Structural Parallel Algorithmics

Uzi Vishkin*

University of Maryland &
Tel Aviv University

Abstract

The first half of the paper is a general introduction which emphasizes the central role that the PRAM model of parallel computation plays in algorithmic studies for parallel computers.

Some of the collective knowledge-base on non-numerical parallel algorithms can be characterized in a structural way. Each structure relates a few problems and technique to one another from the basic to the more involved. The second half of the paper provides a bird's-eye view of such structures for: (1) list, tree and graph parallel algorithms; (2) very fast deterministic parallel algorithms; and (3) very fast randomized parallel algorithms.

1 Introduction

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 experience is that the design of parallel algorithms requires new paradigms and techniques, offering an exciting intellectual challenge. We note that it had not been clear that the design of efficient parallel algorithms for "enough" problems is at all possible. Specifically, I recall a discussion with a colleague in 1979. In a thought-off support of a skeptical position, he quoted [KM68], who proved that parallelism will be rather ineffective in the context of binary search; informally, they show that an increase in the number of processors from one to p may cut the time of binary search by a factor of at most $\log p$.

This review paper relates to very introductory as well as rather advanced material on efficient parallel algorithmics. Only a very partial list of possible topics are touched upon. Preference was given to domains of parallel algorithms where more structure, in a sense that is explained later, was found. Omitted is a review of general NC algorithms and the wealth of fundamental results they offer (e.g., for more on this work see [Coo81], [Coo85] and [KR88a]) It was impossible to give a self-contained presentation within the space limitations. An introduction to parallelism, PRAMs, and PRAM algorithms is followed by a review of list,

*Partially supported by NSF grant CCR-8906949.

tree and graph algorithms, most of which are not very recent. The last two chapters bring more recent very fast deterministic and randomized parallel algorithms. Our presentation emphasizes examples where the main contribution of a paper was not principally in the results it presented, but rather in a new idea, that provided new tools, and thereby evolutionized concepts as to what can be done efficiently in parallel. Frequently, this meant identifying and solving subtle key problems that had been previously unnoticed obstacles blocking further development in various areas. This distinction of idea-versus-result driven research is not an easy task, for a standard way of arguing that an idea is powerful is to show that it has many applications and leads to stronger results.

For over two decades there has been an understanding that fundamental physical limitations on processing speed will eventually force high performance computation to be targeted principally at the exploitation of parallelism. Today, just as the fastest cycle times are approaching these fundamental barriers, a second generation of moderately parallel machines is emerging, and a technology of processing elements and communication switches is appearing with sufficient power to accelerate the pace of experimental parallel machine research. At the same time, there has been a corresponding maturation in our understanding of interconnection networks and their performance costs, although the substantial evolution in this area shall doubtlessly continue. In the design of parallel algorithms, progress during the last decade has redirected the principal research focus from an effort to classify the problems that can be solved in $O(\log^k n)$ time on n^l processors, where l and k are constants (NC algorithms), to a growing body of research on how to design efficient algorithms exhibiting good speedup on parallel machines.

The question of how to model parallel computation is subtle, and has significant impact on both the design of parallel systems and the design of parallel algorithms. This problem has no single answer; indeed, investigations touched upon two aspects of the design question: *techniques for application-specific design*, and *general purpose design*.

Application specific problems include methodologies for designing algorithms on special purpose processing organizations. Their use is primarily justified by the enormous performance/cost benefits that can be attained for worthy problems that admit such solutions.

At the other end of the spectrum is the question of how to design general purpose parallel algorithms, which may not be targeted for a specific machine, and which may be too complex to be suitable for low level design. Our principal model for algorithmic design in this area is the PRAM (parallel random access machine), which is the focal point for the present report.

At first glance, the PRAM model of computation might not appear to be suitable as a general model for designing efficient parallel algorithms; indeed, even its original use by the theory community was not, for the most part, to design efficient algorithms. Yet the PRAM has now won fairly widespread acceptance in the theoretical community, as a model for efficient parallel computation.

Loosely speaking, the PRAM model of computation is an idealization that draws its power from three facts and consequences:

- It 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. 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.

- Many of the design paradigms have turned out to be strikingly robust; as a consequence, they have applications in models outside the PRAM domain, including VLSI, where each wire element and gate is carefully accounted in the complexity cost.

- Recent advances have shown PRAM 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 PRAMs.

The following informal statement represents my belief on the future of general-purpose parallel computation: *Unless parallel machines are designed to support the PRAM, or a model of parallel computation which is very close to it, the design of parallel algorithms is doomed to be a very difficult (or even impossible) task*; to avoid misunderstand, it emphasized that computer designers should aspire to make their machine a virtual PRAM, and no statement is being made about the actual design.

In the rest of the paper we take snapshots summarizing some chief characteristics of efficient parallel algorithms.

2 The PRAM Model

We start by reviewing the basics of the PRAM model. A PRAM employs p synchronous processors, all having unit time access to a shared memory. There are a variety of rules for resolving memory access conflicts. The most common are exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW), and concurrent-read concurrent-write (CRCW). 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 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). The survey paper [Vis83] elaborates on the raison d'etre of the "PRAM approach". Survey papers specializing on the class NC are [Coo81] and [Coo85]. More recent review articles include [EG88], [KR88a], and [KRS88], as well as [Ata90b], which is devoted to parallel computational geometry. Books on the topic include [Akl89], [GR88], [JáJ91], [Par87] and [Rei91].

