We describe several algorithms for computing the convex hull of N points in the plane: (1) N processor EREW PRAM algorithm requiring time $O(\log^2 N)$, (2) $N/\log N$ processor EREW PRAM algorithm requiring time $O(\log^2 N)$, (3) $N/\log^2 N$ processor Butterfly algorithm requiring time $O(\log^3 N)$. Algorithms 2 and 3 give linear speedup in the worst case since the time for computing convex hulls sequentially is $\theta(N \log N)$.

We also describe Brent's scheduling principle which is used for automatically generating algorithm (2) above from algorithm (1).

We also consider computing convex hulls on other networks e.g. 2D mesh. For the 2D mesh, we can show that N the number of points must be extremely large to obtain linear speedup.

1 Preliminaries

The input to the algorithm is an array Points[1..N], which holds N points, each consisting of an x-coordinate and a y-coordinate. We assume that our processors can perform arithmetic operations on coordinates in constant time.

The convex hull of a set S of points is the smallest convex polygon such that each point is either on the boundary or in the interior of the polygon. Intuitively, if we have nails sticking out of the plane at each point, the convex hull is the shape a rubber band wrapped around them would take. The convex hull can be divided into two parts, an upper hull and a lower hull. The upper hull is simply the convex hull of $S \cup (0, -\infty)$. The lower hull of S likewise is the convex hull of $S \cup (0, \infty)$. We shall consider the problem of computing the upper hull; the lower hull can be computed similarly from which the convex hull can be computed easily.

2 N processor, $\log^2 N$ time algorithm

The algorithm has a preprocessing step in which it sorts the points by their x-coordinate. After that the algorithm is recursive. The algorithm computes the upper hull of the left $N/2$ points using $N/2$ processors; and the upper hull of the right $N/2$ points using $N/2$ processors. Then the hulls are merged into a single hull. The algorithm is given in figure 1.

The procedure Merge works as follows. The input to it consists of two hulls stored in subvectors U1 = Points[1..Hullsize1] and U2 = Points[N/2+1..N/2+Hullsize2]. The procedure has three parts: (1) We first identify the number n1 of (leftmost) points from U1 and the number n2 of the (rightmost) points in U2 that comprise the upper hull of the points on U1 and U2. (2) We then move Points[N/2+1..N/2+n2] into Points[n1+1..n1+n2]; so that the points on the common hull are now stored in Points[1,n1+n2]. (3) Finally we return Hullsize=n1+n2 as the result.

Step 1 can be done in $O(\log N)$ time using a binary search procedure described in Preparata-Shamos. For performing step 2, we need to broadcast the values of n1 and n2 to all processors. This takes $O(\log N)$ time since this is an EREW PRAM. Processor i can then move Points[i] if necessary. The total time is $O(\log N)$. Exculding the preprocessing step, this gives us the recurrence:

$$T(N) = T(N/2) + O(\log N)$$

which has the solution $T(N) = O(\log^2 N)$. The preprocessing step requires time $O(\log N)$ using Cole's Mergesort, for a total time of $O(\log^2 N)$.

(comment: Computes the upper hull of the N points given. Returns an integer Hullsize denoting the number of points in the hull as the result. The array Points[1..N] is destructively modified so that Points[1..Hullsize] contains the points on the hull in a left to right order at the point of return.)

```
Algorithm UpperHull(N,Points[1..N])
    In parallel compute
        Hullsize1 = UpperHull(N/2,Points[1..N/2])
        Hullsize2 = UpperHull(N/2,Points[N/2+1..N])
    Hullsize = Merge(Hullsize1, Hullsize2, N, Points[1..N])
    Return Hullsize
End Algorithm UpperHull
```

Figure 1: Algorithm for computing the upper hull

3 Brent's scheduling principle

An algorithm is said to be a *fixed schedule algorithm* if the instruction executed by each processor at each step is known at the start of execution. The notion of *instruction* is somewhat unusual in this definition: we allow conditional instructions of the form "if condition then action", where the condition and the action should each be a primitive operation.

