In *Journal of Algorithms* **26** (1998), pp. 1–33.

# Ultra-Fast Expected Time Parallel Algorithms

Philip D. MacKenzie[1]
Google Research

Quentin F. Stout[2]
Computer Science and Engineering
The University of Michigan

## Abstract

It has been shown previously that sorting $n$ items into $n$ locations with a polynomial number of processors requires $\Omega(\log n/\log\log n)$ time. We sidestep this lower bound with the idea of Padded Sorting, or sorting $n$ items into $n + o(n)$ locations. Since many problems do not rely on the exact rank of sorted items, a Padded Sort is often just as useful as an unpadded sort. Our algorithm for Padded Sort runs on the Tolerant CRCW PRAM and takes $\Theta(\log\log n/\log\log\log n)$ expected time using $n\log\log\log n/\log\log n$ processors, assuming the items are taken from a uniform distribution. Using similar techniques we solve some computational geometry problems, including Voronoi Diagram, with the same processor and time bounds, assuming points are taken from a uniform distribution in the unit square. Further, we present an Arbitrary CRCW PRAM algorithm to solve the Closest Pair problem in constant expected time with $n$ processors regardless of the distribution of points. All of these algorithms achieve linear speedup in expected time over their optimal serial counterparts.

[1]Research done while at the University of Michigan and supported by an AT&T Fellowship.
[2]Supported by NSF/DARPA grant CCR-9004727.

# 1 Introduction

For a given problem with $n$ inputs, we define an *ultra-fast* parallel algorithm as one which uses a linear number of processors and runs in $O((\log\log n)^{O(1)})$ time. Some examples of problems with known ultra-fast parallel algorithms include merging two lists of size $n$ [24] and finding the maximum of $n$ numbers [36]. We also define an *ultra-fast expected time* parallel algorithm as one which uses at most a linear number of processors and runs in $O((\log\log n)^{O(1)})$ expected time. In this paper, we will develop ultra-fast expected time parallel algorithms for sorting and many geometric problems. These algorithms will also achieve linear speedup in expected time over their serial counterparts.

Because there is some ambiguity involved in the term "expected time," we will define it more carefully. If we assume the inputs to an algorithm come from a specific probability distribution, we call the expected time analysis *distribution dependent*. For a *deterministic* algorithm, the expected time is simply the average running time over the distribution of inputs. For a *randomized* algorithm, the expected time is the average running time over the distribution of inputs and the possible random choices within the algorithm.

Alternatively, if we assume nothing about the input distribution, we call the expected time analysis *distribution independent*. For a deterministic algorithm, the expected time is simply equal to the worst case time. For a randomized algorithm, the expected time is the average running time over the possible random choices for the worst case input, or equivalently, the maximum over all inputs of each input's average running time.

We will define a *randomized parallel algorithm* as one in which each processor can make independent random choices. In this paper, we present randomized parallel algorithms that solve the following problems in $\Theta(\log\log n/\log\log\log n)$ expected time with linear speedup ($n\log\log\log n/\log\log n$ processors).

**Padded Sort** Given $n$ values taken from a uniform distribution over the unit interval $[0,1]$, arrange them in sorted order in an array of size $n+o(n)$, with the value NULL in all unfilled locations.

**All Nearest Neighbors** Given a set $S$ of $n$ points taken from a uniform distribution over the unit square, for each point $p \in S$ find the point in $S-\{p\}$ that is closest to $p$ in the Euclidean metric.

**Relative Neighborhood Graph** Given a set $S$ of $n$ points taken from a uniform distribution over the unit square, for each point $p \in S$, construct an edge from $p$ to every other point $q \in S$ where $p$ and $q$ are relative neighbors, i.e., for all $r \in S$ where $r \neq p, q$, the distance from $p$ to $q$ is less than than the maximum of the distance from $p$ to $r$ and the distance from $q$ to $r$.

**Voronoi Diagram** Given a set $S$ of $n$ points taken from a uniform distribution over the unit square, for each point $p \in S$ find the maximal polygon around it which has the property that any point in the polygon is closer to $p$ than to any other point in $S$.

**Delaunay Triangulation** Given a set $S$ of $n$ points taken from a uniform distribution over the unit square, find a triangulation of $S$ with the property that the interior of the circumcircle of every triangle is empty. A triangulation of $S$ is defined as a planar subdivision inside and including the convex hull of $S$, whose vertices are exactly the points of $S$, and whose regions are all triangles.

**Largest Empty Circle** Given a set $S$ of $n$ points taken from a uniform distribution over the unit square, find the largest circle that contains no points of $S$ and whose center is internal to the convex hull of $S$.

Given $n^{1+\epsilon}$ processors, the above problems can be solved in constant time. We do not consider these ultra-fast algorithms due to the super-linear number of processors, but we include descriptions of these algorithms for completeness.

We also present a randomized parallel algorithm to solve the following problem in constant expected time given $n$ processors. Note that there is no dependence on the distribution of points in this problem.

**Closest Pair** Given a set $S$ of $n$ points in the plane, determine the pair of points in $S$ which are closest to each other in the Euclidean metric.

All of the problems above have trivial linear lower bounds on the expected times of their solutions, and serial algorithms have been developed for all of them which attain this lower bound. The solution for sorting is well known. Bentley, Weide, and Yao [7] exhibit linear expected time algorithms for All Nearest Neighbors and Voronoi Diagram, and linear expected time Delaunay Triangulation and Largest Empty Circle algorithms follow immediately. Katajainen, Nevalainen, and Teuhola [22] exhibit a linear expected time algorithm for the Relative Neighborhood Graph. Rabin [29] has given a randomized linear expected time algorithm for Closest Pair which is distribution independent.

There has been a great amount of work on parallel algorithms for distribution independent versions of the problems above. We give the most recent results here, and indicate whether the algorithms are PT-optimal, which means that the processor time product is equal to the serial lower bound, or simply optimal, meaning that for the number of processors used, the time is equal to a known lower bound. When the processor time product of an algorithm is $O(n)$, the PT-optimality is obvious, and we will sometimes omit this indication.

Leighton's [25] modification to the AKS sorting network [3], and Cole's parallel merge sort [13] both use $n$ processors and achieve $\Theta(\log n)$ worst case time for sorting, which is PT-optimal. We note that any PT-optimal algorithm for sorting must use at least $\log n$ time [4]. Reischuk [33] gives a PT-optimal randomized $n$ processor, $\Theta(\log n)$ time algorithm for sorting. Rajasekaran and Reif [30] give a randomized algorithm for general sorting which achieves $\Theta(\log n/\log\log n)$ time with $n\log^{\epsilon} n$ processors for any $\epsilon > 0$, which is optimal, a randomized algorithm for integer sorting which achieves $\Theta(\log n/\log\log n)$ time with $n(\log\log n)^2/\log n$ processors, and a PT-optimal randomized algorithm for integer sorting which achieves $\Theta(\log n)$ time with $n/\log n$ processors. Cole and Goodrich [14] and Willard and Wee [37] both present PT-optimal $n$ processor, $\Theta(\log n)$ worst case time algorithms for solving the All Nearest Neighbor and Closest Pair problems. Aggarwal *et al.* [1] present an $n$ processor, $\Theta(\log^2 n)$ worst case time algorithm for finding the Voronoi Diagram. Cole, Goodrich, and O'Dunlaing [15] give algorithms for finding the Voronoi Diagram in $\Theta(\log^2 n)$ worst case time with $n/\log n$ processors, and in $\Theta(\log n\log\log n)$ worst case time with $n\log n/\log\log n$ processors. Reif and Sen [32] have recently given PT-optimal $n$ processor, $\Theta(\log n)$ expected time randomized algorithms for constructing the Voronoi Diagram and finding All Nearest Neighbors. We note that the Largest Empty Circle can be found in $\Theta(\log n)$ worst case time with $n$ processors given the Voronoi Diagram, so the time and processor bounds above also apply to finding the Largest Empty Circle.

Berkman *et al.* [8] give $n/\log\log n$ processor, $\Theta(\log\log n)$ worst case time algorithms for some other geometric problems, but these problems are highly constrained. On the other hand, we solve much more general problems and use randomness and a knowledge of input distribution to obtain linear speedup and $o(\log\log n)$ expected time solutions.

Two groups have previously done work on parallel distribution dependent expected-time geometry. Stout [35] shows that given $n$ points taken from a uniform distribution over the unit square, the maximal points, extreme points, diameter, smallest enclosing rectangle, smallest enclosing circle, and closest pair can all be found in constant expected time with $n$ processors. These results can also be extended to more general regions. Levcopoulos, Katajainen, and Lingas [26] show that the Voronoi Diagram of $n$ points taken from a uniform distribution over the unit square can be constructed in $\Theta(\log n)$ expected time with $n/\log n$ processors. Katajainen, Nevalainen, and Teuhola [22] show that the Relative Neighborhood graph of $n$ points taken from a uniform distribution over the unit square can also be constructed in $\Theta(\log n)$ expected time with $n/\log n$ processors.

Distribution dependent parallel sorting has been worked on by Chlebus [12]. He obtains a $\Theta(\log n)$ expected time, $n/\log n$ processor algorithm to sort $n$ random integers in the range $[1, n]$.

Following the work in this paper, MacKenzie [27] has proven a lower bound of $\Omega(\log^* n)$ expected

time for Padded Sort and Hagerup and Raman [21] have shown that this is optimal by giving an $O(\log^* n)$ expected time algorithm for Padded Sort. We note that their algorithm also improves on the algorithm given here in that it does not rely on any assumption about the distribution of inputs.

## 2 Preliminaries

Some probabilistic tools and other equations which will be useful in our analyses are given in the appendices. Throughout the paper we will assume that $n$ (the number of inputs) is large enough so that our analyses hold.

For all our algorithms except for Closest Pair we will be using the Tolerant Concurrent Read, Concurrent Write (CRCW) Parallel Random Access Machine (PRAM) model. In this model, if two or more processors write to a cell simultaneously, then the cell remains unchanged. An algorithm for the Tolerant CRCW PRAM implies algorithms with the same time and processor bounds on stronger models, such as the Collision and Arbitrary models (see, for example, Hagerup and Radzik [20]). For the Closest Pair algorithm we will be using the Arbitrary CRCW PRAM model. In this model, if two or more processors write to a cell simultaneously, the one which succeeds in writing is chosen arbitrarily.