For sequential computation, it has been of considerable advantage to deal with an abstraction of the von-Neumann machine, namely the RAM or Random Access Machine (see a standard textbook, such as [AHU74]). Two major advantages of such an abstraction are that it makes the algorithm designer's task less complex, and it eliminates obstacles to algorithm portability. A third reason for the success of the RAM model is that its cost complexity generally provides an accurate approximation of the running time on real sequential machines: by and large, efficient RAM algorithms translate into efficient programs on specific machines that are properly designed. Similar motivations justify the use of the PRAM model for parallel computation.

While the PRAM model is demonstrably simple, and provides a clean medium for expressing algorithms, its power depends equally on the wealth of high performance algorithms that have been inspired by the model.

Given two parallel algorithms for the same problem one is *more efficient* than the other if: (1) primarily, its time-processor product is smaller, and (2) secondarily (but important), its parallel time is smaller. *Optimal* parallel algorithms are those whose time-processor product is asymptotically equal to the serial complexity of the problem. They correspond to optimal (often linear) time sequential algorithms. A *fully-parallel* algorithm is a parallel algorithm that runs in constant time using an optimal number of processors. The notion of fully-parallel algorithm represents an *ultimate theoretical goal* for designers of parallel algorithms. Research on lower bounds for parallel computation indicates that this goal is unachievable for almost any interesting problem. These same results often preclude much weaker time bounds for the same problems. Consequences of the above discussion are: (1) the evolving theory of very fast parallel algorithms cannot benefit from the theory of not-as-fast parallel algorithms; and (2) any result that approaches the fully-parallel performance goal is somewhat surprising. The quest for fast and processor-efficient parallel algorithms has also contributed towards establishing a *tradition of excellence* similar to the one implied by the quest for fast serial algorithms.

While lower-bound techniques are not the focus of this paper, we mention here several lower-bound results whose circumvention provided motivation for much of the research in sections 5 and 6: (1) $\Omega(\log n/\log\log n)$ time using a polynomial number of processors for the parity problem [BH87]; (2) for finding the maximum among n elements [Val75], and merging [BH85] on a parallel comparison model of computation; and (3) for CREW PRAM computation of the *OR* function of n bits [CDR86].

As explained elsewhere (e.g., [KR88a], [KRS88] or [Vis83]), the PRAM should be viewed as a virtual design-space for a parallel machine and not as a parallel machine, and improvement in the parallel running time of a PRAM algorithm can benefit us in reducing the actual running time. An important application area, where this is desired, is deadline-driven computing. Starting from the applications and trying to design very fast parallel algorithms for them is a natural approach. However, the fact that only few very fast algorithms are known makes this approach hard to pursue. How can one design a very fast parallel algorithm for a specific application without having some algorithmic paradigms that can be followed?! A knowledge-base of deadline-driven parallel algorithms is needed. We suggest the following first step towards building such a knowledge-base: develop a core of problems that can be computed very fast, as well as very fast computational paradigms. Another line of additional justification follows [KRS88], [Val90] and [Vis84a] that advocate slackness in processors. Let us explain. Suppose we are given an efficient PRAM algorithm and a (real) parallel machine with p_1 processors, on which we wish to simulate the algorithm. Suppose that the PRAM algorithm is efficient for up to p_2 PRAM processors. In this case, *processor slackness* is defined as the ratio p_2/p_1. Informally, each of these three papers argues that even if p_1 is fixed, having a larger p_2 (and therefore larger processor slackness) leads to a more efficient simulation by the real machine.

Let us sum up. *Getting the fastest possible time by a processor-efficient algorithm is a primary intellectual challenge; the techniques developed are likely to have practical importance.*

3 PRAM Algorithms

A considerable body of PRAM algorithms has been discovered over the past several years; many of them are for fundamental problems that have been recognized as classical in the theory of serial algorithms. The benefit from the PRAM model is not only in the extensive list of efficient and fast parallel algorithms that have been designed. Fundamental paradigms and design techniques have emerged, which are of use in many, if not all, models of physically available parallel machines. These techniques have led to efficient fast parallel algorithms in a diversity of areas, including computational geometry, graph problems, pattern matching, and comparison problems.

The PRAM was first proposed as a model for parallel computation in a joint complexity theoretic and algorithmic context in a 1979 thesis [Wyl79] and in a paper [FW78]; this original instance concerned the CREW PRAM model. [Sch80b] also advocated using a PRAM for studying the limits of parallel computation at around the same chronological time. [Gol82] was the first to propose the CRCW PRAM model in a complexity theoretic context (he called it a SIMDAG). [Pip79] identified and characterized the class NC. [SV81] suggested using the CRCW PRAM in an algorithmic context.

