Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications

John H. Reif* and Sandeep Sen*
Computer Science Department
Duke University
Durham, N.C. 27706

Abstract

There are now a number of fundamental problems in computational geometry that have optimal algorithms on PRAM models. We present randomized parallel algorithms which execute on an n-processor butterfly inter-connection network in $O(\log n)$ time for the following problems of input size n: trapezoidal decomposition, visibility, triangulation and 2-D convex hull. These algorithms involve tackling some of the very basic problems like binary search and load-balancing that we take for granted in PRAM models. Apart from a 2-D convex hull algorithm, these are the first non-trivial geometric algorithms which attain this performance on fixed connection networks. Our techniques use a number of ideas from Flashsort which have to be modified to handle more difficult situations; it seems likely that they will have wider applications.

1 Introduction

1.1 Motivation and overview

In the past decade, we have witnessed a systematic growth in the state-of-art of parallelizing algorithms in the PRAM environment. As a result of this, a number of basic problems have been recognized and many sophisticated techniques have been developed which can be viewed as a 'tool-kit' for tackling increasingly complex problems. There is a general consensus that PRAM models are appropriate for the algorithm designer but these algorithms have to be implemented on fixed-connection networks to be of any practical significance. By well known general-purpose emulation schemes, all these algorithms can be implemented to run on butterfly (or a hypercube) network with a $O(\log n)$ multiplicative factor degradation in time complexity. So the crucial question is if the PRAM algorithms can be extended to fixed-connection networks without this logarithmic penalty in running time.

*Research supported in part by DARPA/ARO contract DAAL03-88-K-0195, Air Force Contract AFOSR-87-0386, DARPA/ISTO contract N00014-88-K-0458, NASA subcontract 550-63 of prime contract NAS5-30428
A preliminary version of this paper appears in the Proc. of the 2nd Annual ACM Symp. on Parallel algorithms and architectures, Greece, 1990

Preprint of paper appearing in SIAM Journal of Computing 23:3, June, 1994, pp.633-651.

In a top-down approach to algorithm design, complicated algorithms are built on top of less complex procedures. The answer to the above question would depend on how far down in this hierarchy can one go without running into problems that cannot be mapped optimally on the fixed-connection network. Moreover, this would also depend on the nature of the algorithm itself. One of the most basic problem in this hierarchy is that of sorting. For example, Reishchuk's[19] $O(\log n)$ time, n processors randomized PRAM sorting algorithm was successfully extended to networks by Reif and Valiant[18] to run in $O(\log n)$ time by using additional new sampling techniques for problem-size control. In contrast, Cole's deterministic $O(\log n)$ time parallel mergesort algorithm seems prohibitively difficult to implement (without a logarithmic slowdown) on the networks because of its liberal use of pointers. Consequently a number of algorithms that use this approach on PRAM models would be at least as difficult to be adapted to network models.

Although the eventual goal of this paper is to present efficient algorithms on interconnection networks, we would like the reader to view this in the more general context of mapping certain kinds of PRAM algorithms on fixed-connection networks and the difficulties associated therewith. We encountered several problems which appear to be very basic for this line of research and we believe that these will have much wider applications.

REMARK: *In this paper, the term fixed-connection network has been used to allude to networks which have* $O(\log n)$ *diameter for* n*-node networks. There already exists a large body of literature for geometric algorithms on grid-like networks where the diameter is a bottleneck for achieving the kind of time complexity we are aiming for.*

One of the underlying problems is doing binary search optimally in a model that does not allow concurrent reads. A common scenario is the following: we are given a tree whose leaves represent intervals and n keys for which we have to determine the interval that each key lies in. If the depth of this tree is d then it is trivial to do this sequentially in $O(nd)$ time. In case of PRAM models which allow concurrent reads, the problem is again quite simple. With n processors, we can simultaneously do the search for all keys in $O(d)$ parallel time, thus resulting in an optimal speed-up. The main difficulty associated with this problem stems from the possibility that the keys may be very unevenly distributed among the intervals. In this paper we have described a randomized algorithm to do this in an *EREW* PRAM in $O(\log n)$ time and refined it further to run in the same asymptotic time on an n-node butterfly network.

An additional problem is that of allocation of sub-problems to sub-networks for recursive calls. Unlike *PRAM* models the network topology imposes severe constraints on processor allocation - not only the number of processors should match with the sub-problem sizes but should also be inter-connected in a certain manner. Our solution to this problem could be applied to a more general situation than the applications described in this paper. Some of the ideas are similar to *Flashsort* where one does *splitter-directed routing* to route the keys to the appropriate sub-networks. However, unlike the *Flashsort* we may be confronted with situations where we have to dynamically allocate resources as the sub-problems could have varying sizes. One of our results is that we have near-optimal solutions to the above problems that should have applications to a wide class of algorithms. These basic procedures serve as crucial link between the PRAM algorithms and inter-connection networks.

1.2 Geometry on fixed-connection networks and main results

Designing efficient parallel algorithms for various fundamental problems in computational geometry has received much attention in the last few years. There have been two fundamentally distinct approaches to this area of research, namely the deterministic methods and algorithms that use random sampling. One of the earliest work in this area is due to Chow [4], who developed algorithms for a number of fundamental problems which were deterministic and executed in inter-connection networks with polylogarithmic running time. A more general approach for deterministic PRAM algorithms was pioneered by Aggarwal et al. [1] who developed some new techniques for designing efficient parallel algorithms for fundamental geometric problems. However a majority of the algorithms were not optimal in $P \cdot T$ bounds. A number of the the most efficient deterministic PRAM algorithms are due to Atallah, Cole and Goodrich [2] who extended the techniques used by Cole [8] for his parallel mergesort algorithm. Their technique is called *Cascaded merging* and has been subsequently used (independently by Chandran [3]) for a number of other problems. Note that most of the geometric problems in the context of research in parallel algorithms have sequential time complexity of $\Omega(n \log n)$ and a typical performance that one aims for is $O(\log n)$ parallel time using an optimal number of processors.

In an independent development, Reif and Sen [16] were also able to derive $O(\log n)$ time optimal algorithms for point-location and trapezoidal decomposition which were randomized. Later they extended their methods to give optimal algorithms for 3-D convex hulls (and hence 2-D Voronoi diagrams) on the *CREW* PRAM model. At the core of their algorithms were random sampling techniques which had also been introduced by Clarkson [5, 6, 7] and Haussler and Welzl [10]. In addition, a new resampling technique called **Polling** was used successfully to derive the parallel algorithms. In essence, it is an efficient resampling procedure which enables us to choose a sample which satisfies certain properties with high probability from a set of samples for which only expected behavior is known. While no deterministic algorithms have been developed for some of the above problems that attain optimal bounds, we feel that the real impact of randomized techniques will be in the domain of parallel algorithms on fixed-connection networks.

In spite of interesting developments in the PRAM world, the state-of-art of geometric algorithms in the case of small diameter fixed-connection networks is lagging far behind. Presently, the only known $O(\log n)$ time algorithm for the network model is a 2-D convex hull algorithm due to Miller and Stout [13]. Consequently, the *Cascaded-merge technique* of Atallah, Cole and Goodrich [2], in spite of its elegance on the PRAM models appear to be of little use in a fixed-connection model. The only obvious way to implement pointer updates takes $O(\log n)$ time per step of the PRAM algorithm which would result in an $\tilde{O}(\log^2 n)$ time algorithm.

