A Simple Parallel Convex Hulls Algorithm for Sorted Points and the Performance Evaluation on the Multicore Processors

Masaya Nakagawa, Duhu Man, Yasuaki Ito, Koji Nakano
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, JAPAN
$\{$*nakagawa, manduhu, yasuaki, nakano*$\}$*@cs.hiroshima-u.ac.jp*

***Abstract*—Finding a vast array of applications, the problem of computing the convex hull of a set of sorted points in the plane is one of the fundamental tasks in pattern recognition, morphology and image processing. The main contribution of this paper is to show a simple parallel algorithm for computing the convex hull of a set of n sorted points in the plane and evaluate the performance on the dual quad-core processors. The experimental results show that, our implementation achieves a speed-up factor of approximately 7 using 8 processors. Since the speed-up factor of more than 8 is not possible, our parallel implementation for computing the convex hull is close to optimal. Also, for 2 or 4 processors, we achieved a super linear speed up.**

***Keywords*-Parallel algorithm; Convex hull; Multicore processor**

I. Introduction

The convex hull for a set P of points in an xy plain is the minimum convex set containing all points in P (Figure 1). Computing the convex hull is one of the most fundamental problems in the area of computational geometry [1]. We assume that the problem of computing the convex hull is a problem to list all convex hull points that constitute the border of the convex hull. The convex hull is partitioned, using the leftmost and the rightmost points, into *the upper hull* and *the lower hull* as illustrated in Figure 1. Clearly, by computing the upper hull and the lower hull and combining them, we can obtain the convex hull. Also, any algorithm to compute the upper hull can compute the lower hull. Thus, in this paper, we will show a parallel algorithm for computing the upper hull.

One of the fundamental heuristics in pattern recognition, morphology, image processing, and robot navigation, involves approximating real-world objects by convex sets. For obvious reasons, one is typically interested in the convex hull of a set P of points, defined as the smallest convex set that contains P [2], [3]. In robotics, for example, the convex hull is central to path planning and collision avoidance tasks [4], [5]. In pattern recognition and image processing the convex hull appears in clustering, and computing similarities between sets [6], [2], [7], [8]. In computational geometry, the convex hull is often a valuable tool in devising efficient algorithms for a number of seemingly unrelated problems [7], [8]. Being central to so many application areas, the convex hull problem has been extensively studied in the literature, both sequentially and in parallel [1], [6], [2], [3], [7], [5], [9], [10], [11].

It is well known that the convex hull of n points in the plane can be computed in $O(n \log n)$ time [12]. Also, if n points are sorted by their x-coordinates in the plane, the convex hull can be computed in $O(n)$ time [13]. Further, theoretically optimal algorithms for computing the convex hull of sorted set of n points have been presented [14]. More specifically, it was shown in [14] that the convex hull can be computed in PRAM (Parallel Random Access Machine) [15]

- $O(\log n)$ time using $\frac{n}{\log n}$ processors on the EREW PRAM, and
- $O(\log \log n)$ time using $\frac{n}{\log \log n}$ processors on the CRCW PRAM.

A parallel algorithm is *asymptotically cost optimal* if the product of computing time t and the number k of processors is asymptotically equal to the computing time of the best sequential algorithm. More specifically, if a parallel algorithm runs in time t using k processors, and the best sequential algorithm runs in T time, then it is asymptotically cost optimal if $tk = O(T)$. Thus, both parallel algorithms above are asymptotically optimal. We also say that a parallel algorithm is cost optimal if the product of the computing time t and the number k of processors is equal to the computing time of the best sequential algorithm. In other words, it is cost optimal if $tk = T$. Since $t = \frac{T}{k}$, the parallel algorithm achieves a speedup factor of k using k processors, it also attains *optimal speedup*. However it is not easy to achieve optimal speedup due to miscellaneous overhead. So, we say that it attains *nearly optimal speedup* if a speedup factor is close to the number of processors.