Figures 1-3 are a focal point for this short tutorial paper. The figures illustrate some of the structure of PRAM algorithmics. The research itself seems to have been led by a desire to solve some involved problems; however, these figures reveals that in order to accomplish this, a variety of techniques, as well as solutions to more fundamental underlying problems, have been introduced. This structure of problems and techniques (from the basic to the more involved) adds *elegance* to parallel algorithmics. Such fine structure (or actually much finer) exists in a few classical fields of Mathematics, but is rather unique in combinatorics-related theories. The first observation that such interesting structures are possible was in the context of list, tree and graph problems; this particular structure is described in the next section, and illustrated in Figure 1. Most of this work was done between 1980 and 1988. We highlight structure-related issues of this work, primarily for background; an elaborate overview of most of this material can be found in [EG88] and [KR88a]. The algorithms for most "target problems" in this figure (these are the more involved and known problems; usually, they are at the top or slightly below the top of the figure) run in logarithmic time. The other figures (and sections) are on doubly-logarithmic time, or faster algorithms; this work was done recently; while it was hard to anticipate the structure of Figure 1 beforehand, searching for a similar structure became one of the research goals for the later work.

4 List, Tree and Graph Algorithms

A basic routine that is used most often in parallel algorithms is undoubtedly that for the *prefix sums* problem [LF80]. The fact that the prefix-sums problem appears at the bottom of Figure 1 is meant to convey the basic role of this problem. A faster CRCW algorithm for prefix-sums also exists [CV89]. A generalization of this problem to pointer structures, the *list ranking* problem, was identified in [Wyl79]; list ranking has proven to be a key subroutine in parallel algorithms. In fact, obtaining optimal algorithms for list ranking and (undirected) graph connectivity proved to be central to obtaining optimal algorithms for a considerable number of list, tree and graph problems. First randomized, and later deterministic, optimal parallel algorithms for list ranking were given [Vis84b], [CV86b], [CV86a], [AM88] and [CV89]. The deterministic algorithms are based on a deterministic arbitration technique, dubbed *deterministic coin tossing* [CV86b]. Extensions of this technique for sparse graphs and other applications were given [GPS87], [CZ90], and [HCD87].

Key techniques for parallel algorithms on trees are reviewed next: (1) The *Euler tour* technique [TV85] reduces the computation of many tree problems to list ranking. (2) The *tree contraction* technique [MR85] led to a number of optimal randomized logarithmic-time algorithms for tree problems, including expression tree evaluation; optimal deterministic versions were also given [GR86], [CV88], [ADKP87] and [KD88]. Implicit use of tree contraction in a non-standard parallel algorithmic setting appeared in [Bre74]. (3) *Centroid decomposition* of a tree, as implicitly used in [Win75] for $O(\log^2 n)$ time computations. Accelerating centroid decomposition was the motivation for the tree contraction version of [CV88].

Two logarithmic time connectivity algorithms were given: (1) a deterministic one which is optimal on all except very sparse graphs [CV86a]; (2) a randomized optimal one [Gaz86]. For Figure 1, the deterministic algorithms builds on a restricted *union find* problem, a scheduling problem, dubbed *duration unknown task scheduling*, and the Euler tour technique, as well as ideas from two previous connectivity algorithms [HCS79] and [SV82a]. It should be pointed out that the logarithmic time version of the deterministic connectivity algorithm requires the use of expander graphs and thus is highly impractical at present; however, a slightly less parallel version involves much smaller constants.

The graph connectivity problem turned out to be the main obstacle to deriving optimal logarithmic time algorithms for several graph problems, including: *biconnectivity* [TV85], *finding Euler tour in a graph* [AV84], [AIS84] and *orienting the edges of an undirected graph to get a strongly connected digraph ("strong orientation")* [Vis85a]. We also note some recent parallel algorithms for k *(edge and vertex) connectivity* problems [KS89] and [CT91].

The problem of achieving optimal speedups on sparse graphs for the strong orientation and biconnectivity problems turned out to depend on an efficient solution for yet another fundamental problem: preprocessing of a rooted tree so that a query requesting the *lowest common ancestor (LCA)* of any pair of nodes can be processed in $O(1)$ operations. Parallel algorithms for this problem [SV88] and [BV89] use the Euler tour technique.

Depth first search (DFS) is perceived by many as the most useful technique known for designing sequential algorithms for graph problems. Unfortunately, it is not known how to implement DFS efficiently in parallel. A technique called *ear decomposition search* (EDS) was suggested as a replacement for DFS in the context of efficient and fast parallel algorithms [MSV86] and [MR86], after an earlier suggestion in [Lov85] for computing EDS in parallel in a fast but inefficient manner. The EDS method implies alternative algorithms for *biconnectivity* and *strong orientation*. More powerful applications were for finding an *st-numbering* of a graph, again in [MSV86], as well as for triconnectivity algorithms [MR87] and [RV88]. An *st*-numbering is used in the *planarity testing* algorithm of [KR88b]. The most recent algorithms for triconnectivity [FRT89] and planarity testing [RR89b], are very nice examples of reaching target problems by building an even higher level in the structure of Figure 1, and using effectively many of the previous techniques.