The randomized algorithms seem to be more amenable to mapping on fixed-connection networks although it is far from straight-forward. We derive an optimal $O(\log n)$ time randomized algorithms for constructing the trapezoidal decomposition. Using this, we can triangulate a simple polygon in $O(\log n)$ time.

1.3 Model of computation and notation

Throughout this paper we will be using the butterfly inter-connection model where the processors operate in a synchronous fashion and have bounded buffer size. These assumptions are consistent with some of the existing machines like BBN Butterfly and the Connection Machine. During every

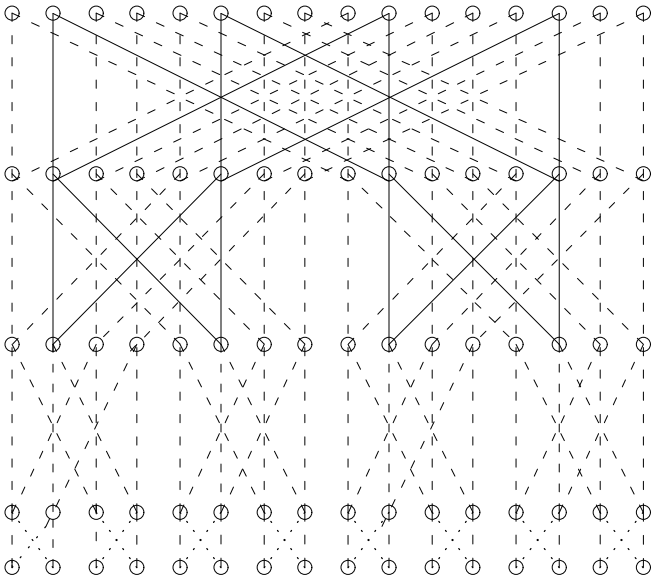

Figure 1: A butterfly network of size 4. The solid lines illustrates a sub-network isomorphic to BF_2

step, each processor is allowed to perform a real-arithmetic operation consistent with standard models used for sequential geometric algorithms. Moreover, each processor has access to a random-number generator that returns in unit time a truly random number of $O(\log n)$ bits. One of the primary reasons for choosing the butterfly network is because of its nice 'recursive' nature. A butterfly network of size k (will be referred to as BF_k) has k levels of 2^k nodes each (i.e. it has $k2^k$ nodes). Each node has an address (w, t) where $w \in \{0,1\}^k$ and $0 \leq t < k$ (see Figure 1). The significance of this network is that there are numerous 'copies' of BB_l in BB_k for $l < k$. We shall make crucial use of the following fact.

Fact 1 *For any w_1, w_2 such that $|w_1| + |w_2| = k - l$, then the subgraph of BF_k spanned by the nodes $\{(w_1 w w_2, i) | w \in \{0,1\}^l\}$ and $|w_1| \leq i \leq |w_1| + l$ is isomorphic to BF_l.*

Moreover, we may have to use emulate a larger butterfly in a work-preserving fashion. The following simple result can be used.

Fact 2 *A BB_k can emulate a BB_{k+c} within $O(2^c + c)$ slowdown where c is a positive integer.*

This is similar to Brent's slowdown lemma except that we have to construct a mapping of the processors of BB_{k+c} to BB_k respecting the interconnection topology. In this case it is very straightforward. Assume that $c = 1$ and map the processors with addresses (xw, i) where $x \in \{0,1\}$ and $|w| = k$ and $i \leq k$ to (w, i). One can verify that the processors of BF_{k+1} that were neighbors are neighbors in the smaller network. Each processor in the smaller network has to do at most twice the amount of work and require twice the amount of local memory. Only the processors of rank k have to do the extra work of emulating the processors of rank $k + 1$. This scheme can be extended directly to yield the claimed bound. For a fixed c, this implies that there a constant factor slow-down. In this paper this scheme will be often used with the value of c being 1 or 2.

The term *very high likelihood (probability)* is used in this paper to denote probability $> 1 - n^{-\alpha}$ for some $\alpha > 1$ where n is the input size. Just like the big-O function serves to represent the complexity bounds of deterministic algorithms, we shall use $\tilde{O}$ to represent complexity bounds of the randomized algorithms. We say that a randomized algorithm has resource bound $\tilde{O}$ (f(n)) if there is a constant c such that the resource used by the algorithm is no more than $c\alpha f(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for any $\alpha > 1$ for an input of size n. (An equivalent definition will be bounding the resource by $\alpha \cdot f(n)$ with probability greater than 1 - $n^{-c\alpha}$ and in the rest of the paper they will be used in an interchangeable manner).

The rest of the paper is organized as follows. In section 2, we briefly review two very important sub-routines in fixed-connection network, namely that of sorting and routing. In section 3, we describe an algorithm for doing binary search. This is used in section 4 to develop fast methods for searching in arrangements. We give a brief description of the algorithm for trapezoidal decomposition. We focus mainly on those portions for which we need different methods from the PRAM algorithms - the reader is referred to a previous paper (Reif and Sen [16]) for a more detailed description of the algorithms that were developed for the *CREW* PRAM model.

2 Overview of sorting and routing on fixed-connection networks

Our algorithms use sorting and routing extensively at various stages and a brief review of these routines will help us in understanding the latter algorithms that use them as building blocks. The problem of *packet routing* involves routing a message from processor i to $\Pi(i)$ where Π is a permutation function. There has been a long and rich history of routing algorithms for fixed connection networks (see [23, 22, 15, 11]) and these can be summarized as following

Lemma 1 *There exists an algorithm for permutation routing on a n-node butterfly network that executes in $\tilde{O}(\log n)$ steps and uses only constant size queues to achieve this running time.*

A more general result has been proved by Maggs et al. [22] for *layered* networks. A *layered* network is one whose nodes can be assigned layer numbers and the edges connects a layer i node to a layer $i+1$ node (butterfly is an example of such a network). Let d denote the maximum distance traveled by any packet and c the largest number of packets that must traverse a single edge (c is also called the congestion of the network). These parameters are fixed for a given selection of paths by all the packets to be routed. Then there exists a scheme for scheduling the movements of the packets such that with high probability the routing can be completed in $O(c + d + \log n)$ steps where n is the size of the network and $O(n)$ packets are being routed.

REMARK *Given the above result and also the fact that $d = O(\log n)$ for most path selection strategies (especially in a butterfly network), it remains to bound the value of c to get a bound on the routing time. For packets being routed to a random location, c can be bounded by $O(\log n)$ with high probability.*

The first optimal $\tilde{O}(\log n)$ time sorting algorithm called *Flashsort* for the butterfly network was due to Reif and Valiant [18]. It was based on a PRAM sorting algorithm due to Reishchuk [20] but required several additional techniques because of the constraints imposed by the network connectivity. A slightly simplified version can be presented as the following.

1. Select n^{ϵ}, $\epsilon < 1/2$ size random subset from the given set of n keys.

2. Sort these, using a simple method like doing all the pairwise comparisons and ranking them.
3. Use these keys to set up a binary tree such that the leaves of the tree correspond to the intervals defined by a pair of consecutive splitter keys. Over-sampling techniques are used to ensure that these intervals partition the remaining keys into roughly equal sized subsets. This eliminates the need for dynamic load-balancing in the special case of sorting. The keys are assumed to be in random locations initially. For each subset a sub-network of appropriate size is set aside and the keys that belong to this subset are routed to this part of the network. This is done using a procedure called *Splitter Directed Routing* which will referred as SDR in future references. Since this is a crucial component of our algorithm, we describe it in more detail in the appendix.
4. These steps are applied recursively until the size of the subproblems is no more than $\log^2 n$.