The first contribution of this paper is to show a simple parallel algorithm for computing the upper hull of n sorted points. Our parallel algorithm runs in $O(\frac{n}{k} + k(\log \frac{n}{k})^2)$ using k processors. Thus, our algorithm is asymptotically optimal whenever $k \le \frac{\sqrt{n}}{\log n}$. From theoretical point of view, our algorithm is asymptotically optimal for smaller ranges of k than the previously published results [14]. However, the

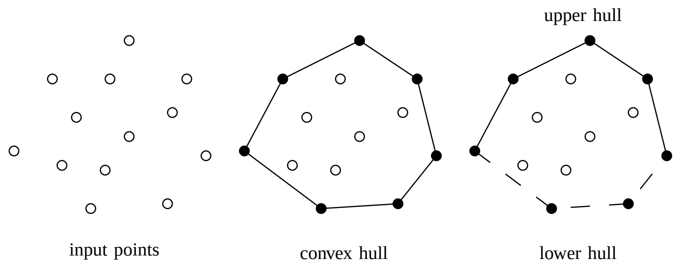

Figure 1. The convex hull, the upper hull, and the lower hull of a set of points

previously published parallel algorithms are too complicated, and the implementation is not easy. From practical point of view, we believe our algorithm is fast.

Further, we analyze the details of the performance. Let $T_u(n)$ be the computing time necessary to compute the upper hull of n points using a single processor. Also, let $T_c(n)$ be the computing time necessary to copy n integers in an array. Clearly, $T_c(n) = O(n)$. Also, from [13], $T_u(n) = O(n)$. We show that our parallel algorithm runs in $T_u(\frac{n}{k}) + T_c(\frac{n}{k}) + O(k(\log \frac{n}{k})^2)$ time using k processors. From the implementation result that we are going to show, $T_u(n) \approx 7 \cdot T_c(\frac{n}{k})$. Thus, our parallel algorithm runs in $\frac{8}{7}T_u(\frac{n}{k}) + O(k(\log \frac{n}{k})^2)$ time. Since at least $T_u(\frac{n}{k})$ time is necessary to compute the upper hull of n points using k processors, our parallel algorithm is close to optimal.

Our second contribution is to implement our convex hull algorithm in multicore processors (or processor cores). The experimental results show that, our convex hull parallel algorithm achieves near optimal speed-up, that is, our parallel algorithm achieves a speed-up factor of approximately 7 using 8 processors. Also, for 2 and 4 processors, we achieves a super linear speed up. We believe that the reason why our parallel algorithm achieves super linear speed up for 2 and 4 processors is the size of L2 cache. The algorithm for a single processor uses 4MB L2 cache. On the other hand, our parallel algorithm for multi-processors uses 8MB L2 cache. Thus, we have achieved a super linear speed up.

Further, we have presented that sorting can be done in efficiently [16]. More specifically, we have shown a parallel sorting with speed-up factor of 6 using 8 processors. Thus, we can obtain a nearly cost optimal convex hull parallel algorithm for non-sorted set of points in multicore processors.

II. A SEQUENTIAL ALGORITHM FOR COMPUTING THE UPPER HULL

Let $P = \{p_0, p_1, \ldots, p_{n-1}\}$ be the set of points in the plane. We assume that points P is sorted for their x-coordinates, that is, $x(p_i) < x(p_{i+1})$ for all i, where $x(p_i)$ denotes the x coordinate of p_i.

Let us review a sequential algorithm for finding the upper hull points [13]. We use a stack to find all upper hull points. Let us define the following operations for the stack S with each element taking an index of a point in the array, that is, an integer value:

$\text{push}(S, i)$	Push the index i of point p_i in stack S.
$\text{pop}(S)$	Pop (or remove) the point index in the top of stack S.
$\text{top}(S)$	Returns the point index of the stack top.
$\text{second}(S)$	Returns the index of the second point of the stack.

Note that the stack can be implemented by an array and a pointer storing the index of the stack top. Thus, these operations can be implemented to be completed in $O(1)$ time.

