreasing vertices (or vertex numbers) till it hits a tree root, and therefore no such vertex can be on a cycle. So, Claim 1 implies that Theorem 11.4 holds after steps 1,2, and 4 of every iteration. Claim 2 implies that Theorem 11.4 holds after step 3 of every iteration, as well.

Theorem 11.5: *The algorithm terminates within $O(\log n)$ iterations, and when it does each connected component of the input graph G is represented by a single rooted star.*

Proof We show that the algorithm terminates within $O(\log n)$ iterations. Given a rooted tree T, we define its *height*, $h(T)$, to be the number of edges in the longest path from a leaf to the root; however, if T consists of a single vertex then we define $h(T) = 1$. Suppose we are immediately after Step 1 of some iteration. Let $T_1, T_2, \ldots, T_\alpha$ be all the rooted trees in the pointer graph of some connected component of the graph. In case $\alpha = 1$ and T_1 is a rooted star then it should have quited in Step 1. Henceforth, we assume that we are not in this case. Let $H = h(T_1) + h(T_2) + \ldots + h(T_\alpha)$ be the total height of the trees. Let $S_1, S_2, \ldots, S_\beta$ be the rooted trees, of the same connected component following (Step 4 of) the same iteration, and denote their total height by $\bar{H}(= h(S_1) + h(S_2) + \ldots + h(S_\beta))$.
Claim 3 $\bar{H} \leq 2H/3$.
Before proving Claim 3, we note that it readily implies that the algorithm terminates within $O(\log n)$ iterations, and this part of Theorem 11.5 follows.
Proof of Claim 3 Assume without loss of generality that the hookings in steps 2 and 3 form a single tree Q out of trees $T_1, T_2, \ldots, T_k$ and following the parallel pointer jumping Q becomes S_1. Claim 3 follows from the following three observations:
(1) $2 \leq h(T_1) + h(T_2) + \ldots + h(T_k)$.
(2) $h(Q) \leq h(T_1) + h(T_2) + \ldots + h(T_k)$. To see this, note that a root can be hooked on a leaf only if the leaf is itself a singleton rooted star.
(3) $h(S_1) \leq 2h(Q)/3$ if $h(Q) \geq 2$.
The proof that, when the algorithm terminates, each connected component is represented by a single rooted star is left to the reader.

Step 3 is crucial for obtaining the logarithmic running time. To see this, consider modifying the second connectivity by omitting Step 3. Figure 35. gives an example for which the modified algorithm takes $n-1$ iterations. After one iteration vertex n points at

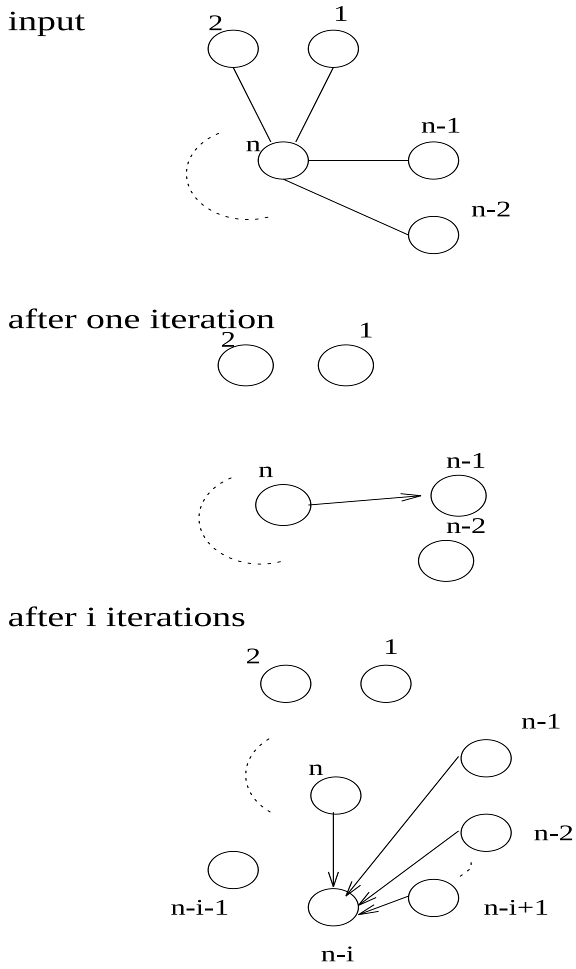


Figure 35: An example where Step 3 is omitted in the second connectivity algorithm

vertex 1 and the other vertices form singleton trees in the pointer graph. After iteration i vertices $n-i+1, n-i+2, \ldots, n-1$ and n points at vertex $n-i$, while the other vertices still form singleton trees.

