Approximate Parallel Prefix Computation and Its Applications

(Preliminary Version)

Michael T. Goodrich[*] Yossi Matias[†] Uzi Vishkin[‡]

Abstract

In this paper we address two fundamental problems in parallel algorithm design—parallel prefix sums and integer sorting—and show that both of them can be approximately solved very quickly on a randomized CRCW PRAM. In the case of prefix sums the approximation is in terms of the accuracy of the sums and in the case of integer sorting it is in terms of allowing some *gaps* between consecutive elements in the ordered list. By introducing approximation in these ways we are able to solve these problems in $o(\lg\lg n)$ time, and thus avoid the near-logarithmic lower bounds by Beame and Håstad that hold for the exact versions of these problems. Nevertheless, we demonstrate that these approximations are strong enough to be used as subroutines in fast randomized algorithms for some well-known problems in parallel computational geometry. Perhaps the most succinct way to describe the power of the new tools which are presented is by observing that prior to this work it was known how to solve the interval allocation problem fast. In the present work we show how to solve the *ordered* version of the problem.

[*]Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218. This research supported by the NSF and DARPA under Grant CCR-8908092, by the NSF under Grants CCR-9003299 and IRI-9116843, and by The Bureau of the Census under Contract JSA-91-23.

[†]Institute for Advanced Computer Studies, University of Maryland; and the Department of Computer Science, Tel Aviv University, Israel. Partially supported by NSF grants CCR-9111348 and CCR-8906949. Current Address: AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. This work is part of this author's PhD thesis [17].

[‡]Institute for Advanced Computer Studies and Department of Electrical Engineering, University of Maryland, College Park, MD 20742. Also with the Department of Computer Science, Tel Aviv University, Israel. Partially supported by NSF grants CCR-9111348 and CCR-8906949.

1 Introduction

Computing all prefix sums for a list of group or semigroup elements is perhaps the most frequently used subroutine in parallel algorithms today. Deterministically, one can find all such sums in logarithmic time using an optimal number of processors, as shown by Stone [21] and Ladner and Fischer [15]. In the CRCW PRAM model one can do even better, in that, as shown by Cole and Vishkin [7], one can achieve a running time of $O(\lg n/\lg\lg n)$ using an optimal number of processors. This time was shown to be best possible with any polynomial number of processors by Beame and Håstad [4], even if a randomized algorithm is sought (using a theorem by Adleman [1]).

This lower bound result drove several researchers searching for algorithms with significantly sub-logarithmic running times to abandon using the prefix sums problem as a subroutine in favor of new problems for which they could produce constant-time or near-constant time algorithms. Indeed, several problems were suggested recently, which may be viewed as much relaxed versions of the prefix sums problem, and for which nearly-constant time algorithms can be developed [9, 10, 11, 13, 17, 18]. These problems include the linear approximate compaction problem [18], the load balancing problem [9], the interval allocation problem [13], and the density partitioning problem [11]. While these problems can be used, often in concert, to replace parallel prefix for some applications, the goal of the present paper is return to the prefix sums problem.

We show that one can solve an approximate version of the prefix sums problem in $o(\lg\lg n)$ parallel time with high probability, using an optimal number of processors on a CRCW PRAM. We show that, with high probability, each prefix sum given by our method is within a small $(1+\epsilon)$ factor of the true value, with ϵ being $1/\lg^d n$ for any constant $d \geq 0$. Specifically, the approximate prefix sums produced are strictly non-decreasing in such a way that the difference between consecutive computed prefix sums is always not smaller than the difference between the exact prefix sums.

We show the utility of this result by deriving a fast randomized parallel algorithm for a "relaxed," but still quite natural, version of the integer sorting problem, known as padded integer sorting [16]. In this version of integer sorting we allow for *gaps* in the ordered listing, so long as the total space needed for the array containing these elements is still linear. This contrasts with recent fast solutions for the strictly-weaker *integer chain sorting* which is little more than a reduction of sorting to the list ranking problem [2, 7] (which, as can be derived by the lower bounds of Beame and Håstad, has a near-logarithmic lower bound). Even though this is a relaxed version of sorting it is still quite powerful, as we demonstrate by giving several applications to well-known problems in parallel computational geometry, including convex hull construction, point set triangulation, and 2-dimensional hidden-line elimination.