Using these stack operations, we can find the upper hull of P. The following algorithm computes the upper hull of P. The resulting upper hull points are stored in stack A.

```
$\text{push}(S, 0)$;
$\text{push}(S, 1)$;
**for** $i = 2$ **to** $n$ **do**
    **begin**
        **while** (point $p_{\text{top}(S)}$ is below line $p_{\text{second}(S)}p_i$)
            $\text{pop}(S)$;
        $\text{push}(S, i)$;
    **end**
```

Note that each element of the stack is an integer, which stores the index of a point. A naive implementation uses a stack with x and y coordinates of points. If each coordinate is a 128-bit long double float number, we need to perform stack operations of 256-bit data. To reduce the data movement, we use the stack with integers. From practical point of view, a stack with 32-bit unsigned integers are sufficient if the number of input points is less than $2^{32} = 4$ billion.

Figure 2 illustrates how the upper hull is computed by this algorithm. Suppose four points $DCBA$ constituting the interim upper hull, and we are now in position to add point p_i to it. Note that, these points are stored in the stack. Since point A, which is stored in the top of the stack, is below line Bp_i, point A cannot be an upper hull point and it is removed from the stack by the pop operation. Similarly, point B is below line Cp_i, point C is removed from the stack. Since point C is above line Dp_i, point C is the interim upper hull point, and p_i is pushed into the stack. In this way, new interim upper hull DCp_i is computed. Clearly, when the algorithm terminates, the upper hull is stored in the stack correctly.

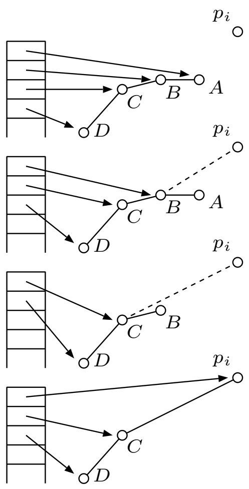

Figure 2. Illustrating the algorithm to compute the upper hull

For each point p_i, the push operation is executed exactly once. Also, by the pop operation, each point is removed at most once. Thus, it is easy to see that the upper hull algorithm above runs in $O(n)$ time.

III. Sequential algorithms for computing the tangents

The main purpose of this section is to review a sequential algorithm for computing the tangent from a point to the upper hull [14]. We then go on to show a sequential algorithm for computing the common tangent of two upper hulls.

Let an upper hull $P = \{p_0, p_1, \ldots, p_{n-1}\}$ with n points and a point q. We assume that points in P are sorted by x-coordinates. Let us pick an arbitrary point p_i in P and draw a line p_iq. Clearly, p_iq is the tangent if both points p_{i-1} and p_{i+1} are below p_iq. Figure 3 shows the tangent of P from q. Clearly, if p_{i-1} is below line p_iq but p_{i+1} is above, then the contact point is in the right-hand side of p_i. Similarly, if p_{i-1} is above line p_iq and p_{i+1} is below, then the contact point is in the left-hand side of p_i. Therefore, the tangent of P from q can be computed by the binary search technique in an obvious way. Since P has n points, the common tangent can be computed in $O(\log n)$ time.

Suppose two upper hulls $P = \{p_0, p_1, \ldots, p_{n-1}\}$ with n points and $Q = \{q_0, q_1, \ldots, q_{m-1}\}$ with m points are given. Our goal is to compute the common tangent of P and Q. Let us pick a point q_j and draw the tangent from q_j to P. We can draw the tangent of P from q_j in $O(\log n)$ time using the binary search. Let p_iq_j be the tangent thus obtained. Clearly, if both q_{j-1} and q_{j+1} are below p_iq_j, then p_iq_j is the common tangent of P and Q. If q_{j-1} is below p_iq_j but

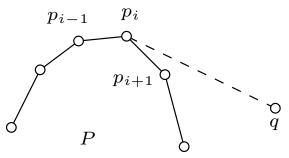

Figure 3. The tangent of P from q.

q_{j+1} is above, then the contact point of Q of the common tangent is in the right-hand size of q_j. Similarly, if q_{j-1} is above p_iq_j and q_{j+1} is below, then the contact point of Q of the common tangent is in the left-hand size of q_j. Hence, the common tangent of P and Q can be computed by the binary search. Since P and Q have n and m points respectively, The common tangent can be computed in $O(\log m \log n)$ time. Thus we have

Lemma 3.1: Suppose that we have two upper hulls P and Q with n and m sorted points. Their common tangent can be computed in $O(\log m \log n)$ time using a single processor.

