MAX-PLANCK-INSTITUT FÜR INFORMATIK

Fast Deterministic Processor Allocation

Torben Hagerup

MPI-I-92-149 November 1992

Im Stadtwald
W 6600 Saarbrücken
Germany

Fast Deterministic Processor Allocation

Torben Hagerup

MPI-I-92-149 November 1992

Fast Deterministic Processor Allocation*

Torben Hagerup†

Abstract

Interval allocation has been suggested as a possible formalization for the PRAM of the (vaguely defined) processor allocation problem, which is of fundamental importance in parallel computing. The interval allocation problem is, given n nonnegative integers $x_1, \ldots, x_n$, to allocate n nonoverlapping subarrays of sizes $x_1, \ldots, x_n$ from within a base array of $O(\sum_{j=1}^{n} x_j)$ cells. We show that interval allocation problems of size n can be solved in $O((\log\log n)^3)$ time with optimal speedup on a deterministic CRCW PRAM. In addition to a general solution to the processor allocation problem, this implies an improved deterministic algorithm for the problem of approximate summation. For both interval allocation and approximate summation, the fastest previous deterministic algorithms have running times of $\Theta(\log n/\log\log n)$. We also describe an application to the problem of computing the connected components of an undirected graph.

1 Introduction

The integer *prefix summation* problem, i.e., given n integers $x_1, \ldots, x_n$, compute $\sum_{j=1}^{i} x_j$, for $i = 1, \ldots, n$, is among the most fundamental problems in the field of parallel computing. To a large extent, this is due to its folklore application to the problem of *compaction*, i.e., given an array of n cells, k of which contain an object, place the k objects in (distinct cells of) an array of size k. The compaction problem, in turn, derives much of its importance from its close connection to processor allocation. E.g., if each object represents a constant-size task to be executed by some processor, the execution of the tasks can be carried out with optimal speedup (i.e., every processor does useful work all of the time) once the objects have been compacted, since then they can be distributed evenly among the available processors; without compaction, much processing power may be wasted because the processors cannot be allocated to the work at hand in an efficient manner.

It is known that the prefix sums of n integers of absolute size $n^{O(1)}$ can be computed in $O(\log n/\log\log n)$ time with optimal speedup, i.e., with a time-processor product of $O(n)$ [5]. Because of its fundamental nature, however, prefix summation is the bottleneck in algorithms for many other problems, and a faster prefix summation algorithm would be a great asset. Unfortunately, as shown by Beame and Hastad [3], any PRAM algorithm that solves the problem using $n^{O(1)}$ processors must have a running time of $\Omega(\log n/\log\log n)$, even if $x_j \in \{0, 1\}$, for $j = 1, \ldots, n$, so that the algorithm of [5] is as fast as possible. The lower bound was originally stated for deterministic algorithms, but can be shown to hold for the expected running time of randomized algorithms as well.

In an attempt to sidestep the lower bound of Beame and Hastad, various researchers considered the problem of *linear approximate compaction*, which is defined as that of (exact) compaction, except that the k objects are to be placed in an array of size $O(k)$, rather than k. Linear approximate compaction does not solve the prefix summation problem, but it serves almost as well in the major application to processor allocation. Using ideas developed previously by Raman [21], Matias and Vishkin [19] showed how to solve linear approximate compaction problems of size n in $O(\log^* n)$ expected time on a randomized n-processor CRCW PRAM (they actually required an upper bound m on k to be given as part of the input and compacted into $4m$ cells rather than $O(k)$ cells, but this is immaterial). Their result and later improvements and extensions of it led to surprisingly fast randomized algorithms for a number of fundamental problems [19, 10, 11, 2, 8, 9, 12, 13, 1, 16, 15].

As mentioned above, it is known that the introduction of randomization by itself is not sufficient to circumvent the $\Omega(\log n/\log\log n)$ lower bound of Beame

*Supported by the ESPRIT Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II).

†Departament de LSI, Universitat Politècnica de Catalunya, E–08028 Barcelona, Spain. Author's present address: Max-Planck-Institut für Informatik, Im Stadtwald, W–6600 Saarbrücken, Germany.

and Hastad. In other words, taking the step from exact to approximate compaction is essential. It was not known whether, once approximate compaction has been substituted for exact compaction, it is additionally necessary to resort to randomization (technically speaking, if we restrict attention to algorithms that are either uniform or reasonably efficient). We answer this question in the negative by showing that linear approximate compaction problems of size n can be solved deterministically in $O((\log\log n)^3)$ time with optimal speedup.

Instead of linear approximate compaction, we actually consider the more general *interval allocation* problem introduced by Hagerup [10], which is also known to have $O(\log^* n)$-time randomized solutions [8, 10]. The interval allocation problem of size n is, given n nonnegative integers $x_1, \ldots, x_n$, to allocate (i.e., compute the offsets of) n nonoverlapping subarrays of sizes $x_1, \ldots, x_n$ from within a base array of size $O(\sum_{j=1}^{n} x_j)$; linear approximate compaction can be seen to be precisely the special case in which $x_j \in \{0, 1\}$, for $j = 1, \ldots, n$. As argued in [13], interval allocation allows the solution of more general processor allocation problems than does linear approximate compaction; in addition, the algorithm of the present paper depends in an essential way on the greater generality of interval allocation: Even if the original problem is one of linear approximate compaction, its solution generates subproblems that are general interval allocation problems. For our present purpose, however, the expression $O(\sum_{j=1}^{n} x_j)$ for the size of the base array is too crude; we have to be more specific about the constant factor hiding in the "big oh". Hence for all $\lambda \geq 0$, define the interval allocation problem with *padding factor* λ to be as above, but with the base array allowed to be of size at most $(1+\lambda)\sum_{j=1}^{n} x_j$, rather than $O(\sum_{j=1}^{n} x_j)$. Our actual result is that for every constant l, interval allocation problems of size n and with padding factor $(\log\log n)^{-l}$ can be solved deterministically in $O((\log\log n)^3)$ time with optimal speedup.

In fact, although this is of secondary interest, we obtain a more general result by exploiting a tradeoff between running time and padding factor ([16] paraphrased a similar situation "waste makes haste"): For all $t \geq (\log\log n)^3$, we can solve interval allocation problems of size n and with padding factor $2^{-t\log\log\log n/(\log\log n)^3}$ in $O(t)$ time with optimal speedup.