In all our algorithms, we will also assume that we can perform floor and ceiling functions in constant time. These are used to perform various types of "bin-sorting" procedures, which are crucial to our algorithms.

We will use the following CRCW PRAM algorithms as subprocedures.

**Prefix** The prefix operation takes an array $A = [a_0, a_1, \ldots, a_{n-1}]$ and an associative binary operator $\oplus$ as input, and outputs an array $S = [s_0, s_1, \ldots, s_{n-1}]$, where $s_i = a_0 \oplus a_1 \oplus \ldots \oplus a_i$. When $\oplus$ is addition, and the input array consists of $n$ numbers, each of $O(\log n)$ bits, the prefix operation can be performed in $\Theta(\log n / \log\log n)$ time with $n \log\log n / \log n$ processors on a CRCW PRAM [16]. *Compression*, in which $m$ marked records out of a total of $n$ records must be compressed to the front of the output array, can easily be reduced to prefix addition, and thus can be performed in the same time bounds. Using only the processors assigned to the marked records, those marked records can be compressed in $\Theta(\log n)$ time [18]. We will call this a *marked compression*. When $\oplus$ is maximum or minimum, the prefix operation can be performed in $\Theta(\log\log n)$ time using $n/\log\log n$ processors [9, 34]. This obviously implies that the maximum or minimum of $n$ elements can be found in $\Theta(\log\log n)$ time with $n/\log\log n$ processors [36]. However, if we can use $n^{1+b}$ processors, for any $b > 0$, the maximum of $n$ elements can be found in constant time [36].

A special type of prefix operation is the *segmented prefix operation* in which the input array is divided into contiguous groups and a prefix operation is performed within each group in parallel. A segmented prefix operation can be performed in the same time bounds as a normal prefix operation.

**Merge** Two lists of size $n$ can be merged in $\Theta(\log\log n)$ time using $n/\log\log n$ processors [24].

**Sort** A list of $n$ items can be sorted in $\Theta(\log n)$ time using $n$ processors [3, 13]. With $n^2$ processors, the items can be sorted in $\Theta(\log n / \log\log n)$ time, by finding the position of each item using separate prefix operations.

**Global OR** Assuming each processor in some set of processors contains a 0 or a 1, the global OR of those processors' values can easily be found in constant time on an Arbitrary CRCW PRAM. However, on the Tolerant CRCW PRAM, we need to have exactly one designated processor to allow us to perform a global OR in constant time as follows. The designated processor initially writes 0 to a memory location. Then it writes 1 to that memory location while all the other processors write 1 only if they contain a 1. The designated processor now reads the location. If the location contains 0 then at least one other processor contains a 1 and so the designated processor writes 1 there. If the location contains 1 then no other processor contains a 1 and the designated processor writes its own value there. Note

that if we have a known block of processors, we can simply use the first processor in the block as the designated processor. Also note that we can perform a Global AND in a similar fashion.

**Broadcast** One processor can broadcast a value to any other set of processors in constant time by simply writing the value to a global memory cell. Each processor that wants to participate in the broadcast then can read the cell.

Now we list some lemmas that will be useful in our analysis. We use the term "randomly" to mean "with uniform distribution." In some lemmas, we will assume that a set of $n$ bins is partitioned into blocks of $\log\log n$ consecutive bins, or into superblocks of $\log^4 n$ consecutive blocks (i.e., $\log^4 n \log\log n$ consecutive bins).

**Lemma 2.1** *Given $n$ items randomly placed into $n$ bins, the probability that more than $6n/\log^8 n$ blocks of size $\log\log n$ will all contain at least $8\log\log n$ items is less than $1/n^3$.*

**Proof:** Use a Chernoff bound to place an upper bound on the probability that $6n/\log^8 n$ blocks each have at least $8\log\log n$ items, and multiply this by the number of choices of $6n/\log^8 n$ blocks out of $n/\log\log n$ blocks. □

**Lemma 2.2** *Given $n$ items randomly placed into $n$ bins, the probability that more than $4\log n$ fall into any block of size $\log\log n$ is less than $1/n^3$.*

**Proof:** Use a Chernoff bound to place an upper bound on the probability that $4\log n$ items fall into any block and multiply this by the number of blocks. □

The ideas in the following lemma were previously used in Stout [35] and Rajasekaran and Sen [31]

**Lemma 2.3** *For any $b > 0$, $n$ processors can each be allocated a position in an array of $n^{1+b}$ positions in constant time (which depends on $b$) with probability of failure less than $1/n$ in the CRCW PRAM model. (We assume that each processor has a unique identification number.)*

**Proof:** A processor attempts to allocate a position for itself in an array by writing its unique identification number to a random position. It then reads that position. If it contains its own identification number, then that processor has succeeded. Otherwise, it has failed.

For $b \geq 2$ the allocation takes one step. The probability of a processor failing is bounded by $1/n^2$, and thus the probability of any failing is bounded by $n(1/n^2) = 1/n$.

For $b < 2$, we assume without loss of generality that $n$ is large enough (depending on $b$) so that the following analysis holds. We divide the array into $T = \lceil 5/b \rceil - 1$ equal size subarrays, and perform the allocation in $T$ steps. At each step $i$ we attempt to allocate each of the remaining unallocated processors a position in subarray $i$. We first note that the probability of a processor failing at a step is less than $T/n^b$. Thus, if $n^a$ processors must be allocated in step $i$, for some $2b/5 < a \leq 1$, and $Z_i$ is the number which fail, then $Z_i$ is dominated by a random variable $Z_i' \sim B(n^a, T/n^b)$. Then by a Chernoff bound,

$$\Pr(Z_i \geq n^{a-(b/5)}) \leq \Pr(Z_i' \geq n^{a-(b/5)}) \leq 2^{-n^{a-(b/5)}} \leq 1/n^2.$$

At each step $t$ we see that we will have less than $n^{1-(tb/5)}$ processors left to allocate, and thus after $T-1$ steps, we will have less than $n^{2b/5}$ processors to allocate. The probability of a processor failing on the last step is bounded by $T/n^{1+(3b/5)}$ and so the probability of any processor failing on the last step is less than $n^{2b/5}(T/n^{1+(3b/5)}) \leq T/n^{1+b/5}$. □

Note that in the Padded Sort algorithm, we will always use the previous lemma with $b = 2$, and thus allocation will be accomplished in one step with high probability.

**Lemma 2.4** *Given $n$ items randomly placed into $n$ bins, the probability that more than $(\log^4 n + \log^3 n)\log\log n$ items fall into any superblock of $\log^4 n$ blocks is $O(n^{-2})$.*

**Proof:** Use a Chernoff bound to put an upper bound on the probability that $(\log^4 n + \log^3 n)\log\log n$ items fall into any superblock and multiply this by the number of superblocks. □

**Lemma 2.5** *For $k \geq 16$,*

$$\binom{n}{k}\left(\frac{2}{n}\right)^k\left(1-\frac{2}{n}\right)^{n-k} \leq e^{-(k\log k)/4}.$$

**Proof:** For $k \geq 16$,

$$\begin{aligned}
\binom{n}{k}\left(\frac{2}{n}\right)^k\left(1-\frac{2}{n}\right)^{n-k} &\leq \binom{n}{k}\left(\frac{2}{n}\right)^k \\
&\leq \left(\frac{en}{k}\right)^k\left(\frac{2}{n}\right)^k \\
&\leq \left(\frac{2e}{k}\right)^k \\
&\leq 2^{-k(\log k - \log 2e)} \\
&\leq e^{-(k\log k)/4}.
\end{aligned}$$

□

# 3 Padded Sort

Beame and Hastad [6] have shown that finding the parity of $n$ bits in any PRAM model requires $\Omega(\log n/\log\log n)$ time using any polynomial number of processors. From Ajtai and Ben-Or [2] and Chandra, Stockmeyer, and Vishkin [11], we can see that this lower bound applies even when randomization is allowed and/or the bits are chosen at random. This implies that sorting $n$ items into $n$ locations requires $\Omega(\log n/\log\log n)$ time using any polynomial number of processors, even if the items are taken from a uniform distribution. Fortunately, the lower bound of Beame and Hastad does not apply to a Padded Sort, in which $n$ items are sorted into $n+o(n)$ locations with some locations left blank, and we have found a way to use the distribution assumption and randomization to achieve a PT-optimal $\Theta(\log\log n/\log\log\log n)$ expected time algorithm.

A Padded Sort does not convey as much information as an unpadded sort, but if the important information is simply the relative ordering of items, and not the exact rank of the sorted items, then a Padded Sort will be as useful as an unpadded sort. Examples of problems in which a Padded Sort is directly useful include the Maximum Gap problem and the one dimensional All Nearest Neighbors problem.

The Padded Sort algorithm we present uses many of the same ideas as the serial algorithm for sorting items taken from a uniform distribution, so we will describe the serial algorithm here. The main idea is that since the $n$ items are taken from a uniform distribution over an interval, they will be roughly evenly distributed over that interval. Thus if we divide the interval into subintervals of equal length $1/n$ and assign a bin to each interval, we know that on average, 1 item will fall into each bin. It can be shown that the expected time to sort the items in a single bin will be constant, and thus the expected time to sequentially sort all the items in all the bins will be linear. The algorithm then simply places items in bins (usually implemented as linked lists) and sorts the items in each bin.

Unfortunately, a straightforward parallelization of the above technique does not lead to an efficient algorithm. The first problem is placing items in bins. The expected maximum number of items in a bin is

$\Theta(\log n/\log\log n)$ [19], and thus it would take that many naive attempts before all items were placed in bins. The other problem is that assuming each processor took one bin to sort, the expected maximum time would be $\Omega(\log n/\log\log n)$. However, we show that we can still use the bin technique, but in a much more careful way, to achieve an ultra-fast expected time parallel algorithm.

