On Optimal Parallel Algorithm for Building A Data Structure for Planar Point Location

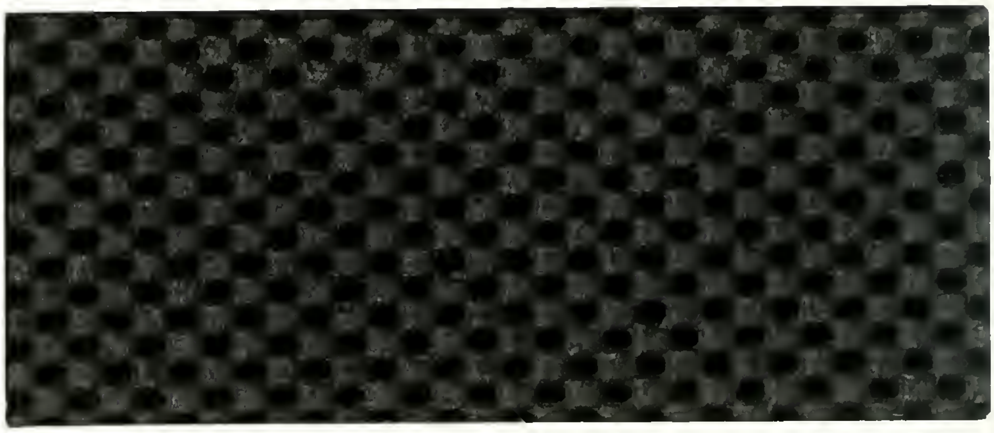
by

Richard Cole
Ofer Zajicek

Ultracomputer Note #128
Computer Science Technical Report #316
October, 1987

Ultracomputer Research Laboratory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012



On Optimal Parallel Algorithm for Building A Data Structure for Planar Point Location

by
Richard Cole
Ofer Zajicek

Ultracomputer Note #128
Computer Science Technical Report #316
October, 1987

ABSTRACT

Using a generalization of the Appoximate Parallel Scheduling of Cole and Vishkin [CoV87a], we give an optimal parallel algorithm for generating a linear size data structure for planar point location that supports an $O(\log n)$ query time. Our algorithm runs on a EREW PRAM in time $O(\log n)$ using a $n/\log n$ processors.

This work was supported in part by the National Science Foundation grants DCR-84-01633 and CCR-87-02271, the Office of Naval Research grant N00014-85-K-0046, and by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under contract DE-AC02-76ER03077.

1 Introduction

The model of parallel computation used in this paper in the exclusive-read exclusive-write (EREW) parallel random access machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. Simultaneous access by multiple processors to the same memory location is not allowed. Let $seq(n)$ be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem at hand. Obviously the best upper bound on the parallel time achievable with p processors, without improving the sequential result, is of the form $O(seq(n)/p)$. An algorithm achieving this running time is said to have *optimal-speedup* or more simply to be *optimal.*

In [CoV87a], Cole and Vishkin defined and solved the duration unknown task scheduling problem (DUTS). This led to the first optimal $O(logn)$ time algorithm for the list ranking problem as well as improved PRAM upper bounds for a variety of problems including connectivity, biconnectivity and st-numbering [CoV87b]. The DUTS problem is defined as follows. n tasks are given (a task can be thought of as a program); it is guaranteed that no task has length greater than $e \log n$, and that the total length of all the tasks is bounded by cn, where e and c are constants. However, the lengths of the individual tasks are not known in advance; in fact, they may vary depending on the order of execution of the tasks. The problem is to schedule the tasks among $n/\log n$ processors so that all tasks will complete in $O(\log n)$ time. Cole and Vishkin gave an optimal algorithm for the DUTS problem. The DUTS framework provides a powerful tool to the algorithm designer: an algorithm can be described as a set of tasks, without concern for the order of execution of the tasks. Of course, there is an associated difficulty: the algorithm must work correctly regardless of the order of execution of the tasks.

In the DUTS problem, all the tasks were defined in advance. The actual execution of a task, and its duration could vary depending on the order of execution, but no new tasks could be introduced. For some applications, it is hard to define all the tasks in advance while being able to limit the length of each task to $O(\log n)$. We would like to generalize the DUTS problem to allow the creation of new tasks. We explore one such generalization in providing an optimal $O(\log n)$ solution to the Planar Point Location problem (PPL).