While not using any specific results from that paper, the present paper owes much of its spirit to [20].

2 Overview of the algorithm

At the topmost level our algorithm uses what has been called the "divide-and-crush paradigm", whereby "crush" refers to the activity of solving small subproblems fast using a number of processors that far exceeds the size of these subproblems. An application of the divide-and-crush paradigm requires, loosely speaking, that the problem under consideration can be broken up into "local" subproblems, solutions to which can be combined to form a "global" problem of the same kind, whose solution is a solution to the original problem. The problem of computing the maximum among a number of input elements, for instance, has this property: If the input elements are partitioned into groups and the maximum within each group is computed, the overall maximum can be obtained as the maximum of the group maxima. Another requirement is that there is indeed a suitable "crushing" subroutine. Continuing the example, the maximum of m elements is easily computed in constant time with m^2 processors by carrying out all m^2 pairwise comparisons between input elements and declaring to be the maximum the unique element that never loses a comparison. Whenever, as in the example, the "crushing" subroutine uses constant time and $m^{O(1)}$ processors on inputs of size m, the divide-and-crush paradigm yields an algorithm that uses $O(\log\log n)$ time and n processors on inputs of size n. If the problem additionally admits a linear-time sequential solution, the running time can be kept at $O(\log\log n)$, while the number of processors is reduced to (the optimal) $O(n/\log\log n)$. For the algorithm studied in this paper, the "crushing" subroutine needs $\Theta(\log\log m)$ time and essentially $m^{\log\log m}$ processors on inputs of size m, which leads to an optimal algorithm with a running time of $O((\log\log n)^3)$. To our knowledge, this is the first application of the divide-and-crush paradigm with a "crushing" subroutine of superpolynomial complexity.

When nothing else is stated, our model of computation is the standard ARBITRARY CRCW PRAM. Theorems 4.1, 5.1 and 6.1 actually hold for any CRCW

PRAM variant capable of computing the OR of n bits using constant time, n processors and $O(n)$ space, and Theorem 6.2 holds for any variant that in addition has the so-called *self-simulating* property, i.e., a machine with p processors can simulate each step of a machine with n processors in $O(\lceil n/p \rceil)$ time, for arbitrary $n, p \in \mathbb{N}$.

Throughout the paper, $\log x$ denotes $\max\{\log_2 x, 1\}$, for arbitrary $x > 0$.

3 Nonoptimal linear approximate compaction

In this section we describe the main ideas of our "crushing" subroutine for the special case of linear approximate compaction. Our solution makes crucial use of techniques developed in the theory of hashing, and the concept of *collisions* under a hash function is central. Given a function h and a finite subset X of the domain of h, define the *collision number* of h on X as the number of elements in X whose image under h is not unique, i.e., as the quantity $|\{x \in X : \text{For some } y \in X \backslash \{x\},\ h(x) = h(y)\}|$. The following lemma was essentially proved by Fredman *et al.* [7]. We use the convenient formulation given in [14].

LEMMA 3.1. *Let $m, k, s \in \mathbb{N}$ and let $p \geq m$ be a prime. Then for every subset X of $\{0, \ldots, m-1\}$ with $|X| \leq k$, there exists an integer a with $1 \leq a < p$ such that the collision number on X of the function $x \mapsto (ax \bmod p) \bmod s$ is less than $2k^2/s$.* □

In our applications of Lemma 3.1, the exact value of p will be inessential: Any prime $p \geq m$ with $p = O(m)$ will do; since the set $\{m, \ldots, 2m\}$ contains at least one prime, such a prime can be found in constant time with m^2 processors, which will always be available. We can interpret Lemma 3.1 as a statement about approximate compaction in the following way: If we are given an array $A[0..m-1]$ of size m containing at most k objects, let X be the set of indices of those cells in A that contain an object. In order to place the objects in a destination array $B[0..s-1]$ of size s, we might attempt to place the object previously in cell number x in A in cell number $(ax \bmod p) \bmod s$ in B, for some value of a with $1 \leq a < p$. The lemma states that there is a value of the multiplier a for which fewer than $2k^2/s$ objects claim a cell in the destination array that is also claimed by some other object; let us call such a multiplier *magical*. In other words, if we attempt to move the objects to the destination array according to the function associated with a magical multiplier, there will be fewer than $2k^2/s$ left-over objects that found no free cell. We may subsequently attempt to handle (most of) these in the same way.

For a numerical example, suppose that we first use the simple compaction scheme outlined above with $s = s_1 = 8k$. Assume for a moment that we happen to know a magical multiplier a. As argued above, we can then place all except fewer than $2k^2/s_1 = k/4$ objects in the destination array of size s_1. In a next stage we may use a new destination array of size $s_2 = 4k$, in which case the number of active objects decreases below $2(k/4)^2/(4k) = k/32$, where an *active* object is one that has not yet been placed. Taking $s_3 = 2k$ leaves fewer than $2(k/32)^2/(2k) = k/32^2$ active objects after the third stage. It seems justified to expect that if we continue in this way with $s_{i+1} = s_i/2$, for $i = 1, 2, \ldots$, then the number of active objects will be bounded by k/r, where r is approximately squared in each stage. This intuition is indeed correct, insofar as the process will come to an end after at most $\lceil \log \log k \rceil$ stages, with all objects placed — it is straightforward to verify by induction that the number of active objects after i stages is less than $k \cdot 2^{-(2^i+i-1)}$, for $i = 1, 2, \ldots$ The total number of destination cells used is $s_1 + s_2 + \cdots = O(k)$. In other words, given an oracle that promptly delivers magical multipliers whenever they are needed, we can solve linear approximate compaction problems of size n in $O(\log \log k) = O(\log \log n)$ time using n processors (if the prime p is given for free).

There is little hope of constructing such an oracle. What we can do, however, is to try out all possible multipliers in parallel. Note that we cannot tell until the very end whether a particular multiplier was good, so that we are forced to combine all possible multipliers in Stage 1 with all possible multipliers in Stage 2, etc. This structures the computation as a $(p-1)$-ary tree of depth at most $\lceil \log \log k \rceil$, all branches of which are explored in parallel, starting from the root. Each stage needs $p-1$ times more processors than the previous one, so that the overall processor requirements are $\leq n \cdot (p-1)^{\lceil \log \log k \rceil} \leq p^{\log \log n + 2}$. In fact, since $p = O(n)$ and since it is easy to see that we can reduce the number of stages by any desired additive constant by

multiplying s_1 by a suitable constant, we have proved:

LEMMA 3.2. *For every constant c, linear approximate compaction problems of size n can be solved in $O(\log\log n)$ time using at most $\lceil n^{\log\log n - c}\rceil$ processors.* □

The above proof implicitly assumes k to be known. If this is not the case, we can simply execute the algorithm in parallel with $k = 0, 1, \ldots, n$ and subsequently choose the smallest destination array into which the objects were successfully compacted. The extra cost in terms of processors can be absorbed into the constant c.