The algorithm for Padded Sorting is logically divided into two sections. The first is the placement of items into bins, and the second is sorting within bins. These sections are further divided into separate stages. Between the stages we will use a processor reallocation procedure described below. This processor reallocation could fail either because not enough items succeeded in the previous stage or because the items that did not succeed were badly arranged. If this happens, we revert to a deterministic logarithmic parallel sorting algorithm. We will show that the probability of this happening is less than $O(n^{-1})$, so it will not increase the expected running time of our algorithm.

### 3.1 Processor Reallocation

Here we give a $\Theta(\log\log n/\log\log\log n)$ expected time algorithm that can be used to approximately evenly distribute the processors over a random set of problems which need to be solved. (For the Placement in Bins section, this will distribute processors evenly to unplaced items, and for the Sorting within Bins section, this will distribute processors evenly over bins which have not been completely sorted.)

Formally, let $c \geq 1$ and $m \geq n\log\log\log n/\log\log n$ be integers, and let $p$ be a value between $1/\log^c n$ and $\log\log\log n/4\log\log n$. Assume we have an array $A$ of size $n$ in which there is a set of marked positions, and this set is taken from a distribution with the properties that (1) for any given number of marked positions $s$, any set of positions of size $s$ is equally likely to be marked, and (2) the probability that the number of marked positions is larger than $np$ is $O(n^{-2})$. Then with $m$ processors and in $\Theta(\log\log n/\log\log\log n)$ time we can with high probability allocate from one to $m/4np$ unique processors to each marked position. The probability of failing will be $O(n^{-2})$. We need the following Lemma.

**Lemma 3.1** *If we partition $A$ into $n/4\log^{c+1} n$ groups of size $4\log^{c+1} n$, then the probability that there are more than $16p\log^{c+1} n$ marked positions in any group is $O(n^{-2})$.*

**Proof:** First we can assume that $A$ contains at most $np$ marked positions, since the probability that it does not is $O(n^{-2})$. Now we analyze the probability of more than $16p\log^{c+1} n$ marked positions in a given group. Note that for a given position $j$ in this group, even conditioning on any other positions in this group being marked or unmarked, the probability of position $j$ being marked is at most $2p$. Let $Z$ be the number of marked positions in this group. Then $Z$ is a random variable which is dominated by a random variable $Z' \sim B(4\log^{c+1} n, 2p)$. Using a Chernoff bound we obtain

$$\begin{aligned}\Pr(Z \geq 16p\log^{c+1} n) &\leq \Pr(Z' \geq 16p\log^{c+1} n)\\ &\leq \frac{1}{2^{32p\log^{c+1} n/9}}\\ &\leq \frac{1}{n^{3.5}}.\end{aligned}$$

Now we can bound the probability of having over $16p\log^{c+1} n$ marked positions in any group by summing the probability of this occurring in one group over all groups. (Note that because we are simply summing probabilities, this bound applies regardless of dependencies between groups.) Since there are fewer than $n$ blocks, the total probability of any group having more than $16p\log^{c+1} n$ marked positions is at most $n^{-2.5}$. □

The Processor Reallocation algorithm proceeds as follows. We assign $4m\log^{c+1} n/n$ processors to each group and perform a compress operation in $\Theta(\log\log n/\log\log\log n)$ time. We then perform a prefix sum

in each group to count the number of marked items. If a processor finds that one of the prefix sums is greater than $16p\log^{c+1} n$, then we say this processor has failed. We perform a global OR to inform all the processors of a failure. It is the responsibility of the caller of this reallocation procedure to decide what to do on a failure. If there is no failure, then in each group, we will have $m/4np$ processors to allocate to each marked item. Let $k$ be the number of processors we wish to allocate to each marked item, where $1 \leq k \leq m/4np$. To the marked item at position $i$ in the compressed array, we will allocate processors $ki$ to $k(i+1)$.

### 3.2 Random Placement Procedure

We will also use a randomized procedure for placing (and compressing and sorting) $m$ items into an array of size $k$, assuming that each item has a processor assigned to it. Assume $A$ is the array of size $k$, and assume processor $p_i$ is assigned to item $i$. Processor $p_i$ then performs the following procedure. First it chooses a random $r$ between 0 and $k-1$, and writes $i$ into $A[r]$. Then it reads $A[r]$. If it reads $i$, it sets a local variable Succ $= 1$, else it sets Succ $= 0$. If Succ $= 1$, it participates in a compression over $A$ of the successful writes. Assume it ends up in position $s$ after compression. If $s = 0$, it considers itself the designated processor. Then (even if Succ $= 0$ or $s \neq 0$) it participates in a global AND over the Succ values of all $m$ processors. Now if Succ $= 1$, it sets AllSucc to be the result of the global AND. Otherwise, it sets AllSucc $= 0$. (Note that AllSucc $= 1$ if and only if all processors were successful in their writes. The key to this fact is that if any processor had Succ $= 1$, then there will be a designated processor, and the global AND will return the correct answer.) Now if AllSucc $= 1$, processor $p_i$ writes the actual item $i$ to $A[s]$, and participates in a Sort of the compressed list of items.

When this procedure is used in the Padded Sorting algorithm, we will specify the algorithms to be used for compression and sorting.

### 3.3 Processor Choice Procedure

We will encounter the situation in which we would like to choose one processor out of many, only knowing that each processor has a unique index from 1 to $k$. (Note that there might be fewer than $k$ processors, and for any given index $i$, it might be that no processor has that index.) To do this we use an array of size $k$, and have each processor mark a position according to its index. Then we perform a compression in the array, and choose the processor that ends up in the first position.

When this procedure is used we will specify the algorithm to be used for compression.

### 3.4 Padded Sort Algorithm

In this section, we give the algorithm for padded sort. It is straightforward to show that assuming we do not revert to a deterministic sorting algorithm, the running time of each stage is bounded by $O(\log\log n/\log\log\log n)$. Therefore in the analysis, we simply prove that the probability of reverting to a determinstc sorting algorithm is $O(n^{-1})$

#### 3.4.1 Placing Items in Bins

In this section, we assume that the unit interval is divided into $n$ equally sized subintervals, and a bin is associated with each subinterval. Placing the items in bins will be carried out in five stages, with processor reallocations between the first four stages. The fifth stage will simply merge the results from the first four stages into an array of size $n+o(n)$ in which items are sorted by bins, items in the same bin are in consecutive locations, and at most the first $16\log\log n/\log\log\log n$ items in each bin are not sorted.

| Stage | Number of processors assigned to each unplaced item | Items placed into | Total unplaced items (with high probability) |
|---|---|---|---|
| 1 | $1/v$ | bin array | $n/4v$ |
| 2 | 1 | array of size $8(\log\log n)^3$ per block | $n/32(\log\log n)^3$ |
| 3 | $8\log\log n$ | array of size $8(\log\log n)^3$ per block | $n/256\log^5 n$ |
| 4 | $64\log^4 n$ | array of size $4\log^3 n$ per block | 0 |
| 5 | gather items (sorted by bins) into an array $F$ of size $n + o(n)$ | | |

Table 1: Summary of the five stages for placing items in bins.

Assume the input is given in array $I$. For each item $i$, Bin$(i) = \lfloor n \cdot I[i] \rfloor$, or equivalently, the bin in which item $i$ should be placed. Also Block$(i) = \lfloor$Bin$(i)/\log\log n \rfloor$, or the block (of $\log\log n$ bins) in which item $i$ should be placed. Let PO be an array that stores for each input its position in its bin if it is placed in Stage 1, and otherwise $-1$. Let UNP be an array that stores for each input a binary value indicating whether it has been placed. Let CO be another array of size $n$ indicating the number of items placed in each bin in Stage 1. Let $v = \log\log n/\log\log\log n$. Table 1 summarizes the five stages for placing items in bins.

**Initialization** Initialize each element of PO to $-1$, each element of UNP to 1, and each element of CO to 0. Also for another array $D$ of size $n$, initialize each element to 0.

**Stage 1:** Split the $n$ items into $16v$ groups. For each group perform the following constant time procedure. Each processor $i$ ($0 \leq i < n/16v$) writes $i$ to $D$[Bin$(i)$], and then reads $D$[Bin$(i)$] to see if its write was successful. If so, it writes 0 to $D$[Bin$(i)$], sets UNP$[i] = 0$, sets PO$[i]$ to CO[Bin$(i)$], and increments CO[Bin$(i)$].

**Analysis of Stage 1:** For each item there will be at most a $1/16v$ probability of not being assigned to the lowest numbered processor writing to its bin. By a Chernoff bound, the probability of this occuring to more than twice the average number of items in a group is $O(n^{-2})$. Each of these items implies at most 1 other processor in a write conflict, and thus the probability of more than $4n/(16v)^2 = n/64v^2$ items from a group not being written is than $O(n^{-2})$. The location of these items within the group is obviously random. Thus, the following reallocation procedure will perform with probability of failure $O(n^{-2})$

**Reallocation 1:** Let $n/v^2$ processors be associated with each group of $n/v$ items. Then in each group of items in parallel, perform the Processor Reallocation procedure to assign one processor to each marked (i.e. unplaced) item. If the reallocation fails, revert to the deterministic sorting algorithm.

**Stage 2:** For each block $b$, using the processors assigned to each item $j$ such that Block$(j) = b$, perform a random placement procedure into an array $A_b$ of size $(8\log\log n)^3$. In this procedure, use marked compression as the compression algorithm, and Cole's parallel merge sort for the sort algorithm. If the procedure succeeds let each processor mark its item as placed. If the procedure fails for block $b$, (i.e. AllSucc $= 0$ for each processor) then repeat the procedure until either it succeeds, or the procedure has been run four times.

