1996

An Improved Hypercube Bound for Multisearching and its Applications

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Report Number:
96-021

AN IMPROVED HYPERCUBE BOUND FOR MULTISEARCHING AND ITS APPLICATIONS

Mikhail J. Atallah

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD-TR 96-021
April 1996
(Revised May 1997)

An Improved Hypercube Bound for Multisearching and its Applications*

Mikhail J. Atallah†

Abstract

We give a result that implies an improvement by a factor of $\log\log n$ in the hypercube bounds for the geometric problems of batched planar point location, trapezoidal decomposition, and polygon triangulation. The improvements are achieved through a better solution to the multisearch problem on a hypercube, a parallel search problem where the elements in the data structure S to be searched are totally ordered, but where it is not possible to compare in constant time any two given queries q and q'. Whereas the previous best solution to this problem took $O(\log n(\log\log n)^3)$ time on an n-processor hypercube, the solution given here takes $O(\log n(\log\log n)^2)$ time on an n-processor hypercube. The hypercube model for which we claim our bounds is the standard one, SIMD, with $O(1)$ memory registers per processor, and with one-port communication. Each register can store $O(\log n)$ bits, so that a processor knows its ID.

Keywords: parallel algorithms; hypercube; multisearching; trapezoidal decomposition; point location; polygon triangulation.

1 Introduction

Consider the situation depicted in Figure 1: We have a horizontal slab partitioned by a set S of n nonintersecting segments. For a set Q of n

*This work was supported in part by the National Science Foundation under Grant CCR-9202807. The author also author gratefully acknowledges the support of the COAST Project at Purdue and its sponsors, in particular Hewlett Packard, DARPA, and the National Security Agency.

†Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA.
Email: mja@cs.purdue.edu
Fax: 317-494-0739

points, we need to determine for each point which region of the slab it belongs to. Both the segments and the points are initially stored in an n-processor hypercube; the segments are given in left to right sorted order, but the points are not given in any particular order.

This problem would be trivial, if the partitioning segments were vertical, but the fact that they are slanted makes it impossible to solve the problem by (e.g.) simply mergesorting $S \cup Q$ according to x-coordinates. The method we give for solving this multisearch problem works for more general versions of this problem: The basic assumption is that any pair x, y in a processor can be compared in constant time if $x \in S \cup Q$ and $y \in S$, but not so if both x and y are in Q (hence the method works if we have algebraic curves instead of segments). Dehne and Rau-Chaplin [6] have given an $O(\log^2 n)$ time algorithm for this problem. The algorithm is more general in the sense that it allows multiple queries in parallel to traverse a data structure and to create and delete queries on the fly. The algorithm is easy to implement and thus of practical interest, and it was later generalized for doing fractional cascading on a hypercube [7]. A further $O(\log^2 n)$ time algorithm was given by Lee and Preparata [8] as a subroutine of a batched planar point location algorithm. Furthermore, a randomized $O(\log n)$ expected time scheme for multisearching was given by Reif and Sen [11]. Since searching is related to sorting and there is a deterministic $O(\log n \log\log n)$ time sorting algorithm for the hypercube [5], the question is how far from that sorting bound the deterministic complexity of multisearching is (one expects it to be somewhat higher because of the difficulty introduced by being unable to compare two query elements in constant time; in the special case of vertical segments, one can of course solve it within the same complexity as sorting, by sorting the query points). A solution that is within a factor of $(\log\log n)^2$ of the sorting bound was given by Atallah and Fabri [3], where an $O(\log n(\log\log n)^3)$ time algorithm on an n-processor hypercube was given. Here we present a solution that is only a factor of $\log\log n$ worse than the best known sorting bound, i.e., an algorithm with time complexity $O(\log n(\log\log n)^2)$ on an n-processor hypercube. The consequences of this are corresponding improvements (i.e., also by a factor of $\log\log n$) to the time complexities of the related problems of batched planar point location, trapezoidal decomposition, and polygon triangulation. The new bounds for these problems are $O(\log n(\log\log n)^2)$ time on an $n\log n$-processor hypercube. As was the case for the previous best bounds for these problems, the new bounds too are more of theoretical than of practical interest, because they too rely on the sorting algorithm of Cypher and Plaxton [5] as a subroutine. However,

any practical improvement to hypercube sorting would immediately make our bounds more practical. Any future theoretical improvement to hypercube sorting also translates into a similar improvement to our multisearching bounds, and hence to the complexity bounds of the other related problems mentioned.