Lower-level description details There are two processors standing by each edge (u, v) of the graph, one considers the edges for (possible hooking of) vertex u and the other for (possible hooking of) vertex v. In addition, there is a processor standing by each vertex for a total of $n+2m$ processors. The model of computation is an Arbitrary CRCW PRAM. For Step 1 we do the following to determine which tree roots do not quit (see also Exercise 39):
1.1 **for** each edge-processor **pardo**

- (focus on the processor of edge (u, v) for vertex u)
- **if** $D(u) \neq D(v)$
- **then** mark $D(u)$ for not quitting in the current iteration

1.2 **for** each root of a tree **pardo**
- **if** it is not the root of a star (see Exercise 39)
- **then** mark the root for not quitting in the current iteration
- quit unless marked not to

For Step 2 we do the following:

for each edge-processor **pardo**
- (focus on the processor of edge (u, v) for vertex u)
- **if** $D(u)$ is the root of a star and $D(v) < D(u)$
- **then** $D(D(u)) := D(v)$

The above instruction may result in concurrent write and is resolved using the Arbitrary CRCW convention. Step 3 is similar to Step 2 and Step 4 is similar its counterpart in the first connectivity algorithm.

Exercise 39: *1. Explain why Step 1.1 takes $O(1)$ time and $O(m)$ work (on a common CRCW).*
2. Show how to find all vertices which are roots of stars (for Step 1.2) in $O(1)$ time and $O(n)$ work (on a common CRCW).
3. Explain why the lower-level description of Step 1 marks a root for quitting if and only if it is the root of a star whose supervertex is a whole connected component.

Complexity Each iteration takes $O(1)$ time. The algorithm runs in $O(\log n)$ time and $O((n + m) \log n)$ work on an Arbitrary CRCW. The algorithm does not achieve optimal speed up.

Exercise 40: *Consider a slight enhancement of the second connectivity algorithm as follows. Change Step 2 so that roots of trees (and not only stars), are hooked, but maintaining the rule that they can be hooked only on smaller vertices. Show that the algorithm is still correct and that the same complexity analysis carries through.*

11.3. Minimum spanning forest

Let $G(V, E)$ be an undirected graph. Each edge $e \in E$ has a **weight** $w(e)$ which can be any real number, and assume (without loss of generality, as explained below) that all edge weights are pairwise different. If G is connected then a **spanning tree** of G is a tree $G(V, \bar{E})$ where the set $\bar{E}$ is a subset of E. For a general graph G, a **spanning forest** of

G is a collection of one spanning tree for each connected component of G. The weight of a spanning forest is the sum of the weights of its edges. A **minimum spanning forest (MSF)** of a graph G is a spanning forest of G whose weight is minimum.
(*Comment:* There is no loss of generality in assuming that edge weights are pairwise different since if instead of $w(e)$ we can define the the weight of edge $e = (u, v)$ to be the triple $(w(e), max(u, v), min(u, v))$ and then apply a lexicographic order to these triples.)

Exercise 41: *1. Show how to derive a (not necessarily minimal) spanning forest algorithm from the first connectivity algorithm (with the same time and work complexity).*
2. Show how to derive a spanning forest algorithm from the second connectivity algorithm (with the same time and work complexity).

All known efficient algorithms for the MSF problem appear to be based on the following theorem.

Theorem 11.6: The MSF Theorem *Let $G(V, E)$ be a weighted graph, as above, and let U and $V - U$ be two non-empty subset of V. Consider the set H of all edges with one endpoint in U and the other in $V - U$, $H = \{(u, v); u \in U$ and $v \in V - U\}$. Suppose that this set of edges is not empty and let e be the edge of minimum weight in the set. Then e is in the MSF of G.*

Proof Let F be an MSF of G and assume in contradiction that $e = (u, v)$ is not in F. Since F is an MSF there must be a simple (i.e., cycle free) path between u and v in F. This path must contain an edge $f \in H$. Now, replacing f by e gives another spanning forest of G whose weight is smaller than F, contradicting the possibility of not including e in F. The theorem follows.