IV. A parallel algorithm for computing the upper hull

The main purpose of this section is to show a parallel algorithm for computing the upper hull.

Again, let $P = \{p_0, p_1, \ldots, p_{n-1}\}$ be the sorted set of points in the plane. Suppose that we have k processors $\mathrm{PE}(0), \mathrm{PE}(1), \ldots, \mathrm{PE}(k-1)$. The outline of the parallel algorithm is as follows:

- Step 1 Partition the sorted points into k equal sized subsets $P_0, P_1, \ldots,$ and P_{k-1}, and compute the upper hull of each P_i using $\mathrm{PE}(i)$. The upper hull points are stored in k stacks.
- Step 2 Remove all points that are not the upper hull points of P.
- Step 3 Copy all the upper hull points in an array.

In Step 1, P is partitioned into k subsets such that each $P_i = \{p_{i\cdot\frac{n}{p}}, p_{i\cdot\frac{n}{p}+1}, \ldots, p_{(i+1)\cdot\frac{n}{p}-1}\}$. We use the sequential algorithm in Section II to compute the upper hull of each P_i $(0 \le i \le k-1)$ using $\mathrm{PE}(i)$.

Suppose that each P_i $(0 \le i \le k-1)$ has $u(i)$ upper hull points. Further, let $p_{i,0}, p_{i,1}, \ldots, p_{i,u(i)-1}$ denote the upper hull points of P_i. The goal of Step 2 is to find an indexes $l(i)$ and $r(i)$ such that $p_{i,l(i)}, p_{i,l(i)+1}, \ldots, p_{i,r(i)}$ are the upper hull points of P. Note that P_i may not have the upper hull points of P. If this is the case, we assume that $l(i)$ and $r(i)$ takes any values such that $l(i) > r(i)$. In Step 2, we compute the value of each $l(i)$ and $r(i)$ $(0 \le i \le k-1)$ using k processors.

Figure 4 illustrates the parallel algorithm for merging four convex hulls into one. Clearly, the upper hull points of P are $p_{0,0}, p_{0,1}, p_{1,1}, p_{1,2}, p_{1,3}, p_{3,2}, p_{3,3}, p_{3,4}$, and $p_{3,5}$. Hence, $l(0) = 0$, $r(0) = 1$, $l(1) = 1$, $r(1) = 3$, $l(3) = 2$, and

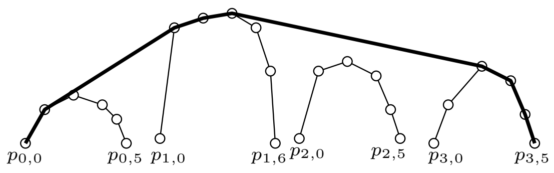

Figure 4. Illustrating the merging of the upper hulls

$r(3) = 5$ hold and $l(4)$ and $r(4)$ can be any value satisfying $l(4) > r(4)$.

Next, we will show how the values of $l(i)$ and $r(i)$ are computed. For two upper hulls P_i and P_j, let $p_{i,c(i,j)}p_{j,c(j,i)}$ denote the common tangent of these upper hulls. Figure 5 illustrates the common tangent $p_{i,c(i,j)}p_{j,c(i,j)}$ of two convex hulls P_i and P_j. Clearly, points of $p_{i,c(i,j)+1}$, $p_{i,c(i,j)+2}$, $\ldots$, $p_{i,c(i,j)-1}$ and $p_{j,0}$, $p_{j,1}$, $\ldots$, $p_{i,c(j,i)-1}$ are below the common tangent, and they cannot be the upper hull points of P.