The paper is organized as follows. In Section 2 we review the definition of a hypercube interconnection network and some basic algorithms for this parallel machine. In Section 3 we give a preliminary solution to the multisearch problem for an asymmetric version of the problem, one where there are more query points than slab segments, i.e., m query points and n slab segments where $m \geq n$. The time complexity for the preliminary solution to the asymmetric problem is $O(\log m(\log\log m)^2)$ time on an m-processor hypercube, but it requires that each hypercube processor have $\Theta((n/m)\log\log m)$ memory registers (rather than $O(1)$ registers). Section 4 uses the preliminary asymmetric algorithm as a subroutine to give the algorithm that achieves the improved bounds we claim for the symmetric case ($m = n$) without requiring more than $O(1)$ memory registers per processor. Section 5 revisits the asymmetric case ($m \geq n$) and gives a better solution than the preliminary one, in that it achieves an $O(\log m + \log n(\log\log n)^2)$ time with m processors, and only $O(1)$ memory registers per processor (so not only is it better than the preliminary one in the sense of requiring less space per processor, but it is also faster if n is substantially smaller than m). That section also considers the other kind of asymmetry, when $m < n$. In that case the bounds are similar to those for the $m \geq n$ case except that the roles of m and n are interchanged in the complexity bounds: $O(\log n + \log m(\log\log m)^2)$ time with n processors (and still $O(1)$ space per processor).

Section 6 points out consequences (all of which are improvements by a time factor of $\log\log n$) to related geometric problems; the link between these problems and the multisearch problem is well known [3, 6, 7, 8].

Before going into the details, we briefly discuss, in general terms, how the algorithms given below differ from (and improve over) the previous ones. The main new ingredients in the improved recipe are:

- When given a symmetric problem ($m = n$) the previous algorithms made recursive calls to problems of the same nature, i.e., also symmetric. Here we gain something by first giving a preliminary (and apparently bad) solution to an asymmetric problem ($m \geq n$) - the ability to recurse on asymmetric subproblems buys us a solution to

the asymmetric case that looks "bad" but whose real purpose is to serve as a subroutine for a better solution to the symmetric case (of course this is not achieved by simply calling the "bad" solution for the $m \geq n$ case on an instance that happens to have $m = n$).

- Even when we are ultimately interested in solving a subproblem for queries Q' and slab segments S', we refrain from recursing on Q' and S'. Instead, we recurse on Q' and *other* slab segments S'' that have the property that, once we know the solution for Q' with respect to S'' we can obtain it with respect to S' without having to use recursion again. We do this in situations when many parallel subproblems involve S' but none involves S''. That is, instead of having to store S'' (for later usage) and wastefully make many copies of S', this substitution trick (of artificially involving S'' in a recursive call that normally would involve S') achieves the following: (i) It helps avoid having to store S'' separately (since the recursive call brings S'' back anyway when it returns), and (ii) it makes it possible to make fewer duplicate copies of S'.

- We use different partitioning schemes of the subproblems (in order to exploit the above two ideas) and postprocessing of the solutions returned by recursion (occasionally even using a brute force, quadratic-processor method on judicious subproblems - not ones of constant size, but small enough and few enough that no damage is done to the processor complexity).

The above description is necessarily an over simplification, and only a careful look at the details can reveal the exact interplay between the above ideas, as well as the exact nature of each.