4 Optimal but wasteful interval allocation

In this section we proceed from linear approximate compaction to interval allocation using a simple reduction introduced in [10], restated in [8] and discussed more fully in [13]. We also embed the "crushing" subroutine in an optimal algorithm according to the divide-and-crush paradigm.

Assume that we are given an input $x_1, \ldots, x_n$ to the interval allocation problem and that we want to solve the problem with constant padding factor (i.e., according to the original definition of interval allocation). Each input number x_j can be viewed as a "request" for a block of x_j contiguous cells. The central observation is that we can pretend that there are only $O(\log n)$ different request sizes. Indeed, if the maximum request is M, any nonzero request smaller than M/n can be replaced by $\lceil M/n\rceil$ without increasing the sum of all requests by more than a constant factor; recall from Section 2 that M can be computed in $O(\log\log n)$ time. Similarly, each nonzero request can be rounded to the nearest larger power of 2, which leaves $O(\log n)$ distinct request sizes, as desired. Henceforth, a "color class" will be the set of requests of one particular common size. At a cost of an extra factor of $O(\log n)$ in the number of processors, we can now apply the linear approximate compaction algorithm of the previous section separately to each color class. What remains is to allocate space for the base arrays of the $O(\log n)$ color classes (i.e., to solve an interval allocation problem of size $O(\log n)$); this can be done in $O(\log\log n)$ time using the standard prefix summation algorithm.

THEOREM 4.1. *For every constant c, interval allocation problems of size n (with unspecified constant padding factor) can be solved in $O(\log\log n)$ time using at most $\lceil n^{\log\log n - c}\rceil$ processors.* □

We now apply the divide-and-crush principle. Given an input of size n to the interval allocation problem, we can begin spending $\Theta(\log\log n)$ time computing prefix sums within groups of input numbers of size $\Theta((\log n)^2)$ using the standard parallel algorithm, thereby in effect reducing the problem size by $\Theta((\log n)^2)$. Since we will use $\Theta(n/(\log\log n)^3)$ processors (see below), this gives us an initial processor advantage of $\Omega(\log n)$, where the *processor advantage* is defined as the number of processors divided by the problem size. We now execute a number of rounds. Assume that at some point we have achieved a processor advantage of $v = \Omega(\log n)$. The next round then divides the problem at hand into the smallest possible number of subproblems of size at most $m = \lfloor v^{1/\lceil\log\log v\rceil}\rfloor$ each, solves these in $O(\log\log m) = O(\log\log n)$ time using the algorithm of Theorem 4.1 (note that $m^{\log\log m} \le v$) and combines the solutions of the subproblems into a global problem by letting the size of the base array of each solution become a request in the global problem. We leave to the reader the easy details of how to keep track of the subarrays allocated to the original requests. As for the running time, however, the important fact is that the global problem, if it is of size ≥ 2, will be smaller than the (global) problem of the previous round by a factor of at least $m/2$. For sufficiently large values of n, we hence go from a processor advantage of v to a processor advantage of at least $v^{1+1/(2\log\log v)} \ge v^{1+1/(2\log\log n)}$. But then, as long as the problem size is not reduced to 1, $2\log\log n$ rounds at least square the processor advantage. Since the processor advantage cannot exceed n, $O((\log\log n)^2)$ rounds suffice to solve the original problem.

The above gives us an interval allocation algorithm with optimal speedup and a running time of $O((\log\log n)^3)$. However, the algorithm suffers from an important drawback: Each application of Theorem 4.1 introduces a constant padding factor, i.e., the $\Theta((\log\log n)^2)$ successive rounds accumulate a total padding factor of $2^{\Theta((\log\log n)^2)}$. In the next section we show how to reduce the padding factor from this value to $o(1)$. This is the most complicated part of our argument; since all important applications of interval alloca-

tion require the padding factor to be at most constant, however, it is well worth the effort.

5 Reducing the padding factor

Assume that we succeed in modifying the algorithm of Lemma 3.2 so that it compacts with padding factor $O((\log\log n)^{-l-3})$, where l is a given positive integer with $l = O(\log n/(\log\log n)^3)$. The better padding factor then carries over to Theorem 4.1, except that the number of color classes must be increased to $\Theta(\log n(\log\log n)^{l+3})$ (note that $(1 + (\log\log n)^{-l-3})^{(\log\log n)^{l+3}} = \Theta(1)$, so that each original color class splits into $\Theta((\log\log n)^{l+3})$ new color classes). This causes a negligible increase in the number of processors needed and an increase to $\Theta(\log\log n + l\log\log\log n)$ in the running time; the latter will turn out to be negligible as well. Applying the divide-and-crush principle as in the previous section gives an algorithm that uses as many processors and as many rounds as before, but with padding factor

$$(1 + O((\log\log n)^{-l-3}))^{O((\log\log n)^2)} - 1 = e^{O((\log\log n)^{-l-1})} - 1,$$

which for sufficiently large values of n is bounded by $(\log\log n)^{-l}$.