**Analysis of Stage 2:** From Lemmas 2.1 and 2.2, we see that the probability that there are over $24n/\log^7 n$ items from blocks with over $8\log\log n$ items in each is $O(n^{-3})$. By Lemma 2.3 the probability that

a block with $\leq 8 \log\log n$ items will fail in all four random placement attempts is $\leq (8 \log\log n)^{-4}$. By a Chernoff bound, we can show that the probability of over twice the average number occuring is $O(n^{-3})$. Thus the probability that we have over

$$\frac{2n(8\log\log n)}{(\log\log n)(8\log\log n)^4} + \frac{24n}{\log^7 n} \leq \frac{n}{32(\log\log n)^3}$$

items which have not been placed is $O(n^{-2})$. Within each group these items are at random positions. Also, given any distribution of items between the groups (and even if all items were in the same group) the following reallocation procedure will perform with probability of failure $O(n^{-2})$.

**Reallocation 2:** Let $n/v^2$ processors be associated with each group of $n/v$ items. Then in each group of items in parallel, perform the Processor Reallocation procedure to assign $8 \log\log n$ processors to each unplaced item. (We may assume that each processor, in addition to knowing the item it is assigned to, also knows its rank in the list of processors assigned to its item.) If the reallocation fails, revert to the deterministic sorting algorithm.

**Stage 3:** For each block $b$, let $B'_b$ be an array of size $8 \log\log n$ and let $A'_b$ be an array of size $(8 \log\log n)^3$. For each block $b$, and for each $i \in \{0, \ldots, 8\log\log n - 1\}$, let $C'_{b,i}$ be an array of size $(8 \log\log n)^3$. Now for each block $b$ and each $i \in \{0, \ldots, 8\log\log n - 1\}$, using the processors with rank $i$ assigned to each item $j$ such that Block$(j) = b$, perform a random placement procedure into array $C'_{b,i}$. In this procedure, use marked compression as the compression algorithm, and Cole's parallel merge sort for the sort algorithm. If the procedure succeeds for a given pair $(b, i)$, then the designated processor $p$ for $(b, i)$ participates in a processor choice procedure in array $B'_b$. In this procedure use marked compression as the compression algorithm. The processor that is chosen then broadcasts that fact to other processors with the same rank, and these processors mark their items as placed, and transfer their list of sorted items to array $A'_b$.

**Analysis of Stage 3:** From Lemmas 2.1 and 2.2, we see that the probability that there are over $24n/\log^7 n$ items from blocks with over $8 \log\log n$ items in each is $O(n^{-3})$. By Lemma 2.3 the probability that a block with $\leq 8 \log\log n$ items will fail in all $8 \log\log n$ random placement attempts is $\leq 2^{-8\log\log n}$. By a Chernoff bound, we can show that the probability of over twice the average number occuring is $O(n^{-3})$. Thus the probability that we have over

$$\frac{2n(8\log\log n)}{\log^8 n} + \frac{24n}{\log^7 n} \leq \frac{n}{256\log^5 n}$$

items which have not been placed is $O(n^{-2})$. Within each group these items are at random positions. Also, given any distribution of items between the groups (and even if all items were in the same group) the following reallocation procedure will perform with probability of failure $O(n^{-2})$.

**Reallocation 3:** Let $n/v^2$ processors be assigned to each group of $n/v$ items. Then in each group of items in parallel, perform the Processor Reallocation procedure to assign $64 \log^4 n$ processors to each unplaced item. If the reallocation fails, revert to the deterministic sorting algorithm. Otherwise, let Item$(p)$ be the item assigned to processor $p$, and let Rank$(p)$ be the rank of processor $p$ in those processors assigned to Item$(p)$. Let the superrank of a processor $p$ be $\lfloor$Rank$(p)/64\log^3 n\rfloor$, and let the subrank of $p$ be Rank$(p) \pmod{64\log^3 n}$. Let a processor $p$ with subrank 0 be called a main processor, and any processor $p'$ assigned to the same item as $p$ and with the same superrank as $p$, be called an auxiliary processor for $p$.

**Stage 4:** For each block $b$, let $B''_b$ be an array of size $\log n$ and let $A''_b$ be an array of size $(4 \log n)^3$. For each block $b$, and for each $i \in \{0, \ldots, (\log n) - 1\}$, let $C''_{b,i}$ be an array of size $(4 \log n)^3$. Now for

each block $b$ and each $i \in \{0, \ldots, (\log n) - 1\}$, using the main processors with superrank $i$ assigned to each item $j$ such that Block$(j) = b$, perform a random placement procedure into array $C''_{b,i}$. In this procedure, use the auxiliary processors to perform compression by computing prefix sums over the number of succesful writes before their main processor's successful write (if the main processor was successful). In the same manner, use these auxiliary processors to sort by computing prefix sums over the number of items smaller than their main processor's item. If the random placement procedure succeeds for a given $(b, i)$, then the designated processor $p$ for $(b, i)$ participates in a processor choice procedure in array $B''_b$. In this procedure, again use the auxiliary processors to perform compression, as above. The processor that is chosen then broadcasts that fact to the other main processors with the same superrank. Then these processors mark their items as placed, and transfer their list of sorted items to array $A''_b$. After this, perform a global OR to determine if any block failed in all of its random placement attempts. If so, revert to the deterministic sorting algorithm.

**Analysis of Stage 4:** In this stage we make $\log n$ independent attempts in each block to place its items into random positions in arrays of size $(4 \log n)^3$. ¿From Lemma 2.3, we know that for each independent attempt, if the number of items in the block is $\leq 4 \log n$, then the probability of not placing the items in one attempt is at most $1/4 \log n$. Then by a Chernoff bound, we can see that the probability of all $\log n$ attempts failing is $O(n^{-2})$. ¿From Lemma 2.2, the probability that any block has over $4 \log n$ items is $O(n^{-3})$, so the probability that any block will fail to have a conflict-free placement must be $O(n^{-2})$.

**Stage 5:** For each block $i$, merge the items placed in Stages 2, 3, and 4 (from arrays $A_i$, $A'_i$, and $A''_i$) using the processors assigned to them in Reallocation 1. Then using segmented prefix sums, count the number of items in each bin $b$ placed in Stages 2, 3, and 4. Add this to CO$[b]$. Now for each superblock $j$ of $\log^4 n$ consecutive blocks, perform the following. Use $\log^4 n \log \log \log n / \log \log n$ processors to perform a prefix operation over the $\log^4 n \log \log n$ positions in the CO array corresponding to the bins in this superblock. This will find for each bin $b$, the number of items that precede it in the superblock. Now using a global OR, check if there are more than $(\log^4 n + \log^3 n) \log \log n$ items in any superblock. If so, revert to a deterministic sorting algorithm. Otherwise, let $F$ be the output array of size $n + (n/\log n)$, and let $x = (\log^4 n + \log^3 n) \log \log n$. Let $S(b)$ be the number of items that precede bin $b$ in the superblock. Now for each group of $n/16v$ items perform the following constant time procedure. Each processor $i$ $(0 \leq i \leq n/16v)$ writes item $i$ to $F[jx + S(b) + \text{PO}[i]]$, if PO$[i] \geq 0$. After this, place the items merged from Stages 2, 3, and 4 after the last item placed from Stage 1.

**Analysis of Stage 5:** By Lemma 2.4, the probability that more than $(\log^4 n + \log^3 n) \log \log n$ items fall into any superblock is $O(n^{-2})$.

### 3.4.2 Sorting Within Bins

Assume we have correctly placed the $n$ items into their corresponding bins. Now we must sort the items within each bin. This procedure is divided into two stages, with a reallocation of processors between the stages. Again let $v = \log \log n / \log \log \log n$.

**Stage 1:** The $n$ bins are divided equally among the processors, and each processor is given at most $144v$ steps to try to sort the items in its $v$ bins. The processors can use a standard serial sorting algorithm, such as MergeSort.

**Analysis of Stage 1:** The left hand side of the inequality in Lemma 2.5 is the maximum probability of exactly $k$ items landing in one bin given that we have seen $\leq n/2$ other bins. Now assume we have a serial sorting algorithm that sorts $k$ items in exactly $ck \log k$ steps, for some constant $c$. Then the

probability that it takes exactly $4ct$ steps to sort a bin will be less than $1/e^t$. Thus the time to sort a bin is bounded by a simple exponential distribution. Without loss of generality, we can assume $4ct \leq 8t$.

In this stage, each processor has $v$ bins to sort and is given a total of $144v$ steps. $80v$ steps will take care of sorting all bins with fewer than 10 items. Now we will bound the probability that more than $n/4v(\log\log n)^2$ groups of $v$ bins (each with over 10 items) take more than $64v$ steps each to sort. From the discussion above, the time to sort a bin is bounded by an exponential distribution. Let $Z$ be the time to sort $n/4(\log\log n)^2$ bins. Then $Z$ is dominated by a random variable $Z' \sim \Gamma(n/4(\log\log n)^2, 1/8)$. We can bound the tail of $Z'$ using a Chernoff bound, and multiplying by the number of possible choices of $n/4v(\log\log n)^2$ out of $n/v$ groups, we see that $\Pr(Z \geq 16n/(\log\log n)^2) \leq O(n^{-2})$. Also, the processors that fail are randomly distributed, so the following reallocation procedure will perform with failure probability $O(n^{-2})$.

**Reallocation 1:** Perform the Processor Reallocation procedure to assign $(\log\log n)^2$ processors to each unfinished processor, and more specifically, $\log\log n$ to each of the unfinished processor's bins. If the reallocation fails, revert to the deterministic sorting algorithm.

**Stage 2:** For each bin $b$, deterministically sort the at most $16v$ unsorted items in $b$. Now bin $b$ has at most 2 sorted sublists in it. If $\mathrm{CO}[b] \leq 2\log\log n$, merge the lists using the $\log\log n$ processors assigned to the bin in the Reallocation 1 of Sorting Within Bins. Otherwise, merge the lists using the processors assigned in Reallocation 1 of Placing Items into Bins to the items placed in this bin in Stages 2, 3, and 4. In either case, there will be at most twice as many items as processors.

**Analysis of Stage 2:** Stage 2 is deterministic and always succeeds.

It is a simple matter to make sure that each unused location contains the value NULL, and this completes Padded Sort.

We have therefore proven the following theorem.