We describe two MSF algorithms, one is an adaptation of the first connectivity algorithm and the other is an adaptation of the second.

For simplicity we will describe the MSF algorithms for the Priority CRCW. Exercise 43 mentions a possibility for relaxing this assumption. Our algorithms start with the following step.

Step 1 Allocate a processor to each edge, as follows. Sort the edges of the graph by weight, and allocate to them processors so that the heaviers the edge, the higher the serial numbers of the processor.
Given a set of edges, this enables picking the edge of minimum weight in the set in a single round, as follows: for each edge a write attempt into the same shared memory location is made; the Priority CRCW convention guarantees that the edge of smallest weight in the set is picked, since its processors has the smallest serial number.

11.3.1. A first MSF algorithm Each iteration consists of one round of hookings and $\log n$ rounds of parallel pointer jumping. The hookings are going to be different than in the first connectivity algorithm. Each star root r finds the edge of minimum weight $e = (u, v)$ among the edges that connect a vertex u in its own supervertex with a vertex v in another supervertex. Root r hooks itself onto the root of v; in case two roots are hooked on one another, we cancel the hooking of the smaller (numbered) root. As a result several rooted stars form a rooted tree, whose root is the root of one of these rooted stars. If a root does not have an adjacent root, then the rooted star together with all its vertices *quit* the algorithm, since they form a complete connected component. The parallel pointer jumpings are as in the first connectivity algorithm.

The algorithm indeed finds an MSF based on the MSF Theorem.

Exercise 42: *Prove that the pointer graph never contains cycles.*

Exercise 43: *1. As described, the algorithm sorts the edges in $O(m \log n)$ work and $O(\log n)$ time and then runs in $O(\log^2 n)$ time and $O(n \log^2 n + m \log n)$ work on a Priority CRCW. Fill in the details for achieving such a performance.*
2. Get an MSF algorithm which runs in $O(n^2)$ work and $O(\log^2 n)$ time on a CREW. Suggested approach: The first connectivity algorithm was described for a CRCW model but then an exercise showed how to adapt it for an CREW with the same performance (of $O(n^2)$ work and $O(\log^2 n)$ time). Generalize these CREW version into an MSF algorithm. (Comment. Clearly, the above step for sorting the edges of the graph by weights will not be needed.)

11.3.2. A second MSF algorithm Following the step in which all edge are sorted by increasing weight we apply the iterations of the second connectivity algorithm. Only the hookings steps (steps 2 and 3) change. Perhaps surprisingly they become simpler since there is only one hooking step. The hookings are similar to the first MSF algorithm. Each star root r finds the edge of minimum weight $e = (u, v)$ among the edges that connect a vertex u in its own supervertex with a vertex v in another supervertex. Root r hooks itself onto $D(v)$; in case two roots are hooked on one another, we cancel the hooking of the smaller (numbered) root.

The algorithm indeed finds an MSF based on the MSF Theorem.

Exercise 44: *Prove that the pointer graph never contains cycles.*

Complexity $O(m \log n)$ work and $O(\log n)$ time on a Priority CRCW.

Exercise 45: *A 2-coloring of an undirected graph G is an assignment of black or white to each vertex so that no two vertices with the same color share an edge of G.*
(1) Give an $O(\log n)$ time, $O(n)$ work parallel algorithm to 2-color an undirected tree T.
(2) An undirected graph G is bipartite if the nodes can be partitioned into two sets A and B, so that there are no edges of G which are internal to A (i.e., both endpoint of the edge are in A), or internal to B. Give an $O(\log n)$ time, $O(\ (n+m)\log n)$ work parallel algorithm to determine whether a given undirected graph is bipartite. Explain why your algorithm is correct.