Brent's Scheduling Theorem: Let the number of operations performed by a fixed schedule PRAM algorithm be N, and let T be the total execution time. Then the algorithm can be executed in time $O(T + \frac{N}{P})$ using a P processor PRAM of the same kind as in the original algorithm.
Proof: We prove the theorem for an EREW PRAM. Let N_t denote the number of processors active at step t, $\sum_t N_t = N$. We know which processors of the algorithm are active at step t, and hence we can have the P available processors simulate their execution for step t in time $\lceil N_t/P \rceil$. The total time is therefore $\sum_t O(\left\lceil \frac{N_t}{P} \right\rceil) = O(\sum_t \frac{N_t}{P} + 1) = O(\frac{N}{P} + T)$. The proof for other kinds of PRAMs is only slightly more complicated and left as an exercise. $\square$

Note that the number of processors involved in the original algorithm do not figure in the final time.

Example: The prefix of N elements can be computed in $O(\log N)$ steps using N processors organised as a binary tree. Remember that at any time step only processors in a single level are active, and the total number of operations performed by the algorithm is only $O(N)$. The simple minded PRAM algorithm simulates the tree execution using N processors, and the simulation is clearly fixed schedule, with $O(N)$ operations. However, we can use $P = N/\log N$ processors and apply Brent's theorem. The time is $O(\frac{N}{P} + \log N) = O(\log N)$. We have thus managed to reduce the number of processors by a $\log N$ factor without increasing the time by more than a constant factor.

4 $N/\log N$ processor, $\log^2 N$ time algorithm

Exercise: Show that our N processor convex hull algorithm is fixed schedule and performs $O(N \log N)$ operations, even though it executes for $O(\log^2 N)$ steps.

We then get the result by choosing $P = N/\log N$.

Butterfly Algorithm: We can simulate the PRAM algorithms to run on a P processor Butterfly; but this will require a slowdown of $\Omega(\log P)$, and we would not get linear speedup. You might think that the PRAM simulation stores the data structures of the algorithm in a random manner among the memories in the Butterfly; and typically it takes $\Omega(\log P)$ time for access to any datum, causing the slowdown. However, it seems that even if we carefully decide where to store the data structures of the PRAM algorithm on the Butterfly a slowdown is unavoidable. This is because in the merge step we move a lot of data. To get linear speedup we need a slightly different strategy.

5 A Tree algorithm

We show that a complete binary tree with P leaves can compute the convex hull of N points in time $O(N/P)$, provided $N = \Omega(P \log^3 P)$! The catch is that we require the points to be input at the leaves in sorted order. Suppose that the leaves of the tree are numbered 0 through $P-1$ left to right. Then we require that each leaf processor inputs N/P points, and the points input by leaf i have no smaller x-coordinates than those input by leaf $i+1$. If the input is in sorted order, then it is possible to compute the convex hull in time $O(N)$ on a sequential processor, so our algorithm gives linear speedup.

The algorithm is quite similar to the PRAM algorithm, except that the data structures are modified to reduce data movement. As in the PRAM, the input is read into an array Points[1..N]. The array is distributed amongst the leaf processors, so that leaf processor i holds subarray Points[si+1..si+s], with $s = N/P$. As mentioned above, we are guaranteed that the points are read in sorted order of their x-coordinates.

The algorithm for computing upper hulls is as follows:

Step 1: Each leaf processor i computes the upper hull of the points in Points[si+1..si+s], and stores the result in an array $U(i)$. Each processor i also maintains three additional integers: $Count(i)$ which holds the length of $U(i)$. The other two $Rank(i)$ and $Size(i)$ will be explained below, but they are initially set so that $Rank(i) = 0$, and $Size(i) = Count(i)$. Notice that this step is entirely local to each processor, and since the points are already sorted, takes time $O(s) = O(N/P)$.

Step 2: The P hulls are merged pairwise to produce a common upper hull, in a total of $p = \log P$ substeps: $2.1, 2.2, \ldots, 2.p$. Each substep will be seen to require time $O(\log P \log N)$, for a total time of $O(\log^2 P \log N)$ for the entire step.

The key to mergeing hulls quickly is the representation used to store the hull in a distributed manner. In general, at the beginning of substep 2.i+1, leaf processors $j2^i$ through $(j+1)2^i - 1$ together hold a single hull U_j. In particular, $U_j = U(j2^i)||..||U((j+1)2^i - 1)$, where $||$ denotes concatenation of the sequences. The other data structures maintained by processor x have the following interpretation. $Count(x)$ denotes the length of $U(x)$, $Rank(x)$ denotes the number of points appearing in $U_{\lfloor x/2^i \rfloor}$ before the first point in $U(x)$, and $Size(x)$ denotes the number of points in $U_{\lfloor x/2^i \rfloor}$. The initialization in step 1 is consistent with this interpretation.