Although the original analysis showed that $\tilde{O}(\log n)$ buffer size may be required the more recent results on routing enables one to do with a constant amount of storage in each buffer([22]). The overall running time of the sorting algorithm was analyzed using the property that the problem size at recursive call at depth i is no more than n^{ϵ^i} and an appeal to the following theorem (Reif and Sen [17])

Lemma 2 *Given a process-tree which has the property that a procedure at depth i from the root takes time T_i such that*

$$P[T_i \quad \geq \quad kc\alpha \log n(\epsilon_0)^i] \leq \quad 2^{-(\epsilon_0)^i c\alpha \log n}$$

then, all the leaf-level procedures are completed in $\tilde{O}(\log n)$ time.

We would need a generalization of the above result where the process tree is modified in the following manner. Instead of all the sub-routines from a node proceeding independently, all the subroutines for a fixed (constant) depth subtree are required to finish before proceeding to the next level (of subtrees). This can be reduced to the previous case if we contract a subtree of fixed depth (see Figure 2) into a single node of the tree. By appropriate adjustment of constants, we can prove the following result on similar lines as the previous lemma.

Corollary 1 *All the leaf level procedures of this modified tree will terminate in $\tilde{O}(\log n)$ steps.*

The above two results will be used repeatedly in the course of the remaining paper.

3 Binary search without Concurrent Reads

One of the frequently encountered problems in case of sequential and parallel algorithms is that of doing a binary search on a tree structure. In particular, given a binary tree of depth $O(\log n)$ whose leaves represent certain intervals, and $O(n)$ keys with one processor per key, we would like to locate the interval that the key belongs to in $O(\log n)$ parallel time (for each key simultaneously). This problem is trivial in a model allowing concurrent-reads. However, the problem becomes more complicated when concurrent reads are not permitted in the model which is the case with inter-connection networks. The simple algorithm uses concurrent reads in an inherent fashion and for situations where the distribution of the keys is not known, this problem appears even more formidable. To make the exposition simpler we shall first describe a scheme for the $EREW$ PRAM and then modify it for the butterfly network. We also note that in certain cases where the intervals and the keys are chosen from the same total ordering the problem reduces to that of merging which

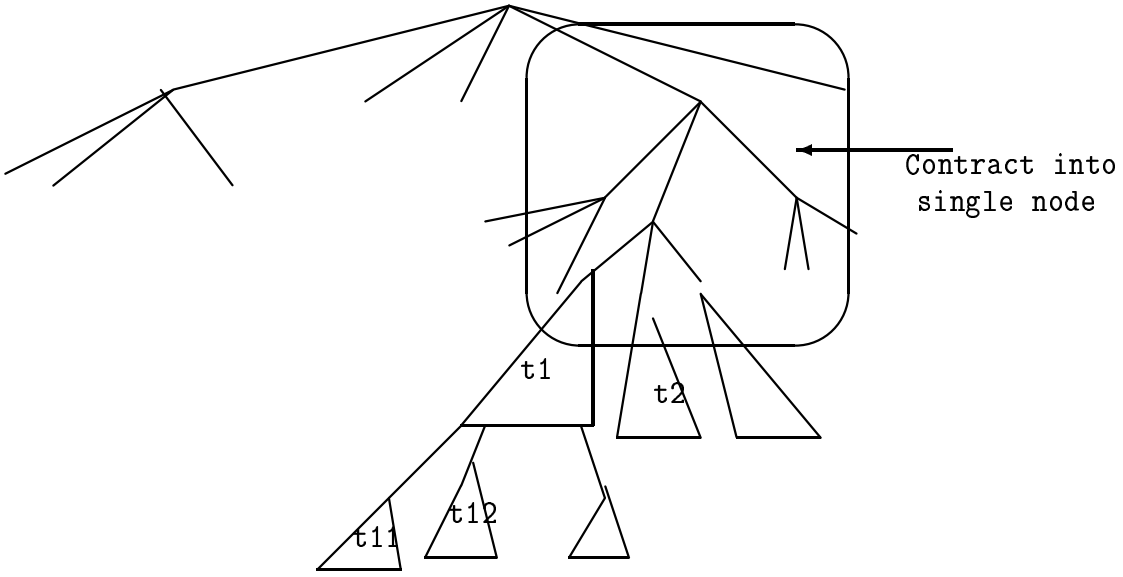

Figure 2: By contracting subtrees of a fixed depth, we get another tree satisfying the preconditions of the lemma

can be done efficiently. However, we are concerned about cases where the intervals may induce an ordering different from the ordering among the keys (see Figure 3).

We shall look at a special case where the number of leaves is n^ϵ for $0 < \epsilon < 1/2$ and the search tree is roughly balanced, so it has depth $O(\log n)$. The basic strategy is the following. We first try to get a reasonably accurate estimate of the number of keys associated with each of the leaf nodes. Following this, we simulate an inter-connection network like the butterfly and allocate appropriate number of sub-networks based on the estimate. Next we route the keys to their destination sub-networks using a scheme similar to the *splitter directed routing* (SDR) used in **Flashsort**.

The analysis for SDR carries through in this case although some of the nodes of the splitter tree may be 'artificial' (i.e. both the edges departing from a node leaves the packet in the same

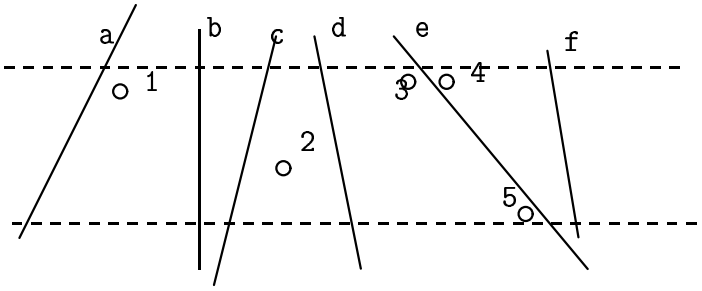

Figure 3: Points 4 and 5 are in different relative orderings with respect to each-other and the segments

interval which is a large sub-network in our case). We show that each of these steps succeeds with high probability.

To obtain an estimate of the number of keys in each range, we make use of a technique used earlier in Reif and Sen [17] and Rajasekaran and Reif [14]. The result can be summarized in the following manner.

Lemma 3 *Let $S_j, 1 \leq j \leq m \leq n$ be disjoint subsets such that $\sum_i S_i = n$. If we sample elements independently and uniformly from the union of the subsets with probability p and s_j is the number of elements in the sample from S_j, then for all j such that $p|S_j| \geq d \log n$, s_j is within a constant factor of $p|S_j|$ with high probability. Here d is a constant and the ratio of s_j to $|S_j|$ depends on d.*

As a consequence of this lemma, we set aside $max\{d \log n \cdot 1/p, E_j\}$ for each subset which are intervals in this case. E_j is an estimate for S_j computed from s_j and is $O(1/p|S_j|)$. Note that, for most applications we would choose a value of p such that $md \log n/p \leq n$. We start by selecting each key with probability $\frac{1}{n^{1-\beta}}$; the exact value of β will be determined later. From Chernoff bounds, it follows that the number of keys in the sample is $O(n^\beta)$ with high probability. For each of these keys, we can determine which interval they belong to by using a brute-force method (simply checking each key against every interval). This can be done in $O(\log n)$ time if we choose $\epsilon + \beta < 1$. From the previous lemma, it follows that if the total number of keys N_i in an interval i exceeds $cn^{1-\beta} \log n$, where c is a constant independent of n, we can get estimates E_i (of N_i) within a constant factor. So we can set aside $max(cn^{1-\beta} \log n, E_i)$ space for each interval. For $n^{1-\beta} \cdot n^\epsilon \log n < n$, the total space can be bounded by kn for some constant k. From the previous two inequalities involving ϵ it can be seen that $\epsilon < 0.5$. A possible choice for the parameters ϵ and β can be 0.49 and 0.5 respectively. After routing (i.e. by simulating a bounded queue butterfly network), we have the keys in the appropriate interval. We can now sort them in $\tilde{O}(\log n)$ time and determine exactly how many keys are there in each interval.