12. Bibliographic Notes

Parallel algorithms have appeared in the literature since the late 1960's. Still, it is not clear which among the following a proper historical perspective should emphasize: (1) the first papers that present parallel algorithmics in various models of parallel computation, such as [KM68], [Bre74], [Win75], [Val75], [Arj75], [Hir76] and [Eck77]; or (2) the first papers that defined the PRAM model of computation from a complexity theoretic point of view, such as [Gol78] and [FW78]; or (3) later work which implies that all these papers are actually addressing the same issues. For instance, in response to a question in Valiant [Val75], [SV81] a suggested PRAM-style model of computation which permitted one of his comparison model algorithms (for finding the maximum among n elements; later [BH85] and [Kru83] showed that Valiant's merging comparison model algorithm is also a basis for a PRAM algorithm. In another paper [SV82b] observed that a scheduling principle which was proposed by Brent [Bre74] for his implicit and very weak algebraic model of computation can be used for an informal high-level description of a rather involved parallel algorithm, and led me towards the emphasis on the Informal Data-Parallel model (see the section on the algorithm for the selection problem), as a "key to thinking in parallel", in these notes. (Interestingly, [MR89] led [CV88a] into pointing out that even Brent's original informal data-parallel algorithm for the evaluation of an arithmetic expression can be recasted as a PRAM algorithm.) In a complexity theoretic setting, [Ruz81] states that alternating Turing machine are equivalent to bounded fan-in circuits and [SV84] describe another strong relationship between unbounded fan-in circuits and PRAMs.

Relating PRAM algorithms to machines began with Schwartz [Sch80], who dubbed as *Paracomputer* a theoretical model of parallel computation which is similar to the PRAM, as opposed to the *Ultracomputer*, which is a model of a machine. He suggested the Paracomputer for theoretical studies but apparently not as a programmer's model. Gottlieb et al [GGK+83] and the position paper [Vis83b] suggested the PRAM as a programmer's model for parallel computation; the former paper based the suggestion on a second generation design outline of the New York University Ultracomputer, while the latter relied on two aspects: (1) the formal efficient emulatability of the PRAM on a fixed degree interconnection network, as was first established in [MV84] and [Vis84b]; and (2) the existing and potential wealth of PRAM algorithms. Enriching the PRAM model of parallel computation by stronger models was considered in [GGK+83] using the Fetch-and-Add construct, in Blelloch [Ble90] using a simpler Scan construct, and as a principle, in [Vis83a].

These notes complement the following literature: the book [JáJ92] which is aimed at a more advanced course, and includes much more material and the even more advanced edited book [Rei93]. A sequence of informative survey papers on the topic of parallel algorithms has appeared. This includes [EG88] and [KR90] to mention just a few. References to the "NC theory" are [Par87], a chapter in [JáJ92] and the chapter by R. Greenlaw in [Rei93].

Blelloch et al [BCH+93] describe a portable data-parallel language, called NESL, whose efficient implementation has been demonstrated on several parallel and serial machines. The recent programming language international standard *Fortran 90*, as described in [MR90], enables coding data-parallel (or PRAM) algorithms. The standard is meant to guide machine designers for supporting such a language.

The balanced binary tree parallel prefix-sums algorithm is based on [Sto75] and [LF80]. The simple merging-sorting algorithm is from [SV81]. Cole [Col88] gave a powerful sorting algorithm which runs in $O(\log n)$ time and $O(n \log n)$ operations; while we do not present his algorithm, we refer to it in several places in the text.

The parallel selection algorithm is based on [Vis87], whose main idea appeared in [BFP+72]. The presentation of the selection algorithm is used to demonstrate the "accelerating cascades technique" (as outlined in [CV86a]). The scheduling principle that came out of [Bre74] brought [SV82b] to propose the following methodology for describing parallel algorithms: (1) give an outline in a model similar to the informal data-parallel (IDP) model; then try to (2) give a detailed description on a PRAM model. In these notes, we slightly refine this methodology by adding an intermediate step: (1.5) give a description on the data-parallel (DP) model. The introduction section brings a formal emulation of the Data-Parallel model by a PRAM (i.e., from Step 1.5 to Step 2), and the section on the selection problem discusses informal data-parallelism; that is after describing an algorithms as per Step 1 translate it, using ad-hoc methods, into a DP algorithm; this methodology of trying to recast an Informal Data-Parallel description of a parallel algorithm as a Data-Parallel one has turned out to be very successful, as many example demonstrate. General methods that replace these ad-hoc methods are given in [GMV91] and [Goo91], where processor allocation problems are solved.