2 The Model of Computation

This section is a brief review of the model, and in particular of some operations on that model that we will make use of.

Unless otherwise specified, the hypercube model we use is the standard one, with $O(1)$ memory registers per processor, and with one-port communication. Each register can store $O(\log n)$ bits, so that a processor knows its ID. Recall that a hypercube of dimension d consists of $n = 2^d$ processors which are uniquely labeled with bitstrings of length d. Two processors are

connected *along dimension* i, iff their labels differ in exactly the i^{th} bit. In this paper we are interested in SIMD (Single Instruction Multiple Data) machines, that is, all processors execute the same instruction simultaneously. An instruction is either an operation on data in the local memory, or a communication step with a processor adjacent along a particular dimension. An instruction takes time $O(1)$.

We shall use as subroutines certain operations on sequences of size n, with time complexity $O(\log n)$. These operations include *segmented parallel prefix* and *monotonic routing* which together allow a *monotonic read.* Recall that a segmented prefix computation consists of a sequence of prefix operations that are individually applied to the various pieces of a given partition of the input string. A routing is monotonic iff the relative order of the packets is preserved, i.e., iff for any pair of processors p_i and p_j such that the packet at p_i has destination p_h and the packet at p_j has destination p_k, $i < j$ implies $h < k$. A read operation is monotonic, iff for any pair of processors p_i and p_j, with $i < j$, which want to read data on processors p_h and p_k, we have $h \leq k$. We refer the reader to the work of Nassimi and Sahni [10] and to Leighton's book [9] for detailled discussions of these operations. Another operation we use is sorting n numbers, which can be done in time $O(\log n \log\log n)$ [5].

We shall occasionally need to solve problems on *subcubes* of a hypercube. We can obtain subcubes of dimension $\hat{d} \leq d$ by selecting all $2^{\hat{d}}$ nodes matching a constant bitpattern on $d-\hat{d}$ bits. Two patterns which occur frequently are the following. Fixing the first $d/2$ bits yields $\sqrt{n}$ *consecutive* subcubes, fixing the last $d/2$ bits yields $\sqrt{n}$ *interlaced* subcubes. We can easily copy the contents of one of the consecutive subcubes to the other consecutive subcubes in $O(\log n)$ time, by broadcasting in parallel the contents of each of its nodes v to the $\sqrt{n}$ nodes that have the same last $d/2$ bits as v but whose first $d/2$ bits differ from v's (each such broadcast takes place in the interlaced subcube defined by fixing the last $d/2$ bits to be the same as v's last $d/2$ bits).

3 Preliminary Algorithm for $m \geq n$

Recall that m denotes $|Q|$ (the number of query points) and n denotes $|S|$ (the number of segments in the slab). This section deals with the asymmetric case of $m \geq n$, and gives for it a preliminary solution of time complexity $O(\log m(\log\log m)^2)$ on an m-processor hypercube, but where each hyper-

cube processor has $O((n/m)\log\log m)$ memory registers (rather than $O(1)$ registers). (In a later section we revisit the asymmetric problem and bring the space needed per processor down to $O(1)$, without any deterioration in any of the other bounds.)

In the algorithm that follows, the invariant that $m \geq n$ is maintained through any recursive calls that are made.

1. Partition Q (arbitrarily) into $\sqrt{m}$ chunks of size $\sqrt{m}$ each. If $\sqrt{m} < n$ then we use $S^{(i)}$, $0 \leq i < n/\sqrt{m}$, to denote the subsequence of $\sqrt{m}$ segments in S that are at positions of the form $i + k(n/\sqrt{m})$ in S, $0 \leq k < \sqrt{m}$. That is, if $S = s_0, \cdots, s_{n-1}$ then $S^{(i)} = s_i, s_{i+(n/\sqrt{m})}, s_{i+2(n/\sqrt{m})} \cdots$.