2 Statement of the planar point location problem

Given a triangulated planar graph G, construct, in parallel, a data structure which, given a point in the plane, will enable a single processor to locate the triangle containing the given point in $O(\log n)$ time. Define a *Kirkpatrick-Decomposition* (the definition was introduced in [ACGOY85]) to be a sequence of planar triangulated graphs G_1, G_2, ..., G_m, such that:

1. $G_1 = G$

2. For $1 < i \leq m$, the area of each triangle in graph G_i is contained in a constant number of triangles of G_{i-1}.

   If in addition we have:

3. the final graph, G_m contains only three vertices (i.e. is a triangle)

   we call it a *complete Kirkpatrick Decomposition.*

Our solution is based on a sequential algorithm due to Kirkpatrick. Kirkpatrick's algorithm [Kir83] chooses a constant K, and then constructs a complete Kirkpatrick decomposition in $m - 1$ stages. In stage i, G_{i+1} is constructed, as follows. All vertices in G_i with degree lower than K are considered. A maximal independent set of these vertices is chosen and deleted from the graph. After the deletion of the independent set, the graph is retriangulated, yielding G_{i+1}. With an appropriate choice of K (say K=12), at each stage a constant fraction of the nodes is deleted, hence $m = O(\log n)$. The preprocessing takes linear time and requires a linear amount of space. A query is satisfied as follows: given a point, we determine whether it is in the triangle G_m in constant time. Now recursively, given we know the triangle that contains the point in G_i, by condition 2 above, we can find the triangle of G_{i-1} that contains the point in constant time. Hence we can answer the query in $O(m)(= O(\log n))$ time.

[DaK87] gave a parallel algorithm for constructing a complete Kirkpatrick Decomposition with $m = O(\log n)$; their algorithm used $O(\log n \log^* n)$ time and n processors. [HCD87] showed how to get the same running time with (optimal) $n/(\log n \log^* n)$ processors. Parallel algorithms to construct different data structures for the PPL problem were considered in [ACGOY85], [AtG86], [ACG87]. These data structures used $O(n \log n)$ space; they were constructed with n processors in time $O(\log^2 n)$ [ACGOY85], $O(\log n \log\log n)$ [AtG86], and $O(\log n)$ [ACG87], respectively.

We give an optimal $O(\log n)$ time algorithm (i.e. using $n/\log n$ processors) to generate a linear sized data structure for PPL. We use a generalization of the DUTS problem to generate in $O(\log n)$ time a Kirkpatrick Decomposition with G_m containing $n/\log n$ vertices, and $m = O(\log n)$. We then use the algorithm presented in [ACG87] to generate a PPL data structure for the reduced graph, G_m. To answer a query we use the Query algorithm of [ACG87] to locate the triangle in G_m containing the given point, and proceed from there as in Kirkpatrick's algorithm. In the following sections we will show how to obtain the reduced graph.

3 Sketch of the Algorithm

The algorithm proceeds in $m-1 = O(logn)$ stages. In stage i, graph G_{i+1} of a Kirkpatrick Decomposition is constructed, by deleting an independent set of vertices from G_i. The $m-1$ stages yield a graph of size $O(n/logn)$. For ease of presentation we break the process into two phases. In the first phase, the algorithm reduces the number of vertices to $O(f(n)n/\log n)$, where $f(n)$ is $O(\sqrt{\log n})$. In the second phase the same algorithm is used to reduce the graph to size $O(n/\log n)$.

Each vertex has an associated task, whose job it is to delete the vertex. Each processor has a queue of tasks. $n/\log n$ processors are used. Initially, there are n tasks, one per vertex; the tasks are distributed arbitrarily, $\log n$ to each queue. Each non-empty queue has a task at the top of the queue; this task is called the *active* task. Each stage is divided into two steps: task distribution (among the processor queues) and task execution. The precise work performed by the tasks will be discussed shortly; the collective goal of the tasks is to delete the vertices.

4 Assumptions and definitions

We choose a constant MAX_{DEG}. We say that a vertex, v, is *heavy* if its degree, $deg(v) > MAX_{DEG}$; it is *light* otherwise. Define a path in the graph to be *light* if it includes only light vertices. We say two vertices, u and v are *light-distance l apart*, if there is a light path from u to v of length l.