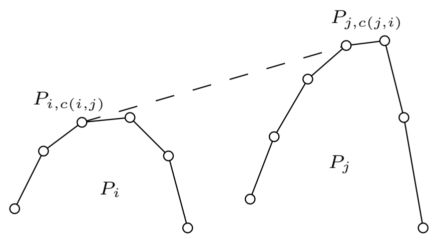

Figure 5. The common tangent $p_{i,r(i,j)}p_{j,l(j,i)}$ of two convex hulls P_i and P_j

From this observation, we can compute the values of $l(i)$ and $r(i)$ as follows:

$$\begin{aligned} l(i) &= \max\{c(i,j) \mid j < i\} \\ r(i) &= \min\{c(i,j) \mid i < j\} \end{aligned}$$

It should be clear that, if $l(i) \leq r(i)$ then $p_{i,l(i)}$, $p_{i,l(i)+1}$, $\ldots$, $p_{i,r(i)}$ are the upper hull points of P, and the other points in P_i are not the upper hull points. If $l(i) > r(i)$ then no point in P_i is the upper hull point of P. Therefore, once we have the values of $c(i,j)$ for all pairs of i and j, we can compute the values of $l(i)$ and $r(i)$ and obtain all the upper hull points.

For a pair of convex hulls P_i and P_j the values of $c_(i,j)$ can be computed in $O(\log u(i) \log u(j))$ using a single processor. Since $u(i) \leq \frac{n}{k}$, the computing time is $O((\log \frac{n}{k})^2)$. Since we have k processors, all the pairs $c_{i,j}$ can be computed in $O(k(\log \frac{n}{k})^2)$ time.

Finally, in Step 3, we move the indexes of all upper hull points in k stacks into an array. Let $n(i) = r(i) - l(i) + 1$ be the number of upper hull points in the i-th stack. We assume that $n(i) = 0$ if $r(i) > l(i)$. To copy all the upper hull points in parallel, we compute the prefix sums of $n(0), n(1), \ldots, n(k-1)$. This can be done in $O(k)$ time using a single processor. After that, PE(i) copies $n(i)$ indexes in its stack to array with positions $n(0)+n(1)+\cdots+n(i-1)$ to $n(0) + n(1) + \cdots + n(i) - 1$. This can be done in $O(n(i)) \leq O(\frac{n}{k})$ time.

Finally, we have

Theorem 4.1: The upper hull of n sorted points can be computed in $O(\frac{n}{k} + k(\log \frac{n}{k})^2)$ using k processors.

Since the upper hull can be computed in $O(n)$ time using a single processor. Thus, our parallel upper hull algorithm is asymptotically optimal if $O(\frac{n}{k} + k(\log \frac{n}{k})^2) = O(n)$ holds. It follows that our algorithm is asymptotically optimal whenever $k \leq \frac{\sqrt{n}}{\log n}$.

Let us discuss the details of the computing time of our parallel upper hull algorithm. We assume that, every processor core can access data in the shared memory independently without overhead of simultaneous access. Let $T_u(n)$ be the computing time necessary to compute the upper hull of n points using a single processor. Also, let $T_c(n)$ be the computing time necessary to copy n integers in an array. Clearly, $T_u(n) = T_c(n) = O(n)$.