Our goal therefore is, for given $l \in \mathbb{N}$, to modify Lemma 3.2 to obtain a padding factor of $O(\lambda)$, where $\lambda = (\log\log n)^{-l-3}$. We meet this goal by introducing a preprocessing phase that places all except at most λk objects in an array of size k, after which the remaining objects can be dealt with using (the original) Lemma 3.2. Our approach is based on the following lemma, of the same flavor as Lemma 3.1:

LEMMA 5.1. *Let $m, s \in \mathbb{N}$ and let $p \geq m$ be a prime. For all $a \in \{1,\ldots,p-1\}$, $b \in \{0,\ldots,s-1\}$ and $x \in \{0,\ldots,m-1\}$, let $h_{a,b}(x) = ((ax \bmod p) + b) \bmod s$. Then for all sets $X \subseteq \{0,\ldots,m-1\}$ and $F \subseteq \{0,\ldots,s-1\}$ with $|X| = |F| \leq s/4$, there exist integers $a \in \{1,\ldots,p-1\}$ and $b \in \{0,\ldots,s-1\}$ such that*

$$|\{x \in X \ : \ h_{a,b}(x) \in F, \text{ and for all } y \in X\backslash\{x\}, h_{a,b}(x) \neq h_{a,b}(y)\}| \geq \frac{|X|^2}{2s}.$$

Proof. Choose $a \in \{1,\ldots,p-1\}$ such that the collision number of $h_{a,0}$ on X is less than $\frac{2|X|^2}{s} \leq \frac{|X|}{2}$, possible according to Lemma 3.1, and let X' be the set of elements in X that do not collide under $h_{a,0}$, i.e., $X' = \{x \in X : \text{For all } y \in X\backslash\{x\},\ h_{a,0}(x) \neq h_{a,0}(y)\}$. By the choice of a, $|X'| \geq \frac{|X|}{2}$. Since the elements in X' do not collide under $h_{a,b}$ for any b, the task now is simply to show that $|h_{a,b}(X') \cap F| \geq \frac{|X|^2}{2s}$ for some $b \in \{0,\ldots,s-1\}$. But $\sum_{b=0}^{s-1} |h_{a,b}(X') \cap F| = |X'||F| \geq \frac{|X|^2}{2}$, since each element of F is "hit" exactly once by each element of X' as b varies from 0 to $s-1$. The claim now follows, since at least one of the s terms of the sum $\sum_{b=0}^{s-1} |h_{a,b}(X') \cap F|$ must be at least $\frac{1}{s} \cdot \frac{|X|^2}{2}$. □

The introduction of the set F allows us to declare certain values in $\{0,\ldots,s-1\}$ (namely those not in F) "invalid" in the sense that even if an object is the only one mapped to such a value, it cannot be placed in the corresponding cell. We use this in two ways. Firstly, although our intention is to compact into k cells, we can take s to be considerably larger than k, as required by Lemma 3.1, simply by ensuring that $F \subseteq \{0,\ldots,k-1\}$. Secondly, we can reuse the same k cells over several stages; F will be the subset of $\{0,\ldots,k-1\}$ of indices of those cells that were not occupied in earlier stages. If we start out with k active objects, the number of active objects will then clearly equal $|F|$ after any number of stages, as necessary for a repeated application of Lemma 5.1.

In order to compact most of k objects into k cells, we hence take $s = 4k$ and execute a number of stages, in each of which we try in parallel for all pairs $(a,b) \in \{1,\ldots,p-1\} \times \{0,\ldots,s-1\}$ to map the active objects according to $h_{a,b}$, an object being placed if it hits one of the k cells, if this cell was not occupied in a previous stage and if it is not claimed by any other object in the present stage. Lemma 5.1 states that there exists a "magical" pair (a,b) that lets us go from $|X|$ active objects to at most $|X| - \frac{|X|^2}{2s}$ active objects. The following lemma quantifies the efficiency of this procedure.

LEMMA 5.2. *Let q_0 and s be positive integers with $q_0 \leq s$ and suppose that $\{q_j\}_{j=1}^{\infty}$ is a series with $q_{j+1} \leq q_j - \frac{q_j^2}{2s}$, for $j = 0, 1, \ldots$ Then for all $r \in \mathbb{N}$, $q_{16r} \leq \frac{s}{r}$.*

Proof. Suppose that $q_i \leq \frac{s}{g}$, for some integers $i \geq 0$ and $g \geq 1$. Then $q_{i+8g} \leq \frac{s}{2g}$. For if this were not the case, we would have $\frac{s}{2g} < q_j \leq \frac{s}{g}$ and hence $q_{j+1} \leq q_j - \frac{q_j^2}{2s} < q_j - q_j \cdot \frac{s}{4gs} \leq q_j - \frac{q_i}{8g}$ for all integers

j with $i \leq j < i + 8g$, a contradiction. Using this for $g = 1, 2, 4, \ldots, 2^{\lfloor \log r \rfloor}$, we find that $q_i \leq \frac{s}{r}$, where $i = 8 \cdot (1 + 2 + 4 + \ldots + 2^{\lfloor \log r \rfloor}) \leq 16r$. □

Assume that we are solving a (sub-)problem of size $m \leq n$. We can conclude from Lemma 5.2 that $O(\log \log m)$ stages suffice to reduce the number of active objects by a factor of $\Theta(\log \log m)$. At this point we use Lemma 3.2 to compact the remaining free cells into an array of size $f = O(\lceil k / \log \log m \rceil)$. This allows us to prune away all branches, except one, from the computation tree (recall that we try out all sequences of pairs (a, b) in parallel), essential in order to keep the processor count at $m^{O(\log \log m)}$: We simply pick a branch for which the compaction into f cells succeeds. The compaction also allows us to redefine s as $4f$, after which we can proceed as before, i.e., another $O(\log \log m)$ stages reduce the number of active objects by another factor of $\Theta(\log \log m)$, etc. After altogether $O(\log \log m \log(1/\lambda) / \log \log \log m) = O(\log \log n \log(1/\lambda) / \log \log \log n) = O(l \log \log n)$ stages we are left with at most λk active objects, as desired. Using this with $l = \lceil t / (\log \log n)^3 \rceil$, we obtain:

THEOREM 5.1. *For all given $t \geq (\log \log n)^3$, interval allocation problems of size n and with padding factor $2^{-t \log \log \log n / (\log \log n)^3}$ can be solved in $O(t)$ time using $\lceil n/t \rceil$ processors and $O(n)$ space.* □

6 Applications

The first result below follows immediately from Theorem 5.1 and was used implicitly in its proof, but still deserves to be formulated explicitly.

The problem of *approximate summation* with *relative error bound* $\lambda \geq 0$ is, given n nonnegative integers $x_1, \ldots, x_n$, to compute an estimate $\hat{s}$ of the number $s = \sum_{j=1}^{n} x_j$ such that $s \leq \hat{s} \leq (1 + \lambda)s$.