**Theorem 3.1** *Given $n$ values taken from a uniform distribution over the unit interval, in $\Theta(\log\log n/\log\log\log n)$ expected time and using $n\log\log\log n/\log\log n$ processors, these values can be arranged in sorted order in an array of size $n + o(n)$ with the value NULL in all unfilled locations.*

We note that although the size of the output is only $n+o(n)$, we actually use superlinear ($n \cdot \mathit{polylog}(n)$) space during the Padded Sort algorithm.

# 4 Applications of Padded Sort

By having each processor choose a random number uniformly from $[0, 1]$ and performing a Padded Sort, we will obviously be left with the processors in random order. We can easily obtain a random cycle of the processors from this using an algorithm for chaining [10], and we can obtain a random permutation of the processors by compressing the padded list using a prefix sum operation. (In a random cycle of processors, each processor contains a link to another processor, the links form a simple cycle, and each possible cycle is equally likely.) Thus, by using the Padded Sort algorithm given above, a random cycle can be constructed in $\Theta(\log\log n/\log\log\log n)$ expected time with $n\log\log\log n/\log\log n$ processors, and a random permutation can be constructed in $\Theta(\log n/\log\log n)$ expected time with $n\log\log n/\log n$ processors.

This result is not optimal, as shown by Gil, Matias, and Vishkin [17], who give an algorithm to construct a random permutation in $\Theta(\log^* n)$ expected time using $n/\log^* n$ processors, where $\log^{(1)} n \equiv \log n$, $\log^{(i)} n \equiv \log(\log^{(i-1)} n)$ for $i > 1$, and $\log^* n \equiv \min\{i : \log^{(i)} n \leq 2\}$.

We note that the Padded Sort algorithm given above can also be viewed solely as a item distributing procedure, which simply places items into their corresponding bins. This item distributing property makes the Padded Sort algorithm very useful in solving many other "proximity" problems in $\Theta(\log\log n/\log\log\log n)$ expected time with $n\log\log\log n/\log\log n$ processors. We show some important examples here.

### 4.1 All Nearest Neighbors

In this problem we are given $n$ points taken from a uniform distribution over the unit square, and we are asked to find each point's nearest neighbor. To do this, we follow the technique of Bentley, Weide, and Yao [7] and divide the square into $n$ equal subsquares in a $\sqrt{n}\times\sqrt{n}$ grid. The points can be placed into bins corresponding to these subsquares just as in the Padded Sort algorithm. To find the nearest neighbor to a point, we simply examine the subsquares around that point in a spiral fashion until we are sure we have found the nearest neighbor. The probability of a search taking more than $Ki$ steps has been shown to be $e^{-i}$, for some K [7]. Thus the search time is bounded by a simple exponential distribution, and this enables us to use techniques similar to the ones used in Padded Sorting (where sorting time per bin was bounded by an exponential distribution) to find the nearest neighbors in $\Theta(\log\log n/\log\log\log n)$ time using $n\log\log\log n/\log\log n$ processors. The details here are not complicated and are left to the reader.

### 4.2 Relative Neighborhood Graph

The procedure for constructing the Relative Neighborhood Graph is very similar to the one for solving All Nearest Neighbors. We refer the reader to Katajainen, Nevalainen, and Teuhola [22] for the details of the differences.

### 4.3 Voronoi Diagram

We construct the Voronoi Diagram with two separate procedures, one which constructs the part inside the unit square, and one which constructs the part outside the unit square. The procedure for constructing the Voronoi Diagram inside the unit square will be very similar to the one given above for solving All Nearest Neighbors. We refer the reader to Bentley, Weide, and Yao [7] for the details of the transformation. Constructing the Outer Voronoi Diagram, that part of the Voronoi Diagram which lies outside the unit square, requires more work, but it too can be accomplished in $\Theta(\log\log n/\log\log\log n)$ expected time using $n\log\log\log n/\log\log n$ processors. We show the details here.

We will use the following lemmas.

**Lemma 4.1** *Given a set of $n$ points taken from a uniform distribution in the unit square, and given a region $R$ within the unit square of area $p$, where $\log n/n \le p \le 1$, the probability that over $4pn$ points lie within this region is less than $1/n^2$.*

**Proof:** Let $Z$ be the number of points which lie within the region $R$. Then $Z \sim B(n,p)$, and by a Chernoff bound,

$$P(Z \ge 4np) \le 4^{-np} \le 4^{-\log n} \le \frac{1}{n^2}.$$

□

**Lemma 4.2** *Given a set of $n$ points in the plane, one can find the Voronoi cell around a point in constant time with $n^{2+\epsilon}$ processors.*

**Proof:** To find the Voronoi cell around a point $p$ we simply need to find the intersections of the halfspaces defined by the perpendicular bisectors between $p$ and the other $n-1$ points. Thus, for each perpendicular bisector $b$, we must find the section (if it exists) which is not behind the other perpendicular bisectors, when viewed from $p$. Each of the other bisectors will restrict the visible section of $b$, and with $n^{1+\epsilon}$ processors we can find the most restrictive limits in constant time. We will do this simultaneously for each perpendicular bisector. If the most restrictive limits on a bisector define a non-empty interval then this interval is an edge to the Voronoi cell of $p$. □

**Corollary 4.1** *Given a set of $n$ points in the plane, one can find the Voronoi diagram in constant time with $n^{3+\epsilon}$ processors.*

Now we define three strips around the edge of the unit square. Let $R$ be a strip of width $2\log n/n^{0.5}$, $S$ be a strip of width $2/n^{0.7}$, and $T$ be a strip of width $1/n^{0.7}$ (see Figure 1). Note that $T \subset S \subset R$. Also note that the probability of any points not in $R$ contributing to the Outer Voronoi Diagram is less than $1/n$ (shown in Bentley, Weide, and Yao), and so we only need to be concerned with points in $R$. We will describe how to find the Outer Voronoi Diagram from the points along one side of the unit square. The other sides and the corners will follow with similar arguments. The points in $R$ along one side are uniformly distributed so we assume they can be placed in sorted order using a Padded Sort.

First we will find the contribution to the Outer Voronoi Diagram from the points in the area $R-S$. From Lemma 4.1 we see that the probability of more than $O(n^{0.5}\log n)$ points in $R-S$ is less than $1/n^2$. Also, given a point $p$ in $R-S$, the probability that no points lie in $T$ for a length of $2\log^2 n/n^{0.3}$ in either direction from $p$ is less than $e^{-2\log n} \leq 1/n^2$. Then it is easy to see that only points within a distance $4\log^2 n/n^{0.3}$ from $p$ in $R$ would have any affect on the Outer Voronoi Cell of $p$ (see Figure 2). By Lemma 4.1, we see that the probability of more than $O(n^{0.2}\log n)$ points in this region is less than $1/n^2$. Now if we assign $O(n^{0.45})$ processors to each point in $R-S$, we can find all of their Voronoi cells in constant time by Lemma 4.2.

Now we must find the Outer Voronoi Cell for each point in $S$. We note that by Lemma 4.1 the probability that there are more than $O(n^{0.3})$ points in $S$ is less than $1/n^2$, and by the above discussion, the probability that over $O(n^{0.2}\log n)$ points in $R-S$ affect the Outer Voronoi Cell of a point in $S$ is less than $1/n^2$. Thus for each point $p$ in $S$, to construct the Outer Voronoi Cell, we simply must consider the $O(n^{0.3})$ points in $S$ and the $O(n^{0.2}\log n)$ points closest to $p$ in $R-S$. Then to each point $p$ in $S$ we can assign $n^{0.65}$ processors and use them to find the Outer Voronoi Cell of $p$ in constant time, according to Lemma 4.2. Since $S$ contains $O(n^{0.3})$ points, all the Outer Voronoi Cells can be found simultaneously, and this completes the construction of the Voronoi Diagram.

## 4.4 Delaunay Triangulation

From Preparata and Shamos [28], we know that the Delaunay Triangulation is simply the straight line dual of the Voronoi Diagram. Thus we can find the Voronoi Diagram as above, and easily construct the dual in $\Theta(\log\log n/\log\log\log n)$ time.

## 4.5 Largest Empty Circle

From Preparata and Shamos [28], we know that the midpoint of the Largest Empty Circle must be on a vertex of the Voronoi Diagram, or on the intersection of a line segment from the Voronoi Diagram and the Convex Hull.

We find the Voronoi Diagram as above. To find the Convex Hull, first we find the extreme points (i.e. those points which are corners of the Convex Hull) in $\Theta(\log\log n/\log\log\log n)$ time, using the constant time algorithm given in Stout [35], but simulating $\log\log n/\log\log\log n$ processors with 1 processor. Then we use the following lemma.

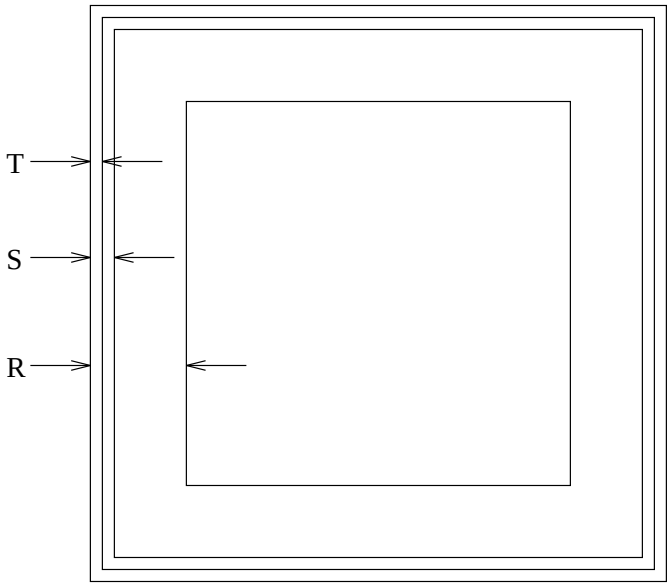


Figure 1: Strips used in computing the Outer Voronoi Diagram.