3.1 Binary search and splitter directed routing

From the scheme described in the previous paragraph, we set aside $max(cn^{1-\beta} \log n, E_i)$ sized sub-network for keys in interval i, K_i. The number of addresses (rows) allocated to a particular interval j is E_j/L. It can be shown that the total number of packets that arrive at any fixed address (that is over all the L levels) does not exceed $O(L)$ with high likelihood. (The expected number of packets arriving at any particular address is $L > O(\log n)$).

REMARK *Note that the 'sub-networks' are not isomorphic to 'butterfly' networks of that size - they have to be routed to 'sub-butterflies' after the termination of the splitter directed routing when we map the algorithm for butterfly network.*

We can assume that E_i divides $cn^{1-\beta} \log n$, so that we can refer to this size as a 'unit' of sub-network. We can also assume for simplicity that this is a power of two. Therefore, we can allocate a number of 'units' of sub-networks from our estimates. As seen from Figure 4 the number of these subnetworks may not be aligned with the binary search data-structure. More specifically at a particular node, there could be sub-networks allocated (for a particular interval) on both the left and right subtrees. We can handle this problem as following: Each packet i that could go either left or right, goes left with probability $L_{k(i)}/(L_{k(i)} + R_{k(i)})$ where $L_{k(i)}$ ($R_{k(i)}$) is the number of 'units' on the left (right) subtree for the interval $k(i)$. The packet goes right with the complementary

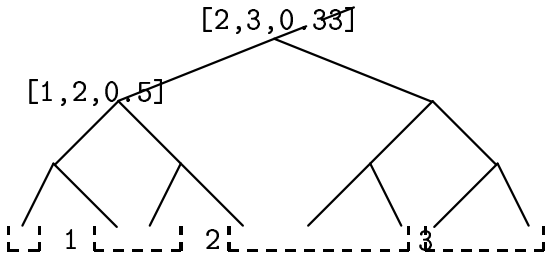

Figure 4: Intervals 1-2 and 2-3 are not aligned with the binary tree. The labels at the nodes indicate the range and the probability that a key in the range taking the left branch. Keys outside this range take the conventional routes.

probability. The number of packets going left (right) is a binomial random variable with mean at least $n^{1-\beta}\log n$ and hence the probability that it deviates from the mean by a small (constant) factor is less than $1/n^{\alpha}$ for any fixed $\alpha > 0$ from Chernoff bounds. The probabilities for this branching can be assigned when the data-structure is set up.

We can now use arguments similar to Maggs [12] to bound the congestion. The number of packets that enter any sub-butterfly is within a constant factor of the size of the sub-butterfly. For a switch at level l, at most $O(L \cdot 2^{L-l})$ rows of the butterfly can be reached for a BF_L. Moreover a switch at level l can be reached from 2^l different inputs. If a packet begins at a random node, the probability of reaching a particular switch is 2^{l-L}. The number of packets that pass through a given switch is a binomial random variable. Using Chernoff bounds it can be shown that this number (and hence the congestion) exceeds OL ($\geq O(\log n)$) is less than $n^{-\alpha}$ for any fixed α. We can therefore conclude from the remark after Lemma 1 that this modified *Splitter Directed Routing* (to be referred as $MSDR$) terminates in $\tilde{O}(\log n)$ steps.

We now apply this procedure recursively in the following manner: For each interval with N_i keys we make $\lfloor N_i/n^{1-\epsilon} \rfloor$ 'copies' of the subtree with L_i as its root. The number of processors allocated to each of these problems is $n^{1-\epsilon}$. For the remaining keys, say r_i, we round it to the nearest power of 2 and 'copy' a subtree with that many nodes and apply this strategy recursively. Clearly, the problem size is decreasing as $n^{1-\epsilon}$ and it can be shown (using Lemma 2) that the entire procedure terminates in $\tilde{O}(\log n)$ time. Notice that when the problem size becomes $O(\log n)$, we can solve the problem in $O(\log n)$ time by pipelining a sequential search algorithm.

Mapping this algorithm on the interconnection is not straight-forward since we do not have the luxury of allocating the required space as in case of PRAMS. Instead, we have to simulate a larger size butterfly network (larger by a constant factor). Moreover, between successive recursive stages we have to do a careful routing to set up the search tree. Unlike the PRAM model this may force us to do global routing to access the appropriate subtrees. So, each recursive call could take $\tilde{O}(\log n)$ time. Note that the sorting algorithm is randomized. Consequently, the expected running time increases by an $O(\log\log n)$ multiplicative factor.

To circumvent the above difficulty we take a different approach. Once the problem size becomes $[O(\log^p n), O(\log^p n)]$ (the number of keys and the size of data-structure respectively), we can solve the problem by emulating PRAM algorithms in an additional $O(\log\log^2 n)$ deterministic time per step. Here p is some fixed integer. This has the following consequence - we can look at a pruned

search tree of size $O(n/\log^p n)$ since if we can determine the subtree where each key belongs to, in an additional $O(\log\log^3 n)$ time we can complete the entire search procedure. We need some more observations:

Lemma 4 *Given a tree data structure T of size $|T|$, we slice off the tree at a certain depth and execute a search on this truncated tree. Then we allocate space to the sub-trees which are not empty i.e. there are keys in that interval and make copies of the sub-trees such that the number of keys in any subtree is no more than the subtree size. Then the total size of the sub-problems is no more than 2 $|T| + N$ where N is the total number of keys.*

Proof For the sub-trees that are full, we can charge the space to the keys in it. For the partially full trees, there can be at most one for each subtree and the lemma follows. □.

This lemma implies that after completing the search on the $O(n/\log^p n)$ size tree, the total space required is less than $2n$ which can be simulated on the network with only a constant factor increase in running time. Moreover, the lemma also says that if we use the above processor allocation strategy, the size of the network required at stage i of the recursion can be bounded as following:

Lemma 5 *The total size of the sub-networks at level i is no more than $i(N + |T|)$, where N is the number of keys and T is the data structure.*

Proof We prove it using induction on the depth of recursion i. Clearly this holds after the first level from the previous lemma. Assume that this is true for level k i.e. the total space required is no more than $k(N + |T|)$. In the next recursive call, we use the strategy described previously (of making 'copies') and use the following charging scheme. For the 'completely-full' sub-trees of level k, the space required for the completely-full sub-trees of level $k + 1$ is charged to the keys. For the partially-full sub-trees of level $k + 1$ resulting from 'completely-full' sub-trees of level k, the extra space can again be charged to the keys since the size of the data-structure in level k was matched with the number of keys. For the partially-full sub-trees from level k, the completely-full sub-trees can be charged to the keys by the previous argument. For the partially-full subtrees of level $k + 1$ resulting from partially-full sub-trees of level k, they are charged to the data-structure of level k.