5 Deterministic Fast Algorithms

Structure that was found in optimal doubly-logarithmic time (or faster), parallel algorithms is highlighted. Figure 2.1 discusses works that can be viewed as using the *doubly-logarithmic tree* paradigm, as per [BBG+89] . Doubly-logarithmic trees are rooted trees with $n = 2^{2^i}$ leaves for some integer $i > 0$. The root has $2^{2^{(i-1)}}$ children, each being the root of a doubly-logarithmic subtree with $2^{2^{(i-1)}}$ leaves. For $i = 0$ a doubly-logarithmic tree consists of a root and two children, which are leaves. Such structure guides the computation in optimal doubly-logarithmic parallel algorithms for finding the *maximum* among n elements [SV81] (using [Val75]), finding the *maximum relative to all prefixes* of an array of elements [Sch87] and [BSV88] (the *prefix-maxima* problem), *merging* two sorted lists [Kru83] and [BH85], finding the *convex hull of a monotone polygon* [BSV91], and finding *all nearest neighbors in a convex polygon* [SV90]. Note that all merging algorithms that are mentioned in this paper may be implemented on a CREW PRAM. *String matching*: For some family of parallel algorithms it is sufficient to consider only non-periodic patterns [Gal85]. A method for eliminating (at least) one among two potential occurrences of a non-periodic pattern string in a text string in [Vis85b] was observed in [BG88] to be similar to comparing two numbers in order to determine which one is larger and together with an algorithm for finding the maximum, led to an optimal doubly-logarithmic *string matching* algorithm; [BG91] showed recently a matching lower-bound for a parallel comparison model of computation. The *all nearest smaller values (ANSV)* problem is: given an array $(a_1, a_2 ... a_n)$, the ANSV problem is to find for each $1 \leq i \leq n$ the nearest j and l, such that a_j and a_l are smaller than a_i (that is, find the smallest $l > i$ such that $a_l < a_i$ and the largest $j < i$ such that $a_j < a_i$). While generalizing two problems - finding the maximum and merging - an optimal doubly-logarithmic algorithm for ANSV was still possible [BSV88]. In the same paper, the ANSV algorithm is shown to lead to optimal doubly-logarithmic algorithms for the following *range-maxima* problem: preprocess an array of numbers $(a_1, a_2 ... a_n)$, so that for any pair of indices i and j, where $1 \leq i < j \leq n$, a *range-maximum* query requesting the maximum among $(a_i, a_{i+1} ... a_j)$ can be processed in constant-time. More remotedly related to the doubly-logarithmic tree paradigm is a matrix searching algorithm [Ata90a]

Remark. Some of the problems mentioned in this section, particularly from here on, may have a rather specific flavor. However, they are still interesting since improvement on the more general problem is either impossible or apparently difficult.

The *surplus-log* approach: suppose the aim is designing a triply-logarithmic (or faster) optimal parallel algorithm; the surplus-log approach suggests the following first step: *design an algorithm with $n \log n$ processors and constant-time.* Uses of the surplus-log approach come in two flavors: (1) As part of a global strategy. (2) As a rule-of-thumb (or "sorcery"); that is, it merely provides an insight that leads to further improvements; in other words, for some reason, which is not fully clear to us, it sometimes helps to follow the surplus-log approach.

A *triply-logarithmic paradigm* [BJK+90] uses the surplus-log approach, in conjunction with doubly-logarithmic algorithms for the same problems, as part of a global strategy. The strategy leads to optimal parallel algorithms for several problems whose running time is triply-logarithmic in the following sense: consider, for instance, the problem of merging two sorted lists of integers drawn from the domain $[1...s]$. The running time obtained is $O(\log\log\log s)$ [BV90]. There are also similar triply-logarithmic results for the prefix-maxima problem [BJK+90] (and thereby for finding the maximum among n elements).

Optimal *log-star* time (i.e., $O(\log^* n)$) time) parallel algorithms seem to be the hardest to fit into a strict structure of paradigms using presently available ideas. See Figure 2.2. However, using the surplus-log approach, as a rule-of-thumb, was helpful for several problems: (1) *String matching* for a preprocessed pattern [Vis91]; (2) prefix-maxima [BJK+91]; there, this prefix-maxima algorithm is also the most time consuming step in an algorithm for routing around a rectangle - a VLSI routing problem ; and (3) for preprocessing a rooted tree, so that any *level-ancestor* query can be processed in constant-time [BV91b]. The input for such query consists of a vertex v and an integer l; the output is the l'th ancestor of v, where the first ancestor of a vertex is its parent and the l'th ancestor is the parent of the $(l-1)$'st ancestor; the Euler tour of the tree is assumed to be given.