Related work: Rajasekaran and Reif [19] gave the first optimal randomized parallel algorithm for integer sorting in $O(\lg n)$ time. An improvement to $O(\lg n/\lg\lg n)$ time, matching the lower bound which is implied by [4], was recently given by [12, 18, 20]. Recently, MacKenzie and Stout [16] gave an algorithm for padded sorting. Their algorithm takes doubly logarithmic time with high probability, but the input is assumed to be taken uniformly at random from the unit interval. They also considered some applications to computational geometry, but it seems that these applications use heavily the assumption that input is taken uniformly at random from the unit square. Hagerup [12] defined the integer chain sorting problem, and gave an optimal randomized algorithm in the doubly logarithmic level. An algorithm in the $O(\lg^* n)$ time level was subsequently given by [10].

As for our applications in parallel computational geometry, previous related results include a randomized method by Ghouse and Goodrich [8] for finding the convex hull of a sorted set of points in almost surely $O(\lg^* n)$ time using an optimal number of processors, an $O(\lg\lg n)$ time method by Berkman *et al.* [5] for triangulating a one-sided monotone polygon, and $O(\lg n)$ time deterministic methods by Atallah, Cole, and Goodrich [3] for solving the 2-dimensional hidden-line elimination problem we address.

Postscript. In an independent work, Hagerup and Raman [14] have recently obtained results similar to ours for padded integer sorting and for approximate prefix sums. Their approach is entirely different, with an emphasis on targeting general padded sorting and then applying it to the more elementary problems of padded integer sorting and approximate prefix sums. Our solution is direct as one would expect for more elementary problems. These results are important since they provide new building blocks for the design of parallel algorithms. We believe that our methods have also intrinsic value as they contribute new ideas.

2 Approximate Prefix Sums

Let $a_1, a_2, \ldots, a_n$ be a given sequence of integers. The *parallel prefix* problem [15, 21] is to compute in parallel the prefix sums sequence, i.e. to compute a sequence $b_1, b_2, \ldots, b_n$ such that, for $i = 1, \ldots, n$, $b_i = \sum_{j=1}^{i} a_j$ for $i = 1, \ldots, n$. A well-known method for solving this problem in parallel (e.g., see [15]) involves solving a related problem, which we call the *summation tree* problem, which is to define a complete binary tree T "on top" of the sequence $a_1, a_2, \ldots, a_n$ and, for each internal node $v \in T$, compute the sum, which we denote by $S(v)$, of the elements stored in v's descendants. (Leaf i of the tree corresponds to item a_i and we assume for simplicity that n is a power of 2.) Indeed, this will also be the approach we follow for solving the approximate parallel prefix problem, which we define as follows:

Approximate parallel prefix. Given some $\epsilon \geq 0$ and a sequence $a_1, a_2, \ldots, a_n$ of non-negative numbers, the *ϵ-approximate parallel prefix* problem is to compute in parallel an *approximate prefix sums sequence*, i.e. a sequence $b_0 = 0, b_1, b_2, \ldots, b_n$, such that, for $i = 1, \ldots, n$,

$$(\sum_{j=1}^{i} a_j)(1+\epsilon)^{-1} \leq b_i \leq (\sum_{j=1}^{i} a_j)(1+\epsilon).$$

We say that such a sequence is *consistent* if $b_i - b_{i-1} \geq a_i$, for $i = 1, \ldots, n$ (so that we automatically get $b_i \geq \sum_{j=1}^{i} a_j$).

An approximate summation tree. Similarly, we can define an *ϵ-approximate summation tree* by building a complete binary tree on top of the sequence $a_1, \ldots, a_n$ and computing a value $\bar{S}(v)$, for each internal node $v \in T$, such that $S(v)(1+\epsilon)^{-1} \leq \bar{S}(v) \leq$

$S(v)(1+\epsilon)$. We say that such an approximate summation tree is *consistent* if $\bar{S}(v) \geq \bar{S}(v_l) + \bar{S}(v_r)$, where v_l and v_r are the children of v in T.