Hence the total amount of extra space required at level $k + 1$ is $N + |T|$. □

From the above observations, we proceed as follows. We consider first a reduced problem where the number of keys $N = n/\log n$ and the size of the tree is $n/\log^3 n$. The $O(n/\log n)$ keys are chosen uniformly at random and the tree is pruned to this size. The processor-allocation strategy is to allocate sub-networks proportional to size $(\log\log n - i)(|T_i| + n_i)$ for subproblem with sub-tree size $|T_i|$ and n_i keys. Notice that even if the number of recursive levels is $O(\log\log n)$, we can do a global routing after every $\log\log n$ levels, so that the size is no more than $3n$ after each of those processor-allocation procedure.

From the previous lemmas, we have enough processors to carry out the reduced search problem. If the keys are chosen uniformly at random with probability $1/\log n$, then, for all the subtrees (of size $\log^3 n$), which have more than $\log^2 n$ keys we have very accurate estimates (Lemma 3). So we can set aside $max\{c\log^n, E_l\}$ sized sub-networks for sub-tree l where E_l is an estimate. Note that $cn/\log n + \sum_l E_l < O(n)$ and perform the SDR on all the n keys this time. Subsequently we count explicitly the number of keys in each subtree and allocate subnetworks with the maximum problem size of $O(\log^3 n)$. We can then emulate any PRAM algorithm adding $O(\log\log^3 n)$ to time complexity.

3.2 Load balancing and processor reallocation

The previous strategy for processor allocation would work if the size of the subproblems always matches with the sub-butterfly size; however it may not always be the case. For example we may be off by a factor of two. So, we have to design a more general processor allocation procedure which can be done dynamically and also evenly distribute the work-load among the sub-butterflies.

Let us denote the subproblem j as $\mathcal{P}_j$ and a sub-butterfly to which the sub-problem is allocated as $\mathcal{N}_j$. Let $|S|$ denote the size of set S. We assume that we have an allocation procedure which achieves the following.

Lemma 6 *Given that $\sum|\mathcal{P}_j| \leq c_1 N$, where N is the total size of the network and c_1 is a constant, there is a processor allocation scheme such that for all j, $|\mathcal{P}_j| \leq c_2|\mathcal{N}_j|$ where c_2 is another constant.*

Proof Assume that the network size is $N \geq k\sum|\mathcal{P}_j|$ since we can always simulate a network of a larger by a factor k. This will be be used as induction hypothesis for the proof. Suppose $N = h \cdot 2^h$. Moreover, wlog assume that all the sub-problem sizes are of the form $t2^t$. (Again this increases the network size by at most a factor of 3 and can be absorbed in the constant k). Let S denote the subproblems whose sizes are larger than $\lfloor h/2 \rfloor 2^{\lfloor h/2 \rfloor}$ and the rest by $\bar{S}$. Allocate subnetworks greedily (the actual procedure is described later) and since the height of the sub-network is at least 1/3 of the total size, we will have misused at most a factor of 3. To apply the same procedure recursively to problems in $\bar{S}$, we may be forced to waste another row of subnetwork of same size. Thus the amount of network remaining is $k(|S| + |\bar{S}|) - 6|S| \geq (k-6)|S| + k|\bar{S}|$. For $k \geq 6$, the inductive assumption holds (that we have a network k times the sum of subproblems) and moreover we have at least two levels of sub-networks where the sub-problems of $\bar{S}$ can be accommodated. Algorithmically, this allocation strategy can be accomplished by sorting the sub-problem sizes, marking the problems whose sizes demarcate two thresholds and doing a prefix to find out what category (i.e. size-wise) a problem falls under. This is followed by routing to appropriate subnetworks and all of the above operations can be done in $\tilde{O}(\log N)$ time. □

Clearly, this scheme in itself is not sufficient to guarantee that over the entire course of a recursive algorithm a sub-network would not be loaded by more than a constant factor. For example, if we were to continue with this scheme recursively, a network could be loaded by a factor of approximately $(c_2)^2$ in the next recursive call. This is undesirable if the depth of recursion is $O(\log\log n)$. However, we can use the previous lemma in the following useful manner. We can apply it for a constant number of levels of recursive calls without increasing the run time by more than a constant factor (in every call the load on a processor increases by a constant factor). We shall show that this suffices for our processor allocation strategy, where we redistribute the load 'evenly' after a fixed number of levels of recursive call. The network topology, where the subnetworks are of sizes $h2^h$ (h is an integer) necessitates this kind of rebalancing strategy. By 'loading' we mean the ratio of the sub-problem to the sub-network size (which is unity in the beginning) and this is also a lower-bound on the slow-down of the algorithm.

We shall start with a special case where the network size is $h2^h$ where $h = 2^l$ for some integer l. In two levels of recursive call we can reduce the sub-problem sizes to $2^{h/2}$. (One call is not sufficient since the subproblem sizes have to be larger than $n^{1/2}$ which is $2^{h/2+\log h}$ in this case). Now we can pack $h/2$ sub-problems into a sub-network of size $(h/2) \cdot 2^{h/2}$. Moreover, there are $2 \cdot 2^{h/2}$ sub-networks of this size (Fact 1) which implies that the entire network is being used (instead of a fraction). Now we are in sub-networks of size $(h/2) \cdot 2^{h/2}$ and hence the procedure can be

applied inductively. Notice however that we have $h/2$ sub-problems in the network (instead of one) and we have to use the previous lemma to allocate sub-networks, thereby loading the sub-network by a constant factor in the next recursive call. But this is only for a fixed (at most 2) levels of recursive call and hence we can maintain the same asymptotic run-time. Moreover, we can also do the following:

Lemma 7 *If a set of processors is loaded by a factor λ, then this load factor does not increase during the course of the processor allocation strategy.*

To generalize this to arbitrary value of network-size, we reduce the problem sizes to 2^{2^k} for some k. This can be done in a constant number of recursive calls since $2^{2^{k+1}} = \left(2^{2^k}\right)^2$. Using Lemma 6, we can allocate sub-problems such that only a constant fraction of the network is unused. This happens if 2^{2^k} does not divide h evenly and consequently some processors may be loaded by a factor of 2. But from this stage we can use the scheme described in the previous paragraph without increasing the load-factor any further (Lemma 7).

In a more general scenario (as will be required later), the subproblems could be of different sizes. However, a key fact is that we can bound the size of the maximum sized subproblem by $n^{1-\epsilon}$ where n^ϵ is a sample size (similar to the previous case). Let $[i]$ denote the largest number less than i which is a power of 2, i.e. if $[i] = 2^j$, then $2^j \leq i \leq 2^{j+1}$. By choosing the sample sizes appropriately (n^ϵ for some $\epsilon > 0$), we can reduce the maximum sub-problem size to $2^{[l]}$. We then greedily pack as many subproblems as we can in a sub-network of size $[h] \cdot 2^{[h]}$. This is done as described before, namely, by sorting the sub-problem sizes and an application of parallel prefix followed by actually routing the sub-problems to the assigned sub-networks. We pack in subproblems as long as there is space, so that we do not 'load' a subnetwork by more than a factor of $(1 + 1/\log n^{1-\epsilon})$ (actually $1 + 1/h$ since we are packing at most $h + 1$ sub-problems in a network which can accommodate h sub-problems).

