Using Approximation Algorithms to Design Parallel Algorithms that May Ignore Processor Allocation

(Preliminary Version)

Michael T. Goodrich*

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

We present a framework for designing parallel algorithms that may ignore processor allocation. We develop a number of fast approximation algorithms, and show how to use these algorithms to simulate any algorithm that fits this framework in a work-preserving fashion on a randomized CRCW PRAM. We also give several applications of our approach to parallel computational geometry.

1 Introduction

A fundamental goal of parallel computation research is to design fast parallel algorithms with small time-processor products (e.g., see [32]). That is, given a parallel computation described by the tasks that some P processors are to perform in T steps, one desires that both T and $T * P$ be small. In fact, if $T * P$ matches a sequential lower bound, then the algorithm is said to be *optimal* [32]. Sometimes, a parallel algorithm "almost" achieves this efficiency goal in that T and W, the sum of all the task lengths (which is usually called the *work* performed by the algorithm), are small, but $T * P$ is not. Fortunately, Brent [9] gives a scheme for converting any algorithm that uses W work into an equivalent algorithm using P processors in time $O(\lfloor W/P \rfloor + T)$. This scheme is only a "meta-theorem" [27] for our purposes, however, in that, in order to apply it on a PRAM, where synchronous processors share a common memory, one must meet two important conditions:

1. One must know n_i, the number tasks active in step i.
2. One must be able to assign P processors to these tasks in $O(\lceil n_i/P \rceil)$ time.

Historically, Condition 2 (processor allocation) has often complicated the design of what would have otherwise been a simple parallel algorithm. Consider, for example, that nearly a decade elapsed before a PRAM implementation of the fast parallel list-merging algorithm of Valiant[1] [50] appeared in the literature [7].

We propose a framework that allows the algorithm designer to assume that processor allocation is handled by the PRAM's "operating system." One of the motivations for this framework is that, while processor allocation is a challenge in PRAM algorithms, it is not necessarily a bottleneck in implementing algorithms on existing parallel machines (e.g., see [48]).

We show how to simulate any algorithm written in our framework in a work-preserving fashion on a randomized CRCW PRAM[2] at a very small cost in time complexity. Moreover, as a demonstration of the power of our approach, we give several applications to parallel computational geometry.

2 Approximation Algorithms

Our simulation theorem depends on the fast solution to a number of approximation problems, which we address in this section.

2.1 Approximate Density Partitioning

Let A be an n-element array with some of its elements marked as *distinguished*. We refer to such an array as *bichromatic*, and let $d(A)$ denote *density* of A, i.e., the number of distinguished elements in A. The first problem we address is the following:

Approximate density partitioning: partition A into subarrays $A_1, A_2, \ldots, A_r$ such that the density of each subarray is approximately ρ, where $\rho < d(A)$.

The method we describe here solves this problem (deterministically) in $O(1)$ time using $O(n)$ processors, and produces a partitioning such that $\rho \leq d(A_k) \leq \rho^4 \log^2 n$, for each k. It makes extensive use of a powerful theorem due to Ragde:

*This research was supported by NSF Grant CCR-9003299 and by NSF/DARPA Grant CCR-8908092.

[1] Valiant's model only counted comparison steps, ignoring overhead costs such as processor allocation.

[2] Write conflicts can be resolved arbitrarily.

Theorem 2.1 (Ragde [38]): *Given an integer $k \leq n^{1/4}$, one can either determine that $d(A) > k$ or produce a 1-to-1 mapping of the distinguished elements in A to a k^4-element array B in $O(1)$ time using n processors on a CRCW PRAM.*

We begin our method by dividing A into $N = n/\lceil \log n \rceil$ subarrays $\tilde{A}_1, \ldots, \tilde{A}_N$ of size $\lceil \log n \rceil$ each. We compress each subarray $\tilde{A}_k$ into a single bit b_k, which is 1 if and only if $\tilde{A}_k$ is non-empty (this can easily be tested for all the subarrays in $O(1)$ time using $O(n)$ processors). Let B denote the resulting array of size N.

We initially partition B into subarrays $B_1, B_2, \ldots, B_{N/\rho^4}$, of size ρ^4 each, and we build a complete binary tree T "on top" of this partitioning, so that each leaf is associated with a subarray B_k. We let $B[v]$ denote the subarray of B spanning all the descendents of v, for each internal node v in T. With each node v in T we also associate an array C_v of size ρ^4. In parallel for each v in T, we attempt to write the distinguished elements in $B[v]$ to C_v using Theorem 2.1. (This can be done in $O(1)$ time using $O(N \log N) = O(n)$ processors.) We say that a node v is *good* if the mapping into C_v fails, but the mapping into C_w succeeds for each of v's descendents w. We then say that a node is a *partition* node if it contains one good descendent but its parent has at least two. The union of all the $B[v]$ arrays for partition nodes forms the desired partition of B (if there are no partition nodes, then we define the root as a partition node). We may then expand this partitioning into a partitioning of A by reversing the compression we performed to form B, thereby converting each $B[v]$ into a corresponding $A[v]$. (Note that we can determine all the good nodes and partition nodes in $O(1)$ time using $O(N \log N) = O(n)$ processors; hence the entire method can be implemented in these bounds.) By adding up all the nodes in T on the fringe of the path from w to v that contribute distinguished elements to $B[v]$, one can show that, for each parition node v in T, $\rho < d(B[v]) \leq \rho^4 \log n$. This, in turn, implies the following:

Theorem 2.2: *Given an n-element bichromatic array A, one can partition A into subarrays $A_1, A_2, \ldots, A_r$ such that, for any $\rho < d(A)$, $\rho < d(A_k) \leq \rho^4 \log^2 n$, for each A_k, in $O(1)$ time using $O(n)$ processors on a CRCW PRAM.*

Note: the above indexing of subarrays as A_1, A_2, and so on, is only a notational convience, for the processors assigned to the elements of A_k do not know the value of k, nor do they need to (for they are all associated with the same node in T).

2.2 Approximate Compaction

In this subsection we show how to apply Theorem 2.2 to solve a related approximation problem:

Approximate compaction: map the distinguished elements of a bichromatic array A one-to-one to an array B of size $m \geq (1+\alpha)d(A)$, for some $\alpha \geq 0$.

The parameter α is referred to as the *expansion parameter*, and is desired to be as small as possible. Of course, we can use Theorem 2.1 to solve this problem for $\alpha \geq \min\{d(A)^3, n/d(A)\}$ in $O(1)$ time using n processors, but this α can be quite large. Matias and Vishkin [35] show how to solve it for $\alpha \geq 3$ with a method that runs in $O(\log^* n)$ time with high probability[3], using n processors on a randomized CRCW PRAM. We solve this problem for any constant expansion factor $\alpha > 0$ with a method that almost surely[4] runs in $O(\log^* n)$ time using $O(n/\log^* n)$ processors[5] in the same model (or almost surely in $O(1)$ time for some alternate formulations).

2.2.1 A Leveraged Approach

Our method consists of four phases, where Phase 1 is a density partitioning step, Phase 2 is a "dart throwing" [28] procedure, Phase 3 is a "processor throwing" procedure, and in Phase 4 we use the successful processors from Phase 3 as a "lever" against the unsuccessful darts from Phase 2.

