Optimal Parallel Approximation Algorithms for Prefix Sums and Integer Sorting *

(Extended Abstract)

Michael T. Goodrich † Yossi Matias ‡ Uzi Vishkin §

Abstract

Parallel prefix computation is perhaps the most frequently used subroutine in parallel algorithms today. Its time complexity on the CRCW PRAM is $\Theta(\lg n/\lg\lg n)$ using a polynomial number of processors, even in a randomized setting. Nevertheless, there are a number of non-trivial applications that have been shown to be solvable using only an approximate version of the prefix sums problem. In this paper we resolve the issue of approximating parallel prefix by introducing an algorithm that runs in $O(\lg^* n)$ time with very high probability, using $n/\lg^* n$ processors, which is optimal in terms of both work and running time. Our approximate prefix sums are guaranteed to come within a factor of $(1+\epsilon)$ of the values of the true sums in a "consistent fashion", where ϵ is $o(1)$. We achieve this result through the use of a number of interesting new techniques, such as *overcertification* and *estimate-focusing*, as well as through new adaptations of known techniques, such as *failure-sweeping* and *bit-thinning*.

We give a number of non-trivial applications of our approximate parallel prefix routine. Perhaps the most interesting application is for *padded integer sorting*, an approximation version of another fundamental problem in parallel algorithm design—integer sorting—where one wishes to sort n integers into an array of size $O(n)$, allowing for gaps between consecutive elements. We show that this problem can also be solved in $O(\lg^* n)$ time, with very high probability, using a linear amount of work, which is also optimal in both time and work. Finally, we show several applications to integer-coordinate (non-approximate) problems in computational geometry, such as convex hulls and hidden-line elimination, as well as for approximate selection.

1 Introduction.

Let $A = (a_1, a_2, \ldots, a_n)$ be a given sequence of non-negative integers. The *parallel prefix* problem [19, 28] is to compute in parallel all the prefix sums

$$b_i = \sum_{j=1}^{i} a_j,$$

for $i = 1, \ldots, n$. Deterministically, one can find all such sums in logarithmic time using an optimal number of processors, as shown by Stone [28] and Ladner and Fischer [19]. 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 the best possible with any polynomial number of processors by Beame and Håstad [3], even if a randomized algorithm is sought (using a theorem by Adleman [1]).

Interestingly, this lower bound does not hold for the approximation version of the parallel prefix sums problem, however, and, as we show in this paper, there are several applications where one needs only an approximate prefix sums sequence. Specifically, given some $\epsilon \geq 0$, the *ϵ-approximate parallel prefix* problem [13, 16] 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 $b_i \geq \sum_{j=1}^{i} a_j$).

*To appear in *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, January 1994.

†Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218. E-mail: goodrich@cs.jhu.edu. This research supported by the NSF and DARPA under Grant CCR-8908092, and by the NSF under Grants IRI-9116843 and CCR-9300079.

‡AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. E-mail: matias@research.att.com.

§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. E-mail: vishkin@umiacs.umd.edu. Partially supported by NSF grants CCR-9111348 and CCR-8906949.

1.1 Our results. In this paper we give a randomized parallel algorithm for constructing a consistent ϵ-approximate prefix sums sequence with ϵ being $o(1)$ in $O(\lg^* n)$ time using $O(n)$ work, with very high probability. Our method clearly beats the lower bound for the exact version of this problem, and it improves the previous approximate parallel prefix algorithms of Goodrich, Matias, and Vishkin [13], which ran in $O(\lg\lg n/\lg\lg\lg n)$ time and $O(n)$ work, with very high probability, and of Hagerup and Raman [16], which ran in $O(\lg\lg n)$ time and $O(n)$ work, with very high probability. Our method establishes the true complexity for this approximation problem, for our method is optimal, by a simple reduction from the load balancing problem, for which MacKenzie [20] has established an $\Omega(\lg^* n)$ lower bound, even in a randomized setting. Moreover, our randomized algorithm is of the *Las Vegas* type [18], indicating that the randomization affects the running time of the method, not the accuracy of our approximation, for our method always produces an accurate approximation. This was not the case, for example, with our previous method [13], which was of the alternate *Monte Carlo* type. We believe the Las Vegas nature of our new method is somewhat surprising, since computing a sum exactly (and then checking it against our approximation) has a near-logarithmic time lower bound [3].

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 [13, 21, 20, 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. Our method runs in $O(\lg^* n)$ time and $O(n)$ work, with very high probability, which is optimal [20] and improves the previous algorithm of Hagerup and Raman [16], which runs in $O(\lg\lg n)$ time and $O(n)$ work, with very high probability. Even though this is a relaxed version of sorting it is still quite powerful, as we demonstrate by giving several applications to integer-coordinate versions of many well-known problems in parallel computational geometry, including convex hull construction, point set triangulation, 2-dimensional hidden-line elimination, and several dominance problems. This class of inputs is motivated by applications in computer vision and computer graphics, where coordinates are determined by integer grid points. Finally, we give an improved constant-time method for approximate selection.