THEOREM 6.1. *For all given $t \geq (\log \log n)^3$, approximate summation problems of size n and with relative error bound $2^{-t \log \log \log n / (\log \log n)^3}$ can be solved in $O(t)$ time using $\lceil n/t \rceil$ processors and $O(n)$ space.* □

Our second result deals with the processor allocation problem, which we must therefore formalize; see [13] for a fuller discussion. Assume that a parallel algorithm operates with a dynamically changing collection of *virtual processors*, and that the algorithm is charged one operation for each virtual processor for each time step in which the virtual processor is in existence (i.e., the algorithm pays only for resources that it actually uses). Suppose further that the mechanism available to the algorithm for manipulating the collection of virtual processors is an instruction *Allocate*$(x_1, \ldots, x_r)$, executed in unison by all currently existing virtual processors, each of which contributes a single nonnegative integer argument x to the instruction. $x = 0$ means that the processor should be removed (presumably because it has completed its task), $x = 1$ means that the processor is to be left alone, and $x \geq 2$ is a request that $x - 1$ new virtual processors be allocated to the virtual processor that contributed the argument x. The algorithm is charged one time unit for each execution of an *Allocate* instruction. An algorithm with these properties is called *loosely specified* in [8] and *standard* in [13]; as concerns aspects of processor allocation, practically all published parallel algorithms can be formulated within this framework. Observe that the efficient implementation of a standard algorithm on a PRAM is far from obvious, since the PRAM lacks machine-level facilities for dynamic task management. Loosely speaking, the processor allocation problem is precisely to provide such an implementation. More precisely, we want to execute the standard algorithm on a PRAM in such a way that the time and the number of operations consumed by the PRAM are as close as possible to the time and the number of operations charged to the standard algorithm, respectively, which means that the overhead due to processor allocation should be low. The proof of Theorem 6.2 below is a slight modification of the proof of Theorem 5.2 in [13], which is a counterpart of Theorem 6.2 for randomized standard algorithms.

THEOREM 6.2. *For all given $t, t', n \in \mathbb{N}$, every deterministic standard algorithm that uses at most t time steps, t' calls of Allocate and n operations can be executed on a deterministic PRAM using $O(t + t'(\log \log n)^3)$ time and $O(n)$ operations.* □

Theorem 6.2 can possibly be used to improve the best known deterministic algorithms for a number of problems in computational geometry — see [9] for a list of candidate problems. We will not pursue this line of investigation, but instead finally consider the problem of computing the connected components of general undirected graphs on a deterministic CRCW

PRAM. The best published algorithm for this problem, due to Cole and Vishkin [6], uses $O(\log n)$ time and $O((n+m)\alpha(m,n)/\log n)$ processors on input graphs with n vertices and m edges, where α is a slowly-growing "inverse Ackermann" function — see [6] for the exact definition of α. The algorithm derives part of its speed from a deterministic load balancing scheme based on the use of expander graphs. As an application of our results on processor allocation and approximate summation, we can give a version of the algorithm of Cole and Vishkin that avoids the large constant factors associated with the use of expander graphs; this serves as an example of how to apply our results in a "real-life" situation. We are also able to simplify the algorithm to some extent. The necessary modifications are small; for the reader's convenience, however, we describe them in the context of the "smallest enclosing black box", the "Main Connectivity Algorithm" of [6]. The reader is alerted to the fact that another and quite different algorithm that can also take the place of the "Main Connectivity Algorithm" of [6] was developed in independent work by Iwama and Kambayashi [17]. Their algorithm avoids not only the use of expander graphs, but of nontrivial load balancing altogether, and appears to be superior to ours in practical terms.

We consider each (undirected) edge $\{u,v\}$ to be composed of two antiparallel directed *darts* (u,v) and (v,u). We assume that an input graph $G=(V,E)$ is presented as an array of size $|V|$ specifying the (integer) names representing the vertices in V, together with an array D of size $2|E|$ listing the $2|E|$ darts of G, each of which is represented by (the names of) its head and tail. The algorithm of [6] in addition requires the darts in D to be semisorted by their tails, the meaning of which is that the darts out of any given vertex occupy a contiguous part of D. Since a standard adjacency list representation can be converted to this format using $O(\log n)$ time and $O(n+m)$ operations by means of (essentially) list ranking, the theorem below need not distinguish between the two representations.

THEOREM 6.3. [6] *The connected components of an undirected graph with n vertices and m edges represented via adjacency lists can be computed in $O(\log n)$ time on an* ARBITRARY *CRCW PRAM with $O((n+m)\alpha(m,n)/\log n)$ processors.*

Proof. Let the input graph be $G_0=(V_0,E_0)$ and take $n=|V_0|$ and $m=|E_0|$. We can assume without loss of generality that $n \geq 2^{16}$ and, as demonstrated by Cole and Vishkin (the "Reduction Procedure"), that $m \geq n\log n$. We describe an algorithm for computing the connected components of G_0 in $O(\log n)$ time using $p=\Theta(m/\log n)$ processors; the reader is referred to [6] for the implementation details.

The algorithm successively constructs undirected graphs $G_i=(V_i,E_i)$, for $i=1,2,\ldots$, which may contain multiple edges and loops. For $i=1,2,\ldots$, G_i is constructed from G_{i-1} by computing a partition $\mathcal{U}$ of V_{i-1}, each of whose sets spans a connected subgraph of G_{i-1}, and then contracting the vertices of each set $U \in \mathcal{U}$ into a single new vertex said to *contain* the vertices in U. Each edge $\{u,v\}$ in E_{i-1} is replaced by the edge $\{u',v'\}$ in E_i, where u' and v' are the vertices in V_i containing u and v, respectively, except that the algorithm discards a number of loops in the construction of G_i from G_{i-1}. Extend the relation of containment to its reflexive and transitive closure, such that each vertex in G_i contains a subset of V_0, for $i=0,1,\ldots$, and extend the relation further to subgraphs of G_i in the obvious way (a vertex in V_0 is contained in a subgraph of G_i if and only if it is contained in one of its vertices).

Let $d_0=\lfloor\sqrt{m/n}\rfloor \geq 4$ and $d_i=\lfloor d_{i-1}^{3/2}\rfloor$, for $i=1,2,\ldots$, and assign to each vertex in G_0 a *weight* of d_0^2. For $i=0,1,\ldots$, define the weight of a vertex in G_i or of a subgraph of G_i as the total weight of the vertices in G_0 that it contains. Obviously the weight of each of $G_0,G_1,\ldots$ is bounded by m. Also define a vertex in G_i or a connected subgraph of G_i to be *complete* if and only if the vertices in G_0 that it contains span a full connected component of G_0, and *incomplete* otherwise.