In Step 1, the upper hull of $\frac{n}{k}$ points are computed independently. Thus, Step 1 takes $T_u(\frac{n}{k})$ time. Step 2 runs in $O(k(\log \frac{n}{k})^2)$ is time, which is much smaller than $T_u(\frac{n}{k})$. Finally, Step 3 runs in $T_c(\frac{n}{k})$ time. However, Step 3 can be much smaller. Let U be the maximum number of local upper hull points over all subsets. Clearly, $U \leq \frac{n}{k}$ and in most practical cases, $U \ll \frac{n}{k}$ holds. Then, Step 3 takes $T_c(U) \ll T(\frac{n}{k})$ time. Consequently, the computing time of our parallel algorithm runs in $T_u(\frac{n}{k}) + O(k(\log \frac{n}{k})^2) + T_c(U)$ time. As we are going to show later in Section V $T_u(n) \approx 7 \cdot T_c(n)$ holds. Also, since $U \ll \frac{n}{k}$ holds for practical input, $T_u(\frac{n}{k})$ is dominant in the computing time. Further, $T_u(\frac{n}{k}) \approx \frac{T(n)}{k}$ holds, we can archive a optimal speed up factor of k using k processors. Note that, this discussion is correct under the assumption that each processor core can access arrays in a shared memory without overhead. If we have too many processor cores connected to the shared memory, the overhead may be dominant and the optimal speed up may not be possible. As we are going to show the next section, near optima speed up is possible for $k \leq 8$.

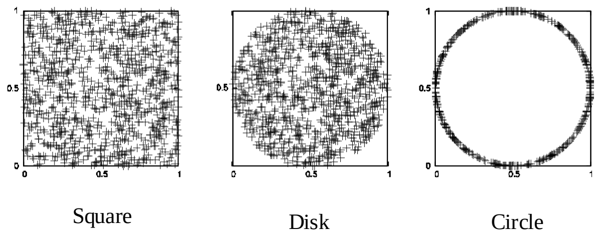

Figure 6. Randomly generated sets of points

V. Experimental results

We have implemented and evaluated the performance of our parallel algorithm in a Linux server (CentOS 5.1) with two quad-core processors (Intel Xeon X5355 2.66GHz [17]), that is, we have used eight processor cores. The software has been implemented in C language with OpenMP 2.0 and compiled by gcc 4.1.2 with -O2 option. The OpenMP (Open Multi-Processing) is an application programming interface that supports shared memory environment [18]. It consists of a set of compiler directives and library routines. By using OpenMP, it is relatively easy to create parallel applications in FORTRAN, C, and C++.

To show the experimental results, we use three randomly generated sets of points

- **Square** The coordinates of each point are generated from $[0, 1)$ uniformly at random.
- **Disk** The coordinates of each point are generated from $[0, 1)$ uniformly at random. If they are not in the circle with radius 0.5 and center $(0.5, 0.5)$, it is discarded and a new point is generated. The generation of coordinates are repeated until it is in the circle.
- **Circle** The angle θ is generated from $[0, 2\pi)$ uniformly at random. The coordinates of the point is $(\cos\theta, \sin\theta)$. Every point is either the upper hull or lower hull points.

Figure 6 shows the outline of these three randomly generated sets of points.

Tables I and II show the performance of our implementation when the upper hulls are computed for the above three randomly generated sets of points, for 10M points and 100M points, respectively. Recall that they are sorted by their x-coordinates in the plane. The performance evaluation has been carried out for various numbers p of processor cores. In the tables, Step 1, Step 2 and Step 3 correspond to the steps of the algorithm shown in Section 4. Note that, if $k = 1$, then the implementation performs only Step 1 for the whole input data. However, for reference, the tables include the computing time of Step 3 for $k = 1$. For computing the 100M points using a single processor, Step 1 and 3 runs in 3.129690s and 0.224130s. It follows that, $T_u(100M) = 3.129690$ and $T_c(50M) = 0.224130$. Thus, $T_u(n) \approx 7 \cdot T_c(n)$ holds and the computing the upper hull is approximately 7 times slower than that for simple copy operation.

The tables show that Step 1 is dominant in terms of the computing time. Also, in Step 1, when k is large, the execution time is short because each processor computes the upper hull of n/k points which are simply divided using k processor cores. According to the tables, the number of upper hull points in the set of Square and Disk, is quite small compared with n. Therefore, in Steps 2 and 3, each computing time is quite short compared with Step 1. In contrast, the number of upper hull points in the set of Circle is approximately $n/2$ because every point is either the upper hull or lower hull point. Thus, Steps 2 and 3 for Circle take a lot of time compared with that for Square and Disk. However, Circle is a special case since every point in Circle will be convex hull point. The computing time for a set of points which are used in general may be close to that for Square and Disk. Thus, we can say that the circle is the worst case for computing the convex hull. Although Circle is the worst, Step 1, which computes the local upper hulls independently, is still dominant. For the speed up factor, each value is close to k. For example, for the square and disk input, a speed up factor of more than 6 is achieved for 8 processor cores. Since the speed up factor cannot be more than k if we use k cores, our algorithm is close to optimal.