We achieve our optimal algorithms through the use of a number of interesting techniques, which we believe will find applications in other algorithms that use randomization, parallelism, or approximation:

New techniques:

- *overcertification.* This technique is useful for producing a fast randomized algorithm of the Las Vegas type. It involves replacing a test of correctness (which may be impossible or, as in our case, may take too long) with a collection of local consistency tests, which, if all true, imply correctness. There may be some correct answers that get rejected, but this is fine, since we wish to guarantee the correctness of the output we finally produce, not detect every possible correct output instance.

- *estimate-focusing.* This is a simple, but powerful, technique for boosting the confidence one has in an approximation without actually testing it against the true value. It involves computing several independent approximations and then choosing an approximate median from this group to be the chosen value.

Known techniques adapted to achieve our results:

- *failure-sweeping* [8], also known as the *thinning-out principle* [23] (see also [10]). This technique is a useful method for making probabilistic divide-and-conquer algorithms "act" more like deterministic ones. It involves the compaction of all erroneous recursive calls in a multi-way probabilistic divide-and-conquer algorithm so that their respective subproblems may be re-solved using additional resources. We adapt this technique here to be used in conjunction with our overcertification and estimate-focusing techniques.

- *bit-thinning.* This technique, used in [16, 24], involves reducing the number of significant bits in an approximation so that it is more efficient to use this value as an index in a look-up table. It enables us to achieve constant-time solutions for small subproblems with not-so-small word sizes.

Because our methods are provably optimal in both time and work, we believe our results can be viewed as the completion of a rather long list of results on very fast parallel approximation algorithms for some fundamental combinatorial problems, which we review below.

1.2 Related previous work. As mentioned above, the parallel prefix problem can be solved exactly in $O(\lg n/\lg\lg n)$ time using an optimal number of processors [7], and this is the fastest time possible using a polynomial number of processors [3], even in a randomized setting [1]. Perhaps because of this lower bound result, research on near constant-time parallel algorithms abandoned using the prefix sums problem as a subroutine, in spite of its widely-recognized usefulness in polylogarithmic parallel algorithms. Several problems were suggested instead, which may be viewed as much relaxed versions of the prefix sums problem, and for which nearly-constant time algorithms were developed [9, 11, 12, 15, 22, 23]. These problems include the linear approximate compaction [23], load balancing [9], interval allocation [15], and density partitioning [12]. (See also [22].) While these problems can be used, often in concert, to replace parallel prefix for some applications, their use is not always as natural as is the case for parallel prefix computations in polylogarithmic-time algorithms. Indeed, this deficiency motivated us to re-examine the parallel prefix problem in an approximate setting in [13], and was perhaps also a motivating factor for the similar independent re-examination by Hagerup and Raman [16], which resulted in optimal-work methods with sub-optimal doubly-logarithmic [16] (or slightly sub-doubly-logarithmic [13]) running times. As justification for this re-examination, we gave a number of applications of our result to computational geometry problems [13] (which we review and expand upon in this paper).

Possibly the most significant application of our new method, however, as mentioned above, is for an approximate version of parallel integer sorting, which also has a rich history in the literature. Rajasekaran and Reif [25] 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 [3], was given in [14, 23, 26]. Recently, MacKenzie and Stout [21] 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 these applications use heavily the assumption that input is taken uniformly at random from the unit square. Hagerup [14] 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 Gil *et al.* [11]. While this is also an interesting approximate version of parallel integer sorting, its applications are limited in that it amounts to a reduction of sorting to the well-known list ranking problem [17], for which the near-logarithmic lower bound still holds [3]. This lack of applicability motivated us to re-examine the padded integer sorting problem (allowing for arbitrary inputs) [13], so as to achieve a running time of $O(\lg\lg n \lg^* n/\lg\lg\lg n)$ with very high probability using an optimal number of processors, and may have also been a motivating factor for the independent re-examination by Hagerup and Raman [16], who achieve a running time of $O(\lg\lg n)$ time with very high probability using an optimal number of processors[1].

As for our applications in parallel computational geometry, previous related results include our previous paper [13] (albeit with sub-optimal running times), 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.* [4] for triangulating a one-sided monotone polygon, an $O(\lg\lg\lg(s+n))$ time algorithm by Berkman, Matias and Ragde [5] for the same problem, when the input is taken from the integer domain $[1..s]$, and $O(\lg n)$ time deterministic methods by Atallah, Cole, and Goodrich [2] for solving 2-dimensional hidden-line elimination and the dominance problems we address.

In the sections that follow we present the main ideas behind our results as well as giving the outlines of several applications.

2 Summation Estimation.

A basic subroutine used extensively in our approximate prefix sums algorithm is one which estimates the sum of a given set of numbers. Such an algorithm was given by [11]. However, our application will involve also small sets for which the confidence bounds are not sufficiently high. The next subsection gives a general technique for boosting the confidence bounds of estimation algorithms.

2.1 Estimate-focusing: A technique for boosting approximation confidence. The primary difficulty in boosting the confidence of our parallel approximate summation algorithm is that we have no way of quickly testing if a certain call to the algorithm is correct or not [3], i.e., our method is a randomized algorithm of the *Monte Carlo* type [18]. Thus, if we want a constant-time algorithm with a much higher confidence bound, it is not possible for us to apply the standard confidence-boosting technique of replicating the problem many times and taking the best estimate produced by calling our algorithm on these multiple instances in parallel.