Our algorithm assumes the existence of an oracle, $\mathcal{D}$, that returns $d(\hat{A})$ for any polylog(n)-sized array $\hat{A}$. It also assumes the existence of an algorithm, $\mathcal{S}$, that maps the distinguished elements in a polylog(n)-sized array $\hat{A}$ to the distinguished elements in an polylog(n)-sized array $\hat{B}$, of size at

[3] By "high probability" we mean that the probability is at least $1 - 1/n^c$ for some constant $c > 0$.

[4] By "almost surely" we mean that the probability is at least $1 - 1/c^{n^d}$ for constants $c > 1$ and $d > 0$.

[5] We have recently discovered that this result was independently discovered by T. Hagerup [29] using a method different from ours.

least $3d(\hat{A})$, provided at least two-thirds of the elements in $\hat{B}$ are distinguished.

We begin our method by applying Theorem 2.1 to try to map the distinguished elements in A to an array C of size $\log^4 n$. If this is successful (so that $d(A) \leq \log^4 n$), then we apply $\mathcal{S}$ to C to map the distinguished elements of C to B (in which case we are done). So, let us suppose this application of Theorem 2.1 is unsuccessful, i.e., that $d(A) > \log n$.

Phase 1. We apply Theorem 2.2, with $\rho = \log n$, to partition A into subarrays $A_1, A_2, \ldots, A_r$ such that $\log n < d(A_k) \leq \log^6 n$ (so that $r \leq n/\log n$). We then map each A_k into an array $\hat{A}_k$ of size $\min\{\log^{24} n, |A_k|\}$ by Theorem 2.1.

Phase 2. We assign a processor to each element a_i in an $\hat{A}_k$. If a_i is distinguished, then the processor for a_i selects a random location b_j in B, and sets $b_j := a_i$ if b_j is empty (the processor has just "thrown" a_i at B.) Each processor iterates this assignment until it succeeds or it has made h_1 attempts (where h_1 is to be set in the analysis).

Phase 3. Call $\mathcal{D}$ on each $\hat{A}_k$ to compute $m_k = d(\hat{A}_k)$, and let C_k be an array of size $\alpha m_k/2$. Each entry c_i in C_k has an initially-zero field, $c_i.index$. We assign a processor to each element in C_k, and have this processor choose a random location b_j in B and write its **id** to b_j if b_j is empty (the processor has just thrown "itself" at B). If the processor is successful, then it writes j to the *index* field of entry c_i in C_k to which it is assigned, and then drops out of Phase 3. We repeat this procedure for h_2 rounds (where h_2 is to be set in the analysis).

Phase 4. We apply $\mathcal{S}$ to map the unmapped distinguished elements in $\hat{A}_k$ to the entries of C_k with non-zero *index* fields. If this mapping is successful, we write each remaining element a_i in $\hat{A}_k$ to the location in B whose index is stored in $c_j.index$, where c_j is the target of a_i in C_k. (Note that, if we let m'_k denote the number of remaining distinguished elements in $\hat{A}_k$, then this call of $\mathcal{S}$ will be successful provided $m'_k \leq \alpha m_k/6$ and at least two-thirds, i.e., $\alpha m_k/3$, of the entries in C_k have non-zero *index* fields.)

We complete the procedure by "zeroing out" any entries in B that still contain a processor's id after Phase 4. We described this algorithm to allow for it to fail, but one can easily modify it so that it never fails (given $m \geq (1+\alpha)d(A)$), via repetition, since failure is easily tested. Assuming that it performs correctly, its time complexity is $O(1 + \mathcal{T}(\log^{24} n))$ using $O(n + \sum_{k=1}^{r} \mathcal{P}(|\hat{A}_k|))$ processors, where $\mathcal{T}(*)$ and $\mathcal{P}(*)$ are the time and processor bounds for $\mathcal{D}$ and $\mathcal{S}$.

2.2.2 Establishing Almost Certainty

Let us therefore analyze the probability that this procedure succeeds, assuming $m = (1+\alpha)d(A)$. Since the number of processors throwing darts in Phase 2 is $d(A)$, the probability that a particular processor fails is $p_1^{h_1}$, where $p_1 = 1/(1+\alpha)$ (we assume the processor fails if its dart collides with another). Thus, the expected value of m'_k is $p_1^{h_1} m_k$, for $k = 1, 2, \ldots, r$. Similarly, since the number of processors throwing "themselves" in Phase 3 is $\sum_{k=1}^{r} \alpha m_k/2 = \alpha d(A)/2$, the probability that a particular processor fails is $p_2^{h_2}$, where $p_2 = (1+\alpha/2)/(1+\alpha)$. Thus, the expected value of m''_k, the number of successful processors assigned to $\hat{A}_k$ in Phase 3, is $(1 - p_2^{h_2})m_k$. By the Chernoff bounds [11] of Raghavan[6] [39], we can set h_1 and h_2 so that the probability that $m'_k \leq \alpha m_k/6$ and $m''_k \geq \alpha m_k/3$, for each $k = 1, 2, \ldots, r$, is at least $1 - 1/n^{b_0}$ for any constant $b_0 > 0$ (which we establish in the full version). This, in turn, implies that we can force the entire procedure to succeed with probability $1 - 1/n^b$ for any constant $b > 0$ (by taking $b_0 = b + 1$).

In fact, we can ramp this probability up further, to be almost sure, by applying the *failure sweeping* technique of Ghouse and Goodrich [22]:

Case 1: $m \leq n^{1/5}$. In this case, we preface our method by an application of Theorem 2.1 (which must succeed in this case) to map the distinguished elements of A to an array C of size $n^{4/5}$. We make $O(n^{1/5})$ copies of C and B and perform our approximate compaction method on each pair of copies in parallel. The probability that our method succeeds for at least one of these is at least $1 - 1/n^{bn^{1/5}}$.

Case 2: $m > n^{1/5}$. In this case we perform our approximate compaction algorithm as described above. After it completes, however, we then apply Theorem 2.1 to map any remaining unmapped elements from A to an array C of size $m^{4/5}$. Note that this mapping succeeds if m', the number of remaining unmapped elements from A, is at most

[6] Raghavan [39] shows that if X is the number of 1's in a collection of independent Bernoulli trials, then $\Pr(X > (1+\delta)\mu) < [\exp(\delta)/(1+\delta)^{1+\delta}]^{\mu}$ and $\Pr(X < (1-\delta)\mu) < [\exp(-\delta)/(1-\delta)^{1-\delta}]^{\mu}$, where $\mu = E(X)$.

$m^{1/5}$, an event which (as we show in the full version) occurs almost surely. Since there are at least $(\alpha/(1+\alpha))m$ free positions left in B, we then assign $(\alpha/(2+2\alpha))m^{1/5}$ processors to each element in C and perform h_3 more dart throwing attempts, where h_3 is a constant set so that the probability of success is at least $1 - 1/c^{n^{1/25}}$ for any constant $c > 1$.

2.2.3 A Suboptimal Implementation

Having established the probability bounds, let us now give implementations for $\mathcal{D}$ and $\mathcal{S}$, both of which are based on solutions to the following problem:

Multiple-array approximate compaction: given μ (compressed) arrays, $A_1, A_2, \ldots, A_\mu$, each of size at most λ, map the ν elements of these arrays into an array B of size $(1+\alpha)\nu$, where $\alpha \geq 0$.