We apply this procedure recursively such that the extra loading factor at level i of recursion is no more than $1 + 1/\log n^{(1-\epsilon)^i}$. Note that in the first level of recursion, we could misuse a constant fraction of the network (because of problem size mismatch) but thereafter the degree of misuse can be bound by the above quantity by our processor reallocation strategy. If the subproblems are never smaller than $\log n$ then the load-factor can be bound by $\prod_{i=1}^{O(\log\log n)}(1 + 1/\log n \cdot (1-\epsilon)^i)$ which is less than e. Note that the 'load-factor' is maintained inductively in the sub-networks, by redistribution of work-load. The slow-down at any stage is at least 2 and no more than e (< 3). For example, after the first level only a fraction of the processors is 'loaded' by a factor of 2 but that also slows down the other processors because of the control structure of the algorithm. During the final stages a much larger number of the processors are loaded by a factor of 3.

We summarize our result as follows.

Theorem 1 *Given a binary search tree with $O(k)$ leaves, we can do binary search for $O(k)$ keys in $\tilde{O}(\log k)$ time in a butterfly network using k processors.*

4 Applications to Computational Geometry

In this section we apply the techniques developed in the previous sections to map some geometric algorithms efficiently on the butterfly network. The description relies partly on the work of the

authors on randomized PRAM algorithms. For a better understanding of the original algorithms, the reader is encouraged to refer to the papers [16, 17].

4.1 Searching in Arrangements

We focus our attention the following problem. Given an arrangement of n^γ lines, $\gamma < 1/3$, we want to find out for n given points the region that it belongs to. In particular we would like to do this in $O(\log n)$ time on a butterfly network. Dobkin and Lipton had described a very simple method for solving this problem using the following data structure. Find out all the pairwise intersections (there are $n^{2\gamma}$ in this case) and project them on the X axis. Within each interval, the lines can be totally ordered and one can set up a binary tree corresponding to this ordering. Thus, there are $n^{2\gamma}$ binary trees each of size n^γ. To find the region (which is a trapezoid by the above partitioning scheme) that a point belongs to, we do two binary searches - one along each direction.

To implement this algorithm in parallel, the data structure can be set up very easily in parallel by sorting. To search n points, we first sort the n points by their x coordinates and merge them with the end-points of the intervals. This saves the binary search in the x direction. If I_i is the set of points in the i-th interval, we allocate a subcube of size $|I_i| + n^\gamma$ nodes. Since $n + n^{3\gamma} < 2n$, for appropriate choice of γ and moderately large n, this can be done by simulating a network twice the size. The binary search can then be done within each sub-cube in $\tilde{O}(\log n)$ time using the procedure outlined in the previous section. Thus the overall algorithm runs in $\tilde{O}(\log n)$ time.

We shall see in the next section that these trapezoidal regions define a finer partition of equivalence classes (which are the regions in the original partition).

4.2 A framework for randomized divide-and-conquer

Given a set of non-intersecting line-segments and points, we wish to determine for each point, the line segment(s) that are the first to intersect the vertical rays drawn through these points. There may be 0,1 or 2 such edges which are called the trapezoidal edges.

We shall recapitulate the main steps of the algorithm for trapezoidal-decomposition described in [21] in a more general context of a divide-and conquer algorithm

(1) Select $O(\log n)$ subsets of random objects (in case of 2-D hulls these were half-planes) each of size $\lfloor n^\epsilon \rfloor$ for some $0 < \epsilon < 1$. Each such subset is used to partition the original problem into smaller sub-problems. A sample is 'good' if the maximum sub-problem size is less than $O(n^{1-\epsilon} \log n)$ and the sum of the sub-problem sizes is less than $\bar{c}n$ for some constant $\bar{c}$. From the probabilistic bounds proved in Reif and Sen [16] (for problems considered in the paper) and Clarkson [7] (for very general situations which are applicable to our problems), it is known that the first condition for a 'good' sample holds with high probability. However, the second condition is satisfied in the with probability at least 1/2.
(2) Select a sample that is 'good' with high probability using *Polling*. At least one of the $\log n$ samples in the previous case is 'good' with high probability. *Polling* [17] is a sampling technique which allows us to choose a 'good' sample efficiently.
(3) Divide the original problem into smaller sub-problems (the maximum size can be

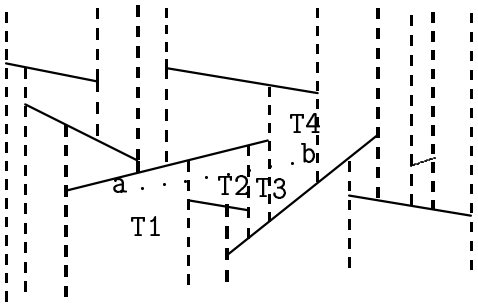

Figure 5: Convex Map of non-intersecting line-segments

bound by $O(n^{1-\epsilon} \log n)$) using the 'good' sample.
(4) Use a *Filtering* procedure to bound the sum of the sub-problem sizes by some fixed measure like the output size or input size. The reason for this being that the probabilistic bounds in step (1) bounds the sum of the sub-problems by $\bar{c}n$. If this increase by a multiplicative constant continues over each recursive level, after $O(\log \log n)$ depth, the input size will have increased by a polylogarithmic factor. This *filtering* procedure is problem dependent and uses the specific geometry properties of a problem. The purpose is to bound the number of processors.
(5) If the size of a sub-problem is more than a threshold (usually it is chosen to be $O(\log^k n)$ for some constant k), then call the algorithm recursively else solve the problem using some direct method.

The algorithms presented in this paper are based on this approach. However the implementation of some of these steps depend heavily on the specific problem. The procedure used for dividing the subproblems depends on the problem at hand and this is also closely linked with *Polling*. Polling involves selecting $n/\log^2 n$ input objects and partition them using a random subset chosen in step (1) instead of the entire input set. Since there are $O(\log n)$ subsets, this saves the extra work we would have to do if we tested the 'goodness' of the sample on the entire input. The **Polling lemma** [17] guarantees that with high probability we can choose a good sample using this method. Since the test for 'goodness' is carried out independently for each of the sample, this part of the algorithm is inherently parallelizable even on the networks. Perhaps the step that is most specific to a problem is the *Filtering* step where we have to use some geometric properties of the problem.

In the context of mapping algorithms to inter-connection networks, step 3 turns out to be the most difficult. Steps 1 and 2 are inherently parallel and step 4 for the problems that we are interested in problem is not very involved. We shall show that step (3) reduces to searching in an arrangements of lines in two dimensions.

4.3 Trapezoidal decomposition

For trapezoidal decomposition, we sample line-segments and build its convex-map (see Figure 5). We can build this using the following brute-force approach. For every segment end-point, we order the line-segments by their y coordinates. For every segment, we order the projection of the end-

points those are visible (from the bottom and top). From this information, we can construct the trapezoids by simulating 'pointer-jumping' a fixed (at most 6) number of times.