2. Recurse in parallel on all of the $\sqrt{m}$ chunks of Q, as follows.

   (a) If $\sqrt{m} \geq n$ then solve each chunk of Q recursively with respect to that chunk's own private copy of S (that is, we first make $\sqrt{m}$ copies of S, one for each chunk, before recursing). This is done in parallel for all chunks and, when the parallel recursive calls return, the algorithm terminates.

   (b) If $\sqrt{m} < n$ then we solve the ith chunk of Q recursively with respect to $S^{(j)}$ where $j = i \bmod \lceil n/\sqrt{m} \rceil$. This is done in parallel for all chunks, and when the recursive calls return the algorithm proceeds to the next substep (c), which determines (without using recursion) the solution of each chunk of Q with respect to $S^{(0)}$.

   (c) Use the outcome of the previous step to determine, for each chunk of Q, its solution with respect to $S^{(0)}$. This is easy to do, since we already know the solution of the ith chunk with respect to $S^{(j)}$ where $j = i \bmod \lceil n/\sqrt{m} \rceil$: For every segment s in S, let the *leader of* s be the nearest segment of $S^{(0)}$ that is to the right of s. Letting each s know its leader is easily done in logarithmic time by a segmented parallel copy. Assume this has already been done. Now, suppose that a query point p was determined, in the previous substep (b), to belong between segments s' and s'' of $S^{(j)}$. Then a comparison of p to the leaders of s' and s'' determines the position of p with respect to $S^{(0)}$.
   Note: The reader may be wondering why we do not just make $\sqrt{m}$ copies of $S^{(0)}$ and solve all chunks recursively with respect to $S^{(0)}$, instead of solving (in substep (b)) the ith chunk with respect to $S^{(i \bmod \lceil n/\sqrt{m} \rceil)}$ and then later (in substep (c)) using

the answer to obtain the solution with respect to $S^{(0)}$. If we did that, however, the space complexity would increase beyond repair: The segments not participating in the recursion would have to be stored somewhere, in addition to the $\sqrt{m}$ copies of $S^{(0)}$. How damaging this would be can be revealed by an analysis of the total storage space needed by all the processors (it would turn out to be $m \log\log m$ rather than the desired $m+n\log\log m$).

3. Let $S_1, S_2, \ldots, S_{\sqrt{m}}$ be the partition of S induced by the elements of $S^{(0)}$, i.e., $|S_i| = n/\sqrt{m}$ and S_i is to the left of S_{i+1}. Let Q_i denote the subset of Q which belongs in S_i. Partitioning Q into $Q_1, \ldots, Q_{\sqrt{m}}$ is easily done by sorting the queries of Q based on which S_i they belong to.

4. Partition every Q_i (arbitrarily) into $l_i = \lceil |Q_i|/\sqrt{m} \rceil$ pieces of size $\sqrt{m}$ each (except for the last piece which may be smaller), call them $Q_{i,j}$, $1 \le j \le l_i$. That is, each $Q_{i,j}$ is of size $\sqrt{m}$ except that Q_{i,l_i} might be smaller. We next recurse in parallel on *all* of the $Q_{i,j}$'s, but in a manner that depends on the size of each $Q_{i,j}$:

   (a) For each $Q_{i,j}$ such that $|Q_{i,j}| = \sqrt{m}$, we recurse on $Q_{i,j}$ with respect to that $Q_{i,j}$'s own private copy of S_i (we therefore use up to l_i copies of S_i). We say that such a subproblem defined by $Q_{i,j}$ and S_i is *full*. Note that a full subproblem satisfies the asymmetry invariant, because

   $$|Q_{i,j}| = \sqrt{m} \;\ge\; n/\sqrt{m} = |S_i|$$

   where the fact that $m \ge n$ was used.