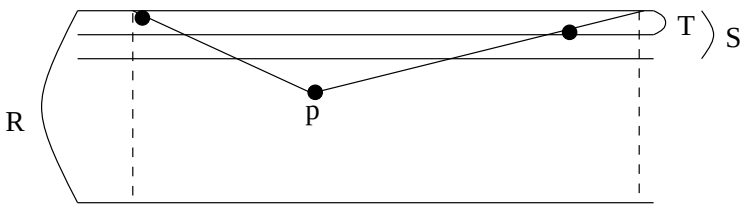


Figure 2: Only points within the dotted lines can affect the Outer Voronoi Cell of point $p$.

**Lemma 4.3** *Given $n$ points taken from a uniform distribution in the unit square, the probability that there will be more than $\log^2 n$ extreme points is $O(1/\log^3 n)$.*

**Proof:** It is easy to see that the number of extreme points is less than or equal to the total number of maximal points found in the following four orientations of the plane: standard, rotated by $90°$, rotated by $180°$, and rotated by $270°$. Since the probabilities for all four orientations are equivalent, we simply need to show that the probability of more than $\log^2 n$ maximal points in the standard orientation is less than $O(1/\log^3 n)$.

Let us number the points from 1 to $n$ by increasing $y$ coordinate. Since the $x$ coordinates and $y$ coordinates of points chosen from a uniform distribution are independent, we see that if points are placed in order by decreasing $x$ coordinate, then any permutation of 1 to $n$ is equally likely. We consider position $k$ a maximal position if the number stored at position $k$ is larger than all the numbers stored at positions before $k$. The number of maximal positions is then equal to the number of maximal points.

The analysis of the number of maximal positions is given in Knuth [23]. The average is less than $\log n$ and the standard deviation is less than $\sqrt{\log n}$. Then by Chebyshev's Inequality, the probability that there are more than $\log^2 n$ extreme points is $O(1/\log^3 n)$. □

We attempt to place the extreme points into an array of size $\log^4 n$. By Lemma 2.3 and Lemma 4.3 the probability of failing is less than $1/\log^2 n$. If we do fail, we simply find the Largest Empty Circle deterministicly. It is trivial to show that this takes $O(\log^2 n)$ time, so that it will add only a constant to the expected time.

If we succeed, then in $\Theta(\log\log n/\log\log\log n)$ time we can sort the extreme points and construct the Convex Hull. The probability that any point not in the outer $2\log n/\sqrt{n}$ width strip around the outside of the unit square is not inside the Convex Hull is less than $1/n^2$, so we only must check the $O(\sqrt{n}\log n)$ vertices of the Voronoi Diagram in this strip to see if they lie inside the Convex Hull. We can assign $\log^2 n$ processors to each vertex and check each one of these simultaneously in constant time. We then perform a $\max$ operation on the at most $n$ empty circles corresponding to those vertices inside the Convex Hull. Similarly, we find the intersections of the Voronoi Segments with the Convex Hull, and perform a $\max$ operation on the at most $n$ empty circles corresponding to those intersections. The larger of the two circles found will be the Largest Empty Circle.

We summarize the results given in this section in the following theorem.

**Theorem 4.1** *Given $n$ points taken from a uniform distribution over the unit square, in $\Theta(\log\log n/\log\log\log n)$ expected time and using $n\log\log\log n/\log\log n$ processors, we can*

1. *solve the All Nearest Neighbors problem,*
2. *construct the Relative Neighborhood Graph,*
3. *construct the Voronoi Diagram,*
4. *construct the Delaunay Triangulation, and*
5. *find the Largest Empty Circle.*

# 5 Closest Pair

Rabin [29] suggests the following algorithm for finding the closest pair of $n$ points in linear time. Take a random sample of $n^{2/3}$ points and find the closest pair of this sample recursively. Then form a lattice with a mesh size equal to the distance between the closest pair in the sample. Obviously the points of the closest pair must be within one square of each other, and Rabin shows that with high probability only $O(n)$ combinations of points have this property. Finding the minimum of these will thus take linear time, and the recurrence is bounded, so the total time of the algorithm is also linear.

A trivial parallelization of this algorithm will not work well because there is only a bound of $O(\sqrt{n})$ on the number of points in any square. This will cause major problems when we try to sort points by squares. Moreover, we are not able to apply any techniques from Padded Sort because there is no assumption on the distribution of points. Fortunately, we can still use most of Rabin's ideas and simply strengthen his intermediate results to obtain a good parallel algorithm. Before we begin to dicuss these, though, we will present some technical lemmas to aid our analysis.

**Lemma 5.1** *Given $k < 1$, a random sample of size $n^k$ is drawn from a set $S$ of size $n$. Then the probability that $n^{1-k} \log n$ or more items from the set $S$ are smaller than the minimum item in the random sample is less than $1/n$.*

**Proof:** The probability that any item chosen from the set $S$ is greater than $n^{1-k} \log n$ items is at most $1 - (\log n / n^k)$. Then the probability that every item from the random sample has this property is less than $(1 - (\log n / n^k))^{n^k} \leq e^{-\log n} \leq 1/n$. □

**Lemma 5.2** *The minimum (or equivalently, the maximum) of $n$ items can be found in constant expected time with $n$ processors.*

**Proof:** We assume one item is assigned to each processor. We take a random sample by having each processor decide to include its item in the random sample with probability $1/n^{1/2}$. Using a Chernoff bound, we see that the probability of a sample with more than $(3/2)n^{1/2}$ items or less than $(1/2)n^{1/2}$ items is less than $1/n$. ¿From Lemma 2.3, these items can be placed into an array of size $n^{3/4}$ in constant time with probability of failure less than $1/n$. Now with $n$ processors we can find the minimum of the items in the random sample in constant time. By Lemma 5.1 the probability that there are more than $2n^{1/2} \log n$ items which are less than the minimum in the sample is less than $1/n$, and by Lemma 2.3 these can be written to an array of size $n^{3/4}$ in constant time with probability of failure less than $1/n$. Then the minimum of these items (which is the minimum of all $n$ items) can be found in constant time with $n$ processors. If any step fails, we simply run a $\Theta(\log \log n)$ time deterministic algorithm to find the minimum. This will not add more than a constant to our expected time. □

We now return to the description of a parallel closest pair algorithm. We begin by defining the notation used in Rabin [29]. Let $S = \{x_1, \ldots, x_n\}$ be a set of points in the plane. If $S = S_1 \cup \ldots \cup S_k$ is a decomposition $D$ of $S$ and $|S_i| = n_i$, then the measure of $D$ is defined as $N(D) = \sum_{i=1}^{k} n_i(n_i - 1)/2$. This is simply the number of possible pairings of points which are both in the same subset of the decomposition. If we know that the nearest pair of points is within one of the $S_i$, then we simply need to find the minimum distance between these $N(D)$ pairs of points.

Let $\Gamma$ be a square lattice of mesh size $\delta$. Let $\delta(S)$ be the distance between the closest pair in $S$. If $\delta(S) < \delta$, then the closest pair must lie within the same square of $\Gamma$ or in two squares with a common corner. Let $N(\Gamma)$ denote the measure of the decomposition of $S$ formed by the lattice $\Gamma$. Also let $N'(\Gamma, t)$ denote the measure of the decomposition of $S$ formed by the lattice $\Gamma$ but only including those subsets which have more than $t$ points.

**Lemma 5.3** *Let $\Gamma$ be a lattice of mesh size $\delta$. Construct a lattice $\Gamma_1$ by choosing a fixed lattice point $y$ of $\Gamma$ as a lattice point of $\Gamma_1$ and forming the lattice with mesh size $2\delta$ and lines parallel to those of $\Gamma$. Then for a fixed set $S$ of $n$ points,*

$$N(\Gamma_1) \leq 16N(\Gamma) + 24n$$
$$N'(\Gamma_1, 4) \leq 32N(\Gamma).$$

**Proof:** The first inequality is proven by Rabin. Now consider a square in $\Gamma_1$ with at least $5$ points. It must have a subsquare in $\Gamma$ with at least $2$ points and thus the measure in the square is at most

$$\frac{4k(4k-1)}{2} \leq \frac{32k(k-1)}{2},$$

where $k \geq 2$ is the maximum number of points in any of the four subsquares. We obtain the second inequality in the lemma by summing over all squares in $\Gamma_1$ with at least $5$ points. □

**Corollary 5.1** *There exists a constant $c$ such that for every $S$, $\Gamma$, and $\Gamma_1$ as above, if $N(\Gamma) \leq n^{0.4}$, then $N(\Gamma_1) \leq cn$ and $N'(\Gamma_1, 4) \leq cn^{0.4}$.*

If we start from a lattice in which all points are in individual squares, we can keep doubling this lattice as in Lemma 5.3 until we find $N(\Gamma_1) \geq n^{0.4}$. At this point, $N(\Gamma) \leq n^{0.4}$, so the conditions of Corollary 5.1 still hold for $\Gamma_1$. Let $\Gamma_0 = \Gamma_1$ at this point, and let $\delta_0$ be the mesh size of $\Gamma_0$.

**Lemma 5.4** *Let $D$ be a partition of the set $S$, $|S| = n$, and $n^{0.4} \leq N(D)$. If $n^{0.9}$ pairwise different points are drawn at random from $S$, then the probability that two elements will be chosen from the same set of $D$ is at least $1 - 2e^{-cn^{0.1}}$, for some $c$.*

**Proof:** Rabin shows that we can substitute a partition $D'$ for $D$ in which exactly one subset has more than $1$ element, $\lambda n^{0.4} \leq N(D')$ for some constant $\lambda$, and the probability that two elements will be chosen from the same set of $D'$ is greater than the equivalent probability in $D$. Thus if $p$ is the size of the only non-singular set in $D'$, then $2\lambda n^{0.4} \leq p(p-1)$ so that $cn^{0.2} \leq p$ for $c \sim \sqrt{2\lambda}$.