Lemma 8 *The trapezoidal map of n^ϵ segments segments can be constructed in $\tilde{O}(\log n)$ time in a $n^{3\epsilon}$ processor butterfly network.*

The remaining segments (not part of the sample) are partitioned into subproblems defined by the trapezoidal map. The partitioning step in trapezoidal decomposition can also be reduced to the problem of searching in arrangements of linear constraints in two dimensions (see Reif and Sen [16]). This is also known as the *locus* method where the problem reduces to finding out the region of the arrangement a query point lies in. These regions are pre-processed so that after the point location, one has to do a table look-up to determine the answer. In this case it is a string of trapezoids (that a segment intersects). Since this can be of various length, it has to be done with a little care. First the number of trapezoids is determined (this is just a number) and then an appropriate number of processors is delegated the responsibility of determining the actual trapezoids. This allocation is done with the help of a prefix computation. Subsequently, each of this processor do a table look up. The table look-up is done by emulating a single step of the CREW PRAM in $\tilde{O}(\log n)$ steps.

For the filtering step, we need to keep track of parts of segments that completely span a trapezoid. Such segments within a trapezoid have to be processed for binary search such that for end-points lying within the trapezoid, we can quickly determine its closest visible (upper and lower) segments. This can be done in $\tilde{O}(\log n)$ time using the binary search algorithm described in section 3. Moreover, only the segments that partially or completely lie within a trapezoid are needed for further recursive calls.

At each stage we keep track for each end-point, which are its closest (upper and lower) segments and hence at the end we have its trapezoidal edge(s). From this information, we can decompose a simple polygon into one-sided monotone polygons (see Figure 7). This can be accomplished by an application of sorting and prefix computation (Goodrich [9]). Using Yap's [24] technique, further calls to trapezoidal decomposition within these one-sided monotone polygons enables us to determine all the triangulation edges. The procedure is as follows.

Step 1 Construct the horizontal trapezoidal decomposition of the one-sided monotone polygon i.e. for every vertex v determine the edges of the polygon that a horizontal line through v while the line is contained within the polygon. Assume that the distinguished edge (the one with which the polygon is monotone) is horizontal.

Step 2 Denote the left visible edge by $l(v)$ and right visible edge by $r(v)$ and let $\lambda(v)$ and $\rho(v)$ denote vertices of $l(v)$ and $r(v)$ respectively which have the lower altitude.

Stepp 3 The set of edges $E = \{v\lambda(v)\} \cup \{v\rho(v)\}$ form a triangulation of the polygon.

Trapezoidal decomposition also enables us to solve the problem of determining visibility of a set of non-intersecting line segments when they are projected orthogonally. Sort the end points of the line segments projected on the x-direction and choose a point (say the mid-point) in every interval. For each of these points, determine its trapezoidal edge using the trapezoidal decomposition algorithm described above.

We can state the main result of this section as the following.

Theorem 2 *There exist algorithms for problems of Trapezoidal Decomposition, Triangulation of Simple Polygons and Visibility that execute in time $\tilde{O}(\log n)$ on an n-processor butterfly network where n is the input size of the problem.*

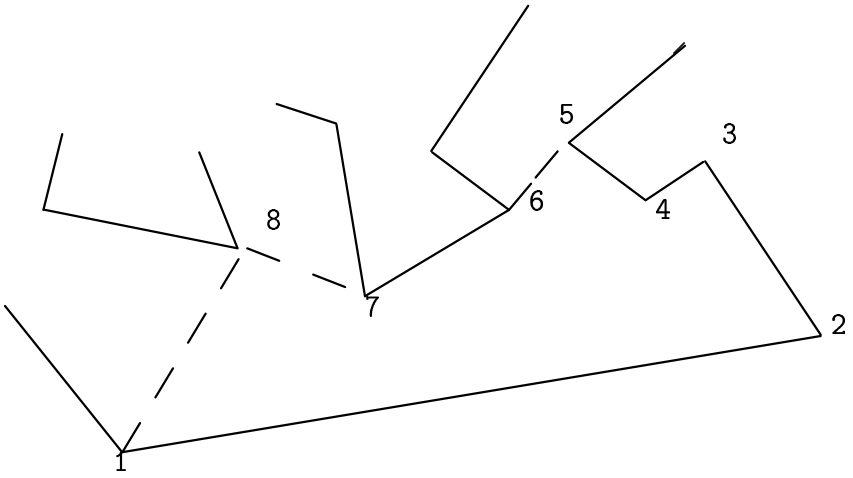

Figure 6: Decomposition into one sided monotone polygons. The dotted lines indicate some of the triangulation edges. The distinguished edge is 1,2.

4.4 2-D convex hulls

We need the following preliminary results:

Lemma 9 *The convex hull of n^α, $\alpha < 1/2$ half-planes can be constructed in $O(\log n)$ time in an n-node butterfly network.*

Assume that the intersection is non-empty and bounded and then a brute-force algorithm suffices.

Given this hull and a point that lies in the final hull, we divide this hull into triangular sectors and find out for the remaining half-planes which of the sectors they intersect. For this we go to the dual plane, and search in the arrangements of the (duals of) vertices of the convex hull of the sampled half-spaces. This can be done using the procedure described in the previous section - the details are straight-forward and are omitted for brevity.

The filtering scheme for 2-D convex hulls involves an application of 2-D maxima.

Lemma 10 *The maximal points of n points in a plane can be computed in $\tilde{O}(\log n)$ time in an n-node butterfly network.*

Proof: We use a variation of the sequential algorithm in the following manner. We sort the points based on their x-coordinates and do a prefix computation on the operation maximum on this sorted set. If $p_1, p_2 ..., p_n$ are the points sorted on the x-coordinates then we compute an array MAX such that MAX_i has the maximum y coordinate for all points p_j such that $j > i$. If the y-coordinate of p_i is smaller than MAX_i, then it is not a maximal element. □

Finally we can state the result of this section as

Theorem 3 *The convex hull of n points in a plane can be constructed in $\tilde{O}(\log n)$ in an n node butterfly network with bounded buffer size.*

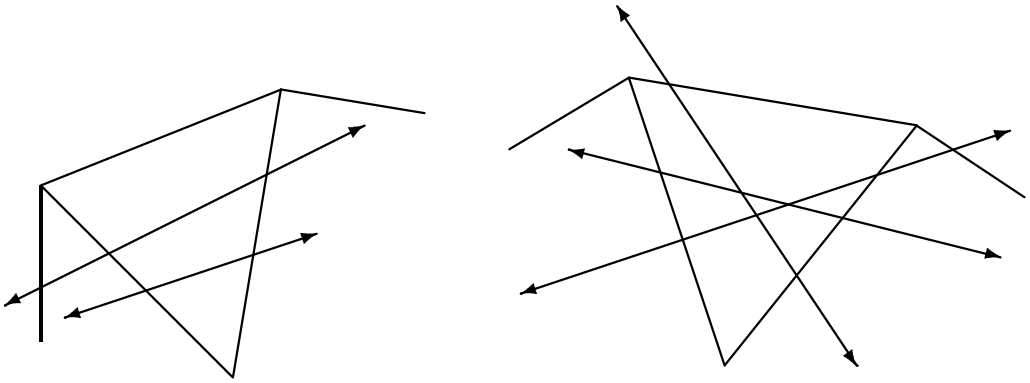

Figure 7: In Case 1, the redundant half-planes can be identified by an application of the 2-D maxima algorithm where the coordinates are the rank of the intersection of the half-planes on the sides of a sector. In case 2, we may not be able to detect such redundant half-planes but for the other sectors the redundant half-plane can be detected from case 1. Hence we shall carry at most one extra copy of a half-plane and hence the size can be bound by the input and output size.