Optimal *inverse-Ackermann* time (i.e., $O(\alpha(n))$ time , where α is the inverse-Ackermann extremely slow growing function) parallel algorithms actually use the surplus-log approach in a methodological way, overviewed below. Benefiting from a construction on unbounded fan-in circuits in [CFL83], the inverse-Ackermann paradigm [BV89] works by designing a series of algorithms; the first in the series should run in $O(1)$ time using $n\log n$ processors; then, in a certain way, slight increase in time implies significant decrease in the number of processors. The $\alpha(n)$'th algorithm in the series runs in $O(\alpha(n))$ time using $n\alpha(n)$ processors, and finally an optimal algorithm that uses $(n/\alpha(n))$ processors and $O(\alpha(n))$ time is derived. See Figure 2.3 for the sequel. The most basic problem that was solved using the inverse-Ackermann paradigm is for the *nearest-one* problem (see also [Rag90], who calls it the *chaining problem*): given an array of bits $(a_1,...,a_n)$, find for each $1 \le i \le n$, the two nearest j and l such that $a_j = a_l = 1$ (that is, find the smallest $j > i$ such that $a_j = 1$ and the largest $j < i$ such that $a_j = 1$). Inverse-Ackermann time for chaining is best possible in an "oblivious" model of parallel computation, even with n processors [Cha90]. The nearest-one algorithm has been used to reduce a general version of the merging problem to the problem of finding all nearest neighbors (ANN) of vertices in a convex polygon; a consequence is that a doubly-logarithmic time lower-bound for merging extends to the ANN problem, resulting in a simpler proof than in [SV90]. Wherever reducibilities are more efficient than lower bounds they become promising tools for the theory of lower bounds. Before proceeding we make two comments: (1) in all problems below the input is assumed to come from the domain of integers $[1...n]$; (2) we avoid redefining problems that were defined earlier. Problems for which optimal inverse-Ackerman algorithms were given include: (1) the *all nearest smaller value* (ANSV) problem; this leads to: (2) *parentheses matching*: given the level of nesting for each parenthesis in a legal sequence of parentheses, find for each parenthesis its match; the last two results are in [BV91a]; (3) the *nearest-one complementation* problem: given is an array of bits $(a_1, ..., a_n)$ and suppose for each $a_i = 1$, the two nearest indices j and l, such that $a_j = a_l = 1$, are known; find for each $a_i = 0$, $1 \leq i \leq n$, the two nearest j and l such that $a_j = a_l = 1$ (that is, find the smallest $j > i$ such that $a_j = 1$ and the largest $j < i$ such that $a_j = 1$); this leads to: (4) *merging* two sorted lists; the nearest-one complementation and the merging algorithms are for a CREW PRAM; the last two results are in [BV90].

The following two problems involve preprocessing and query retrieval: (1) preprocessing for *range-maxima* queries; the preprocessing is done by an optimal inverse-Ackermann parallel algorithm and processing a query takes inverse-Ackermann time; the series of algorithms obtained as part of the inverse-Ackermann paradigm also implies trading-off slightly slower, but still optimal, preprocessing for faster (e.g., constant-time) query retrieval; (2) preprocess a rooted tree so that a query requesting the *lowest-common-ancestor* (LCA) of any pair of vertices can be quickly processed; results are similar to the ones for range-maxima, assuming that the Euler tour of the tree is given; the algorithm is new, and interestingly also simpler than previous LCA algorithms [HT84] and [SV88].

6 Randomized Fast Algorithms

Randomization has shown to be very useful for both the simulation of PRAM-like shared memory models of parallel computation by other models of parallel machines (e.g., in [KU86], [KRS88], [MV84], [Ran87], [KPS90] and [MSP90]), and for the design of parallel algorithms (e.g., in [ABI86], [AM90], [Gaz86], [GM91], [KR87], [Lub86], [MR85], [MV90], [MV91], [RR89a], [RS89], [Rei81], [Sch80a], [Sen89] and [Vis84b]).

All "target algorithms" in this section are randomized, and their running time is at the doubly-logarithmic level, or faster. By the *doubly-logarithmic level*, we mean $O(f(n) \log\log n)$ where the function $f(n)$ is $o(\log\log n)$.

Several constant-time optimal randomized algorithms were given: (1) for finding the maximum among n elements [Rei81]; and its generalization (2) for linear programming in fixed dimension [AM90]; (3) for finding approximate median [Sen89]; (4) for the nearest one problem (as in [BV89] and [Rag90]), under the assumption that there is some upper bound on the number of ones, [Ram90].

Several parallel deterministic and randomized algorithms, that run in time proportional to $\log n/\log\log n$ ("logarithmic level") or slower, were given for sorting [AKS83], [Bat68], [BN89], [Col88], [Hir78] [Pre78], [RV87], and [SV81], and integer sorting [BDH$^+$89], [Hag87], [Hag91a], [MV90], [MV91], [RR89a], [Ram90] and [Ram91]. The lower-bound in [BH87] implies that faster algorithms are possible only by relaxing the definition of the problem: (1) [MS91] gave a doubly-logarithmic level result, assuming the input comes from a certain random source; the output is given in a "padded" representation; (2) [Hag91a] allows general integer inputs from the range $[1..n]$; the output is given in a linked list which is sorted in a non-decreasing order.