Since this paper generalizes the DUTS scheme of [CoV87a], we use routines and notations introduced in that paper. For the sake of completeness we state these results here.

p processors are used. With each processor we associate a collection of tasks. The *size* of collection i, $size_i$, is the number of tasks in the collection. The weight of collection i is defined to be $size_i^2$, and the total weight $W = \sum_{i=1}^{p} size_i^2$.

Define WMIN to be the minimum weight possible for the current number of tasks. There are two constants g_1 and g_2 such that if the total weight is bounded by either $g_1 n/\log n$ or g_2WMIN, the collections are said to be *balanced.* Cole and Vishkin gave a redistribution procedure (RP) that redistributes the tasks among the collections such that:

- The total weight is not increased.
- The maximum number of tasks in any collection does not increase.
- If the collections are not balanced, the total weight of the collections is reduced by a multiplicative constant factor.
- The redistribution takes constant time.

Remark: Due to the use of expander graphs, the constants in the running time for the above procedure are large.

5 Symmetry Breaking

To break symmetries in the graph, we use the following generalization of Cole and Vishkin's deterministic coin tossing, which was given in [GPS87]: each vertex has a unique number (in the range $[1, n]$) which is its vertex number. In addition, each vertex has a *potential*, which is used for tie-breaking.

The coin tossing is performed by executing the following steps synchronously for all vertices:

- **Step 1**: For each light vertex, set its potential to be its vertex number. For the other vertices set the potential to be equal to zero.

  Perform the following step l times (the constant l will be chosen later):

- **Step 2**: For each light vertex, v, of degree d, and each of v's neighbors, u_i, compute v_i = index of the least bit for which the potential of v and the potential of u_i differ, and let b_i be the v_i bit of the potential of v. We consider the bits to be numbered from 0. Define the new potential of v to be the concatenation: New Potential(v) = $b_1|v_1|b_2|v_2|\ldots|b_{DEG_{MAX}}|v_{DEG_{MAX}}$, where $|$ stands for the concatenation operator, and $b_i = v_i = 0$ for $d < i \leq MAX_{DEG}$. (We note that

v_i is considered to be a k-bit number, including leading zeros, where k is the bit length of the current potential. The length of all potentials is initially $\lceil \log(n+1) \rceil$.)

Property 1: No two adjacent light vertices have the same potential.

Property 2: If the potentials have K bits before an iteration of Step 2, then after a single iteration of Step 2 the length of the potentials will be $MAX_{DEG}(1 + \lceil \log K \rceil)$.

Proof of properties: (similar to the proof of [GPS87].) Property 1 is true following Step 1, since each light vertex has a unique number associated with it. We have to show that the invariant holds after each execution of Step 2.

Assume that before the execution of Step 2 no two adjacent light vertices have the same potential. Consider two adjacent light vertices v and u. Assume that the least bit in which they differ is the j^{th} bit. We can further assume that the j^{th} bit of v is a 1, and that of u is a 0. Hence one of the components of the new potential of v is $1|j$. For the two new potentials to be the same after the execution of the step, the new potential of u must have a component $1|j$, but this cannot be, since the j^{th} bit of u is a zero.

Property 2 is true by definition. ¶

If we have n vertices initially, each vertex number has $\lceil \log(n+1) \rceil$ bits. When we iterate the second step l times, the final potential will have length $O(\log^{(l+1)}(n+1))$. (Recall MAX_{DEG} is constant.)

Note: When we perform the coin tossing procedure, we execute step 1 once, and step 2 l times. Thus the potential of a vertex v is a function of all its neighbors light-distance $\leq l$ away, and only these neighbors.

Remark: It is convenient to store, with each vertex, the l intermediate values computed when determining the potential.

6 Task Definition

In each step, each processor tries to delete a vertex and to retriangulate the local graph around the deleted vertex. When two processors try to delete adjacent vertices, only the one with larger potential proceeds. More formally, for each vertex v of G we generate a task. When the task of vertex v becomes active we say v is active. The task of vertex v comprises the following substeps:

Substep 1: If v is light and has no active neighbors with larger potential, determine that v is to be deleted.

Otherwise, the task halts (and does not proceed to Substep 2).