Let $t_{ac}(n)$ denote the time needed to compute $d(A)$ and perform approximate compaction with expansion parameter α for a bichromatic array A, using $p_{ac}(n)$ processors.

Lemma 2.3: *One can solve multiple-array approximate compaction with expansion parameter α in $O(t_{ac}(\mu))$ time on a CRCW PRAM with $O(p_{ac}(\mu)\log\lambda + (\nu + \lambda\log\nu\log\lambda)/t_{ac}(\mu))$ processors.*

Proof: The proof is based on a reduction similar to that used by Chandra et al. [10] for summation. Construct an $\mu \times \lceil\log\lambda\rceil$ table L, where $L[i,j]$ contains the j-th bit of $|A_i|$, and associate a subarray of A_i of size 2^j with $L[i,j]$ in the natural way if this bit is 1. (This can easily be done in $O(1)$ time using $\nu+\mu\log\lambda$ processors.) Compute l_j, the number of 1's in column j of L, and use approximate compaction to compress column j into an array of size $(1+\alpha)l_j$. This induces a compression of the subarrays associated with column j into an array of size $(1+\alpha)l_j 2^j$. We can then compress these arrays into a single array, by computing the parallel prefix sums of all the $l_j 2^j$ values (and then multiplying by $1+\alpha$). We can compute this parallel prefix in $O(1)$ time by the reduction of [10] to a simple table look-up for summation, using an exponential number of processors for each bit in the answer. In our case, this requires $O(\lambda\log\nu\log\lambda)$ processors, since there are $\lceil\log\lambda\rceil$ values whose sum can be represented with $\lceil\log\nu\rceil$ bits. □

The proof of Lemma 2.3 involved a reduction to density computation and approximate compaction via the computation of a small number of prefix sums. But the computation of prefix sums, itself, can be used for density computation and approximate compaction (in fact, we get $\alpha = 0$ in this case). Thus, we can use the method of [10] as the density-computing oracle $\mathcal{D}$, which, since it is always applied to polylog(n)-sized subarrays, requires $O(n^\delta \log\log n)$ processors for any constant $\delta > 0$. The proof of the following lemma uses a similar "brute force" approach to derive an implementation for the procedure $\mathcal{S}$.

Lemma 2.4: *One can implement the procedure $\mathcal{S}$ in $O(1)$ time using $O(n^\epsilon)$ processors on a CRCW PRAM, for any constant $\epsilon > 0$.*

Proof: Recall that the problem is to map the distinguished elements of a polylog(n)-sized array A to the distinguished elements of a polylog(n)-sized array B (where $|B| \geq 3d(A)$ and $d(B) \geq 2|B|/3$). Note that it is sufficient to compress A and B, respectively, and then match-up these two compressed arrays. To perform this compression, for, say A, we divide A into $\log n$ subarrays, compress each subarray recursively in parallel, and then merge these compressed arrays into a single compressed array. This merging step is exactly the multiple-array compaction problem; and, if we perform this merge $\delta\log n$ subarrays at a time using the above brute-force prefix-sum computation, where δ is some constant such that $0 < \delta < \epsilon$, then we can still achieve a running time of $O(1)$ time. Thus, by applying Lemma 2.3 with $\mu = \delta\log n$, $\nu = \log^c n$, and $\lambda = \log^c n$, for some constant $c \geq 1$, then our method requires $O(n^\delta \log\log n + \log^c n(\log\log n)^2) = O(n^\epsilon)$ processors. □

This, in turn, leads to the following:

Theorem 2.5: *One can perform approximate compaction for any constant expansion parameter $\alpha > 0$ in $O(1)$ time with probability $1 - 1/c^{n^{1/25}}$ using $O(n^{1+\epsilon}/\log n)$ processors[7] on a randomized CRCW PRAM, for any constants $c > 1$ and $\epsilon > 0$.*

[7] We note that we could have simply written the processor bound as n^ϵ; we give the bound as we did here in order to simplify our discussion in Section 2.3.

2.3 Density Approximation

One of the bottlenecks in the proof of Theorem 2.5 is the computation of $d(A)$. Our method for eliminating this bottleneck is to content ourselves with an approximation to $d(A)$. Thus, let us address the following problem:

Density approximation: given a bichromatic array A, compute an approximation for $d(A)$.

The best previous constant-time method for solving this problem is based on a randomized tournament paradigm (see, e.g. [28, 41, 44]), and gives an estimate that is within a polylogarithmic factor of $d(A)$ with high probability, using $O(n \log n / \log\log n)$ processors [28]. We give a constant-time procedure that produces an approximation that is almost surely within the interval $[(1-\beta)d(A), (1+\beta)d(A)]$ for any consant $\beta > 0$, which we call the *estimation parameter.* Our method is based on the approximate compaction method of the previous subsection and the "polling" paradigm of Reif and Sen [43].

We begin our method by attempting to map the distinguished elements of A into an array B of size $h_4^4 n^{8/9}$ using Theorem 2.1, where h_4 is a constant to be set in the analysis. If this mapping fails, then we take B to be a random sample of A of size $n^{8/9}$ (note that $m > h_4 n^{2/9}$ and $E(d(B)) = m/n^{1/9} > h_4 n^{1/9}$ in this case). We then perform $O(\log n)$ applications of Theorem 2.5, with $\epsilon = 1/8$ and α chosen so that $(1+\alpha)^2 \leq 1+\beta$ (e.g., take $\alpha = \beta/3$). Each subproblem i is an attempt to map the distinguished elements of B to an array of size $(1+\alpha)*(1+\alpha)^i$. Let k be the smallest index such that this mapping succeeds. If B contains all m distinguished elements, then we take $m = (1+\alpha)^k$, and if B is our random sample, then we take $m = (1+\alpha)^k n^{1/9}$.

Theorem 2.6: *Given a bichromatic array A, one can compute an estimate m in $O(1)$ time using $O(n)$ processors on a randomized CRCW PRAM, such that, for any constant estimation parameter $\beta > 0$, $m \in [(1-\beta)d(A), (1+\beta)d(A)]$ with probability $1 - 1/c^{n^{1/25}}$, for any constant $c > 1$.*

Proof sketch: The time and processor bounds follow from Theorem 2.5. Using the bounds from Theorem 2.5 we can show that all $O(\log n)$ subproblems perform correctly with probability $p_0 = 1/c_0^{n^{1/25}}$, for any constant $c_0 > 1$. Thus, with probability p_0, $(1+\alpha)^{k-1} < d(B) \leq (1+\alpha)^{k+1}$. If we chose $m = (1+\alpha)^k$, i.e., $d(B) = d(A)$, then $m \in [(1-\alpha)d(B), (1+\alpha)d(B)]$ with this same probability by Theorem 2.5 (since $1-\alpha \leq 1/(1+\alpha)$), which establishes the theorem in this case. Otherwise, if we chose $m = (1+\alpha)^k n^{1/9}$, then, by applying the Chernoff-type [11] bounds of Angluin and Valiant[8] [2], we can set h_4 so that $d(B) \in [(1-\alpha)d(A), (1+\alpha)d(A)]$ with probability $p_1 = 1 - 1/c_1^{n^{1/9}}$, for any constant $c_1 > 1$. Thus, we have that $m \in [(1-\alpha)^2 d(A), (1+\alpha)^2 d(A)]$ with probability at least $p_0 p_1$, which establishes the theorem in this case. □

2.4 Optimal Approximate Compaction

In this subsection we show how to optimally perform approximate compaction. Our approach is implement the oracle $\mathcal{D}$ using Theorem 2.6 and to implement the procedure $\mathcal{S}$ using the algorithm of Section 2.2.1 recursively.