We proceed to Figure 3, the main structure in this section. At the most basic level,

Figure 3 has the *d-polynomial approximate compaction (d-PAC)* problem (for $d = 3$ or 4). Given is an array of n cells; we know that only m of them contain one item each, and the rest are empty; the problem is to insert all items into an array of size m^d. A constant-time algorithm using n processors has been given for this fundamental problem in [Rag90]. The *linear approximate compaction (LAC)* problem is harder: using the same input, the items are to be inserted into an array whose size is linear in m, say $4m$. An optimal randomized algorithm for LAC, whose running time is at the log-star level was given [MV91]. Unless mentioned otherwise, all log-star level results are from this paper. The algorithm uses the d-PAC algorithm. A somewhat similar use of the d-PAC algorithm for a different problem can be found in [Ram90]. Using the log-star-time deterministic algorithms for the nearest-one and prefix-maxima problems, mentioned earlier, as well as the LAC algorithm, an optimal log-star level for *generating a random permutation* was given. Other methods for this problem are at the logarithmic level [MR85] and [RR89a]; [Hag91b] gives a doubly-logarithmic level algorithm that produces random permutations in a non-standard representation. The LAC algorithm required a new algorithmic paradigm. This paradigm has been extended, within the same performance bounds, to cope with the more general and well-investigated problem of *hashing*: given a set of n input elements, build a linear size table that supports membership queries in constant-time. Logarithmic level hashing [MV90], and doubly-logarithmic level hashing [GM91] preceded this result. Some log-star level ideas for a non-standard algorithmic model, where cost of counting, as well as assignment of processors to jobs, are ignored were given in [GMW90]. An $\Omega(\log^* n)$ time lower-bound using n processors is also given in [GMW90]; the lower bound is for a model of computation that admits the log-star level algorithm. We mention here only one application of hashing; see [MV90] for reference to several parallel algorithms with excessive space requirements that become space-efficient by using parallel hashing; the penalties are increase in time (as required by the hashing algorithm) and switching from a deterministic to a randomized algorithm.

Assignment of processors to jobs is a typical concern in parallel algorithms; for instance, one of the most powerful methodologies for designing parallel algorithms is to have a first design in terms of total work and time; extending this first design into a "full PRAM" design is guided by a theorem due to [Bre74]; the problem, however, is that the theorem holds only for a non-standard model of parallel computation, where assignment of processors to jobs can be done free of charge; the methodology was first used for the design of a PRAM algorithm in [SV82b], and is elucidated in [Vis90] and [JáJ91], who call it the *work-time* framework; typical applications of this methodology solve the processor assignment problem in an ad-hoc manner; however, sometimes proper processor assignment can be achieved using general methods for balancing loads among processors. Load balancing can be achieved by a simple application of a prefix-sums algorithm (e.g., [Vis84b]), with a logarithmic-level time overhead. A family of load balancing algorithms are treated in [Gil90], with a doubly-logarithmic multiplicative overhead; [MV91] treats a more specific family, with log-star level additive overhead, using the LAC algorithm. Load balancing and hashing methods, including the ones in [GM91], led to a doubly-logarithmic level "*dictionary*" extension of hashing, where, insertion and deletion queries are also supported [GMV90]; an algorithm in [DM89] solves the dictionary problem with running time of the form $O(n^\epsilon)$.

Routines for the prefix-sums problem play a major role in parallel algorithms, as indicated in Section 4. An additional perspective with respect to some problems in the previous paragraph is given by simply ordering them according to how well they "approximate" the prefix-sums problem, as follows: (1) the d-PAC problem is a first approximation; (2) the LAC problem; (3) the *load balancing* problem, which is defined as follows: given is an array of n cells $c_1, ..., c_n$; cell c_i contains t_i tasks, $1 \leq i \leq n$, where $\sum_{i=1}^{n} t_i \leq N$ (each cell i has the count t_i and a pointer to an array of size t_i; the array has a task at each of its entries); the problem is to redistribute the tasks among the cells such that each cell gets $O(N/n)$ tasks.

Acknowledgement. Helpful comments by O. Berkman, J. JáJá, S. Khuller, Y. Matias and R. Thurimella are gratefully acknowledged.

References

[ABI86] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7:567–583, 1986.

[ADKP87] K. Abrahamson, N. Dadoun, D. A. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. Technical Report 87-30, The University of British Columbia, 1987.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, 1974.

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

[Akl89] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Engelwood Cliffs, New Jersey, 1989.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. of the 15th Ann. ACM Symp. on Theory of Computing*, pages 1–9, 1983.

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

[AM90] N. Alon and N. Megiddo. Parallel linear programming almost surely in constant time. In *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 574–582, 1990.

[Ata90a] M.J. Atallah. A faster algorithm for a parallel algorithm for a matrix searching problem. In *Proc. 2nd SWAT*, volume LNCS 447, pages 192–200. Springer-Verlag, 1990.

[Ata90b] M.J. Atallah. Parallel techniques for computational geometry. Technical Report CS-1020, Purdue University, 1990.

[AV84] M.J. Atallah and U. Vishkin. Finding Euler tours in parallel. *J. Comp. Sys. Sci.*, 29,3:330–337, 1984.

[Bat68] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 32(1968).

[BBG+89] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly-parallelizable problems. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing*, pages 309–319, 1989.

[BDH+89] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. Technical Report TR 15/1989, Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, W. Germany, November 1989.

[BG88] D. Breslauer and Z. Galil. An optimal $O(\log \log n)$ parallel string matching algorithm. To appear in SIAM J. Comput., 1988.

[BG91] D. Breslauer and Z. Galil. A lower bound for parallel string matching. In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, 1991.

[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.

[BH87] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 83–93, 1987.