Substep 2: For each light vertex, u, within light-distance r of v, such that u is not on any queue, create a new task and add the task to v's queue. The constant r will be chosen later. If more than one processor tries to add u to its queue, only one succeeds.

Substep 3: Delete v and retriangulate the local graph.

Substep 4: For each light vertex, u, within light-distance $r - 1$ of a vertex that was a neighbor of v, such that u is not on any queue, create a new task and add the task to v's queue. If more than one processor tries to add u to its queue, only one succeeds.

Lemma 1 *The work performed by all the tasks combined is bounded by $c'n$ for some constant c'.*

Proof: We will show that over the entire execution of the algorithm, there are only a linear number of tasks, and each task runs in constant time.

Initially there are a linear number of tasks. Each light vertex has a constant number of neighbors within light-distance r. Hence each task which deletes a vertex will create at most a constant number of new tasks. Since new tasks are created only when a vertex is deleted, the total number of tasks created during the running of the algorithm is linear.

It remains to show that each task can be executed in constant time. The task of vertex v proceeds as follows:

- If v is deleted, as v has degree bounded by MAX_{DEG}, the deletion and retriangulation take constant time.

- If v is not deleted, it is marked "free" (i.e. that it does not belong to any queue). This can be done in constant time.

- For each vertex, v, that is to be deleted, the associated processor, p, performs Substep 2 as follows. The processor carries out a sequential Breadth First Search (BFS) on v's neighborhood; the BFS stops whenever it reaches a heavy vertex or a node at depth r, where v is defined to be at depth 0. (A processor, searching from a vertex w, can determine if a neighboring vertex, u, is heavy in constant time. It simply counts the number of edges on the adjacency list of u starting from the edge connecting w to u, until it either reaches the starting edge or counts MAX_{DEG} edges.) The processors all search the vertices at depth i simultaneously; this can be done, for, in each BFS tree, there are at most $(MAX_{DEG})^i$ vertices at level i. Each free light vertex, u, that p encounters in its BFS, is marked as "taken"; for each such vertex u, p creates a task and adds the task to its queue. Since there are only a constant number of vertices within light-distance r of v, it takes p a constant amount of time to create the tasks induced by v's deletion. Note that there are at most a constant number of processors that will try to mark u as "taken"; we show below how to resolve any conflicts in constant time.

The technical details to ensure Substep 2 is EREW follow.

If two BFSs (from v_1 and v_2) reach vertex u simultaneously only one is allowed to proceed; the other is obliged to stop. The BFS of the processor with larger processor i.d. proceeds. This can be determined when testing whether u is heavy: each processor labels the edge from which it starts with its processor i.d. . As the traversal of the adjacency list is performed a processor can determine whether its BFS should proceed. Note that there are no read conflicts here.

A free light vertex is marked "taken" by the first processor to reach it; if several reach it simultaneously, the processor with the largest processor i.d. marks it (this can be determined as in the previous paragraph). Again, there are no read conflicts.

Substep 4 is performed in essentially the same way as Substep 2.

This completes the proof. ¶

6.1 Computing the potentials

In the above task definition we assumed the potentials of the vertices were given. In Section 7 we will show how to compute the initial potentials. To complete the description of the tasks we will now show how to update the potentials in constant time, following execution of Substeps 1-4.

Since no new vertices are added to the graph, the potential of a vertex v can change only if within light-distance l of v a vertex is deleted, a heavy vertex becomes light, or a light vertex becomes heavy. A heavy vertex (resp. light vertex) can become light (resp. heavy) only if a vertex that was adjacent to it was deleted. Hence

Observation: The potential of a light vertex v can change only if either there is a vertex u, within light-distance $l+1$ of v, such that u is to be deleted, or there is a vertex u such that after it is deleted one of the vertices that was adjacent to u, before u was deleted, is now within light-distance l of v.

In order to maintain the potentials it is sufficient to add the following substep to the definition of the task for vertex v:

Substep 5: Recompute the potential of all the vertices that are (resp. were) within light-distance l of a currently (resp. previously) light vertex that was adjacent to v.

For each deleted vertex there are only a constant number of such vertices. By choosing $r \geq l + 1$, we ensure that the light vertices for which a new potential is to be computed are all enqueued; the new potentials can then be obtained using the method of Section 5 (note in particular the remark at the end of Section 5. Computing a new potential takes constant time, thus Substep 5 can be executed in constant time. It is straightforward to ensure that Substep 5 is EREW.