The term "data-parallel" has been suggested in [HS86]. It appears that there is no commonly agreed definition of this term. We feel that we, and actually [JáJ92] before us, did not abuse this term.

The integer sorting algorithm is based on a comment in [CV86b]. The papers [BLM+91] and [ZB91] discuss implementation of several parallel sorting algorithms including this integer sorting one on several parallel machines. A randomized logarithmic time integer sorting algorithm, which runs in optimal work for a restricted range of integers, appeared in [RR89].

The 2-3-tree algorithm is based on [PVW83]; randomized parallel algorithm which support search, insert and delete operations, using optimal work were given in [DM89], and [GMV91].

The algorithm for finding the maximum among n elements is from [SV81] which is based on Valiant's parallel comparison model algorithm [Val75]. Doubly logarithmic trees and their role for parallel algorithms is discussed in [BSV93]. The randomized algorithm is related to [Rei81]. Exercise 18 is actually related to a much earlier paper [Hoa61]

The Euler tour technique is from [TV85]. The standard list ranking algorithm is from [Wyl79]. The randomized one is based on [Vis84c]. The right tail theorem is from [CLR90]. The deterministic coin tossing (DCT) technique and the deterministic list ranking algorithm are based on [CV86b]. The variant of DCT used is similar to [GPS87], from which we took also Exercise 27. Alternative list ranking algorithms were given in: [AM88], [CV88b] and [CV89].

Tree contraction as a paradigm, appeared in [MR89]. The algorithm given is similar to [ADKP89], and [KD88].

The first parallel graph connectivity algorithm is based on [HCS79] and the second is based on [SV82a], using a variant of [AS87]. Exercise 38 is from [CLC82] and Exercise 37 is from [Vis84a].

13. Acknowledgement

Anat Bremler drew a first draft of most of the figures. This help is gratefully acknowledged.

References

[ADKP89] K.N. Abrahamson, N. Dadoun, D.A. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *J. Algorithms*, 10(2):287–302, 1989.

[AM88] R.J. Anderson and G.L. Miller. Optimal parallel algorithms for list ranking. In *Proc. 3rd Aegean workshop on computing, Lecture Notes in Computer Science 319, 1988, Springer-Verlag*, pages 81–90, 1988.

[Arj75] E.A. Arjomandi. *A Study of Parallelism in Graph Theory*. PhD thesis, PhD thesis, Dept. Computer Science, University of Toronto, 1975.

[AS87] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. *IEEE Trans. on Computers*, 36:1258–1263, 1987.

[BCH+93] G.E. Blelloch, S. Chatterjee, J.C. Harwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proc. of the 4th ACM PPOPP*, pages 102–111, 1993.

[BFP+72] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *J. Comp. Sys. Sci.*, 7(4):448–461, 1972.

[BH85] A. Borodin and J.E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *J. Computer and System Sciences*, 30:130–145, 1985.

[Ble90] G.E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[BLM+91] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proc. of the 3rd SPAA*, pages 3–16, 1991.

[Bre74] R.P. Brent. The parallel evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.*, 21:302–206, 1974.

[BSV93] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.

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

[CLR90] T.H Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990. Second Printing.

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

[CV86a] R. Cole and U. Vishkin. Deterministic coin tossing and accelating cascades: micro and macro techniques for designing parallel algorithms. In *Proc. of the 18th Ann. ACM Symp. on Theory of Computing*, pages 206–219, 1986.

[CV86b] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70:32–53, 1986.

[CV88a] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–348, 1988.

[CV88b] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17:128–142, 1988.

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

[DM89] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 360–368, 1989.

[Eck77] D. M. Eckstein. *Parallel Processing Using Depth-First Search and Breadth-First Search.* PhD thesis, Computer Science Department, University of Iowa, Iowa City, Iowa, 1977.

[EG88] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.*, 3:233–283, 1988.

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

[GGK+83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAulife, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing an MIMD shared memory parallel machine. *IEEE Trans. on Comp*, C-32:175–189, 1983.

[GMV91] Y. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 698–710, 1991.