Reducing approximate parallel prefix to approximate summation tree Our method for computing a consistent ϵ-approximate parallel prefix sequence is based on a reduction to the problem of constructing a consistent ϵ-approximate summation tree. So, suppose we have an ϵ-approximate summation tree built on top of the sequence $a_1, \ldots, a_n$. Given the approximation $\bar{S}(v)$ for each node v in T, we compute an approximate parallel prefix sequence as follows. For a leaf v we define $L_T(v)$ to be the set of nodes $u \in T$ on the *left fringe* of the path from v to the root, i.e., such that (i) u is a left child, (ii) u is not an ancestor of v, and (iii) the parent of u is an ancestor of v. For the i'th leaf v we compute

$$b_i = \sum_{u \in L_T(v)} \bar{S}(u) + a_i \ .$$

This can either be done, for each leaf v, by "brute force" in $O(1)$ time using $O(n^{1/k})$ processors, where $k \geq 1$ is a constant, or in $O(\lg\lg n/\lg\lg\lg n)$ time with $O(\lg n)$ work [7].

Lemma 2.1 *If the tree T is a consistent ϵ-approximate summation tree, then the sequence $b_0 = 0, b_1, \ldots, b_n$ is a consistent ϵ-approximate parallel prefix sequence.*

Proof. The proof follows from the definition of a consistent ϵ-approximate summation tree, and is omitted here. ■

Therefore, it is sufficient for us to construct a consistent ϵ-approximate summation tree. However, this computation is complicated by a number of factors, not the least of which is that previous summation approximation schemes [10, 11, 13] do not yield consistent sums. So, let us address this consistency issue first.

Achieving consistency. Suppose we have an ϵ-approximate summation tree T. We wish to make T consistent, but do not wish to disturb the accuracy of the approximate partial sums in T by too much.

Lemma 2.2 *Given an ϵ-approximate summation tree T, one can convert T into a consistent $(4\epsilon\lg n)$-approximate summation tree in $O(1)$ time using $|T|$ processors.*

Proof. Our method is actually quite simple. Let $S'(v)$ denote the approximate value stored at $v \in T$. For each v in T, which is, say, at height is i, we let

$$\bar{S}(v) = (1+\epsilon)^{2i} S'(v) \ .$$

This choice is sufficient to guarantee consistency (i.e., $\bar{S}(v) \geq \bar{S}(v_l) + \bar{S}(v_r)$). For, by the definition of an ϵ-approximate summation tree we have

$$\begin{aligned}
\bar{S}(v_l) + \bar{S}(v_r) &= S'(v_l)(1+\epsilon)^{2i-2} + S'(v_r)(1+\epsilon)^{2i-2} \\
&\leq S(v_l)(1+\epsilon)^{2i-1} + S(v_r)(1+\epsilon)^{2i-1} \\
&= S(v)(1+\epsilon)^{2i-1} \\
&\leq S'(v)(1+\epsilon)^{2i} \\
&= \bar{S}(v) \ .
\end{aligned}$$

By well-known inequalities, for $0 \leq \epsilon \leq 1$,

$$(1+\epsilon)^{2i} \leq (1+\epsilon)^{2\lg n} \leq e^{2\epsilon\lg n} \leq 1 + 4\epsilon\lg n,$$

which establishes the claimed approximation factor. ■

Therefore, given an ϵ-approximate summation tree, for a sufficiently small ϵ, we can make T consistent and ϵ'-approximate. Unfortunately, in addition to being inconsistent, known methods for computing approximate sums are probabilistic [10, 11, 13]—they may return some inaccurate approximate partial sums (albeit with small probability). Next, we provide sufficiently accurate algorithms for our purposes.