Nevertheless, we may mimic this approach by using an alternate confidence-boosting technique, which we call *estimate-focusing*. Simply put, the idea is to replicate the problem several times, call our algorithm

[1]Interestingly, the approximate parallel prefix of Hagerup and Raman uses padded integer sorting as a subroutine, whereas the $O(\lg^* n)$-time padded integer sorting routine of the present paper uses approximate parallel prefix as a subroutine.

on these multiple instances in parallel, and then call an approximate median-finding algorithm upon the approximations. As we show in the lemma below, this simple idea significantly improves the confidence of our approximation.

LEMMA 2.1. *Let $\epsilon > 0$ be given, and let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of independent random variables such that, for a given value $s > 0$, s_i fails to satisfy $(1+\epsilon)^{-1}s \leq s_i \leq (1+\epsilon)s$ with probability $p_i \leq 1/12$, for each $i \in \{1, 2, \ldots, n\}$. If s^* is an element whose rank in S is between $n/4$ and $3n/4$, then the probability that s^* does not satisfy $(1+\epsilon)^{-1}s \leq s^* \leq (1+\epsilon)s$ is at most $(4e\mu/n)^{n/4} \leq (e/3)^{n/4}$, where $\mu = \sum_{i=1}^{n} p_i$.*

Proof. Let X denote the number of s_i's that fail to satisfy the above inequality. Observe that the ranks of all the "good" s_i's form a contiguous integral subinterval of $[1, n]$. Thus, observing that $\mu = E(X)$, we can bound the failure of s^* as a good estimate by

$$\Pr(X \geq n/4) = \Pr(X \geq (1+\delta)\mu),$$

for $\delta = n/(4\mu) - 1 \geq 2$ (since $\mu \leq n/12$). By a well-known Chernoff bound this probability is at most

$$\begin{aligned} \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} &= e^{(n/4)-\mu}(4\mu/n)^{n/4} \\ &\leq (4e\mu/n)^{n/4} \\ &\leq (e/3)^{n/4}. \end{aligned}$$

■

Sen [27] gives a constant time randomized parallel algorithm for approximate median finding; for an input set of size n, his algorithm uses n processors to find a set element s^* whose rank in the set is between $n/4$ and $3n/4$ with n-polynomial probability.[2] Sen's algorithm can be adapted to enable finding an approximate median within the same complexity bounds, with n-exponential probability.[3] We therefore have:

LEMMA 2.2. *Assume that an ϵ-accurate estimate for a number x can be computed in time T with probability at least $1/12$, using n processors. Then, using N processors an ϵ-accurate estimate for x can be computed in time $T + O(1)$ with (N/n)-exponential probability.*

Proof. The estimation algorithm is executed N/n times in parallel, to provide N/n estimates, each being ϵ-accurate with probability at least $1/12$. Sen's algorithm [27] is used to compute in constant time an approximate median among the estimates, with N-exponential probability. By Lemma 2.1, the approximate median is an ϵ-accurate estimate with probability $1 - 2^{\Omega(N/n)}$. ■

[2]We say that an event occurs with $f(n)$-polynomial probability it it occurs with probability $1 - f(n)^{-c}$ for some constant $c > 0$.

[3]We say that an event occurs with $f(n)$-exponential probability if it occurs with probability $1 - 2^{-f(n)^c}$ for some constant $c > 0$.

2.2 Computing an approximate sum. When estimating the sum of n numbers we may have some flexibility in obtaining a tradeoff between accuracy and success probability.

LEMMA 2.3. *Suppose an $\epsilon(n)$-estimate for the sum s of n numbers can be computed in time $T(n)$ using n processors with probability $\geq 1/12$. Then, using N processors and for any m, $N \geq m \geq n$, an $\epsilon(m)$-estimate for s can be computed in time $T(m) + O(1)$, with (N/m)-exponential probability.*

Proof. The input set is first duplicated m/n times to be of size m, with its sum becoming ms/n. Now, an $\epsilon(m)$-estimate for ms/n can be computed in time $T(m)$ with probability at least $1/12$. Dividing the estimate by m/n will yield an $\epsilon(m)$-estimate for s, with probability at least $1/12$. The lemma follows by Lemma 2.2. ■

Let $d > 0$ be a constant. Given a set of n numbers, a $(1/\lg^d n)$-estimate for the sum of these numbers can be computed in $O(1)$ time, using n processors, with n-polynomial probability [11]. By Lemma 2.3 we therefore have

COROLLARY 2.1. *Let $A = \{a_1, a_2, \ldots, a_n\}$ and $s = \sum_{i=1}^{n} a_i$. Using N processors an $(1/\lg^d m)$-estimate for s can be computed in $O(1)$ time with (N/m)-exponential probability, for any m, $N \geq m \geq n$.*

3 Constant-Time Approximate Prefix Sums.

Let $A = a_1, a_2, \ldots, a_n$ be a given sequence of non-negative integers. In this section we show how to produce a consistent ϵ-approximate parallel prefix sums sequence for A. Our method runs in $O(1)$ time with very high probability, albeit with a rather large number of processors. Although this method is not work-inefficient, it lays a foundation upon which we will build a very fast optimal-work method.