[Gol78] L. M. Goldschlager. A unified approach to models of synchronous parallel machines. In *Proceedings 10th Annual ACM Symposium on Theory of Computing*, pages 89–94, 1978.

[Goo91] M.T. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. In *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 711–722, 1991.

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

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

[Hir76] D.S. Hirschberg. Parallel algorithms for the transitive closure and the connected components problems. In *Proc. of the 8th Ann. ACM Symp. on Theory of Computing*, pages 55–57, 1976.

[Hoa61] C.A.R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Comm. ACM*, 4,7:666–677, 1961.

[HS86] W.D. Hillis and G.L. Steele. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, 1986.

[JáJ92] J. JáJá. *Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[KD88] S.R. Kosaraju and A.L. Delcher. Optimal parallel evaluation of tree-structured computations by ranking. In *Proc. of AWOC 88, Lecture Notes in Computer Science* No. 319, pages 101–110. Springer-Verlag, 1988.

[KM68] R.M. Karp and W.L. Miranker. Parallel minimax search for a maximum. *J. of Combinatorial Theory*, 4:19–34, 1968.

[KR90] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 869–942. MIT Press, 1990.

[Kru83] C.P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. on Comp*, C-32:942–946, 1983.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *J. Assoc. Comput. Mach.*, 27:831–838, 1980.

[MR89] G.L. Miller and J.H. Reif. Parallel tree contraction part 1: fundamentals. *Advances in Computing Research*, 5:47–72, 1989.

[MR90] M. Metcalf and J. Reid. *Fortran 90 Explained.* Oxford University Press, New York, 1990.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[Par87] I. Parberry. *Parallel Complexity Theory.* Pitman, London, 1987.

[PVW83] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proc. of 10th ICALP, Springer LNCS 154*, pages 597–609, 1983.

[Rei81] R. Reischuk. A fast probabilistic parallel sorting algorithm. In *Proc. of the 22nd IEEE Annual Symp. on Foundation of Computer Science*, pages 212–219, October 1981.

[Rei93] J.H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, San Mateo, California, 1993.

[RR89] S. Rajasekaran and J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18:594–607, 1989.

[Ruz81] W.L. Ruzzo. On uniform circuit complexity. *JCSS*, 22:365–383, 1981.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, 1980.

[Sto75] H.S. Stone. Parallel tridiagonal equation solvers. *ACM Tran. on Mathematical Software*, 1(4):289–307, 1975.

[SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms*, 2:88–102, 1981.

[SV82a] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[SV82b] Y. Shiloach and U. Vishkin. An $o(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3:128–146, 1982.

[SV84] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13:409–422, 1984.

[TV85] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM J. Computing*, 14:862–874, 1985.

[Val75] L.G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4:348–355, 1975.

[Vis83a] U. Vishkin. On choice of a model of parallel computation. Technical Report TR 61, Dept. of Computer Science, Courant Institute, New York University, 1983.

[Vis83b] U. Vishkin. Synchronous parallel computation - a survey. Technical Report TR 71, Dept. of Computer Science, Courant Institute, New York University, 1983. Also: Annual Paper Collection of "Datalogforeningen" (The Computer Science Association of Aarhus, Denmark), 1987, 76-89.

[Vis84a] U. Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Math*, 9:197–207, 1984.

[Vis84b] U. Vishkin. A parallel-design distributed-implementation (PDDI) general purpose computer. *Theoretical Computer Science*, 32:157–172, 1984.

[Vis84c] U. Vishkin. Randomized speed-ups in parallel computations. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 230–239, 1984.

[Vis87] U. Vishkin. An optimal algorithm for selection. *Advances in Computing Research*, 4:79–86, 1987.

[Vis91] U. Vishkin. Structural parallel algorithmics. In *Proc. 18th ICALP - Lecture Notes in Computer Science 510, Springer-verlag*, pages 363–380, 1991.

[Win75] S. Winograd. On the evaluation of certain arithmetic expressions. *J. Assoc. Comput. Mach.*, 22,4:477–492, 1975.

[Wyl79] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Computer Science Department, Conell University, Ithaca, NY, 1979.

[ZB91] M. Zagha and G.E. Blelloch. Radix sort of vector multiprocessors. In *Proc. Supercomputing 91*, 1991.

14. Index

Work-time methodology 35