Computing an ϵ-approximate summation tree. The core of our approximate parallel prefix algorithm is an algorithm that computes ϵ-accurate sums for all the nodes of the tree *simultaneously*, for $\epsilon = 1/\lg^d n$, where $d \geq 0$ is any fixed constant. A nice side benefit of the ϵ term for this method is that, by the previous discussion, it allows us to construct a consistent ϵ'-approximate summation tree (and, hence, an ϵ'-approximate parallel prefix sequence) for $\epsilon' = 1/\lg^c n$, where $c \geq 0$ is any fixed constant (this can be done simply by taking d large enough relative to c). To derive this method, we present an algorithm that for any node computes an ϵ-accurate sum with n-polynomial probability.[1] This algorithm is primarily based on an algorithm due to [10, Thm 6.1]. For an input of size n, their algorithm computes an ϵ-accurate sum, for $\epsilon = 1/\lg^d n$, with n-polynomial probability. We

[1] We say that a probability is n-polynomial if it is $1 - n^{-\gamma}$ for a constant $\gamma > 0$.

need a stronger result however, both in terms of accuracy and of success probability, since we have *many* sub-problems of various sizes—between polylog(n) and n/polylog(n)—and for *each* sub-problem we need to compute $(1/\lg^d n)$-accurate sum with n-polynomial probability. (For such sub-problems, the algorithm of [10] would only compute $(1/\lg\lg n)$-accurate sums with $\lg n$-polynomial probabilities.)

We first present an estimation algorithm stronger than the one given in [10, Thm 4.2]:

Lemma 2.3 (multiple estimation) *Let $m \geq \lg^{2d+1} n$ and assume that we are given a partition of m processors into sets, such that each processor is allocated with an auxiliary set of $\lg n$ processors, and let $d \geq 0$ be a constant. Then, there exists an algorithm which, in $O(\lg\lg n/\lg\lg\lg n)$ time and using $O(m \lg m \lg n)$ space, computes for each set an $(1/\lg^d n)$-estimate for its size, with n-polynomial probability.*

Proof. (Sketch) For each set Φ we do a *geometric decomposition* into the sets $\Phi_1, \Phi_2, \ldots$: Each element in Φ selects itself to be in Φ_i with probability 2^{-i}, for $i = 1, \ldots, \lg m$ (and in none of the subsets Φ_i with probability $1/m$). Now, we will estimate $|\Phi|$ by computing $|\Phi_i|$ for some i such that $|\Phi_i| = \Theta(\lg n)$. Such i exists with n-polynomial probability, and each estimate is then an $(1/\lg n)$-estimate with n-polynomial probability as well. To implement the computation of $|\Phi_i|$, we first insert each subset Φ_i into an array of size $\Theta(\lg n)$ and then compute $|\Phi_i|$ in $O(\lg\lg n/\lg\lg\lg n)$ time [7]. By using the renaming algorithm of [9] we can insert each subset of size at most $\lg n$ into its array in $O(\lg\lg\lg n)$ expected time. By applying the same algorithm $\lg n$ times in parallel into $\lg n$ different arrays, all subsets of size at most $\lg n$ are inserted to their arrays in time $O(\lg\lg\lg n)$ with n-polynomial probability. ∎

We can now get

Lemma 2.4 (approximate sum) *Let $d > 0$ be a constant and d' be another constant dependent on d. Given m numbers, $\lg^{d'} n \leq m \leq n$, each allocated with $\lg n$ teams of $\lg n$ processors each, a $(1/\lg^d n)$-accurate sum can be computed in $O(\lg\lg n/\lg\lg\lg n)$ time with n-polynomial probability.*

Proof. (Sketch) Our algorithm is similar to the algorithm of [10, Thm 6.1] except for several modifications, which include: (1) instead of replacing each number by one of the nearest powers of 2, as done in Step 4 in [10, Thm 6.1], we replace each number by $\leq 2\lg n$ powers of 2, according to its binary representation; and (2) in Step 5 we use the estimation algorithm of Lemma 2.3. Since all other steps contribute errors smaller than $1/m$ with m-exponential probability, the lemma follows. We only give here an overview. The input numbers are first normalized to integers from $[0 \ldots m^2]$. Then, the input is further modified to consist only of numbers which are powers of 2. This implies partitioning the numbers into $2\lg m + 1$ sets. Next, the input is once again modified to get a more favorable distribution of values, where the relatively larger values can only exist in "big" groups; the reason being that misestimating larger values contributes more to an error, and having them in bigger groups enables evaluating their number with higher probability. Now, an estimate for the size of each set is computed and the resulting sum is computed. ∎

We now have

Lemma 2.5 *Using $n\lg^2 n$ processors, the $(1/\lg^d n)$-approximate parallel prefix problem can be solved, for any constant $d \geq 0$, in $O(\lg\lg n/\lg\lg\lg n)$ time with high probability.*