Our method loosely follows an approach we used in [13] for approximate parallel prefix by reducing it to an approximate version of a related problem, which we call the *summation tree* problem. In this problem one considers a balanced tree T defined "on top" of the sequence $a_1, a_2, \ldots, a_n$ and, for each internal node $v \in T$, one wishes to compute the sum, which we denote by $S(v)$, of the elements stored in v's descendants. We previously considered only the case when T was binary [13], but here we allow more general trees. Specifically, we define the *ϵ-approximate summation tree* problem, as that of producing a value $\bar{S}(v)$, for each

$v \in T$, 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

$$\bar{S}(v) \geq \bar{S}(v_1) + \cdots + \bar{S}(v_r),$$

where $v_1, \ldots, v_r$ are the r children of v in T, ordered from left to right. We use $h(T)$ to denote the height of the tree T.

In the remainder of this section we explain how we reduce the ϵ-approximate prefix sums problem to the ϵ-*approximate summation tree* problem, and we then show how to quickly solve this problem. All the steps we describe in this section will run in $O(1)$ time using a rather large number of processors. We postpone until Section 4 our techniques for solving the approximate prefix sums problem optimally.

3.1 A reduction to the approximate summation tree problem. So, suppose we have a consistent ϵ-approximate summation tree, T, built on top of the given sequence $a_1, \ldots, a_n$. We wish to use this to compute a consistent ϵ-approximate prefix sums sequence. We begin with a definition. For any leaf v in a rooted tree T 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., the nodes that are left siblings of v's ancestors in T. Our method for computing a prefix sums sequence, then, is that we define T to be a binary tree, and we compute, for each leaf $v \in T$,

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

where a_i is the value stored at v. Since we desire a constant-time method, we must of course perform this summation in constant time. Fortunately, for each $v \in T$, we need only sum $O(\lg n)$ numbers, each of which can be defined using only $O(\lg n)$ bits; hence, we may apply the following well-known lemma:

LEMMA 3.1. *Suppose one is given m non-negative integers $s_1, s_2, \ldots, s_m$, each of which is defined by a string of $l \leq b$ bits*[4] *stored in a memory cell of size b. Then one can compute $s = \sum_{i=1}^{m} s_i$ in $O(k)$ time using $O(\min\{2^{(ml)^{1/k}}, m2^{b^{1/k}}\})$ processors in the CRCW PRAM model.*

By applying this to compute each b_i as above, then, we have the following:

LEMMA 3.2. *Suppose one is given a sequence $A = a_1, a_2, \ldots, a_n$ of non-negative integers, and a consistent solution to the ϵ-approximate summation tree problem for A and a binary tree T. Then one can construct a consistent ϵ-approximate parallel prefix sums sequence $b_0 = 0, b_1, \ldots, b_n$ in $O(1)$ time using $O(n^{1+1/k})$ processors for any constant $k \geq 1$.*

Proof. The time and processor bounds follow immediately from Lemma 3.1, so let us establish the correctness of this method. Let v be the i'th leaf and v' be the $(i-1)$'st leaf. Consider the lowest common ancestor of v and v' in T, $\mathrm{lca}(v, v')$, and let w be the child of $\mathrm{lca}(v, v')$ that is an ancestor of v'. Let T' be the subtree of T rooted at w. It is easy to verify that $L(v) \setminus L(v') = \{w\}$ and that $L(v') \setminus L(v) = L_{T'}(v')$. We note that v' is the rightmost leaf of T', and therefore, as can be derived from the consistency of T (and easily proved by induction) $\bar{S}(w) \geq \sum_{u \in L_{T'}(v')} \bar{S}(u) + a_{i-1}$. Therefore,

$$\begin{aligned}
b_i - b_{i-1} &= \sum_{u \in L(v)} \bar{S}(u) + a_i - \left(\sum_{u \in L(v')} \bar{S}(u) + a_{i-1} \right) \\
&= \sum_{u \in L(v) \setminus L(v')} \bar{S}(u) - \sum_{u \in L(v') \setminus L(v)} \bar{S}(u) + a_i - a_{i-1} \\
&= \bar{S}(w) - \sum_{u \in L_{T'}(v')} \bar{S}(u) + a_i - a_{i-1} \\
&\geq a_{i-1} + a_i - a_{i-1} \\
&= a_i \ .
\end{aligned}$$

Thus, the sequence is consistent. It is easy to verify that $\sum_{j=1}^{i} a_j = \sum_{u \in L(v)} S(u) + a_i$. Therefore, since T is an ϵ-approximate summation tree,

$$\begin{aligned}
b_i &= \sum_{u \in L(v)} \bar{S}(u) + a_i \\
&\leq (1+\epsilon) \sum_{u \in L(v)} S(u) + a_i = (1+\epsilon) \sum_{j=1}^{i} a_j \ .
\end{aligned}$$

Thus, the sequence is an ϵ-approximate parallel prefix sequence. ∎

Therefore, if T is a binary tree, then it is sufficient for us to construct a consistent ϵ-approximate summation tree for T in order to produce a consistent solution to the approximate prefix sums problem. Producing an ϵ-approximate summation tree is complicated by a number of factors, however, not the least of which is that previous summation approximation schemes [11, 12, 15] do not yield consistent sums.