   (b) For each Q_{i,l_i} such that $|Q_{i,l_i}| < \sqrt{m}$, we recurse on $Q_{i,j}$ with respect to $|Q_{i,l_i}|n/m$ evenly spaced elements of S_i. Hence the spacing between any two consecutive chosen elements of S_i is of size $|S_i|(|Q_{i,l_i}|n/m)^{-1} = \sqrt{m}/|Q_{i,l_i}|$; we use $S_{i,j}$ to denote the jth such spacing. We say that such a subproblem (i.e., one with $|Q_{i,l_i}| < \sqrt{m}$) is *non-full*.

5. The full subproblems need no further processing, but a non-full one (say, Q_{i,l_i}) needs further processing because the output of its recursive call gives which $S_{i,j}$ a query point belongs to, but not where among the $|S_{i,j}| = \sqrt{m}/|Q_{i,l_i}|$ possible positions in $S_{i,j}$ it belongs. Let $Q'_{i,j}$

denote the subset of Q_{i,l_i} that was determined (by the recursive call of Step 4(b)) to belong in $S_{i,j}$. We solve each such pair $Q'_{i,j}$, $S_{i,j}$ by brute force: We use $|Q'_{i,j}||S_{i,j}|$ processors to examine every pair of elements in $Q'_{i,j} \times S_{i,j}$.

The data movements required by the above algorithm (not counting the recursive calls) are dominated by the time complexity of hypercube sorting, which is $O(\log m \log\log m)$ on an m-processor hypercube [5]. If we let $T(m)$ denote the time complexity, then the parallel recursive calls of Step 2 take time $T(\sqrt{m})$, and so do those of Step 4. Therefore the overall time complexity obeys a recurrence of the form $T(m) = 2T(\sqrt{m}) + c \log m \log\log m$ where c is a constant. This implies that $T(m) = O(\log m(\log\log m)^2)$.

That m processors suffice follows from the following analysis. Steps 1, 2(c) and 3 are nonrecursive and clearly take m processors. Step 2(a) takes m processors since it does $\sqrt{m}$ parallel recursive calls of size $\sqrt{m}$ each, and the same holds for Step 2(b). Step 4 takes $\sum_{i=1}^{\sqrt{m}} \sum_{j=1}^{l_i} |Q_{i,j}|$ processors, which equals m. Step 5 takes, for a particular Q_{i,l_i}, a number of processors equal to:

$$\sum_{j=1}^{\sqrt{m}} |Q'_{i,j}||S_{i,j}| \;=\; \sum_{j=1}^{\sqrt{m}} |Q'_{i,j}|(\sqrt{m}/|Q_{i,l_i}|) \;=\; |Q_{i,l_i}|(\sqrt{m}/|Q_{i,l_i}|) \;=\; \sqrt{m}.$$

Therefore the total number of processors for Step 5 is $\sum_{i=1}^{\sqrt{m}} \sqrt{m} = m$.

Let the overall space complexity (taking all processors into consideration) be $S(m,n)$. That $S(m,n)$ is worse than linear is apparent when one considers the fact that, before a parallel recursive call, many segments that are not needed in the recursion might have to be stored nevertheless (they are needed after the parallel recursive call returns). For example, in Step 4, in a non-full subproblem only some selected segments from an S_i are part of the recursive call, but the other segments (that are in between the selected ones) must nevertheless be stored, as they will be needed later, in Step 5, to complete the processing of the non-full subproblems. More formally, Steps 2(a) and 2(b) imply that $S(m,n)$ is at least $\sqrt{m}S(\sqrt{m},\sqrt{m})$. Step 4 implies that $S(m,n)$ is at least $\sum_{i=1}^{\sqrt{m}} \sum_{j=1}^{l_i} S(|Q_{i,j}|, |Q_{i,j}|n/m) + n$, where the additive n term is due to the fact that we need to store segments of S that do not participate in the recursion but are needed after the recursion returns. The other steps, which are nonrecursive, clearly require linear space. Putting

these observations together gives the recurrence:

$$S(m,n) = \max \{ \sqrt{m}S(\sqrt{m},\sqrt{m}) , \sum_{i=1}^{\sqrt{m}} \sum_{j=1}^{l_i} S(|Q_{i,j}|, |Q_{i,j}|n/m) + n \}.$$