For each $u \in V_0$, the vertex in the most recently constructed graph G_i that contains u will be recorded in a variable $T[u]$. The goal therefore is to arrive at a graph G_i without incomplete vertices, since at this point the array T furnishes a solution to the original connected-components problem. The algorithm will ensure that the weight of each incomplete vertex in G_i is at least d_i^2, for $i=0,1,\ldots$ (the *weight condition* for G_i); in particular, the number of incomplete vertices in G_i is bounded by m/d_i^2. Since $\min\{i \in I\!N : d_i^2 > m\} = O(\log\log n)$, the algorithm terminates after constructing G_i for some i with $i=O(\log\log n)$.

What remains is to describe the construction of G_i from G_{i-1}, for $i = 1, 2, \ldots$, which will be called *Stage i*. We first describe an idealized algorithm that ignores a certain complication, and afterwards describe how to deal with the complication.

We will consider each dart (u, v) to "belong to" its tail u. In the idealized description, each stage begins with the darts of each vertex divided into *blocks* of exactly B darts each, where $B = \Theta(\log n/(\log\log n)^2)$, except that each vertex may possess a single *incomplete block* containing fewer than B darts. The darts of each block are stored in a linked list, accessible via a pointer in a *block header*. The block headers of all vertices are stored together in an array of size $O(m/B)$, with the blocks of each vertex occupying a contiguous part of the array. Note that as far as the representation is concerned, a dart still records the endpoints in V_0 of the corresponding dart in G_0; it would be too expensive to repeatedly update the representations of all darts. A dart with recorded tail u and head v therefore actually is a dart from $T[u]$ to $T[v]$.

Define a *spanning forest* of an undirected graph $G = (V, E)$ as any acyclic subgraph of G on the full vertex set V. Each connected component of a spanning forest will be called a *tree* in the forest. For $i = 1, 2, \ldots$, Stage i consists of the following three steps, which are practically the same as those of [6]:

1. Construct a subgraph G' of G_{i-1} on the vertex set V_{i-1} and with at most m/d_{i-1} edges, the weight of each of whose incomplete connected components is at least d_i^2;
2. Construct a spanning forest G'' of G', the weight of each of whose incomplete trees is at least d_i^2;
3. Construct G_i from G_{i-1} by contracting the vertices in each tree of G'' into a single vertex.

G' will be a simple graph, i.e., it includes neither multiple edges nor loops. For each $u \in V_{i-1}$, we will include in G' either d_{i-1} edges incident on u, or else one edge between u and each neighbor of u. Thus each incomplete connected component of G' will contain at least d_{i-1} vertices; since each vertex is of weight at least d_{i-1}^2, by the weight condition for G_{i-1}, the weight of each incomplete component will be at least $d_{i-1}^3 \geq d_i^2$, as required.

The basic operation in Step 1 is that of *processing* a dart. To process a dart $e = (u, v)$ (where u and v are the true endpoints of e, not those recorded in the representation), discard e if $u = v$. Otherwise check by inspecting a variable $ACTUAL[u, v]$ whether a dart from u to v was already found earlier in the present stage. If so, discard e. If not, attempt to record e as the value of $ACTUAL[u, v]$, thereby signaling the discovery of a dart from u to v. Since possibly other darts from u to v are discovered simultaneously, this may involve concurrent writing. The dart actually recorded in $ACTUAL[u, v]$ becomes a *selected dart*; the other darts from u to v are discarded. Note that since the table $ACTUAL$ is too large to be initialized, it should be handled with care: Whenever a dart e is recorded in some cell in $ACTUAL$, a back pointer to this cell should be stored with e and checked by every processor that finds e stored in some cell of $ACTUAL$, and the valid entries in $ACTUAL$ should be cleared between stages.

To process a block is to process the darts in the block one by one, which can be done in $O(B)$ time by a single processor. Step 1 processes blocks until for each vertex u either at least d_{i-1} darts out of u have been selected, or all of u's blocks have been processed. In more detail, call a vertex u *active* as long as the number of selected darts out of u is not known to be at least d_{i-1}, and call a block active if it belongs to an active vertex and has not yet been processed. Step 1 is divided into *iterations*. The first iteration processes all incomplete blocks; since there is at most one incomplete block for each vertex and $p = \Omega(n)$, this takes $O(B)$ time. In the beginning of every other iteration, we first use Theorem 6.1 to compute an estimate $\widehat{b}$, correct up to a constant factor, of the total number of remaining active blocks. Afterwards a vertex u with b_u active blocks requests $\min\{b_u, \lceil b_u p/\widehat{b}\rceil\}$ virtual processors, each of which will process one of u's active blocks. Note that the total number of virtual processors requested is $O(p)$; by Theorem 6.2, the time needed to allocate these processors is $O((\log\log n)^3) = O(B)$ per iteration, i.e., negligible. This processor allocation by means of Theorem 6.2, which replaces the "approximate task scheduling" of [6], is the only change to the algorithm of [6] that is essential in order to avoid the use of expander graphs; other small modifications in the present description are introduced merely in the

interest of simplicity. After each iteration and for each vertex u, the processors allocated to u use Theorem 6.1 to estimate the number s_u of selected darts found for u; specifically, they compute an integer $\widehat{s}_u$ with $s_u \leq \widehat{s}_u \leq 2s_u$. If $\widehat{s}_u \geq 2d_{i-1}$ and therefore $s_u \geq d_{i-1}$, u becomes inactive. When all blocks have become inactive, G' is formed from a subset of the selected darts: a vertex with s selected darts contributes $\min\{s, d_{i-1}\}$ of these, and the undirected versions of all darts contributed in this way form the edge set of G'; since the number of incomplete vertices in G_{i-1} is at most m/d_{i-1}^2, the number of edges in G' will be at most m/d_{i-1}, as required.

The correctness of Step 1 is obvious. The time analysis is as in [6]: Say that (the processing of) a dart in some iteration is of type (a) if the dart is discarded, and of type (b) if the tail of the dart becomes inactive at the end of the iteration under consideration, and say that an iteration is of type (a) or (b) if at least one third of the darts processed in the iteration are of type (a) or (b), respectively. Call an iteration *regular* if it is neither the first nor the last iteration in its stage. For sufficiently large values of n, every regular iteration is either of type (a) or of type (b) (or both), as can be seen from the following observations. Firstly, every regular iteration processes $\Omega(p)$ blocks. Secondly, a dart that is neither of type (a) nor of type (b) is selected, but its tail continues to be active. In Stage i, this happens to at most $2d_{i-1} - 1$ darts per vertex, and hence to $O(m/d_{i-1}) = O(m/\sqrt{\log n})$ darts altogether; for sufficiently large values of n, these darts cannot constitute one third or more of the $\Omega(pB) = \Omega(m/(\log\log n)^2)$ darts processed in a regular iteration.