This requires three minor modifications to our algorithm. First, we must generalize the problem to that of mapping distinguished elements in A to distinguished elements in B. That is, in performing a dart throw or a processor id throw, a processor must also check if the target is distinguished, and only attempt a dart/id throw if so. Second, we must modify Phase 4 to allow for each estimate m_k returned by $\mathcal{D}$ to be an approximation for $d(\hat{A}_k)$. Specifically, we set $\beta = 1/5$ in Theorem 2.6 when we use it for $\mathcal{D}$ and take each C_k to be of size $\frac{5}{12}\alpha m_k$. This forces C_k to be almost surely of size at most $\alpha d(\hat{A}_k)/2$ and at least $\alpha d(\hat{A}_k)/3$, which is sufficient for our purposes. Finally, we must modify our algorithm to allow for the failure of $\mathcal{D}$ or $\mathcal{S}$, since they are now randomized. We may assume, however, that both $\mathcal{D}$ and $\mathcal{S}$ succeed with probability $1 - 1/c^{|\hat{A}_k|^{1/25}}$, for any constant $c > 1$ (this assumption is made inductively for $\mathcal{S}$). Even though this almost certainty is with respect to $|\hat{A}_k|$, not n, it is sufficient enough for us to apply our failure sweeping procedure if we modify Phase 1 to use $\rho = (b \log n)^{25}$.

[8] They show that if X is the number of 1's in a collection of independent Bernoulli trials, then $\text{Prob}(X \leq (1-\delta)\mu) \leq e^{-\delta^2\mu/2}$ and $\text{Prob}(X \geq (1+\delta)\mu) \leq e^{-\delta^2\mu/3}$.

Lemma 2.7: *One can perform approximate compaction for any constant expansion parameter $\alpha > 0$, in time that is $O(\log^* n)$, with probability $1 - 1/c^{n^{1/25}}$ for any constant $c > 1$, using n processors on a randomized CRCW PRAM.*

Proof sketch: The time bound is based on a simple induction argument based on the recurrence relations of Section 2.2.1 and the success of our failure sweeping. □

This is clearly more efficient than Theorem 2.5, but it is still not optimal. We can reduce the number of processors, however, to make it optimal. Specifically, we can reduce the approximate compaction problem to multiple-array approximate compaction by dividing A into n/s subarrays of size s each, where $s = 2^{\log^* n}$, and apply the standard parallel prefix algorithm [33] to compress each subarray. This reduction takes $O(\log^* n)$ time using $O(n/\log^* n)$ processors. Applying Lemma 2.7 with Lemma 2.3 then gives us the following:

Theorem 2.8: *One can perform approximate compaction, for any constant expansion parameter $\alpha > 0$, in $O(\log^* n)$ time, with probability $1 - 1/c^{n^{1/25}}$, for any constant $c > 1$, using $O(n/\log^* n)$ processors on a randomized CRCW PRAM.*

Proof: The number of processors needed is $O(n \log^* n/2^{\log^* n} + n/\log^* n + 2^{\log^* n} \log n \log^* n)$, which is $O(n/\log^* n)$, since $\nu = n$, $\lambda = 2^{\log^* n}$, and $\mu = n/2^{\log^* n}$. □

Incidentally, this theorem improves the bottleneck in many of the algorithms of Matias and Vishkin [35]; hence, it has a number of interesting applications. For example, it implies that one can compute a 2-ruling set[9] almost surely in $O(\log^* n)$ time using $O(n/\log^* n)$ processors on a randomized CRCW PRAM.

2.4.1 Some Constant-Time Methods

We can improve the running time for approximate compaction even further, to be almost surely $O(1)$ using $n \log^{(k)} n$ processors, for any constant $k \geq 1$, by stopping the recursion early and substituting our brute force implementations of $\mathcal{D}$ and $\mathcal{S}$. By a reduction to multiple-array approximate compaction, this, in turn, implies that one can perform optimal approximate compaction in $O(1)$ time if $d(A)$ is $O(n/\log^{(k)} n)$, for some constant $k \geq 1$. If we use the CRCW-bit PRAM model, however, where one allows $O(\log^{(k)} n)$ processors to simultaneously write (a 1 or 0) to different bits in a single word [6], then we can perform optimal approximate compaction almost surely in $O(1)$ time for all problem instances. This result is based on the following observation:

Observation 2.9: *Given a bichromatic array $A = (a_1, a_2, \ldots, a_{\log^{(k)} n})$, one can perform the compaction of A to the first $d(A)$ locations in an array B of size at least $d(A)$ in $O(1)$ time using $\log^{(k)} n$ processors on a CRCW-bit PRAM.*

Proof: Suppose a_i is distinguished (if it is not, then processor i does nothing). Processor i writes a 1 to bit i of a word W, which is initially 0, and then reads W and performs an *and* of W and $2^{i+1} - 1$, assigning the result to a register r. Processor i then writes a_i to b_j, where j is the number of 1's in r. The value of j can be found by table look-up. □

Thus, we also have the following:

Theorem 2.10: *One can perform approximate compaction for any constant expansion parameter $\alpha > 0$ in $O(1)$ time, with probability $1 - 1/c^{n^{1/25}}$, for any constant $c > 1$, using n processors on a randomized CRCW-bit PRAM.*

Theorems 2.8 and 2.10 play important roles in our simulation theorems, as does the next problem we study.

2.5 Summation Approximation

The final problem we address is the following:

Summation approximation: given an n-element array A of integers in the range $[0, n^d]$, for some constant d, compute an estimate s' for $s = \sum_{i=1}^{n} a_i$.

We show how to find an s' such that $s' \in [(1 - \gamma)s, (1 + \gamma)s]$ almost surely, for any constant $\gamma > 1$. We begin by choosing β so that

[9] A 2-ruling set [15] is a subsequence L' of an n-element linked list L, such that the distance in L between two consecutive elements in L' is at least 2 and at most 3.

$(1+\beta)^2 \leq 1+\gamma$. For each element a_i in A we let $\mathrm{col}(a_i)$ denote the value j such that $(1+\beta)^j$ is the smallest power of $(1+\beta)$ greater than a_i. Note that if we add up all the $(1+\beta)^{\mathrm{col}(a_i)}$ values, we would get an approximation to s that is off by at most a $1+\beta$ factor. We use an approach similar to that used to prove Lemma 2.3, in that we build an $n \times O(\log n)$ array L such that $L[i,j]$ is 1 if $\mathrm{col}(a_i) = j$. We then perform density approximation on the columns, i.e., apply Theorem 2.6, with estimation parameter β, and add the results using a brute-force $O(1)$-time method [10]. This gives us the approximation s', since it implies that s' is almost surely off from the sum of the $(1+\beta)^{\mathrm{col}(a_i)}$ values by at most a $1+\beta$ factor. This computation clearly runs in $O(1)$ time using $O(n \log n)$ processors. We can reduce the number of processors to n by using polling and an in-place version of Theorem 2.5 (we give the details in the full version).