A tedious but straightforward proof by induction reveals that the above recurrence has a solution $S(m,n) \leq c_1 m + c_2 n \log\log m$ where c_1 and c_2 are constants. Finally, we must argue that it is possible to "spread" the data roughly evenly among the m processors before we can claim that the space per processor is $O((n/m)\log\log m)$. This, however, can be achieved by spreading the segments that do not participate in the recursion (but are needed when the parallel recursive calls return) among the least space-congested processors (the details, using routing, are straightforward and omitted).

4 Algorithm for $m = n$

The algorithm uses the one in the previous section to achieve, for the symmetric case of $m = n$, $O(\log n(\log\log n)^2)$ time on n hypercube processors each of which has $O(1)$ local memory. It does so as follows.

1. Let $t = \log\log n$. Partition S into n/t chunks of size t each, call these $S_1, \ldots, S_{(n/t)}$. Call $\hat{S}$ the set of n/t elements that are at the boundaries of adjacent chunks.

2. Use the algorithm of the previous section on the set of all n query points and the slab segments in $\hat{S}$. The time complexity is $O(\log n(\log\log n)^2)$, the processor complexity is n, and the space per processor is $O(n^{-1}|\hat{S}|\log\log n)$ $= O(1)$ since $|\hat{S}| = n/t$ and $t = \log\log n$.

3. Let Q_i denote the subset of Q which belongs in S_i. Partitioning Q into $Q_1, \ldots, Q_{(n/t)}$ is easily done by sorting the queries of Q based on which S_i they belong to (which is known from Step 2). Next, do the following in parallel for all the Q_i, S_i pairs: For $j = 1, \ldots, t$ in turn, broadcast (using a segmented parallel prefix) the jth segment in S_i to all the points in Q_i, and through these t iterations have each point in Q_i keep track of the nearest segment of S_i that is to its left, and the nearest one to its right. This step completes the solution, and takes a total of time $O(\log n(\log\log n)^2 + t\log n) = O(\log n(\log\log n)^2)$ time and $\sum_{i=1}^{(n/t)} \max\{|Q_i|, t\} = O(n)$ processors.

5 The Asymmetric Case Revisited

Now that we have the algorithm for the symmetric case of $m = n$, we can revisit the asymmetric case of $m \geq n$ and improve on the earlier bounds we gave for it, in that a processor now needs only $O(1)$ local memory registers. This is done as follows.

1. View the m query points as being partitioned (arbitrarily) into m/n chunks $Q_1, \cdots, Q_{(m/n)}$ of size n each. Make m/n copies of the segments in S, one copy for each Q_i. Time: $O(\log m)$ with m processors (using a segmented parallel copying on n interlaced subcubes of size m/n each).

2. Use the symmetric algorithm of the previous section on each Q_i and that Q_i's private copy of S. Time: $O(\log n(\log\log n)^2)$ with $n(m/n) = m$ processors.

Thus the overall time for the case $m \geq n$ is now $O(\log m + \log n(\log\log n)^2)$ with m processors having $O(1)$ local memory registers each. Note that not only is this better than the preliminary algorithm in the sense of requiring less space per processor, but it is also faster if n is substantially smaller than m.

Finally, we consider another asymmetric case: That when $m < n$. Of course we now need n processors since it takes that many just to store the n segments. The algorithm is as follows.

1. View the n segments of S as being partitioned (in left to right order) into n/m chunks $S_1, \cdots, S_{(n/m)}$ of size m each. Make n/m copies of the points in Q, one copy for each S_i. Time: $O(\log n)$ with n processors (using a segmented parallel copying on m interlaced subcubes of size n/m each).