5 Concluding Remarks

In this paper, we have described a strategy for implementing PRAM algorithms for geometric problems on fixed-connection networks. These methods involve tackling some of the very basic problems like binary search and dynamic load-balancing that we take for granted in PRAM models. Our techniques use a number of ideas from *Flashsort* but they have to be modified to handle more difficult situations, namely searching in partial orders and dynamically allocate sub-networks to recursive calls.

An important goal of our research is to build up a hierarchy of fundamental geometric algorithms for fixed connection networks similar to that of PRAM algorithms. Two very important problems in this regard are that of constructing 2-D Voronoi diagrams 3-D convex hulls in optimal (or near-optimal) time.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Proc. of 25th Annual Symposium on Foundations of Computer Science*, pages 468 – 477, 1985. also appears in full version in Algorithmica, Vol. 3, No. 3, 1988, pp. 293-327.

[2] M.J. Atallah, R. Cole, and M.T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *Proc. of the 28th Annual Symposium on the Foundations of Computer Science*, pages 151 – 160, 1987.

[3] S. Chandran. *Merging in Parallel Computational Geometry.* PhD thesis, University of Maryland, 1989.

[4] A. Chow. *Parallel Algorithms for Geometric Problems.* PhD thesis, University of Illinois at Urbana-Champaign, 1980.

[5] K.L. Clarkson. A probabilistic algorithm for the post-office problem. *Proc of the 17th Annual SIGACT Symposium*, pages 174 – 184, 1985.

[6] K.L. Clarkson. New applications of random sampling in computational geometry. *Discrete and Computational Geometry*, pages 195 – 222, 1987.

[7] K.L. Clarkson. Applications of random sampling in computational geometry ii. *Proc of the 4th Annual ACM Symp on Computational Geometry*, pages 1 – 11, 1988.

[8] R. Cole. Parallel merge sort. *Proc. of the 27th Annual IEEE Symp. on Foundations of Computer Science*, pages 511 – 516, 1986.

[9] M.T. Goodrich. *Efficient Parallel Techniques for Computational Geometry.* PhD thesis, Purdue University, 1987.

[10] D. Haussler and E. Welzl. ϵ-nets and simplex range queries. *Discrete and Computational Geometry*, 2(2):127 – 152, 1987.

[11] A. Karlin and E. Upfal. Parallel hashing - an efficient implementation of shared memory. *Proc. of the 18th ACM STOC*, pages 160–168, 1986.

[12] B. Maggs. *Locality in Parallel Computation.* PhD thesis, Massachusetts Institute of Technology, 1989.

[13] R. Miller and Q. Stout. Efficient parallel convex hull algorithms. *IEEE Transaction on Computers*, 37(12):1605–1618, 1988.

[14] S. Rajasekaran and J. Reif. Optimal and sublogarithmic time radomized parallel sorting algorithms. *Technical Rept, Aiken Computing lab, Harvard University*, 1986. To appear in SIAM Journal on Computing.

[15] A. Ranade. How to emulate shared memory. *Proc. of the 28th IEEE FOCS*, pages 185–194, 1987.

[16] J.H. Reif and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Proc. of the 16th International conference on Parallel Processing*, 1987. To appear in Algorithmica.

[17] J.H. Reif and S. Sen. Polling: A new random sampling technique for computational geometry. *Proc. of 21st STOC*, pages 394 – 404, 1989.

[18] J.H. Reif and L.G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34:60 – 76, 1987.

[19] R. Reischuk. A fast probabilistic parallel sorting algorithm. *Proc. of the 22nd IEEE FOCS*, pages 212 – 219, 1981.

[20] R. Reischuk. A fast probabilistic parallel sorting algorithm. *Proc. of the 22nd Annual FOCS*, pages 212 – 219, 1981.

[21] S. Sen. *Random Sampling Techniques for Efficient Parallel Algorithms in Computational Geometry.* PhD thesis, Duke University, 1989.

[22] B. Maggs T. Leighton and S. Rao. Universal packet routing algorithms. *Proc. of the 29th IEEE FOCS*, pages 256–269, 1988.

[23] L.G. Valiant. A scheme for fast parallel communication. *SIAM J. on Computing*, 11:350 – 361, 1982.

[24] C.K. Yap. Parallel triangulation of a polygon in two calls to the trapezoidal map. *Algorithmica*, 3:279 –288, 1988.

6 Appendix

6.1 Splitter Directed Routing

Let X be the set of cN keys that are totally ordered by the relation $<$. V is the set of nodes in the network. Suppose that for some l $(1 \leq l \leq n)$ we are given a set of *splitters* $\Sigma \subseteq X$ of size

$|\Sigma| = 2^l - 1$. We index each splitter $\sigma[w] \in \Sigma$ by a distinct binary string $w \in \{0,1\}^L$ of length less than L. Let $\prec$ denote the ordering defined as follows: For $u, v, w \in \{0,1\}^L$,$w0u \prec w \prec w1v$. We require that for all $w_1, w_2 \in \{0,1\}^L, \sigma[w_1] < \sigma[w_2]$ if and only if $w_1 \prec w_2$. We assume that a copy of each splitter $\sigma[w]$ is available in each node $V[w]$. $V[w]$ is the set of nodes with rank $|w|$ with addresses prefixed by w (same as in Reif and Valiant [18].

Let $X[\lambda] = X$ where λ is the empty string. Initially we assume that the keys of $X[\lambda]$ are located in $V[\lambda]$, that is, the nodes of V having stage 0. The splitter directed routing tree is executed in l temporarily overlapping stages $i = 0, 1, ..l-1$. For each $w \in \{0,1\}^i$ the set of keys $X[w]$ that are eventually routed through $V[w]$ is defined recursively. The splitter $\sigma[w]$ partitions $X[w] - \sigma[w]$ into disjoint subsets

$$X[w0] = \{x \in X[w] | x < \sigma[w]\}$$

and

$$X[w0] = \{x \in X[w] | x > \sigma[w]\}$$

which are subsequently routed through $V[w0]$ and $V[w1]$ respectively.

In our case, we assume that after each recursive call, the sub-networks (of varying sizes corresponding to different subroutine calls) are relabeled as if these were isolated networks. The $V[w]$'s are then defined accordingly. The time analysis for this procedure is carried out using a *delay-sequence* argument and it can be shown that this takes $\tilde{O}(\log n)$ time in a BF_n.

6.2 Probabilistic inequalities

We say a random variable X upper-bounds another random variable Y (equivalently Y lower bounds X) if for all x such that $0 \leq x \leq 1$, Prob(X $\leq x$) $\leq$ Prob(Y $\leq$ x).

A Bernoulli trial is an experiment with two possible outcomes, namely, success and failure. The probability of success is p.

A binomial variable X with parameters (n,p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. The *probability mass function* of X can be easily seen to be

$$\text{Prob(X} \leq \text{x)} = \sum_{k=0}^{x} \left(\begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{n-k}$$

The tail end of the Binomial distribution can be bounded by *Chernoff* bounds. In particular the following approximations due to Angluin and Valiant are frequently used:

$$\text{Prob(X} \geq \text{m)} \leq (\tfrac{np}{m})^m e^{m-np} \quad (1)$$
$$\text{Prob(X} \leq \text{m)} \leq (\tfrac{np}{m})^m e^{-np+m} \quad (2)$$
$$\text{Prob(X} \leq (1-\epsilon)pn) \leq exp(-\epsilon^2 np/2) \quad (3)$$
$$\text{Prob(X} \geq (1+\epsilon)np) \leq exp(-\epsilon^2 np/3) \quad (4)$$

for all $0 < \epsilon < 1$.