Theorem 2.11: *Given an n-element array A of integers in the range $[0, n^d]$, for some constant d, one can compute an estimate s' such that $s' \in [(1-\gamma)s, (1+\gamma)s]$, with probability $1 - 1/c^{n^{1/25}}$, for any constant $c > 1$, in $O(1)$ time using n processors on a randomized CRCW PRAM, where $s = \sum_{i=1}^{n} a_i$.*

Incidentally, besides being an important ingredient in our simulation theorem (which we give in the next section), Theorem 2.11 also leads to improved parallel methods for a number of other problems, including appoximate selection (which we show in the full version).

3 Our Framework

Having presented our approximation algorithms, we now show how they may be used to simulate algorithms that ignore processor allocation.

3.1 Previous Simulation Theorems

Several researchers [16, 17, 34, 45] have studied the problem of producing a work-preserving simulation of a computation where all tasks are created at the beginning of the computation and each task performs in a non-idle fashion for a (possibly unknown) number of steps and then terminates. Hagerup [27] gives a simulation theorem for a slightly more general version of the problem, where one allows for the entire computation to be terminated, and a new computation to take its place (possibly with a different number of tasks), at an overhead cost of $O(\log\log n / \log\log\log n)$ time.

3.2 Our Simulation Theorem

We show how to simulate an algorithm that ignores processor allocation at a cost that is almost surely $O(\log^* n)$ time per processor allocation step on a randomized CRCW PRAM, and is almost surely $O(1)$ on a randomized CRCW-bit PRAM [6]. Our framework is synchronous, in that it allows one to assume that all the operations for Step i complete before any task performs the operation for Step $i+1$. It also allows for all the usual PRAM operations to be computed in $O(1)$ time, including arithmetic, comparisons, and accesses to shared memory. In addition, it allows the following operations on tasks:

Halt: terminate this task.

Spawn(p): create a new task, to begin, in the next time step, the execution of the procedure specified by the pointer p.

Start(m,p): create n new tasks, to begin, in the next time step, the execution of the procedures specified by the pointer p (which could, for example, point to an array). Only one task may perform this operation in any given step.

In order to avoid confusing this model with the usual PRAM model, where processor allocation is a serious consideration, we refer to this as the *V-PRAM* model. This model subsumes the previous frameworks for task management [16, 17, 23, 27, 35, 42, 45], while still allowing for synchronous computations.

Let n_i denote the number of real operations performed by the active tasks in step i of a V-PRAM computation $\mathcal{A}$ that runs in time T. The *work* performed by a V-PRAM computation is the sum of the lengths of all the tasks created during an execution of the computation, i.e., $W = \sum_{i=1}^{T} n_i$. Our goal is to simulate $\mathcal{A}$ on a PRAM in a work-preserving fashion using P processors. The running time of our simulation is dependent upon T, P, W, and T', the number of steps in $\mathcal{A}$ such that some task performs one of the above task allocation operations. Our simulation does not require the exact value of n_i, however; we can use an approximation $\hat{n}_i$ for n_i such that $\hat{n}_i \leq (3/2)n_i$ almost surely. This is sufficient, for it implies that $\sum_{i=1}^{T} \hat{n}_i$ is almost surely $O(W)$.

We can also make a simplifying assumption based on the echo function defined on the $\hat{n}_i$ values, where, given any non-negative integer function f defined on the domain $\{1, 2, \ldots, m\}$, we define f's *echo function* g by the recurrence relation $g(i) = \max\{f(i), g(i-1)/2\}$, with $g(1) = f(1)$. Note that, for any such f and g, $\sum_{i=1}^{m} g(i) \leq 2\sum_{i=1}^{m} f(i)$. Thus, we can use the echo function defined on the $\hat{n}_i$ values in the same way Brent uses the n_i values.

Theorem 3.1: *Let $\mathcal{A}$ be a V-PRAM algorithm that runs in time T with W work. Then $\mathcal{A}$ can be simulated by P processors on a randomized CRCW PRAM in $O(\lfloor W/P \rfloor + T + T' \log^* P)$ time, where T' is the number of processor allocation steps in $\mathcal{A}$. Alternately, one can simulate $\mathcal{A}$ in $O(\lfloor W/P \rfloor + T)$ time on a randomized CRCW-bit PRAM. The time bounds hold with probability $1 - T/c^{P^{1/25}}$, for any constant $c > 1$.*

Proof: Our proof is based on a simulation of each step i of $\mathcal{A}$ with P processors on a randomized CRCW PRAM in time that is almost surely $O(\lceil N_i/P \rceil + \log^* P)$ (or, alternately, almost surely $O(\lceil N_i/P \rceil)$ on a randomized CRCW-bit PRAM), where N_i is the echo function defined on the $\hat{n}_i$ approximations of the n_i values. Each real processor p is assigned a packet of tasks, the size of which is p's *load*. We maintain the following invariant:

Simulation Invariant: The load of each real processor is at most $16\lceil N_i/P \rceil$.

Our method for simulating step i is as follows. We first perform all the operations of $\mathcal{A}$'s step i that do not affect the pool of processors. If there are no other kinds of operations in this step (which can be tested in $O(1)$ time by an **or** operation), then we are done, for $N_{i+1} = N_i$ in this case.

We next perform the **Halt** operations (if there are any to perform). Let n_i' denote the number of surviving tasks. Of course, we do not know the value of n_i', and don't have time to compute it exactly. So, we use summation approximation to compute an estimate $\hat{n}_i'$ such that $(1/2)n_i' \leq \hat{n}_i' \leq (3/2)n_i'$ almost surely, and let $N_i' = \max\{\hat{n}_i', N_i/2\}$. After performing all the **Halt** operations the load for any real processor will now be at most $32\lceil N_i'/P \rceil$. We redistribute the load among real processors in a fashion similar to that used by Matias and Vishkin [35]. We call a real processor *overloaded* if its load is more than $8\lceil N_i'/P \rceil$. Note that there can be at most $N_i'/(8\lceil N_i'/P \rceil) \leq \lceil P/8 \rceil$ overloaded real processors. Use approximate compaction to map the **id**'s of these processors into an array of size $\lceil P/4 \rceil$, which almost surely takes $O(\log^* P)$ time and $O(P)$ work. By then associating 4 real processors with each element of this array and distributing the load evenly we get that the maximum load for any real processor becomes at most $16\lceil N_i'/P \rceil$.

We then perform the **Spawn** operations. Let n_i'' denote the total number of tasks that result, including the surviving old ones. Again, we use the summation approximation procedure to compute an approximation $\hat{n}_i''$ to n_i'' such that $(1/2)n_i'' \leq \hat{n}_i'' \leq (3/2)n_i''$ almost surely, and let $N_i'' = \max\{\hat{n}_i'', N_i'\}$ $(= \max\{\hat{n}_i'', N_i/2\})$. Since any virtual processor can spawn at most one other virtual processor, we have that this increases the load of any real processor to at most $32\lceil N_i''/P \rceil$. We then redistribute the load among the real processors as above, which reduces the load of every real processor to at most $16\lceil N_i''/P \rceil$.