3.2 Achieving consistency. Suppose we have an ϵ-approximate binary summation tree T. That is, for

[4]Note: we do not require that $s_i \leq 2^l$, just that there is some consistent way to represent each s_i using l bits.

each node v of T we have an approximation $S'(v)$, such that $S(v)(1+\epsilon)^{-1} \leq S'(v) \leq S(v)(1+\epsilon)$. We showed in [13] that if we define $\bar{S}(v) = (1+\epsilon)^{2i} S'(v)$, then the $\bar{S}(v)$'s define a consistent $(4\epsilon \lg n)$-approximate binary summation tree. 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 [11, 12, 15]—they may return some inaccurate approximate partial sums (albeit with small probability).

In using one of these or a similar method for approximate sums it would therefore be desirable if we could test if the probabilistic method returned a correct approximation. Unfortunately, we cannot, in general, quickly test if a node in T satisfies the ϵ'-approximate condition, for that would require an exact summation, which of course has a near-logarithmic time lower bound [3]. Nevertheless, we can test a local condition that will be sufficient for our purposes.

3.3 Overcertification: Using a local condition for approximate summation trees.

We say that a summation tree T is *locally ϵ-bounded* if, for every internal node $v \in T$,

$$\left(\sum_{i=1}^{r} S(v_i)\right)(1+\epsilon)^{-1} \leq \bar{S}(v) \leq \left(\sum_{i=1}^{r} S(v_i)\right)(1+\epsilon),$$

where $\bar{S}(v)$ denotes the approximate sum stored at v, and $v_1, \ldots, v_r$ denote v's children. Note that an ϵ-approximate summation tree is automatically locally ϵ-bounded. Also note that the converse of this statement need not be true. Nevertheless,

LEMMA 3.3. *If T is a locally ϵ-bounded summation tree, then T is a $(2\epsilon\, h(T))$-approximate summation tree.*

Proof. Let $\bar{S}(v)$ denote the approximate sum for v in T. By a simple inductive argument, $S(v)(1+\epsilon)^{-i} \leq \bar{S}(v) \leq S(v)(1+\epsilon)^{i}$, where i is the height of v in T. Given this, we can show, by well-known inequalities, for $0 \leq \epsilon \leq 1$,

$$(1+\epsilon)^{i} \leq (1+\epsilon)^{h(T)} \leq e^{\epsilon\, h(T)} \leq 1 + 2\epsilon\, h(T).$$

∎

Thus, we have a local condition that implies the desired global approximation condition (assuming the ϵ in Lemma 3.3 is small enough relative to the desired global ϵ). This local condition provides a means to test in constant time if an approximate summation binary-tree that is a basis for computing the output sequence is a sufficiently good one, which amounts to an instance of the technique we call overcertification. We also have a scheme for converting an inconsistent approximate summation tree into a consistent approximation tree. Thus, we have only yet to show how to construct a locally ϵ-bounded summation tree.

To construct such a tree using $N \geq n^{1+1/k}$ processors we assign Nn_v/n processors to each node v in T, where n_v denotes the number of leaf descendents of v, and we call Corollary 2.1 to produce an approximate sum, for each $v \in T$ in parallel, of the values stored at v's leaf descendents. Each such sum has accuracy $\epsilon = 1/\lg^c N$ with (N/n)-exponential probability. Since there are $O(n)$ nodes in T, all of these calls succeed with n-polynomial probability. Moreover, if all these calls succeed, then T is locally ϵ-bounded. Thus, it is sufficient to test this local condition to decide if we need to repeat the parallel application of Corollary 2.1 or not. Therefore, we have the following:

THEOREM 3.1. *Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence of non-negative integers. Then one can compute an $(8\epsilon \lg^2 n)$-approximate prefix sums sequence for A in time that is $O(1)$ with n-polynomial probability using N processors in the CRCW PRAM model, where $\epsilon = 1/\lg^k N$ and $N \geq n^{1+1/k}$, for any fixed constant $k \geq 1$.*

4 Optimal Approximate Prefix Sums.

While the method of the previous section is extremely fast, running in constant time with very high probability, it is also quite inefficient, requiring $O(n^{1+1/k})$ processors for a constant $k \geq 1$. In this section we show how to develop a more efficient algorithm. Not surprisingly, our method involves the use of techniques somewhat more sophisticated than those used to derive the constant-time inefficient method. To obtain our main result, we design a series of refined algorithms, beginning with one that runs in constant time with $O(n \lg^c n)$ processors, for some constant $c \geq 1$.