[BJK+90] O. Berkman, J. JáJá, S. Krishnamurthy, R. Thurimella, and U. Vishkin. Some triply-logarithmic parallel algorithms. In *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 871–881, 1990.

[BJK+91] O. Berkman, J. JáJá, S. Krishnamurthy, R. Thurimella, and U. Vishkin. Top-bottom routing is as easy as prefix minima. In preparation (a preliminary and partial version is part of Some Triply-logarithmic Parallel Algorithms, see above), 1991.

[BN89] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines. *SIAM J. Computing*, 18:216–228, 1989.

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

[BSV88] O. Berkman, B. Schieber, and U. Vishkin. Some doubly logarithmic parallel algorithms based on finding all nearest smaller values. Technical Report UMIACS-TR-88-79, Univ. of Maryland Inst. for Advanced Computer Studies, 1988.

[BSV91] O. Berkman, B. Schieber, and U. Vishkin. The parallel complexity of finding the convex hull of a monotone polygon. In preparation, 1991.

[BV89] O. Berkman and U. Vishkin. Recursive *-tree parallel data-structure. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 196–202, 1989.

[BV90] O. Berkman and U. Vishkin. On parallel integer merging. Technical Report UMIACS-TR-90-15, University of Maryland Inst. for Advanced Computer Studies, 1990.

[BV91a] O. Berkman and U. Vishkin. Almost fully-parallel paretheses matching. In preparation, 1991.

[BV91b] O. Berkman and U. Vishkin. Finding level-ancestors in trees. Technical Report UMIACS-TR-91-9, University of Maryland Institute for Advanced Computer Studies, 1991.

[CDR86] S.A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15:87–97, 1986.

[CFL83] A.K. Chandra, S. Fortune, and R.J. Lipton. Unbounded fan-in circuits and associative functions. In *Proc. of the 15th Ann. ACM Symp. on Theory of Computing*, pages 52–60, 1983.

[Cha90] S. Chaudhuri. Tight bounds for the chaining problem. preprint, December, 1990.

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

[Coo81] S.A. Cook. Towards a complexity theory of synchronous parallel computation. *Ensign. Math.*, 27:99–124, 1981.

[Coo85] S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[CT91] J. Cheriyan and R. Thurimella. Algorithms for parallel k-vertex connectivity and sparse certificates. In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, 1991.

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

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

[CV88] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–348, 1988.

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

[CZ90] R. Cole and O. Zajicek. An optimal parallel algorithm for building a data structure for planar point location. *J. Parallel and Distributed Computing*, 8:280–285, 1990.

[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.

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

[FRT89] D. Fussell, V.L. Ramachandran, and R. Thurimella. Finding triconnected components by local replacements. In *Proc. of 16th ICALP, Springer LNCS 372*, pages 379–393, 1989.

[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.

[Gal85] Z. Galil. Optimal parallel algorithms for string matching. *Information and Control*, 67:144–157, 1985.

[Gaz86] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 492–501, 1986.

[Gil90] J. Gil. Fast load balancing on PRAM. Preliminary report; see also: Lower Bounds and Algorithms for Hashing and Parallel Processing, Ph.D. Thesis, Hebrew University, Jerusalem, Israel, 1990.

[GM91] J. Gil and Y. Matias. Fast hashing on a PRAM. In *Proc. of the 2nd Second ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1991.

[GMV90] Y. Gil, Y. Matias, and U. Vishkin. A fast parallel dictionary. In preparation, 1990.

[GMW90] Y. Gil, F. Meyer auf der Heide, and A. Wigderson. Not all keys can be hashed in constant time. In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 244–253, 1990.

[Gol82] L.M. Goldschlager. A universal interconnection pattern for parallel computers. *J. Assoc. Comput. Mach.*, 29:1073–1086, 1982.

[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.

[GR86] A. Gibbons and W. Rytter. An optimal parallel algorithm for dynamic evaluation and its applications. In *Proceedings of the sixth Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 241*, pages 453–469. Springer-Verlag, 1986.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, 1988.

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

[Hag91a] T. Hagerup. Constant-time parallel integer sorting. In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, 1991.

[Hag91b] T. Hagerup. Fast parallel generation of random permutations. In *Proc. of 18th ICALP*, 1991.

[HCD87] T. Hagerup, M. Chrobak, and K. Diks. Parallel 5-coloring of planar graphs. In *Proc. of 14th ICALP*, pages 304–313, 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.

[Hir78] D. S. Hirschberg. Fast parallel sorting algorithms. *Comm. ACM*, 21:657–661, 1978.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, May 1984.

[JáJ91] J. JáJá. *Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1991.

[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.

[KPS90] Z.M. Kedem, K.V. Palem, and P.G. Spirakis. Efficient robust parallel computations. In *Proc. of the 22nd Ann. ACM Symp. on Theory of Computing*, pages 138–148, 1990.

[KR87] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, 31:249–260, 1987.

[KR88a] R.M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division (EECS) U. C. Berkeley, 1988. also, in Handbook of Theoretical Computer Science, North-Holland, to appear.

[KR88b] P. Klein and J.H. Reif. An efficient parallel algorithm for planarity. *J. Comp. Sys. Sci.*, 37, 1988.