The probability that in one choice from $S$ we miss this non-singular subset is $1 - p/n$, which is smaller than $1 - c/n^{0.8}$. Then for $n^{0.9}$ choices the probability of all missing this subset is smaller than

$$\left(1 - \frac{c}{n^{0.8}}\right)^{n^{0.8}(n^{0.1})} \leq e^{-cn^{0.1}}.$$

The probability that two elements are chosen from this set is thus greater than $1 - 2e^{-cn^{0.1}}$. □

Since $N(\Gamma_0) \geq n^{0.4}$, if we take a random sample of size $n^{0.9}$, we know that two points must fall within a single square of $\Gamma_0$ with probability $1 - 2e^{-cn^{0.1}}$. Then the distance $\delta$ between the closest pair of points will be less than $\sqrt{2}\delta_0$. Thus if we consider a lattice $\Gamma$ with mesh size $\delta$, each square will be covered by a square found when quadrupling $\Gamma_0$. We can see then that $N'(\Gamma, 64) \leq O(n^{0.4})$ and $N(\Gamma) \leq O(n)$.

The actual algorithm proceeds as follows. We take a random sample by having each processor decide to include its point in the random sample with probability $1/n^{0.1}$. Using a Chernoff bound, we see that the probability of a sample with more than $(3/2)n^{0.9}$ points or less than $(1/2)n^{0.9}$ points is less than $1/n$. ¿From Lemma 2.3, these points can be placed into an array of size $n^{0.95}$ in constant time with probability of failure less than $1/n$. We find the closest pair in this random sample recursively. Then we form a lattice with a mesh size equal to the closest distance found and find the number of squares vertically and horizontally which we must use to cover all the points. This again can be done with the constant expected time minimum and maximum finding algorithm. We then assign a position in memory to each square in this region.

We will place the points into squares by writing each point simultaneously to the position corresponding to its square. If a point succeeds in being written, then it is placed in an array of size $64$ for its square. Otherwise, it tries again. We do this $64$ times. After this, if any points are left over, their processors write *dense* to the positions corresponding to their squares. All processors can then tell if their points are in dense squares. If so, these points will be called the *dense* points. All the points which are in squares with less than $64$ points will be called *sparse* points.

We have shown that when one forms this lattice, the number of distance comparisons which must be made is between $O(n^{0.4})$ and $O(n)$, but that only $O(n^{0.4})$ comparisons come from dense squares. We will now separately find the closest pair of the dense points, and then find for every point the closest neighboring sparse point.

To find the closest pair of the dense points, we simply write them into an array of size $n^{0.45}$. By Lemma 2.3 this will take constant time with probability of failure less than $1/n$. Then we use $n^{0.9}$ processors to find the distances between all pairs of dense points. The minimum of these $n^{0.9}$ distances can then be found in constant time with $n$ processors.

For each point, to find the closest neighboring sparse point we simply examine the at most $9 \cdot 64$ points in its own and neighboring sparse squares. Then we perform the constant expected time minimum operation from Lemma 5.2 to find the closest pair of these $O(n)$ pairs.

The closest pair of all the points must be two dense points or a point and a neighboring sparse point, so it can be found simply by comparing the distances of the two pairs found above. Given that after 16 stages of recursion, there will be less than $n^{0.45}$ points left, and the closest pair of these can easily be found in constant time, we have the following theorem.

**Theorem 5.1** *Given $n$ points in the plane, we can determine the closest pair of points in constant expected time using $n$ processors.*

# 6 Padded Sort Revisited

We note that the lower bound of $\Omega(\log n / \log\log n)$ time on sorting $n$ items into $n$ locations applies for any polynomial number of processors. However, Padded Sort, and the geometry problems which have similar solutions, all can be solved in constant expected time when one is allowed to use $n^{1+\epsilon}$ processors for any constant $\epsilon > 0$. We show how to do this here.

## 6.1 Padded Sort

First we prove the following technical lemmas.

**Lemma 6.1** *For large $m$, given $m$ items to be randomly placed into an array of size $km$, for some constant integer $k > 1$, the probability that there will be a conflict is less than $1 - e^{-m/k}$.*

**Proof:** Let $A$ be the event that there is a conflict. Then $\Pr(A)$ can be bounded by

$$\begin{aligned}
\Pr(A) &= 1 - \frac{(km)!}{(km-m)!}(km)^{-m} \\
&\leq 1 - \left( \frac{\left(\frac{km}{e}\right)^{km} \sqrt{2\pi km}}{\left(\frac{km-m}{e}\right)^{km-m} \sqrt{2\pi(km-m)} e^{1/12(km-m)}} \right) (km)^{-m} \\
&= 1 - \left( \frac{km}{km-m} \right)^{km-m} e^{-m} \sqrt{\frac{km}{km-m}} e^{-1/12(km-m)} \\
&\leq 1 - \left( \frac{k}{k-1} \right)^{(k-1)m} e^{-m} \sqrt{\frac{k}{k-1}} \left( 1 - \frac{1}{12(km-m)} \right) \\
&\leq 1 - \left( \frac{k}{k-1} \right)^{(k-1)m} e^{-m} \left( 1 + \frac{1}{3(k-1)} \right) \left( 1 - \frac{1}{12(km-m)} \right) \\
&\leq 1 - \left( \frac{k}{k-1} \right)^{(k-1)m} e^{-m}
\end{aligned}$$

$$
\begin{aligned}
&= 1 - \frac{1}{\left(\frac{k-1}{k}\right)^{(k-1)m} e^m} \\
&\leq 1 - \frac{1}{e^{-(1-1/k)m} e^m} \\
&= 1 - e^{-m/k}.
\end{aligned}
$$

(See B for some of the mathematical details.) □

**Lemma 6.2** *Given $m$ items to be randomly placed into an array of size $km$, for some constant $k$, the probability of failing on all of $e^{2m/k}$ attempts is less than $e^{-e^{m/k}}$.*

**Proof:** Let $A$ be the event of failing on all attempts. Then $\Pr(A)$ can be bounded by

$$
\begin{aligned}
\Pr(A) &\leq (1 - e^{-m/k})^{e^{2m/k}} \\
&\leq e^{-e^{m/k}}
\end{aligned}
$$

□

**Lemma 6.3** *Given an array of $m$ values from $0$ to $b-1$, the prefix sums can be computed in constant time with $mb^m$ processors.*

**Proof:** Note that there are $b^m$ possible sequences of $m$ values, and we have $m$ processors for each sequence. Each processor in a sequence will be initialized with the value at its position in the sequence and the prefix sum at its position in the sequence. Each processor can then check to see if its value is the same as the input value at its position. A global Or can be used to test for a failure. Exactly one sequence will not have a failure, and the processors corresponding to this sequence can write their prefix sums to the output array. □

**Lemma 6.4** *Given $m$ values from $0$ to $b-1$, and a constant integer $k > 0$, the prefix sums can be computed in constant time with $mb^{m^{1/k}(1+\log m)}$ processors.*

**Proof:** Compute the prefix sums of groups of $m^{(k-1)/k}$ values recursively, assigning $m^{(k-1)/k} b^{m^{1/k}(1+\log m)}$ processors to each group. Note that the recursion will stop when there are $m^{(k-1)/k}$ groups of $m^{1/k}$ values. At this point each group is assigned more than $m^{1/k} b^{m^{1/k}}$ processors, and by Lemma 6.3, the prefix sums can be computed in constant time.

Now compute the prefix sums over the groups, i.e., over the $m^{1/k}$ values from $0$ to $bm^{(k-1)/k}$ corresponding to the total sum in each group. By Lemma 6.3 these can be computed in constant time, with $m^{1/k}(bm^{(k-1)/k})^{m^{1/k}} \leq m^{1/k} b^{m^{1/k}(1+\log m)}$ processors.

Now we assign one processor per location. This processor can find the prefix value for its location by summing at most $k$ prefix sums computed at different levels of recursion. □

**Lemma 6.5** *Given $m$ items, and a constant integer $k > 0$, these items can be sorted into $m$ locations in constant time with $m^2 2^{m^{1/k}(1+\log m)}$ processors*

**Proof:** We will assign $m2^{m^{1/k}(1+\log m)}$ processors to each item to find its position in the sorted array. For each item $t$, we will compare $t$ with every other item $s_i$ and thus compute an array of length $m$ which contains a $1$ at position $i$ if $t > s_i$, and otherwise contains a $0$. The sum of these bits is the position of $t$ in the sorted list, and by Lemma 6.4, this sum can be found in constant time with the exact number of processors we have assigned to each $t$. □

We now return to the constant expected time Padded Sort algorithm which uses $n^{1+\epsilon}$ processors. Let $j$ be a constant integer such that $4/j < \epsilon$. We attempt to place items randomly into the correct block of $\log n$ bins. By a Chernoff bound, the probability that there are more than $4 \log n$ items to be placed in any block is less than $1/n$. We allocate $n^{4/j}$ processors to each item, so we can make $n^{4/j} = 2^{4 \log n/j}$ independent attempt to place points in the correct block. If we choose the array size for a block to be $16j \log n$, then by Lemma 6.2 (with $m = 4 \log n$, $k = 4j$, and noticing that $2^{4 \log n/j} > e^{2m/k}$) the probability of not having a correct placement in a block is less than $e^{-e^{4 \log n/4j}}$. Thus the probability of any block not having a correct placement is less than $(n/\log n)e^{-n^{1/j}} \leq 1/n$, for large $n$. Using a decision broadcast, we can find a correct placement in each block.

Now we form groups of $\log^3 n$ blocks, so that each group has $\log^4 n$ bins associated with it. We will sort the items within these groups. We have at most $16j \log^4 n$ positions to sort for each group, and $n^{4/j} \log^4 n = (\log^4 n)2^{4 \log n/j}$ processors for each group. By Lemma 6.5 (with $m = 16j \log^4 n$, $k = 8$, and $n$ large enough so that $(16j \log^4 n)^2 2^{(16j \log^4 n)^{1/8}(1+\log(16j \log^4 n))} \leq (\log^4 n)2^{4 \log n/j}$), we can sort these into a compressed list (assuming we mark each position without an item as greater than any item) in constant time. Also, in a proof similar to Lemma 2.4, we can show that the probability that there are more than $\log^4 n + \log^3 n$ items in any group is less than $1/n$. Thus we can now write the items into an array of size $n + o(n)$ without conflicts, and we are finished.