4.1 An $O(n \lg^c n)$-work method.

In the previous section we gave a constant time method for constructing an ϵ-approximate prefix sums sequence using $O(n^{1+1/k})$ processors for some constant $k \geq 1$. In this subsection we show how to use this method to derive a more work-efficient constant-time method.

Our method mimics our work-inefficient method at a high level in that we first construct a consistent approximate summation tree and then use that to build an approximate prefix sums sequence. The main idea is that instead of using a binary tree, we let T be a balanced doubly logarithmic height tree [6]. Such a tree is defined so that any internal node of height $h > 1$ has $2^{2^{h-2}}$ children. An internal node of height one has two children. This implies that the number of leaves in the rooted subtree of any internal node of height $h > 1$ is $2^{2^{h-1}}$ (that is, the square of the number of its children). Clearly, the height of the tree is at most $\lg\lg n + 1$. The leaves of the tree correspond to the n inputs of the problem, and, at a high level, we use it to compute an approximate summation tree as follows:

Step 1. For each internal node v we assign $O(n_v \lg^c n)$ processors, and we compute an ϵ-approximate sum $S'(v)$, where $\epsilon = 1/(\lg\lg n)^c$ for some sufficiently high constant $c \geq 1$. Each such call succeeds with $(\lg^c n)$-exponential probability, so that all of these calls succeed with n-polynomial probability.

Step 2. For each internal node v we define $\bar{S}(v) = S'(v)(1+\epsilon)^{4i}$, where i is the height of v in T (the reason for this scaling will become clear below), and we compute an approximate prefix sums sequence over the $\bar{S}(u)$ values stored at v's children, producing an estimated prefix sum, $s'(u)$ for each child u of $v \in T$. We use the method of the previous section to implement this step for each v in parallel using $O(n_v \lg^c n)$ processors per node $v \in T$ (recall that the number of children of v is $O(\sqrt{n_v})$). If any of these calls takes longer than the expected $O(1)$ time, then we abort this step and start the computation over again with Step 1. Of course, all of these calls succeed with n-polynomial probability.

Step 3. We produce an estimate $\bar{s}(u)$ for each $s'(u)$ value by applying the bit-thinning technique to round $\bar{s}(u)$ so that the number of significant bits needed to represent any $\bar{s}(u)$ is $O(\lg\lg n)$ [24]. This will, of course, cost us at most a factor of $(1+\epsilon)$ in our approximation factor. It is only at this point that we check the tree T for *consistency*, which in this case we define so that we must satisfy, for each $v \in T$, the condition

$$\bar{S}(v) \geq \bar{s}(v_r),$$

where v_r is the right-most child of v. If any node fails, then we start the computation over again with Step 1.

Step 4. For the i'th leaf v compute $b_i = \sum_{u \in L_T(v)} \bar{s}(u) + a_i$ deterministically in constant time using $O(\lg n)$ processors, by Lemma 3.1.

We have:

Lemma 4.1. *The above method constructs a consistent $(8\epsilon h(T))$-approximate summation tree T.*

Proof. Let a node $v \in T$ have children $v_1, \ldots, v_r$. By the above construction we have

$$\begin{aligned}
\bar{s}(v_r) &\leq s'(v_r)(1+\epsilon) \\
&\leq (\bar{S}(v_1) + \cdots + \bar{S}(v_r))(1+\epsilon)^2 \\
&= S'(v_1)(1+\epsilon)^{4i-2} + \cdots + S'(v_r)(1+\epsilon)^{4i-2} \\
&\leq S(v_1)(1+\epsilon)^{4i-1} + \cdots + S(v_r)(1+\epsilon)^{4i-1} \\
&= S(v)(1+\epsilon)^{4i-1} \\
&\leq S'(v)(1+\epsilon)^{4i} \\
&= \bar{S}(v),
\end{aligned}$$

which establishes our consistency condition.

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

$$(1+\epsilon)^{4i} \leq (1+\epsilon)^{4h(T)} \leq e^{4\epsilon h(T)} \leq 1 + 8\epsilon h(T),$$

which establishes the claimed approximation factor. ∎

Thus, we have the following:

Lemma 4.2. *One can solve the ϵ-approximate prefix sums problem in $O(1)$ time with n-polynomial probability for $\epsilon = 1/(\lg\lg n)^c$ using $O(n \lg^c n)$ processors, where $c \geq 1$ is some constant.*

Thus we have significantly improved the work for our computation. We can do even better, however.

4.2 Further improvements. By using a divide-and-conquer method based upon the above algorithm, we can achieve a very fast parallel method with a near-optimal work bound.

Let $m = \lg^c n$, where c is a sufficiently large constant to be determined in the analysis. The main idea of our method is to divide the input sequence into n/m sequences of size $O(m)$ each, and recursively solve the approximate prefix sums sequence problem for each in parallel. Then, using the method of Lemma 4.2, we compute an approximate prefix sums sequence on the sums returned from the recursive calls. This requires only $O(n)$ processors if m is big enough, and we can then pass down the appropriate prefix sums to the recursively-computed prefix sums so as to produce a solution for the entire problem. This, too, can be done in $O(1)$ time using n processors. As for the time bound, it can be characterized by the recurrence relation:

$$T(n) = T(\lg^c n) + O(1),$$

which implies that $T(n)$ is $O(\lg^* n)$.