7 The Algorithm for Stage I

Initially there are n tasks, one for each vertex of the graph, and $n/\log n$ processors. Each processor has a queue of tasks. Initially each queue contains $\log n$ tasks. The task on top of the queue for each processor is the active task.

The algorithm to reduce the input graph to size $O(n/\sqrt{\log n})$ (Phase I) proceeds as follows:

Init: Compute the potential of all the vertices. All heavy vertices get potential 0; all light vertices obtain their potential as described in Section 5 on symmetry breaking.

Repeat the following two step cycle:

Step 1: Run the Redistribution Procedure C_{RP} times to redistribute the tasks. The constant C_{RP} will be chosen later.

Step 2: Each processor executes its active task (if it has one).

Lemma 2 *During the execution of every cycle one of the following three conditions must hold:*

A: *The weight is reduced by a constant factor over the course of the cycle.*

B: *The number of tasks is bounded by $g_1 n/\log n$ at the start of the cycle (and hence following Step 1).*

C: *After Step 1, a constant fraction of the processors have an active task. In this case the weight will grow by no more than a constant multiplicative factor as a result of Step 2.*

Proof: Consider the collections after performing Step 1 of the cycle.

Case 1: the collections are not balanced after Step 1. As in [CoV87a], the total weight of the collections was reduced by a constant fraction at each execution of the redistribution procedure. In Step 2, each queue can grow by at most a constant number of tasks, say g'. Hence the total weight can grow by at most a multiplicative factor of $(g')^2$. By adjusting the number of redistribution calls that are made in Step 1 of the cycle, we can ensure that the total weight will be decreased by a constant fraction in this case, satisfying condition A.

Case 2: the collections are balanced after performing Step 1. One of the following two subcases must hold.

Case 2.1: the total weight is bounded by g_2WMIN but not by $g_1 n/\log n$, and so a constant fraction of the processors have an active task as required by condition C. Since each processor can spawn at most a constant number of tasks per cycle, the weight cannot be increased in this case by more than a constant multiplicative factor.

Case 2.2: the weight is bounded by $g_1 n/\log n$. Then the total number of tasks is bounded by $g_1 n/\log n$, as required by condition B.

This completes the proof. ¶

Corollary 1: There is a constant h such that after $h \log n$ cycles, the number of tasks has been reduced to $c'n/\log n$, for some constant $c' \geq g_1$.

Note: It might be that the number of tasks is reduced to $\leq c'n/\log n$, and subsequently increases to $> c'n/\log n$.

Proof: By Lemma 2, at each cycle one of the three conditions must hold.

- Once condition B of Lemma 2 holds we are done; also, if the weight is bounded by $g_1 n/\log n$ then condition B holds. (Note that condition B need not remain true subsequently.)
- Let N(c) be the number of cycles for which condition C of Lemma 2 holds. By Lemma 1, the total work performed on the tasks is linear. For each cycle for which condition C holds we perform $O(n/\log n)$ work on the tasks, hence $N(c) = O(\log n)$.
- Consider the cycles preceding the first cycle in which condition B holds. There are at most $N(c)$ cycles in which condition C holds. It takes

O(N(c)) cycles in which condition A holds to undo the weight increase due to all the cycles in which condition C holds; it takes a further O($\log\log n$) cycles in which condition A holds to reduce the weight to at most $g_1 n/\log n$, at which point condition B must hold. Thus Condition B must have been true during some cycle, among the first O(N(c)) + O($\log\log n$) cycles.

This concludes the proof of the corollary.

To complete the description of the algorithm we need to show we can limit the number of free vertices per queued vertex. For this purpose we prove the following lemma. Recall l, the parameter used in Step 2 of the coin tossing algorithm, and r, the parameter used in the task definition.

Lemma 3 *If $r > l + 1$, then, from every free light vertex there is a light path to a queued vertex such that the values of the potentials of the vertices along this path are strictly increasing.*

Proof: The proof will be by induction. Initially all the vertices are queued, hence the lemma holds. Consider one iteration of Step 2 of the algorithm. We will show that if the lemma holds before executing Step 2, it will hold after executing each of Substeps 1, 4, and 5. This will conclude the proof.