2. Use the symmetric algorithm of the previous section on each S_i and that S_i's private copy of Q. The symmetric algorithm changes the order of Q within each chunk, therefore after that algorithm returns we restore the original ordering of Q within each of the n/m chunks (by using sorting); the purpose of this reordering of Q within each chunk will become apparent in Step 3 below. Time: $O(\log m(\log\log m)^2)$ with $m(n/m) = n$ processors.

3. There are now n/m copies of each point p of Q, evenly spaced at a distance of m apart, and with the jth copy of p containing the region of S_j in which p lies. Therefore it is easy to "combine" these partial answers (n/m of them for each p) in $O(\log n)$ time by doing a "min" kind of computation on each of m interlaced subcubes of size n/m each (each subcube corresponds to a particular query point p).

Thus the time for the case $m < n$ is $O(\log n + \log m(\log\log m)^2)$ with n processors having $O(1)$ local memory registers each.

6 Applications

The improved algorithm for multisearching implies corresponding improvements in the hypercube complexities of a number of geometric problems:

- Batched planar point location,
- Trapezoidal decomposition,
- Polygon triangulation.

For all of the above problems the improvement is by a factor of $\log\log n$ in their hypercube time complexity. For a discussion of how an improvement in multisearching translates into an improvement in each of these problems, see [3, 6, 7, 8].

Acknowledgement. The author is grateful to two anonymous referees for their careful reading and useful comments.

References

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

[2] M.J. Atallah, R. Cole and M.T. Goodrich. Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms. *SIAM J. on Computing*, 18, pp. 499–532, 1989.

[3] M.J. Atallah and A. Fabri. On the Multisearching Problem for Hypercubes, *Proc. 6th Parallel Architectures and Languages Europe Symp.*, Athens, Greece, 1994, Springer Verlag Lecture Notes in Computer Sci.: 817, 1994, pp. 159–166.

[4] K.E. Batcher. Sorting Networks and Their Applications. *Proc. AFIPS Spring Joint Computer Conference*, pp. 307–314, 1968.

[5] R. Cypher and C.G. Plaxton. Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers. *ACM Proceedings of the 22th Annual ACM Symposium on Theory of Computing*, pp. 193–203, 1990.

[6] F. Dehne and A. Rau–Chaplin. Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry. *J. of Parallel and Distributed Computing*, Vol. 8, pp. 367–375, 1990.

[7] F. Dehne, A. Ferreira, and A. Rau–Chaplin. Parallel Fractional Cascading on Hypercube Multiprocessors. *Computational Geometry: Theory and Applications*, 2, pp. 141–167, 1992.

[8] D.T. Lee and F.P. Preparata. Parallel Batched Planar Point Location on the CCC. *Information Processing Letters*, 33, pp. 175–179, 1989.

[9] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[10] D. Nassimi and S. Sahni. Data Broadcasting in SIMD Computers. *IEEE Transactions on Computers*, C-30(2), pp. 101–107, February 1981.

[11] J.H. Reif and S. Sen. Randomized Algorithms for Binary Search and Load Balancing on Fixed Connection Networks with Geometric Applications. *SIAM J. on Computing*, 23, pp. 633–651, 1994.

[12] C.K. Yap. Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map. *Algorithmica*, 3, pp. 279–288, 1988.

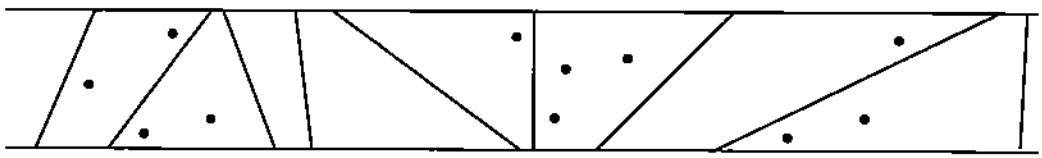

Figure 1. Point location in a subdivided horizontal slab.