We now describe the action of step $2.i+1$, for $i = 0..p-1$.

Step 2.i+1: Let U_j be defined as above, with j even. This step will merge hulls U_j and U_{j+1} to form a single hull U using a binary search like technique as in the PRAM algorithm. The mergeing process is itself performed by the processor Q in level $i+1$ that is the common ancestor of leaf processors $j2^i$ through $(j+2)2^i - 1$ which together hold U_j and U_{j+1}. The binary search requires processor Q to be able to issue requests of the form:

Fetch the kth leftmost point on hull U_j (or U_{j+1}).

This is performed simply by broadcasting the request to the processors in the left or right subtree rooted at Q. Each leaf processor x can determine in constant time if it is holding the requested point using $Rank(x)$ and $Count(x)$, and return the result to Q. The time for a single request is

$O(i) = O(\log P)$. The binary search requires a total of $O(\log s2^i) = O(\log N)$ accesses, together requiring time $O(\log P \log N)$. The binary search enables processor Q to determine the number n1 of (leftmost) points on U_j, and the number n2 of (rightmost) points on U_{j+1} that survive to be included into U. Q then broadcasts n1 and n2 to its subtree, and each processor x updates $U(x)$, $Rank(x)$, $Count(x)$ and $Size(x)$, so as to cooperatively store U. After receiving n1 and n2, each processor in the subtree can perform the update in constant time. Each processor is now ready to enter step 2.i+2.

The total time for all substeps of step 2 is thus $O(\log^2 P \log N)$. Adding the time for step 1, we have the total time to be $O(\log^2 P \log N + N/P)$. If we choose $N = \Omega(P \log^3 P)$, we have total time $O(N/P)$ as claimed.

Notice that we really didnt need a complete binary tree. Any binary tree with height $O(\log P)$ and P leaves will do to get the bounds mentioned above.

6 Butterfly Network

We will show that even if the points are not given in a sorted order, it is possible to compute convex hulls on a P processor Butterfly (processors in all levels) in time $O((N \log N)/P)$. For getting a deterministic algorithm this requires $N = \Omega(P^{1+\epsilon})$ for some constant $\epsilon > 0$. For randomized algorithms we need $N = \Omega(P \log^2 P)$.

The algorithm has two steps: (1) Sort points by x-coordinate, (2) embed a tree of height $O(\log P)$ in the Butterfly and emulate the algorithm given in the previous section.

Step 1 takes time $O((N \log N)/P)$ using the Reif-Valiant Flashsort (randomized), or in the deterministic case using Leighton's columnsort (we need $N = \Omega(P^{1+\epsilon})$ in that case). Step 2 can also be seen take the same time using the analysis of the previous section.

7 Other networks

The idea of the Butterfly algorithm can obviously extended to other hypercubic networks.

If the points are input in an arbitrary (unsorted) order, then linear speedup is acheived on non-hypercubic networks such as the mesh only with very few processors as compared to the number of points. This is because sorting can be reduced to computing convex hulls (see Preparata-Shamos). On a mesh sorting takes time $\theta(N/\sqrt{P})$ because of the bisection width bound. Thus for linear speedup on sorting (and convex hulls) we must have $N/\sqrt{P} = O(N \log N/P)$. This implies that $P = O(\log^2 N)$, i.e. very little parallelism.

Similar arguments can be made for other low bisection width networks.

8 Exercises

Problem 1. Let P denote a set of n points $p_0, p_1, \ldots, p_{n-1}$ in the plane, given in sorted order of their x coordinate. The following is a sketch of an n processor CREW PRAM algorithm for computing their upper hull.

1. $\sqrt{n}$ way recursion: Let $P_i = \{p_{i\sqrt{n}}, \ldots, p_{(i+1)\sqrt{n}-1}\}$. In parallel recursively compute the upper hull of P_i for all i, using $\sqrt{n}$ processors for each recursive call. Let $U_0, U_1, \ldots, U_{\sqrt{n}-1}$ denote the upper hulls computed.

2. Computation of common tangents: Find the common tangents between hull U_i and U_j for all i, j. This is where the concurrent read feature might be important.

3. Mark off the points on each U_i that survive.

4. Compact the points that survive and return them as the joint upper hull.

Show that step 3, determining which points survive can be done in $O(\log n)$ time for all i together. Consider some hull U_i, where $0 < i < \sqrt{n} - 1$. Is it possible that no points on U_i will appear in the upper hull of P? Give an example to justify.