As a dart is discarded only once, the total work expended in all iterations of type (a) is $O(m)$. We will prove that every regular iteration of type (b) reduces the number of active blocks by at least a constant factor. Since the total number of blocks is $O(m/B) = O(p(\log\log n)^2)$, this can happen only $O(\log\log\log n)$ times within one stage. The time spent in all iterations that are not of type (a) over all stages is hence $O(B\log\log n\log\log\log n) = O(\log n)$.

In order to prove that a regular iteration of type (b) reduces the number of active blocks by at least a constant factor, we distinguish between *bad* vertices, those for which at most one block is processed, and the remaining *good* vertices. Say that an *initially active* block is one that is active at the beginning of the iteration under consideration. The proportion of both processed blocks and initially active blocks contributed by bad vertices is $o(1)$ (i.e., tends to zero as n tends to infinity), for the bad vertices contribute at most $n = o(p)$ processed blocks and at most $n\widehat{b}/p = o(\widehat{b})$ initially active blocks, where $\widehat{b}$ is the estimate of the number of initially active blocks computed by the algorithm. On the other hand, if for each vertex we form the ratio of the number of processed blocks to the number of initially active blocks, then the ratios of two good vertices will differ by at most a factor of 2 (because if $\lceil b_u p/\widehat{b}\rceil \geq 2$, then $\lceil b_u p/\widehat{b}\rceil \leq 2b_u p/\widehat{b}$). Using these observations, we can conclude that in a regular iteration of type (b) and for sufficiently large values of n, at least one quarter of the blocks of good vertices processed belong to vertices that become inactive in that iteration, hence that at least one eighth of the initially active blocks of good vertices become inactive (since for each good vertex the "sample" of processed blocks faithfully reflects the number of active blocks, up to a factor of 2), and finally that therefore at least one ninth of all initially active blocks become inactive.

For Step 2 we use a modification proposed in [6] of the connected-components algorithm of Shiloach and Vishkin [22]. The algorithm runs in $O(\log d_{i-1})$ time using $O((n+m/d_{i-1})\log d_{i-1})$ operations and computes a spanning forest of G', each of whose incomplete trees contains at least d_{i-1} vertices and hence has a weight of at least d_i^2, as required. The time and the number of operations needed sum over all stages to $O(\log n)$ and $O(n\log n + m) = O(m)$, respectively.

For Step 3, begin by constructing an Euler tour of each of the trees in G'' computed in Step 2. To do this, convert G'' from the representation as an unordered list of darts to the adjacency list representation, after which constructing the Euler tour in constant time is a trivial matter. The conversion of G'' essentially reduces to sorting its darts by their tails. Using the algorithm of Bhatt *et al.* [4], this can be done in $O(\log n/\log\log n)$ time using $O((n + m/d_{i-1})\log\log n)$ operations, which sums over all stages to $O(\log n)$ time and $O(m)$ operations. We now appeal to the idealizing assumption mentioned in the beginning of the proof, which is that no tree created in Step 2 contains more

than d_{i+1}^2 vertices. Under this assumption $O(\log d_{i-1})$ steps of standard pointer doubling applied to the Euler tour of each tree suffice to select a unique name for each tree, to update the array T accordingly, and to number the vertices of each tree consecutively; the time and the number of operations needed are as for Step 2. The block headers of all vertices can then be rearranged by means of prefix summation so that the headers of each tree appear in consecutive locations; this needs $O(\log n/\log\log n)$ time and $O(m/B)$ operations per stage, which sums over all stages to $O(\log n)$ time and $O(m)$ operations. Processing again the blocks processed in Step 1 and additionally spending another $O(\log n/\log\log n)$ time and $O(m/B)$ operations, we can also for each tree divide the processed darts that were not discarded into new blocks of B darts each, with at most one incomplete block, reuse some block headers of processed blocks for the new blocks and remove the block headers that are no longer needed. Now G_i has been constructed from G_{i-1}.

If we remove the simplifying assumption that each tree created in Step 2 is of size at most d_{i+1}^2, there may be trees that cannot be contracted in the $O(\log d_{i-1})$ time allotted to this operation. We shall still want to consider the vertices in such a tree as forming a new vertex in G_i; since this vertex is not represented by the algorithm in the usual way, we call it a *virtual vertex*. We hence distinguish between a *virtual graph*, the one that the algorithm would have constructed if the tree contraction had been allowed to run to completion, and the *actual graph*, which differs from the virtual graph in that certain contractions have not been carried out. Note that we consider the weight invariants to apply to the virtual graph. We identify the edges in the actual graph with the edges in the virtual graph in the natural manner. The vertices represented in the actual graph but not present in the virtual graph will be called *out-of-date* vertices, and each spanning-tree edge found in Step 2 between two out-of-date vertices will be called an out-of-date edge; note that this does not include all edges between out-of-date vertices. A *current* vertex is a vertex in the actual graph that is not out-of-date. For $i = 1, 2, \ldots$, call a vertex *heavy* in Stage i if its weight is at least d_{i+1}^2; the intention is that the weight of a heavy vertex is sufficient to allow the vertex to be ignored in the following stage. Spending sufficient but still $O(\log d_{i-1})$ time in Step 3, we can easily ensure that each virtual vertex created in Stage i is heavy in Stage i, for $i = 1, 2, \ldots$ The approach for dealing with a heavy vertex, basically, is to "let it lie" until the first stage in which it is no longer heavy, at which point it will be contracted into a normal vertex. This involves a number of changes to the algorithm. First of all, in Step 1 we no longer attempt to select darts out of an out-of-date vertex, which is both unfeasible (since we cannot tell whether a given dart is a loop in the virtual graph) and unnecessary (because the vertex is contained in a heavy vertex in the virtual graph). There may still be selected darts leading into out-of-date vertices from current vertices; however, we arrange it so that a current vertex that encounters a dart into an out-of-date vertex selects one such dart without consulting the table $ACTUAL$, after which it immediately becomes inactive. This ensures that the current vertex will not have to decide whether two darts leading to out-of-date vertices are incident on the same virtual vertex, which would be unfeasible, and the single dart selected suffices to satisfy the weight condition. In Step 2, the spanning-tree algorithm is modified to treat all edges incident on some out-of-date vertex as though they were incident on the same ("canonical out-of-date") vertex. Since this represents the most conservative assumption concerning which out-of-date vertices might be contained in the same virtual vertex, the edge set computed certainly spans an acyclic subgraph, even in the virtual graph. In the virtual graph there may be trees with fewer than d_{i-1} vertices, but each such tree contains at least one virtual vertex and is therefore sufficiently heavy. Finally, in Step 3 we want to attempt to contract each tree of a virtual vertex created in a previous stage. In Stage i, the algorithm of Step 3 is therefore applied to a graph that includes not only the vertices in V_{i-1} and the edges found in Step 2 of Stage i, but also all out-of-date vertices and edges. Since there are at most n out-of-date vertices and at most $n-1$ out-of-date edges (the out-of-date edges form no cycle), this does not increase the complexity of Step 3 by more than a constant factor. With these modifications, the algorithm works without the simplifying assumption and still uses $O(\log n)$ time and $O(m)$ operations. □