Substep 1. Assume the lemma holds before executing the current Step 2. In Substep 1, the only change that occurs is that some active vertices become free. So we have to show that the lemma holds for all light vertices that are freed, and further that it remains true for all light vertices that were free.

Consider a light vertex v that is freed in Substep 1. Since no two adjacent light vertices have the same potential, and since exactly the active vertices with locally maximum potentials will be deleted, there is a strictly increasing light path p from v to a vertex u that is to be deleted. Hence the lemma holds for vertex v following Substep 1.

Consider a vertex v that is free at the start of Substep 1. Consider a shortest light path p of strictly increasing potential from v to a queued vertex u. In Substep 1, the only change that might occur to p is that u might be freed. But then, by the argument of the previous paragraph, there is a strictly increasing path from u to some queued vertex w that is to be deleted. Since there is still a strictly increasing light path from v to u, there is a strictly increasing light path from v to a queued vertex.

Substeps 2-4. In these Substeps there are three changes that might occur to vertices. A light queued vertex might be deleted; a light vertex might become heavy, and a heavy vertex might become light. However, the free vertices are unchanged. It suffices to prove the lemma for two types of vertices: free light vertices that remain light, and free heavy vertices that become light.

Consider a vertex v that is free and light at the start of Substep 2. Let p be a shortest light path from v to a queued vertex u at the start of Substep 2. Over the course of Substeps 2-4, p might change in one of two ways. A vertex w on p might become heavy, or vertex u might be deleted. We consider each in turn.

Let w be the first vertex on the path p that becomes heavy, if any. The vertex w becomes heavy only if a neighbor of w was deleted. But then, if $v \neq w$, vertex w', that precedes w on the path, is queued (since $r \geq 2$). Hence either $v = w$, or $v = w'$ and will be queued, or there is a light path of strictly increasing potential from v to queued w'.

Assume no vertex on p becomes heavy and let u' precede u on path p. If u is deleted, then u' is queued following Substep 2. Since u' remains light by assumption, there is a strictly increasing light path from v to queued u'.

Thus the lemma still applies to free light vertices that remain light. Now consider a free heavy vertex v that becomes light. v must be a neighbor of a vertex deleted in Substep 3; thus v is queued following Substep 4, and the lemma then applies to v.

We conclude the lemma is true following Substep 4.

Substep 5. Consider a vertex v that is free following Substep 4, and let p be a shortest strictly increasing light path from v to a queued vertex. By the Observation at the end of Section 6, if the potential of a vertex w on p is changed, either it was or it is now within light-distance l of the neighbor of a vertex that was deleted. As $r > l + 1$, w is queued presently (so $v \neq w$). Let u be the predecessor of w on p. u is also queued presently. For if u became light in this step, then u was adjacent to a vertex that was deleted, and thus u was queued. While if u was already light, either it was already enqueued, or it became queued in one of Substeps 2 and 4 (since $r > l + 1$). But then the vertex preceding w on p is queued, contradicting the definition of p. Thus the potential of every vertex on p is unchanged and there is still a light path with strictly increasing potentials from v to a queued vertex. Since v was an arbitrary free vertex, the lemma will hold after Substep 5.

This concludes the proof.

Note: Lemma 3 shows that setting the parameter r to be $r = l + 2$ is sufficient. Hence from here on, we assume that $r = l + 2$.

Section 5 (on symmetry breaking) showed that there is a constant k, such that for large enough n, the number of bits in the binary representation of the vertex potentials is $\leq P_{len} = k\log^{(l+1)}(n+1)$. Define P_{max} to be the largest value the potentials can take; clearly, $0 \leq P_{max} < 2^{P_{len}}$.

By choosing an appropriate value for l, we can limit the number of free light vertices to be $O(f(n))$ per queued vertex, i.e. per task (recall $f(n) = \sqrt{\log n}$). For example, if we choose $l = 3$, by Lemma 3, there is a light path of length $O(\log^{(3)} n)$ from every light vertex to a queued vertex. Hence for some constant h_1, there are $O((\log^{(2)} n)^{h_1})$ free light vertices for every task.