We complete the simulation by performing the **Start**(m, p) operation, if there is one. We assign the m new tasks evenly to the P real processors, increasing the load of any processor to at most $\lceil m/P \rceil + 16\lceil N_i''/P \rceil$. Note that $n_{i+1} = n_i'' + m$. So we set $N_{i+1} = \max\{\hat{n}_i'' + m,\ N_i/2\}$. This immediately implies that $m \leq N_{i+1}$ and $N_i'' = \max\{\hat{n}_i'', N_i/2\} \leq N_{i+1}$. Thus, this **Start** operation can increase the load of any real processor to at most $17\lceil N_{i+1}/P \rceil$. So, by a redistribution as above, we can almost surely achieve our invariant for the beginning of step $i+1$, and this completes the simulation of step i.

By Theorem 2.1 and our observation about echo functions, the total time needed for the entire simulation is almost surely $O(\lfloor W/P \rfloor + T + T' \log^* P)$, where T' is the number of steps where **Halt**, **Spawn**, or **Start** operations were performed. This completes the proof. □

Having given our framework for designing V-PRAM algorithms and how they can be efficiently simulated on a randomized CRCW PRAM, let us now give an application of this approach to an important problem in parallel computational geometry: upper envelope construction.

4 Application: Upper Envelopes

Suppose we are given a collection of functions $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$, where each $f_i : \Re \rightarrow \Re$. Moreover, suppose that $\mathcal{F}$ defines a k-intersecting class of functions, that is, there is an integer constant $k \geq 0$ such that, for any pair of functions (f_i, f_j), $i \neq j$, there are at most k x-values such that $f_i(x) = f_j(x)$. We define the *upper envelope* function $\mathbf{f}$ on $\mathcal{F}$ by $\mathbf{f}(x) = \max_{i=1}^{n} f_i(x)$. We define a *piece* of $\mathbf{f}$ to be a maximal restriction of $\mathbf{f}$ to an interval $[x_1, x_2]$ such that $\mathbf{f}(x) = f_i(x)$, for some i, for all $x \in [x_1, x_2]$. A natural combinatorial problem arising from this definition is the characterization of $\lambda_k(n)$, the maximum number of pieces of $\mathbf{f}$. Atallah [3] showed that this number is closely related to the complexity of a class of strings known as Davenport-Schinzel sequences [18], which implies, by a theorem of Szemeredi [47], that $\lambda_k(n)$ is $O(n \log^* n)$ for any constant $k \geq 3$ ($\lambda_k(n)$ is linear for $k = 1, 2$ [3]). Since Szemeredi's bound was established, the bounds for $\lambda_k(n)$ have been refined to $\Theta(n\alpha(n))$ for $k = 3$ [31], and to $O(n\alpha(n)^{O(\alpha(n)^{k-3})})$ for $s > 3$ [46], where $\alpha(n)$ is the very slow-growing inverse of Ackermann's function.

We are interested in the parallel complexity of constructing $\mathbf{f}$, as it has a host of applications, including (1) the computation of convex hulls and closest pairs for moving point sets [3], (2) the computation of Voronoi diagrams in parallel [13], (3) ray-shooting data structures [14], (4) hidden-surface elimination [42], (5) the construction of certain types of arrangements [19, 20], and (6) polygon containment problems [49]. The previous method for solving this problem in parallel is due to Boxer and Miller [8], and runs in $O(\log^2 n)$ time using $O(\lambda_k(n))$ processors on an CREW PRAM.

In this section we give an algorithm for constructing a representation of the upper envelope $\mathbf{f}$ in $O(\log n)$ time with $O(\lambda_k(n))$ processors in Valiant's parallel comparison model (where we count intersection computations as intersections). We also show how to implement our algorithm on a CREW V-PRAM model in $O(t_2(\lambda_k(n)) \log n)$ time with $O(\lambda_k(n)/t_2(n) + p_2(\lambda_k(n)))$ processors, where $t_2(n)$ and $p_2(n)$ are, respectively, the time and processor bounds for constructing a 2-ruling set[10]. By then applying the results derived earlier in this paper, we show that we can construct $\mathbf{f}$ almost surely in $O(\log n \log^* n)$ time using $O(\lambda_k(n)/\log^* n)$ processors on a randomized CRCW PRAM.

Our method takes the same general approach as the sequential divide-and-conquer algorithm used by Atallah [3], where one divides $\mathcal{F}$ into two groups, $\mathcal{F}_1 = \{f_1, f_2, \ldots, f_{n/2}\}$ and $\mathcal{F}_2 = \{f_{n/2+1}, f_{n/2+2}, \ldots, f_n\}$, recursively constructs the upper envelope functions $\mathbf{f}_1$ and $\mathbf{f}_2$, and then merges the lists representing $\mathbf{f}_1$ and $\mathbf{f}_2$ to construct $\mathbf{f}$. Since each piece in an upper envelope is characterized by its associated function f_i and the interval $[x_1, x_2]$ upon which it equals $\mathbf{f}$, this merge step can be implemented by merging the endpoints of the intervals defining the pieces of each upper envelope $\mathbf{f}_1$ and $\mathbf{f}_2$. The pieces of $\mathbf{f}$ are determined by intersecting the functions associated with each pair of overlapping intervals. Since there can be at most k intersections for each such pair, this computation requires $O(1)$ time per pair.

One problem with trying to parallelize this approach, however, is that we do not know *a priori* which old upper envelope pieces will survive and which ones will die after performing a merge. Also, we do not know which pairs of overlapping intervals may contribute k new pieces. Thus, if, say, we wish to pipeline the interval merging process using Cole's parallel mergesort procedure [12], then it seems that we must assume that pieces do not die. Iterating this procedure for $O(\log n)$ iterations without compaction would require $O(n^{\log k})$ processors, however. On the other hand, performing compaction often enough to keep the work optimal would require too much time[11].

We can avoid these difficulties, however, by using the parallel mergesort procedure of Goodrich and Kosaraju [26] to pipeline this divide-and-conquer process. Their method is based on a linked-list representation, rather than an array representation, as is needed by Cole's method [12]; hence, their method does not require compaction be performed after each merge. Applying their method still has some difficulties, however. One such difficulty is that they require that there be a processor assigned to each element in the lists being merged. Fortunately, by implementing our generalization of their method on a V-PRAM, we

[10] The best deterministic method runs in $O(\log^* n)$ time using n processors [15].

[11] Approximate compaction cannot be applied here, since it does not preserve orders.

can ignore this issue.