Proof. Each leaf is allocated with $\lg n$ teams of $\lg n$ processors each; each team will participate in computing the approximate sum of one of the leaf's ancestors. Using the approximate sum algorithm of Lemma 2.4, we compute for each node v an ϵ-accurate sum, for $\epsilon = 1/\lg^{d+2} n$. If the height of v in T is i then we let $\bar{S}(v) = (1+\epsilon)^{2i} S'(v)$. If v is the i'th leaf then we compute $b_i = \sum_{u \in L(v)} \bar{S}(u) + a_i$. Since each leaf is allocated with $\lg n$ processors, this can be done in parallel in time $O(\lg\lg n/\lg\lg\lg n)$ by using the parallel prefix algorithm of [7]. By Lemma 2.1 and Lemma 2.2 the sequence $b_0 = 0, b_1, \ldots, b_n$ is a $(1/\lg^d n)$-approximate parallel prefix sequence. ∎

In order to achieve an optimal algorithm, i.e., using $n/\lg\lg n$ processors, we can now use standard techniques. Specifically, the input sequence is first partitioned into blocks of size $(\lg n)^2$ each. Within each block the prefix sums are computed in $O(\lg\lg n/\lg^{(3)} n)$ time, using the algorithm of [7]. We now consider a sequence $a'_1, \ldots, a'_{n/(\lg n)^2}$, where a'_i is the sum of elements in the i'th block; i.e., $a'_i = \sum_{j=1}^{(\lg n)^2} a_{i-1+j}$. The algorithm of Lemma 2.5 is used to compute the approximate parallel prefix sequence $b'_1, \ldots, b'_{n/(\lg n)^2}$. Finally, the approximate parallel prefix sequence $b_1, \ldots, b_n$ is computed as follows: if j is in block i then b_i is the sum of b'_{i-1} and the prefix sum of j within its block.

We have

Theorem 2.1 *Let d be an arbitrary constant. Given a sequence $a_1, a_2, \ldots, a_n$ of positive numbers, one can construct a $(1/\lg^d n)$-approximate parallel prefix sequence for it with high probability in $O(1)$ time using $O(n^{1+1/k})$ processors, or in $O(\lg\lg n/\lg\lg\lg n)$ time with $O(n)$ work on a CRCW PRAM, for any constants $c, k \geq 1$ and $d \geq 0$.*

3 Padded Integer Sorting

As we show in this section, the approximate parallel prefix problem will prove useful for the following problem.

Ordered allocation. Given a sequence $a_1, \ldots, a_n$, the *ordered allocation* problem is to allocate in order a sequence of non-overlapping intervals in an array of size $(1+\epsilon)\sum_{i=1}^{n} a_i$ so that the i'th interval is of size $\geq a_i$. More formally, compute $I_i = \langle L_i, R_i \rangle$, for $i = 1, \ldots, n$, and L_{n+1} such that for all $i = 1, \ldots, n$, L_i and R_i are integers and

(a) $R_i - L_i + 1 \geq a_i$ (allocation);

(b) $L_{n+1} \leq (1+\epsilon)\sum_{i=1}^{n} a_i$ (approximation);

(c) $R_i < L_{i+1}$ (no overlap and ordering).

The ordered allocation problem can be easily computed from an approximate parallel prefix sequence as follows: Define the sequence $a'_1, \ldots, a'_n$: if $a_i = 0$ then $a'_i = 0$ otherwise $a'_i = a_i + 1$. Compute the approximate parallel prefix sequence $b'_1, \ldots, b'_n$ of $\{a'_i\}$. Let $L_1 = 0$; for $i = 1, \ldots, n$, let $R_i = \lfloor b_i \rfloor$ and let $L_{i+1} = R_i + 1$. It is easy to verify that the resulting sequence $I_1, \ldots, I_n$ is an ordered allocation.

We use ordered allocation to derive our first application of approximate parallel prefix computation, which is for the following problem.