The only problem with this approach is that our merge algorithm is randomized, not deterministic, as this approach seems to require. The difficulty is that the probability that our algorithm succeeds is directly proportional to the number of processors used, which in this case is proportional to the problem size. Thus, if we inductively assume that each subproblem terminates in $T(m)$ steps with polynomial probability, that polynomial is with respect to m, not n. So we can only assume that a subproblem terminates with $(\lg n)$-polynomial probability. Fortunately, however, we can "ramp" up this probability to be n-polynomial by using the thinning-out principle [22], which is also known as the failure-sweeping technique [8].

If we inductively assume that a subproblem terminates after $T(m)$ steps with m-polynomial probability, then after $T(m)$ steps the number of subproblems still active is at most n/m^b, for some constant b, with n-exponential probability. We can then use linear approximate compaction [23] to allocate $\lg n$ processors to each of them. By selecting $c = 2/b$ we obtain that this it is an instance of linear approximate compaction problem with sparse input and can therefore be implemented in constant time (e.g., see [12]). Finally, we can run our method of Lemma 4.2 $\lg n$ times in parallel for each active problem, to have them all successfully solved in $O(1)$ time, with n-polynomial probability. This can be implemented by using the testing technique, based on the local condition. Thus, we have an algorithm that runs in $O(\lg^* n)$ time, with n-polynomial probability, using n processors in the CRCW PRAM model.

All that remains, then, is to analyze the approximation factor for this algorithm, which we can characterize using the following recurrence:

$$\epsilon(n) = (1 + 1/(\lg\lg n)^c)\,\epsilon(\lg^c n),$$

which is bounded by $1+\epsilon$, where $\epsilon \leq O(1/(\lg\lg b)^c)$, where b is the number of processors available to us when we must solve the base problem in the recursion and c is a chosen constant. This parameter ϵ is currently, of course, not $o(1)$, nor is our method work optimal. We can easily deal with both of these issues, however, as we show next.

4.3 The optimal solution. To achieve an optimal algorithm we produce an initial partitioning of the input sequence into $O(n/m)$ sequences of size $O(m)$ each, where m is $2^{O(\lg^* n)}$, and we apply the well-known deterministic method to produce an exact prefix sums sequence for each subproblem in $O(\lg^* n)$ time. We then apply the above method to the sequence of sums produced from each subproblem. This allows us to assign $O(2^{\lg^* n}/\lg^* n)$ processors for each element in the sequence; hence, it allows us to achieve an approximation factor of $1+\epsilon$, where ϵ is $O(1/\lg\lg^* n)$, which is, of course, $o(1)$, as desired. In addition, it also allows us to implement our $O(\lg^* n)$-time method using only $O(n/\lg^* n)$ processors, which is optimal. Therefore, we have the following:

THEOREM 4.1. *Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence of non-negative integers. One can produce an ϵ-approximate prefix sums sequence for A in time that is $O(\lg^* n)$ with n-polynomial probability using $O(n/\lg^* n)$ processors on a CRCW PRAM, where ϵ is $o(1)$.*

5 Padded Integer Sorting.

One of the most common applications of parallel prefix is for allocation of resources, where the allocation is by order of requests.

5.1 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.

By Theorem 4.1 we get

THEOREM 5.1. (optimal ordered allocation) *The ordered allocation problem for a sequence of size n can be solved in $O(\lg^* n)$ time with n-polynomial probability, using $n/\lg^* n$ processors.*

5.2 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.

In [13] we have shown how to use an algorithm for approximate parallel prefix to obtain an algorithm for padded integer sorting that is slower by a factor of $O(\lg^* n)$, with high probability. Using the approximate parallel prefix algorithm of Theorem 4.1 we therefore get a padded integer sorting algorithm that runs in $O((\lg^* n)^2)$ time with high probability, using an optimal number of processors.

The algorithm in [13] is based on a modification of the integer chain sorting algorithm of [11, Sec. 9]. We now show that the latter algorithm can be used to obtain $O(\lg^* n)$ time algorithm for padded integer sorting.

Let Φ_j be the set of elements of value j, $j = 1, \ldots, n$. The integer-chain sorting algorithm in [11, Sec. 9] consists of $t = O(\lg^* n)$ iterations. At iteration i, an array A_i of size $cn/2^i$ is used. For each set Φ_j an interval D_j^i of size $d_j^i = O(\mu_j^i)$ may be allocated in A_i, where μ_j^i is an estimate for $|\Phi_j|$ computed in this step, and a subset $\Phi_j^i \subset \Phi_j$ is injectively mapped into D_j^i. At the end of the $t = O(\lg^* n)$ iterations, each element in Φ_j is mapped into a private cell in one of the intervals D_j^i.

Let $\bar{d}_j^i = \sum_{k=1}^{i} d_j^i$, and $d_j = \bar{d}_i^t$. The prefix sums $\left\{\bar{d}_j^i\right\}_i$, $j = 1, \ldots, n$, can be obtained by straightforward modifications of the integer chain sorting algorithm. After the execution of this algorithm, we can obtain the padded integer sorting sequence in an array B of size $2cn$, as follows:

Step 1. Allocate from B to each set Φ_j a private interval B_j of size d_j, so that the allocation is ordered according to j.