We begin our algorithm by building a complete (balanced) binary tree T with n leaves such that the elements of $\mathcal{F}$ are stored in T one element per leaf. As in [26], we view T as the schematic for a parallel mergesort procedure. We make the notational convention that v is an arbitrary node, x and y are its children, and u is its parent. We define a list $A(v)$ for each v to be the upper envelope of the functions stored in descendents of v. We construct $A(v)$ in a pipelined fashion in a sequence of *stages*. At the end of each stage t we store a list $A_t(v)$ and say that v is *full* when $A_t(v) = A(v)$. We pipeline the merge at v's children to v by maintaining a list $L_t(v)$ to be an approximately-uniform subsequence of $A_t(v)$, for all v in T, and by then defining $A_{t+1}(v) = L_t(x) \cup L_t(y)$. By *approximately-uniform* we mean that the distance in $A_t(v)$ between two consecutive elements in $L_t(v)$ is at least c_1 and at most c_2 for constants c_1 and c_2. Once a node v becomes full, then we iteratively refine $L_t(v)$ until we eventually have $L_t(v) = A_t(v) = A(v)$. This refinement is quite straightforward: for every consecutive pair of elements e and f in $L_t(v)$ we add the element from $A_t(v)$ that is half-way between e and f. Once $L_t(v) = A_t(v)$, we then perform the upper-envelopes merge of $A(x)$ and $A(y)$ to form $A(v)$, at which point v becomes full. Every $\lceil \log c_2 \rceil = O(1)$ stages, the nodes at the next higher level in T become full. This implies that the entire computation requires only $O(\log n)$ stages.

Due to space constraints, we must postpone the details of this process to the full version of this paper. Some of the main ingrediants in the method involve the maintenance of two additional temporary lists at v, $\hat{L}_t(v)$ and $L_t^*(v)$, which facilitate the maintenance of five important stage invariants. We have already mentioned two of the most important of these invariants in the previous paragraph, however: namely, that we maintain the definitions of $A_t(v)$ and $L_t(v)$ after each stage t. Moreover, it is the maintenance of these two invariants that are most different from the corresponding invariants from the mergesorting procedure of Goodrich and Kosaraju [26] (we, of course, show how to maintain all our invariants in the full version). Let us therefore address the first invariant, the definition of $A_t(v)$. So long as v is not full it is simply the merge of the L_t's stored at v's children. Thus, we can use the method of [26] to form $A_{t+1}(v)$ so long as v is not full. Once v becomes full we must then use the merged list of $A(x)$ and $A(y)$ to construct the upper envelope $A(v)$, which we use for $A_{t+1}(v)$. This can easily be done by a (local) computation, where for each overlapping pair of intervals $[x_1, x_2] \in A(x)$ and $[y_1, y_2] \in A(y)$ we compute the intersections of the respective functions associated with these intervals and construct the new set of (at most k) intervals this determines in $[x_1, x_2] \cap [y_1, y_2]$. Of course, we could also determine that some of the endpoints in $\{x_1, x_2, y_1, y_2\}$ are no longer endpoints in the merge. Fortunately, for such each endpoint p we must remove, we can easily determine the (possibly new) endpoint of an interval in $A(v)$ that is immediately to the left of p. For example, if y_1 is made redundant, it must be because it is covered by the function f defined on $[x_1, x_2]$; hence, we can examine x_1 and the (at most k) new intersection points determined by f to choose which one is immediately to the left of p. This solves the problem of correctly constructing $A_{t+1}(v)$ for all v in T, and can be done in $O(1)$ time given the merge of $A(x)$ and $A(y)$.

Unfortunately, this operation makes the maintainence of our last invariant, the definition of $L_{t+1}(v)$, much more difficult. The problem is that the upper-envelopes merge at a node v could cause the removal of a number of consecutive endpoints from $A_t(v)$, so that $L_t(v)$ cannot simply be refined to form a valid $L_{t+1}(v)$, for it may no longer be an approximately-uniform subsequence of $A_{t+1}(v)$. We get around this problem by going ahead and refining $L_t(v)$ as described above, but follow this by forming a 2-ruling set of any chains in this refined list that violate our requirement that any two consecutive elements in $L_t(v)$ be at distance at least c_1 in $A_t(b)$. We leave the details of this to the full version, as well as the "ripple" effect this computation has through the entire pipelining process. Suffice it to say that each stage can be implemented in $O(t_2(N)))$ time, using $O(p_2(N))$ processors on a CREW V-PRAM, where N is the total size of all the $A_t(v)$ lists after any stage t. In fact, if we discount the time required to construct the 2-ruling set (a computation that is not counted in Valiant's parallel comparison model), then each stage requires $O(1)$ time using N processors. We show in the full version that N is $O(\lambda_k(n))$, and derive the following:

Theorem 4.1: *Given $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$, a collection of k-intersection functions, one can con-*

struct the upper envelope **f** *of the functions in* $\mathcal{F}$ *in* $O(\log n)$ *time using* $O(\lambda_k(n))$ *processors in Valiant's parallel comparison model. Alternately, one can construct* **f** *almost surely in* $O(\log n \log^* n)$ *time using* $O(\lambda_k(n)/\log^* n)$ *processors on a randomized CRCW PRAM.*

4.1 Implications

This theorem immediately leads to improved parallel algorithms for a number of other problems [3, 14, 19, 20, 42]. We address some of these applications in the full version of this paper. In fact, it even improves the sequential complexity of an interesting polygon containment problem [49].

5 Other Applications

In addition to problems related to upper-envelope constructions, our methods improve the parallel running times for a number of other parallel computational geometry problems. We briefly mention two in this preliminary version.

5.1 Range Searching

In the full version we show how to apply our framework and a parallel version of the data structure of McCreight to efficiently answer three-sided range queries with an output-sensitive number of processors on a randomized CRCW PRAM algorithm.

5.2 Hidden-Surface Elimination

The parallel method of Reif and Sen [43] for performing hidden-surface elimination in a terrain, and the method of Goodrich, Ghouse, and Bright [25] for performing hidden-surface elimination for a rectilinear scene, are both designed for the CREW V-PRAM model. By applying our framework, we immediately achieve an improvement of almost a $\log n$ factor in the running time (since their simulation algorithms have an $\log n$ time overhead).

6 Conclusion

The results of this paper fall into three distinct catagories—randomized approximation algorithms, PRAM simulations, and parallel computational geometry—where the results in each catagory depend on the ones that come before. Moreover, the dependence upon randomization and approximation for the problems studied in Section 2 seems quite strong, for exact versions of all these problems can have the parity or majority problems reduced to them, implying an $\Omega(\log n/\log\log n)$ lower bound on their running time on a CRCW PRAM with a polynomial number of processors [5].

The methods of Section 2 also make use of look-up tables. The existence of such tables, of course, begs the question of how long they take to build. Indeed, this issue is at the heart of the distinction between uniform and non-uniform computations [10]. Fortunately, since we make repeated use of our approximation algorithms in our simulation theorem, the cost of any table construction can be amortized over the entire simulation. In any case, the construction costs associated with the look-up tables we need are quite small—none requires more than $O(\log^* n)$ time [10, 15].

Acknowledgements

I am truly indebted to Yossi Matias for introducing me to approximate compaction and density approximation. I would also like to thank Mujtaba Ghouse, Torben Hagerup, S. Rao Kosaraju, and Michael Luby for several helpful discussions.

References

[1] R.J. Anderson and G.L. Miller, "A Simple Randomized Parallel Algorithm for List-Ranking," *I.P.L.*, **33**, 1990, 269–273.

[2] D. Angluin and L.G. Valiant, "Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings," *J.C.S.S.*, **18**, 1979, 155–193.

[3] M.J. Atallah, "Some Dynamic Computational Geometry Problems," *Comp. and Maths. with Appls.*, **11**, 1985, 1171–1181.