Padded Integer Sorting. Given a sequence $X = \{x_1, \ldots, x_n\}$ taken from the integer interval $[1, \ldots, n]$, the *padded integer sorting* problem is to compute an injective mapping $\pi : X \mapsto [1, \ldots, \beta n]$ for some constant β, such that π is order preserving; i.e., if $x_i < x_j$ then $\pi(x_i) < \pi(x_j)$. In other words, the problem is to insert the elements of X in a sorted manner into an array $[1, \ldots, \beta n]$, while allowing empty cells between consecutive elements.

We show in this section how to reduce the padded integer sorting problem to the approximate parallel prefix problem. Let $T_{APS}(n)$ be a time upper bound for the approximate parallel prefix problem with n input elements, using an optimal number of processors.

Similar to the approach of [19], it is useful to have a stable sorting when the input is taken from a very small universe size. (A sorting algorithm is *stable* if the ordering within elements with equal values remains the same.) Using an algorithm of [7], with the approximate parallel prefix algorithm serving instead of the usual parallel prefix algorithm, we have

Lemma 3.1 *The padded integer stable-sorting problem with n input elements taken from the integer interval $[1, \ldots, k]$ can be solved in time $O(k + T_{APS}(n))$, using an optimal number of processors.*

We now show

Lemma 3.2 *The padded integer sorting problem can be solved in time $O(T_{APS}(n) \cdot \lg^* n)$ with high probability, using an optimal number of processors.*

Our algorithm is based on the integer chain-sorting algorithm in [10, Sect. 9]. Their algorithm in fact not only computes a chain of the elements in sorted order, but it also groups the elements according to their values. More precisely, the algorithm in [10] consists of $O(\lg^* n)$ iterations, at each of which a subset of the elements are inserted into an array—grouped according to their value. Our algorithm has two modifications: (i) using an ordered allocation algorithm instead of an interval allocation step to get the groups *ordered* according to their values, and (ii) adding a post-processing procedure in which all $O(\lg^* n)$ arrays are merged into a single array.

In step i of our algorithm we map a subset of the active elements into an array of size $cn/2^i$, for some constant c, such that the mapping is injective and order preserving. The algorithm consists of $O(\lg^* n)$ rounds. At each round, a new array serves as a "working space": for each set of elements sharing the same value an interval is allocated within the array, and a subset of these elements is mapped into it. Elements that are mapped do not participate in future rounds. Using the ordered allocation enables us to allocate intervals in order: thus the elements that are mapped into the array always consist of a non-decreasing monotonic sequence. Each round is done in constant number of steps. Thus, after $O(\lg^* n)$ steps we have

$O(\lg^* n)$ arrays, where in each array the elements are in sorted order. In a post-processing step, we merge the arrays into a single array of size $O(n)$. Since the elements are from the integer interval $[1, \ldots, n]$, we can use the $O(\alpha(n))$ integer merging algorithm of [6] to merge all arrays in time $O(\lg\lg^* n \cdot \alpha(n)) = o(\lg^* n)$.

There is a slight complication though: some of the arrays to be merged are of size less than linear; this implies that a universe of size n is too large for the merging algorithm. To overcome this, we first modify the range from which the input is taken to be $[1, \ldots, n/(\lg^* n)^3]$. This is done by considering only the $\lg n - 3\lg\lg^* n$ most significant bits of each input element. After (padded) sorting the modified input, the stable-sorting algorithm of Lemma 3.1 is used to sort according to the $3\lg\lg^* n$ least significant bits, in $O(\lg^* n)$ time. It remains to show that all arrays to be merged can be of size at most $O(n/(\lg^* n)^3)$.

As in the integer chain sorting algorithm of [10], optimal speedup is obtained by employing a preprocessing step in which the input size is reduced, using linear work. Specifically, the input size is reduced to $n/(T_{APS}(n)(\lg^* n)^3)$ in time $O(T_{APS}(n)\lg\lg^* n)$ (details omitted from this preliminary version). Then, the arrays used at each round are of size $O(n/(\lg^* n)^3)$, as required.

We therefore have

Theorem 3.1 *The padded integer sorting problem can be solved with high probability in time $O(\lg\lg n \lg^* n / \lg\lg\lg n)$ using an optimal number of processors.*

Comment: We omit the times achievable with a suboptimal number of processors in this preliminary version.