7 Open problems

We have demonstrated the existence of deterministic algorithms with running times of $o(\log n/\log\log n)$ for problems in the "prefix summation family". Although for practical values of n the running time of $O((\log\log n)^3)$ achieved in this paper is likely to behave like a constant, on the theoretical side there remains a large gap to the $O(\log^* n)$ achievable with randomized algorithms, and to the only known lower bound of $\Omega(\log^* n)$ [18]. Obtaining better upper bounds or showing that the performance of randomized algorithms cannot be matched deterministically is an obvious open problem.

Although the present paper succeeds in derandomizing some of the results obtained in the course of the so-called "log-star revolution" [13], with some loss in speed, other such results are entirely unaffected because they additionally use randomization for purposes unrelated to compaction. Can interesting deterministic algorithms be obtained for these problems?

References

[1] H. Bast, M. Dietzfelbinger, and T. Hagerup, A Perfect Parallel Dictionary, *in* Proc. 17th International Symposium on Mathematical Foundations of Computer Science (1992), Springer Lecture Notes in Computer Science, Vol. 629, pp. 133–141.

[2] H. Bast and T. Hagerup, Fast and Reliable Parallel Hashing, *in* Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (1991), pp. 50–61.

[3] P. Beame and J. Hastad, Optimal Bounds for Decision Problems on the CRCW PRAM, *J. ACM* **36** (1989), pp. 643–670.

[4] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena, Improved Deterministic Parallel Integer Sorting, *Inform. and Comput.* **94** (1991), pp. 29–47.

[5] R. Cole and U. Vishkin, Faster Optimal Parallel Prefix Sums and List Ranking, *Inform. and Comput.* **81** (1989), pp. 334–352.

[6] R. Cole and U. Vishkin, Approximate Parallel Scheduling. II. Applications to Logarithmic-Time Optimal Parallel Graph Algorithms, *Inform. and Comput.* **92** (1991), pp. 1–47.

[7] M. L. Fredman, J. Komlós, and E. Szemerédi, Storing a Sparse Table with $O(1)$ Worst Case Access Time, *J. ACM* **31** (1984), pp. 538–544.

[8] J. Gil, Y. Matias, and U. Vishkin, Towards a Theory of Nearly Constant Time Parallel Algorithms, *in* Proc. 32nd Annual Symposium on Foundations of Computer Science (1991), pp. 698–710.

[9] M. T. Goodrich, Using Approximation Algorithms to Design Parallel Algorithms that May Ignore Processor Allocation, *in* Proc. 32nd Annual Symposium on Foundations of Computer Science (1991), pp. 711–722.

[10] T. Hagerup, Fast Parallel Space Allocation, Estimation and Integer Sorting, Tech. Rep. no. MPI–I–91–106, Max-Planck-Institut für Informatik, Saarbrücken, 1991.

[11] T. Hagerup, Fast Parallel Generation of Random Permutations, *in* Proc. 18th International Colloquium on Automata, Languages and Programming (1991), Springer Lecture Notes in Computer Science, Vol. 510, pp. 405–416.

[12] T. Hagerup, Fast and Optimal Simulations between CRCW PRAMs, *in* Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science (1992), Springer Lecture Notes in Computer Science, Vol. 577, pp. 45–56.

[13] T. Hagerup, The Log-Star Revolution, *in* Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science (1992), Springer Lecture Notes in Computer Science, Vol. 577, pp. 259–278.

[14] T. Hagerup, On a Compaction Theorem of Ragde, *Inform. Process. Lett.* **43** (1992), pp. 335–340.

[15] T. Hagerup and J. Katajainen, Improved Parallel Bucketing Algorithms for Proximity Problems, *in* Proc. 26th Hawaii International Conference on System Sciences (1993).

[16] T. Hagerup and R. Raman, Waste Makes Haste: Tight Bounds for Loose Parallel Sorting, *in* Proc. 33rd Annual Symposium on Foundations of Computer Science (1992), pp. 628–637.

[17] K. Iwama and Y. Kambayashi, A Simpler Parallel Algorithm for Graph Connectivity, manuscript, 1992.

[18] P. D. MacKenzie, Load Balancing Requires $\Omega(\log^* n)$ Expected Time, *in* Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (1992), pp. 94–99.

[19] Y. Matias and U. Vishkin, Converting High Probability into Nearly-Constant Time – with Applications to Parallel Hashing, *in* Proc. 23rd Annual ACM Symposium on Theory of Computing (1991), pp. 307–316.

[20] P. Ragde, The Parallel Simplicity of Compaction and Chaining, *in* Proc. 17th International Colloquium on Automata, Languages and Programming (1990), Springer Lecture Notes in Computer Science, Vol. 443, pp. 744–751.

[21] R. Raman, The Power of Collision: Randomized Parallel Algorithms for Chaining and Integer Sorting, *in* Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science (1990), Springer Lecture Notes in Computer Science, Vol. 472, pp. 161–175.

[22] Y. Shiloach and U. Vishkin, An $O(\log n)$ Parallel Connectivity Algorithm, *J. Alg.* **3** (1982), pp. 57–67.