[4] M.J. Atallah and M.T. Goodrich, "Deterministic Parallel Computational Geometry," to appear in *Synthesis of Parallel Algorithms*, J.H. Reif, ed.

[5] P. Beame and J. Hastad, "Optimal Bounds for Decision Problems on the CRCW PRAM," *19th STOC*, 1987, 83–93.

[6] O. Berkman and U. Vishkin, "Recursive *-Tree Parallel Data-Structure," *30th FOCS*, 1989, 196–202.

[7] A. Borodin and J.E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation," *J.C.S.S.*, **30**(1), 1985, 130–145.

[8] L. Boxer and R. Miller, "Parallel Dynamic Computational Geometry," TR 87-11, SUNY-Buffalo, 1987.

[9] R.P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *J. ACM*, **21**(2), 1974, 201–206.

[10] A.K. Chandra, L.J. Stockmeyer, U. Vishkin, "A Complexity Theory for Unbounded Fan-In Parallelism," *23rd FOCS*, 1982, 1–13.

[11] H. Chernoff, "A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations," *Annals of Math. Stat.*, **23**, 1952, 493–507.

[12] R. Cole, "Parallel Merge Sort," *SIAM J. Comput.*, **17**(4), 1988, 770–785.

[13] R. Cole, M.T. Goodrich, C. Ó Dúnlaing, "Merging Free Trees in Parallel for Efficient Voronoi Diagram Construction," *17th ICALP*, 1990, 432–445.

[14] R. Cole and M Sharir, "Visibility Problems for Polyhedral Terrains," TR 92, Courant Institute, NYU, 1986.

[15] R. Cole and U. Vishkin, "Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms," *18th STOC*, 1986, 206–219.

[16] R. Cole and U. Vishkin, "Approximate Parallel Scheduling, Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time," *SIAM J. Comput.*, **17**(1), 1988, 128–142.

[17] R. Cole and O. Zajicek, "An Optimal Parallel Algorithm for Building a Data Structure for Planar Point Location," *J. Par. and Dist. Comput.*, **8**, 1990, 280–285.

[18] H. Davenport and A. Schinzel, "A Combinatorial Problem Connected with Differential Equations," *Amer. J. Math.*, **87**, 1965, 684–694.

[19] H. Edelsbrunner, L. Guibas, J. Hershberger, J. Pach, R. Pollack, R. Seidel, M Sharir, and J. Snoeyink, "On Arrangements of Jordan Arcs with Three Intersections per Pair," *4th ACM Symp. on Comp. Geom.*, 1988, 258–265.

[20] H. Edelsbrunner, L. Guibas, J. Pach, R. Pollack, R. Seidel, and M Sharir, "Arrangements of Curves in the Plane – Topology, Combinatorics, and Algorithms," TR UIUCDCS-R-88-1477, Univ. of Illinois, 1988.

[21] F.E. Fich and V. Ramachandran, "Lower Bounds for Parallel Computation on Linked Structures," *2nd ACM Symp. on Parallel Alg. and Arch. (SPAA)*, 1990, 109–116.

[22] M. Ghouse and M.T. Goodrich, "In-place Parallel Techniques for Fast Convex Hull Construction," *3nd SPAA*, 1991.

[23] M.T. Goodrich, "Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors," *SIAM J. Comput.*, to appear.

[24] M.T. Goodrich, "Fast Parallel Approximation Algorithms for Searching and Counting Problems," Technical Report, The Johns Hopkins Univ., 1991.

[25] M.T. Goodrich, M. Ghouse, and J. Bright, "Generalized Sweep Methods for Parallel Computational Geometry," *2nd SPAA*, 1990, 280–289.

[26] M.T. Goodrich and S.R. Kosaraju, "Sorting on a Parallel Pointer Machine with Applications to Set Expression Evaluation," *30th FOCS*, 1989, 190–195.

[27] T. Hagerup, "Fast Parallel Generation of Random Permutations," manuscript, 1990.

[28] T. Hagerup, "Constant-Time Parallel Integer Sorting," *23rd STOC*, 1991, 299–306.

[29] T. Hagerup, "Fast Parallel Space Allocation, Estimation and Integer Sorting," Tech. Report MPI-I-91-106, Max-Planck-Institut für Informatik, Saarbrücken, 1991.

[30] T. Hagerup and T. Radzik, "Every Robust CRCW PRAM Can Efficiently Simulate a Priority PRAM," *2nd SPAA*, 1990, 117–124.

[31] S. Hart and M. Sharir, "Nonlinearity of Davenport-Schinzel Sequences and Generalized Path Compression Schemes," *Combinatorica*, **6**, 1986, 151–177.

[32] R.M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," in *Handbook of Theorertical Computer Science: Vol. A*, MIT Press, 1990, 869–942.

[33] Ladner, R.E., and Fischer, M.J., "Parallel Prefix Computation," *J. ACM*, October 1980, 831–838.

[34] Y. Matias and U. Vishkin, "On Parallel Hashing and Integer Sorting," CS-TR-2395, Univ. of Maryland, 1990.

[35] Y. Matias and U. Vishkin, "Converting High Probability into Nearly-Constant Time—with Applications to Parallel Hashing," 23rd STOC, 1991, 307–316.

[36] E.M. McCreight, "Priority Search Trees," *SIAM J. on Comput.*, **14**, 1985, 257–276.

[37] G.L. Miller and J.H. Reif, "Parallel Tree Contraction and its Applications," *26th FOCS*, 1985, 478–489.

[38] P. Ragde, "The Parallel Simplicity of Compaction and Chaining," *17th ICALP*, 1990, 744–751.

[39] P. Raghavan, *Lecture Notes on Randomized Algorithms*, 1989.

[40] S. Rajasekaran and S. Sen, "Random Sampling Techniques and Parallel Algorithms Desing," to appear in *Synthesis of Parallel Algorithms*, J.H. Reif, ed.

[41] J.H. Reif, "An Optimal Parallel Algorithm for Integer Sorting," *26th FOCS*, 1985, 496–504.

[42] J.H. Reif and S. Sen, "An Efficient Output-Sensitive Hidden-Surface Removal Algorithm and Its Parallelization," *4th ACM Symp. on Comp. Geom.*, 193–200, 1988.

[43] J.H. Reif and S. Sen, "Polling: A New Randomized Sampling Technique For Computational Geometry," *21st STOC*, 1989, 394–404.

[44] J.H. Reif and L. Valiant, "Logarithmic Time Sort for Linear Size Networks," *J. ACM*, **34**(1), 1987, 60–76.

[45] G. Shannon, "Optimal On-Line Load Balancing," *1989 SPAA*, 235–245.

[46] M. Sharir, "Almost Linear Upper Bounds on the Length of General Davenport-Schinzel Sequences," *Combinatorica*, **7**, 1987, 131–143.

[47] E. Szemeredi, "On a Problem of Davenport and Schinzel," *Acta Arith.*, **25**, 1974, 213–224.

[48] Thinking Machines Corp., *The Connection Machine System: Paris Release Notes*, 1990.

[49] S. Toledo, "Extremal Polygon Containment Problems," *Proc. 7th ACM Symp. on Computational Geometry*, 1991, 176–185.

[50] L.G. Valiant, "Parallelism in Comparison Problems," *SIAM J. Comput.*, **4**(3), 1975, 348–355.