Step 2. For each j, allocate from B_j to each subset Φ_j^i a private sub-interval of size d_j^i, so that the allocation is ordered according to i. B_j^i

Step 3. Copy the contents of each interval D_j^i into interval B_j^i.

It is easy to verify that the input elements are indeed sorted in the array B. Step 1 is implemented by the ordered allocation algorithm of Theorem 5.1. Step 2 is implemented using the prefix sums sequences $\left\{\bar{d}_j^i\right\}_i$, $j = 1, \ldots, n$. Step 3 is trivial. We have

THEOREM 5.2. (optimal padded integer sorting) *The padded integer sorting problem can be solved in time $O(\lg^* n)$, with n-polynomial probability, and linear space using $n/\lg^* n$ processors.*

6 Applications.

It had been shown in [13] that a number of well-known problems in parallel computational geometry can be solved efficiently and very fast by reductions to padded sort. We apply the new result of padded integer sorting to get improved results. 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.

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. Using an algorithm by [8], we showed in [13] that the problem can be solved in time $O(\lg^* n)$ plus the time for padded integer sorting, using an optimal number of processors. We therefore have an optimal-work integer-coordinate convex hull algorithm that runs in $O(\lg^* n)$ with very high probability. In the full version we show how to use this to determine the separability of two integer-coordinate point sets in $O(\lg^* n)$ time with high probability using n processors.

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. The problem can be solved in $O(1)$ steps plus the time needed for padded integer sorting using $O(n \lg n)$ work, with n-polynomial probability [13].

In the full version we also show how padded sort can be used to solve the planar *dominance counting* problem, where one wishes to determine the number of points in S dominated by point p in S.

Approximate selection revisited. In the full version we show how to apply our overcertification and estimate-focusing techniques to improve Sen's [27] parallel approximate median routine to be able to an element with rank in $[(1+\epsilon)^{-1}k, (1+\epsilon)k]$, for any $k \in \{1, 2, \ldots, n\}$ in constant time with n-polynomial probability, using n processors, with ϵ being $o(1)$.

Acknowledgment. We would like to thank Rajeev Raman for some helpful comments regarding padded integer sorting.

References

[1] Leonard M. Adleman. Two theorems on random polynomial time. In *Proc. of the 19th IEEE Annual Symp. on Foundation of Computer Science*, pages 75–83, 1978.

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

[3] Paul Beame and Johan Håstad. 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.

[4] Omer Berkman, Dany Breslauer, Zvi Galil, Baruch Schieber, and Uzi Vishkin. Highly parallelizable problems. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing*, pages 309–319, 1989.

[5] Omer Berkman, Yossi Matias, and Prabhakar L. Ragde. Triply-logarithmic upper and lower bounds for minimum, range minima, and related problems with integer inputs. In *Proc. of the Third Workshop on Algorithms and Data Structures, Springer LNCS 709*, pages 175–187, 1993.

[6] Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14:344–370, 1993.

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

[8] M. Ghouse and Michael T. Goodrich. In-place techniques for parallel convex hull algorithms. In *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 192–203, 1991.

[9] Joseph Gil. Fast load balancing on a PRAM. In *Proc. of the 3rd IEEE Symposium on Parallel and Distributed Computing*, pages 10–17, December 1991.

[10] Joseph Gil and Yossi Matias. Fast hashing on a PRAM—designing by expectation. In *Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1991.

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

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

[13] Michael T. Goodrich, Yossi Matias, and Uzi Vishkin. Approximate parallel prefix computation and its applications. In *Proc. of the 7th International Parallel Processing Symposium*, pages 318–325, 1993.

[14] Torben Hagerup. Constant-time parallel integer sorting. In *Proc. of the 23rd Ann. ACM Symp. on Theory of Computing*, pages 299–306, 1991.

[15] Torben Hagerup. Fast parallel space allocation, estimation and integer sorting. Technical Report 03/91, SFB 124, Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, Germany, 1991.

[16] Torben Hagerup and Rajeev Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *Proc. of the 33rd IEEE Annual Symp. on Foundation of Computer Science*, pages 628–637, 1992.

[17] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley (Reading, Mass.), 1992.

[18] Richard M. Karp and Vijaya L. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–941. North-Holland, Amsterdam, 1990.

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

[20] Philip D. MacKenzie. Load balancing requires $\Omega(\lg^* n)$ time. In *Proc. of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 94–99, 1992.

[21] Philip D. MacKenzie and Quentin F. Stout. Ultra-fast expected time parallel algorithms. In *Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 414–423, 1991.

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

[23] Yossi Matias and Uzi 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*, pages 307–316, 1991.

[24] Yossi Matias, Jeffrey S. Vitter, and Neal E. Young. Approximate data structures with applications. These proceedings.

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

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

[27] Sandeep Sen. Finding an approximate median with high probability in constant parallel time. *Information Processing Letters*, 34:77–80, March 1990.

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