### 6.2 All Nearest Neighbors

To solve All Nearest Neighbors with $n^{1+\epsilon}$ processors, we will again place items into blocks of $\log^2 n \times \log^2 n$ cells, but use the constant time procedure given above for Padded Sort. Then we can assign each point $n^\epsilon$ processors and each one can perform a constant expected time minimum operation on the distances between itself and the $\log^2 n \leq n^\epsilon$ neighbors closest to it.

### 6.3 Relative Neighborhood Graph

An algorithm similar to the one for All Nearest Neighbors will find the Relative Neighborhood Graph in constant expected time.

### 6.4 Voronoi Diagram

An algorithm similar to the one for All Nearest Neighbors will find the Voronoi Diagram in the unit square in constant expected time, and the Outer Voronoi Diagram can be found in constant time as shown previously.

### 6.5 Delaunay Triangulation

The Delaunay Triangulation can be constructed in constant expected time given the Voronoi Diagram. Since the Voronoi Diagram can be constructed in constant expected time, the Delaunay Triangulation can be constructed in constant expected time.

### 6.6 Largest Empty Circle

The Largest Empty Circle must have its midpoint on a vertex of the Voronoi Diagram, or on the intersection of the Voronoi Diagram and the Convex Hull. With $n^{1+\epsilon}$ processors we can find the Voronoi Diagram and the extreme points in constant time. We see by Lemma 6.5 that we can sort these extreme points in constant time to form the ordered convex hull. Then we can easily find the vertices of the Voronoi Diagram which are inside the Convex Hull in constant expected time, and find the maximum empty circles these imply in constant expected time. Similarly we can find the intersections of the Voronoi Diagram with the Convex

Hull in constant expected time, and find the maximum empty circles these imply in constant expected time. Comparing these two will give us the Largest Empty Circle.

We sum up the results in this section in the following theorem.

**Theorem 6.1** *Given $n$ values taken from a uniform distribution over the unit interval, in constant expected time and using $n^{1+\epsilon}$ processors, these values can be arranged in sorted order in an array of size $n + o(n)$ with the value NULL in all unfilled locations. Also, given $n$ points taken from a uniform distribution over the unit square, in constant expected time and using $n^{1+\epsilon}$ processors, we can*

1. *solve the All Nearest Neighbors problem,*
2. *construct the Relative Neighborhood Graph,*
3. *construct the Voronoi Diagram,*
4. *construct the Delaunay Triangulation, and*
5. *find the Largest Empty Circle.*

# 7 Conclusion

We have defined an ultra-fast expected time parallel algorithm as one which uses a linear number of processors and runs in $O((\log\log n)^{O(1)})$ expected time. We have presented ultra-fast expected time parallel algorithms for Padded Sort, All Nearest Neighbors, Relative Neighborhood Graph, Voronoi Diagram, Delaunay Triangulation, Largest Empty Circle, and Closest Pair. All the algorithms run in $\Theta(\log\log n/\log\log\log n)$ expected time using $n\log\log\log n/\log\log n$ processors and assume that inputs are taken from a uniform distribution except Closest Pair, which runs in constant time with $n$ processors and makes no assumptions on the distribution of inputs.

We note that all of these algorithms are optimal in terms of linear speedup, but only Closest Pair is known to be optimal in terms of running time. It is known that the optimal running time of Padded Sort is $\Theta(\log^* n)$, but in regards to the other problems, an open question is whether any of them can be solved in $o(\log\log n/\log\log\log n)$ expected time with a linear number of processors. As for the case of having $n^{1+\epsilon}$ processors, we showed these problems could all be solved in constant expected time,

# A Probabilistic Tools

One technique we use for bounding the tail of a probability distribution is the Chebyshev Inequality. It states that given a random variable $X$ with mean $\mu$ and standard deviation $\sigma$,

$$P(|X-\mu| > r\sigma) \leq \frac{1}{r^2}.$$

Another technique we use is the Chernoff bound. This can be used when we wish to bound the distribution of a random variable $Z$ which is the sum of $n$ independent random variables. For a binomial random varible $Z \sim B(n,p)$, where $Z$ is the sum of $n$ independent Bernoulli trials with probability of success $p$, Angluin and Valiant [5] show that for $0 < \beta < 1$, one can obtain the bounds

$$P(Z \geq (1+\beta)np) \quad \leq \quad e^{-\beta^2 np/3},$$

and

$$P(Z \leq (1-\beta)np) \quad \leq \quad e^{-\beta^2 np/2}.$$

From this we obtain the bound

$$P(Z \geq 2np) \leq 2^{-4np/9}.$$

Also, for $k \geq 3$ we obtain the bound

$$P(Z \geq knp) \quad \leq \quad k^{-np},$$

and for $k \geq 6$ we obtain the bound

$$P(Z \geq knp) \quad \leq \quad 2^{-knp}.$$

For a random variable with a gamma distribution, $Z \sim \Gamma(n, \lambda)$, where $Z$ is the sum of $n$ independent random variables with exponential distributions with parameter $\lambda$, we obtain for $0 < \beta < 1$ the bound

$$P(Z \geq (1+\beta)n/\lambda) \leq e^{-\beta^2 n/6},$$

and for $k \geq 3$ the bound

$$P(Z \geq kn/\lambda) \leq 2^{-kn/2}.$$

# B Useful Equations and Inequalities

To simplify some of the expressions we derive, we use the following facts.

$$\begin{aligned} 1 + x &\leq e^x, \text{ for all } x. \\ \sqrt{1+x} &\geq 1 + x/3, \text{ for } 0 \leq x \leq 1. \end{aligned}$$

Also, Stirling's Approximation is often useful. Taken from Knuth [23],

$$\left(\frac{x}{e}\right)^x \sqrt{2\pi x} \leq x! \leq \left(\frac{x}{e}\right)^x \sqrt{2\pi x} e^{1/12x}.$$

One use of Stirling's Approximation is to obtain the bound

$$\binom{n}{k} \leq \frac{n^n}{(n-k)^{n-k} k^k} \leq \left(\frac{ne}{k}\right)^k.$$

# References


[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

[2] M. Ajtai and M. Ben-Or. A theorem on probabilistic constant depth computations. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 471–474, 1984.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. *Combinatorica*, 3:1–19, 1983.

[4] N. Alon and Y. Azar. The average complexity of deterministic and randomized parallel comparison sorting algorithms. *SIAM J. Comput.*, 17(6):1178–1192, December 1988.

[5] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *J. Comput. System Sci.*, 18:155–193, 1979.

[6] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. *J. Assoc. Comput. Mach.*, 36(3):643–670, July 1989.

[7] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Trans. Math. Software*, 6(4):563–580, December 1980.

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

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

[10] O. Berkman and U. Vishkin. Recursive *-tree parallel data-structure. In *Proc. 30th Symp. on Found. of Comp. Sci.*, pages 196–202, 1989.

[11] A. K. Chandra, L. J. Stockmeyer, and U. Vishkin. A complexity theory for unbounded fan-in parallelism. In *Proc. 23th Symp. on Found. of Comp. Sci.*, pages 1–13, 1982.

[12] B. S. Chlebus. Parallel iterated bucket sort. *Inform. Process. Lett.*, 31(4):181–183, May 1989.

[13] R. Cole. Parallel merge sort. In *Proc. 27th Symp. on Found. of Comp. Sci.*, pages 511–516, 1986.

[14] R. Cole and M. T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. In *Proc. 4th ACM Symp. on Comp. Geom.*, pages 205–214, 1988.

[15] R. Cole, M. T. Goodrich, and C. O'Dunlaing. Merging free trees in parallel for efficient voronoi diagram construction. In *Proc. 17th Intl. Coll. on Automata, Languages, and Programming*, pages 432–445, 1990.

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

[17] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd Symp. on Found. of Comp. Sci.*, pages 698–710, 1991.

[18] J. Gil and L. Rudolph. Counting and packing in parallel. In *Proc. 15th Intl. Conf. on Parallel Processing*, pages 1000–1002, 1986.

[19] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. Assoc. Comput. Mach.*, 28:289–304, 1981.

[20] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a Priority PRAM. In *Proc. 2nd ACM Symp. on Para. Alg. and Arch.*, pages 117–124, 1990.

[21] T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *Proc. 33rd IEEE Symp. on Found. of Comp. Sci.*, pages 628–637, 1992.

[22] J. Katajainen, O. Nevalainen, and J. Teuhola. A linear expected-time algorithm for computing planar relative neighbourhood graphs. *Inform. Process. Lett.*, 25(2):77–86, May 1987.

[23] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Massachusetts, 1973.

[24] C. P. Kruskal. Searching, merging, and sorting. *IEEE Trans. Comput.*, 32(10):942–947, October 1983.

[25] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput.*, 34(4):344–354, April 1985.

[26] C. Levcopoulos, J. Katajainen, and A. Lingas. An optimal expected-time parallel algorithm for Voronoi diagrams. In *Scandenavian Conf. on Theoretical Comp. Sci.*, 1988.

[27] P. D. MacKenzie. Load balancing requires $\Omega(\log^* n)$ expected time. In *3rd ACM-SIAM Symp. on Disc. Alg.*, pages 94–99, 1992.

[28] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[29] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–39. Academic Press, Inc., New York, New York, 1976.

[30] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, June 1989.

[31] S. Rajasekaran and S. Sen. Random sampling techniques and parallel algorithm design. In J. Reif, editor, *Synthesis of Parallel Algorithms*, pages 411–451. Morgan Kaufmann, San Mateo, CA, 1993.

[32] J. H. Reif and S. Sen. Polling: A new randomized sampling technique for computational geometry. In *Proc. 21st ACM Symp. on Theory of Computing*, 1989.

[33] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM J. Comput.*, 14(2):396–409, May 1985.

[34] B. Schieber. *Design and analysis of some parallel algorithms*. PhD thesis, Tel Aviv University, 1987.

[35] Q. F. Stout. Constant-time geometry on PRAMs. In *Proc. Intl. Conf. on Parallel Processing*, pages 104–107, 1988.

[36] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4:348–355, 1975.

[37] D. E. Willard and Y. C. Wee. Quasi-valid range querying and its implications for nearest neighbor problems. In *Proc. 4th ACM Symp. on Comp. Geom.*, pages 34–43, 1988.