If we choose $MAX_{DEG} = 6$, at least a constant fraction of the vertices are light. We conclude that the number of free vertices (not necessarily light) per task is $O((\log^{(2)} n)^{h_1}) = O(f(n))$. By Corollary 1, in time $O(\log n)$, the number of tasks was reduced, at some point, to $c'n/\log n$. Thus the number of vertices was reduced to $O(f(n)n/\log n) = O(n/\sqrt{\log n})$. Since the number of vertices can only decrease, we conclude that in time $O(\log n)$, using $n/\log n$ processors, the number of vertices is reduced to $O(n/\sqrt{\log n})$.

We have shown how to reduce the problem from a problem of size n to a problem of size $O(n/\sqrt{\log n})$. We can show that if we run the above algorithm for an additional $O(\log n)$ cycles, the problem is reduced to size $O(n/\log n)$. As the argument is somewhat complicated, we use the following alternative approach. At present, we have a problem of size $s = O(n/\sqrt{\log n})$. Repeat the above reduction algorithm, replacing n with s, i.e. with the current problem size and with $p = s/\log s = O(n/(\log n)^{3/2})$. In time $O(\log n)$ the problem will be reduced to size $O(n/\log n)$.

8 Conclusions and Open Problems

In this paper we gave an algorithm to generate a linear size data structure for the Point Location Problem. The scheme presented here is a generalization of Cole and Vishkin's Approximate scheduling algorithm, allowing the dynamic generation of tasks during the execution of the algorithm.

In general, our scheme shows how to combine an efficient algorithm with a fast algorithm so as to obtain an algorithm that is both efficient and fast. In our case we combined an $O(\log n)$ time linear processor algorithm, with an algorithm that performs linear work (i.e. is optimal), to obtain an optimal algorithm that runs in time $O(\log n)$.

An interesting open problem is whether there is a direct approach that will generate a linear sized PPL data structure without using an approximate task scheduling scheme. We expect a direct approach to yield an algorithm with smaller constants.

A related problem is the 5-coloring of planar graphs. [HCD87] gave an optimal parallel algorithm for that problem with running time $O(\log n \log^* n)$. No fast ($O(\log n)$ time) algorithm to solve this problem is known. If there is a fast algorithm for this problem, which is also efficient (i.e. uses $O(n \text{polylog}(n))$ processors), our technique can be used to arrive at a fast optimal algorithm.

References

[ACGOY85] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing and C. Yap, "Parallel Computational Geometry", *Proceedings of the 25th IEEE Annual Symp. on Foundations of Computer Science*, 1985, pp. 468–477.

[AtG86] M. J. Atallah and M. T. Goodrich, "Efficient Plane Sweeping in Parallel", *Proceeding of the 2nd ACM Symposium on Computational Geometry*, 1986, pp. 216–225.

[ACG87] M. J. Atallah. R. Cole and M. T. Goodrich. "Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms", *Proceedings of the 27th IEEE Annual Symp. on Foundations of Computer Science*, 1987.

[CoV87a] R. Cole and U. Vishkin, "Approximate Parallel Scheduling. Part I: The Basic Technique With Applications to Optimal Parallel List Ranking in Logarithmic Time", to appear, *SIAM Journal on Computing.*

[CoV87b] R. Cole and U. Vishkin, "Approximate Parallel Scheduling. Part II: Application to Optimal Parallel Graph Algorithms in Logarithmic Time", Courant Institute Technical Report #291.

[DaK87] N. Dadoun and D. Kirkpatrick, "Parallel Processing for Efficient Subdivision Search", *Proceedings of the 3rd ACM Symposium on Computational Geometry*, 1987.

[GPS87] A. Goldberg, S. Plotkin and G. Shannon, "Parallel Symmetry-Breaking in Sparse Graphs", *Proceedings of the 19th Annual Symp. on Theory of Computing*, New York, May 1987, pp. 315–324.

[HCD87] T. Hagerup, M. Chrobak and K. Diks, "Parallel 5-Colouring of Planar Graphs", *ICALP*, 1987.

[Kir83] D. Kirkpatrick, "Optimal Search in Planar Subdivision", *SIAM Journal on Computing*, Vol. 12, No. 4, February 1983, pp. 28–35.

NYU COMPSCI TR-316 c.2
Cole, Richard
On optimal parallel
algorithm for building a
data structure for planar