[KRS88] C.P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. In *Proc. of 15th ICALP, Springer LNCS 317*, pages 333–346, 1988.

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

[KS89] S. Khuller and B. Schieber. Efficient parallel algorithms for testing connectivity and finding disjoint s-t paths in graphs. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 288–293, 1989.

[KU86] A. Karlin and E. Upfal. Parallel hashing – an efficient implementation of shared memory. In *Proc. of the 18th Ann. ACM Symp. on Theory of Computing*, pages 160–168, 1986.

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

[Lov85] L. Lovasz. Computing ears and branching in parallel. In *Proc. of the 26th IEEE Annual Symp. on Foundation of Computer Science*, pages 464–467, 1985.

[Lub86] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15:1036–1053, 1986.

[MR85] G.L. Miller and J.H. Reif. Parallel tree contraction and its application. In *Proc. of the 26th IEEE Annual Symp. on Foundation of Computer Science*, pages 478–489, 1985.

[MR86] G.L. Miller and V.L. Ramachandran. Efficient parallel ear decomposition and applications. unpublished manuscript, 1986.

[MR87] G.L. Miller and V.L. Ramachandran. A new graph triconnectivity algorithm and its parallization. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 335–344, 1987.

[MS91] P.D. MacKenzie and Q.F. Stout. Ultra-fast expected time parallel algorithms. In *Proc. of the 2nd Second ACM-SIAM Symposium on Discrete Algorithms*, pages 414–424, 1991.

[MSP90] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science*, pages 590–599, 1990.

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

[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.

[MV90] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. In *Proc. of 17th ICALP, Springer LNCS 443*, pages 729–743, 1990. Also, in UMIACS-TR-90-13, Inst. for Advanced Computer Studies, Univ. of Maryland, Aug. 1990 (revised), and J. Algorithms, to appear.

[MV91] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time - with applications to parallel hashing. In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, 1991.

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

[Pip79] N. Pippenger. On simultaneous resource bounds. In *Proc. of the 20th IEEE Annual Symp. on Foundation of Computer Science*, pages 307–311, 1979.

[Pre78] F. P. Preparata. New parallel sorting schemes. *IEEE trans. Computer*, C-27:669–673, 1978.

[Rag90] P. Ragde. The parallel simplicity of compaction and chaining. In *Proc. of 17th ICALP, Springer LNCS 443*, pages 744–751, 1990.

[Ram90] R. Raman. The power of collision: Randomized parallel algorithms for chaining and integer sorting. Technical Report TR-336 (revised version, January 1991), Computer Science Dept., Univ. of Rochester, 1990.

[Ram91] R. Raman. Optimal sub-logarithmic time integer sorting on a CRCW PRAM (note). manuscript, 1991.

[Ran87] A.G. Ranade. How to emulate shared memory. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 185–194, 1987.

[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.

[Rei91] J.H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann, San Mateo, California, 1991.

[RR89a] S. Rajasekaran and J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18:594–607, 1989.

[RR89b] V.L. Ramachandran and J.H. Reif. An optimal parallel algorithm for graph planarity. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 282–287, 1989.

[RS89] J.H. Reif and S. Sen. Polling: a new random sampling technique for computational geometry. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing*, pages 394–404, 1989.

[RV87] J.H. Reif and L.G. Valiant. A logarithmic time sort for linear size networks. *J. Assoc. Comput. Mach.*, 34:60–76, 1987.

[RV88] V.L. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic parallel time. In *Proc. of AWOC 88, Lecture Notes in Computer Science* No. 319, pages 33–42. Springer-Verlag, 1988.

[Sch80a] J. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4):701–717, 1980.

[Sch80b] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, 1980.

[Sch87] B. Schieber. *Design and analysis of some parallel algorithms.* PhD thesis, Dept. of Computer Science, Tel Aviv Univ., 1987.

[Sen89] S. Sen. Finding an approximate-median with high-probability in constant time. Manuscript, 1989.

[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.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[SV90] B. Schieber and U. Vishkin. Finding all nearest neighbors for convex polygons in parallel: a new lower bounds technique and a matching algorithm. *Discrete Applied Math*, 29:97–111, 1990.

[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.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33,8:103–111, 1990.

[Vis83] U. Vishkin. Synchronous parallel computation - a survey. Technical Report TR 71, Dept. of Computer Science, Courant Institute, New York University, 1983.

[Vis84a] U. Vishkin. A parallel-design distributed-implementation (PDDI) general purpose computer. *Theoretical Computer Science*, 32:157–172, 1984.

[Vis84b] U. Vishkin. Randomized speed-ups in parallel computations. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 230–239, 1984.

[Vis85a] U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20:235–240, 1985.

[Vis85b] U. Vishkin. Optimal parallel pattern matching in strings. *Information and Computation*, 67,1-3:91–113, 1985.

[Vis90] U. Vishkin. A parallel blocking flow algorithm for acyclic networks. Technical Report UMIACS-TR-90-11, University of Maryland Inst. for Advanced Computer Studies, 1990.

[Vis91] U. Vishkin. Deterministic sampling - a new technique for fast pattern matching. *SIAM J. Comput.*, 20(1):22–40, February 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.