4 Applications to Parallel Computational Geometry

In this section we show that a number of well-known problems in parallel computational geometry can be solved efficiently and very fast by reductions to padded sort. Each application assumes one is given a set of geometric objects that are specified by integer coordinates in the range $[1..O(n)]$. The motivation for studying this restricted domain is that it is the domain that one may find in computer graphics and computer vision applications, where points that determine the geometric objects are pixel coordinates on a computer screen. Each of the results in this section are specified relative to $T_{PIS}(n)$, the running time of padded sort using an optimal number of processors. We begin with two simple reductions, which require only one call to padded sort, and we then give some more involved applications that require multiple calls.

4.1 Convex Hulls in the Plane

Suppose we are given a set S of n points in the plane. The *convex hull* problem is to produce a representation of the smallest convex set containing all the points of S. Typically, we desire that this representation list the edges of the convex hull (possibly with duplicate entries) in clockwise order. Ghouse and Goodrich [8] show that if one is given a set S of n points in the plane sorted by x-coordinates, then one can compute the convex hull of S in $O(\lg^* n)$ time, with high probability, using $O(n)$ work on a randomized CRCW PRAM. But, in fact, their algorithm never uses rank information, and one can show that their method can be implemented without loss of efficiency even assuming the underlying array of sorted elements contains duplicate entries (we omit the details in this extended abstract). Thus, by prefacing their (modified) method by a single call to padded sort, we can solve the convex hull problem for an unsorted point set in the same time as for the sorted case. Specifically, we have the following:

Theorem 4.1 *Given a set S of n planar points with integer coordinates, one can construct a representation of the convex hull of S in $O(T_{PIS}(n) + \lg^* n)$ time, with high probability, using an optimal number of processors on a randomized CRCW PRAM.*

4.2 Point Set Triangulation

Suppose we are again given a set S of n points in the plane. Related to the problem of constructing a convex hull for a set of points is the problem of triangulating that set, which is the problem of producing a subdivision of the convex hull of S into triangles, so that the interiors of no two triangles intersect and the entire convex hull of S is covered by triangles. Berkman *et al.* [5]. show that one can construct a triangulation of a monotone chain in $O(\lg\lg n)$ time using an optimal number of processors. Again, their method does not actually use rank information. It too only assumes one is given an ordered list of edges stored in an array. One can construct such a list of edges by performing a padded sort followed by a call to the "nearest ones" problem. Thus, as we show in the full version, one can modify their method to achieve the following result.

Theorem 4.2 *Given a set S of n planar points with integer coordinates, one can produce a triangulation of S in $O(T_{PIS}(n) + \lg\lg n)$ time, with high probability, using an optimal number of processors on a randomized CRCW PRAM.*

We have shown above that we can solve problems defined on point sets with integer coordinates very fast in parallel. In the subsections that follow we show analogous results for lines.

4.3 2-Dimensional Hidden Line Elimination

Suppose we are given a set S of n planar line segments that do not intersect, except possibly at endpoints. Suppose further that the endpoints of the segments in S have integer coordinates. The *2-dimensional hidden line elimination problem* is to produce a sorted list of pairs (x_i, y_i) such that x_i is the x-coordinate of a segment endpoint and y_i is the y-coordinate of the point visible from $(0, -\infty)$ at x_i (i.e., the lowest point on a segment in S that intersects the line $x = x_i$). Intuitively, one imagines the point $(0, -\infty)$ to be the "eye" location, and the problem is to produce a representation of what that eye can see assuming each segment is opaque. We show below how to solve this problem in $O(T_{PIS}(n))$ time using $O(n \lg n)$ work. The work complexity matches that of the fastest deterministic algorithm, which is due to Atallah, Cole, and Goodrich [3]; their algorithm takes $O(\lg n)$ time, however.

1. Sort the endpoints of S by x-coordinates by a call to padded sort.

2. Build a binary tree T "on top" of these x-coordinates. Each leaf v of T is associated with a vertical *slab* $\Pi(v)$, which is the set of all points whose x-coordinates fall in the x-interval associated with this leaf. Define for each internal node v a slab $\Pi(v)$ that is the union of the x-coordinate intervals defined by v's descendants. (This can all be easily implemented in $O(1)$ time.)

3. For any segment $s = \overline{pq}$, we say that *s covers v* if $\overline{pq}$ intersects the left and right boundaries of $\Pi(v)$ but does not intersect both boundaries of $\Pi(p(v))$, where $p(v)$ is v's parent. In this step we assign $O(\lg n)$ processors to each segment s and determine the nodes in T that s covers in $O(1)$ time. For each segment s this produces a collection of pairs (v, s) such that s covers v.

4. Perform a padded sort on all the (v, s) pairs so as to construct, for each $v \in T$, an array (which is actually a subarray of a larger array), $C(v)$, which contains all the segments that cover v. We can use padded sort for this step because the range of possible v's is $O(n)$ in size.

5. Note that each of the segments in $C(v)$ intersect $\Pi(v)$ in a well-defined order. In this step we compute the lowest segment in each $C(v)$, which can be done in $O(1)$ time, with high probability, using $C(v)$ processors.

6. For each endpoint p, determine the path from the leaf associated with $x(p)$ to the root, and find the lowest segment from among all the lowest segments in the $C(v)$ lists in this path. This, too, can be done in $O(1)$ time, with high probability, and gives us a solution to the $2-d$ hidden line elimination problem.

Thus, our method makes two calls to padded sort and all the other steps require only $O(1)$ time. Therefore we have

Theorem 4.3 *Given a set S of n non-intersecting planar segments with integer coordinates, one can solve the 2-dimensional hidden line elimination problem for S in $O(T_{PIS}(n))$ time, with high probability, and $O(n \lg n)$ work on a randomized CRCW PRAM.*

Acknowledgments

We would like to thank Yossi Gil for several helpful discussions in early stages of this research.

References

[1] Leonard M. Adleman. Two theorems on random polynomial time. In *FOCS '78*, pages 75–83, 1978.

[2] Richard J. Anderson and Gary L. Miller. Optimal parallel algorithms for list-ranking. *Information Processing Letters*, 33:269–273, 1990.

[3] M. Atallah, R. Cole, and M. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM J. Computing*, 18(3):499–532, 1989.

[4] Paul Beame and Johan Håstad. Optimal bounds for decision problems on the CRCW PRAM. In *STOC '87*, pages 83–93, 1987.

[5] Omer Berkman, Dany Breslauer, Zvi Galil, Baruch Schieber, and Uzi Vishkin. Highly parallelizable problems. In *STOC '89*, pages 309–319, 1989.

[6] Omer Berkman and Uzi Vishkin. On parallel integer merging. *Info. Comp.*, 1992. To appear. A preliminary version appeared in O. Berkman, J. JaJa, S. Krishnamurthy, R. Thurimella and U. Vishkin, Some triply-logarithmic parallel algorithms, 31st FOCS, 871-881, 1990.

[7] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Info. Comp.*, 81:334–352, 1989.

[8] M. Ghouse and M.T. Goodrich. In-place techniques for parallel convex hull algorithms. In *SPAA '91*, pages 192–203, 1991.

[9] Joseph Gil. Fast load balancing on a PRAM. In *SPDP '91*, pages 10–17, December 1991.

[10] Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *FOCS '91*, pages 698–710, October 1991.

[11] Michael T. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. In *FOCS '91*, pages 711–722, 1991.

[12] Torben Hagerup. Constant-time parallel integer sorting. In *STOC '91*, pages 299–306, 1991.

[13] Torben Hagerup. Fast parallel space allocation, estimation and integer sorting. Technical Report 03/91, SFB 124, Fachbereich 14, Universität des Saarlandes, 1991.

[14] Torben Hagerup and Rajeev Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *FOCS '92*, pages 628–637, 1992.

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

[16] Philip D. MacKenzie and Quentin F. Stout. Ultra-fast expected time parallel algorithms. In *SODA '91*, pages 414–423, 1991.

[17] Yossi Matias. *Highly Parallel Randomized Algorithmics.* PhD thesis, Tel Aviv University, Tel Aviv, Israel, December 1992.

[18] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *STOC '91*, pages 307–316, 1991.

[19] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18:594–607, 1989.

[20] Rajeev Raman. Optimal sub-logarithmic time integer sorting on a CRCW PRAM (note). Submitted for publication, 1991.

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