Further, quite surprisingly, a super linear speed up is achieved for 2 and 4 processor cores. For example, a speed up factor of 4.27 is achieved for 100M points. The reason why a super linear speed up is the architecture of two quad-core processors [17]. Each processor has 4MB L2 cache. Thus, the algorithm for a single processor runs on 4MB L2 cache. On the other hand, the parallel algorithm for two or more processors runs on 8MB L2 cache. This may be the reason why the parallel algorithm for 2 and 4 processors achieves a super linear speed up. However, due to the limitation of memory bandwidth, a speedup factor for 8 processors is saturated.

VI. Concluding remarks

The main contribution of this work was to present a parallel algorithm for computing the convex hull of a set of n sorted points in the plane and implement it on the dual quad-core processors.

We have evaluated our algorithm in a Linux server with two Intel quad-core processors (Intel Xeon X5355 2.66GHz). The results have shown that our parallel implementation is 7 times faster than sequential implementation. Since the speed up factor cannot be more than 8 if we use 8 cores, our algorithm is close to optimal. Also, for 2 or 4 processors, we have achieved a super linear speed up.

References

[1] S. G. Akl and K. A. Lyons, *Parallel Computational Geometry*. Prentice-Hall, 1993.

Table I
PERFORMANCE OF OUR PARALLEL ALGORITHM FOR 10M POINTS

	The number of processors	Computing time				Speed up	The number of upper hull points
		Step 1[s]	Step 2[s]	Step 3[s]	Total[s]		
Square	1	0.378099	–	(0.000002)	0.378099	–	22
	2	0.185269	0.000015	0.000005	0.185289	2.04	
	4	0.088754	0.000026	0.000005	0.088785	4.26	
	8	0.055187	0.000035	0.000006	0.055228	6.85	
Disk	1	0.387516	–	(0.000002)	0.380516	–	362
	2	0.189332	0.000024	0.000007	0.189363	2.01	
	4	0.088574	0.000039	0.000006	0.088619	4.29	
	8	0.055322	0.000055	0.000007	0.055384	6.87	
Circle	1	0.314209	–	(0.021787)	0.314209	–	4999600
	2	0.143707	0.000027	0.013560	0.157294	2.00	
	4	0.070297	0.000059	0.009024	0.079380	3.96	
	8	0.059471	0.000117	0.008248	0.067836	4.63	

Table II
PERFORMANCE OF OUR PARALLEL ALGORITHM FOR 100M POINTS

	The number of processors	Computing time				Speed up	The number of upper hull points
		Step 1[s]	Step 2[s]	Step 3[s]	Total[s]		
Square	1	3.791957	–	(0.000002)	3.791957	–	26
	2	1.850652	0.000015	0.000005	1.850672	2.05	
	4	0.888049	0.000027	0.000007	0.888083	4.27	
	8	0.544188	0.000042	0.000005	0.544235	6.97	
Disk	1	3.790487	–	(0.000002)	3.790487	–	791
	2	1.852445	0.000029	0.000006	1.852480	2.05	
	4	0.883241	0.000048	0.000006	0.883295	4.29	
	8	0.543315	0.000072	0.000007	0.543394	6.97	
Circle	1	3.129690	–	(0.224130)	3.129690	–	50000480
	2	1.449058	0.000033	0.136745	1.585836	1.97	
	4	0.698485	0.000072	0.094026	0.792583	3.95	
	8	0.590330	0.000137	0.088430	0.678897	4.61	

[2] R. O. Duda and P. E. Hart, *Pattern classification and scene analysis*. Wiley and Sons, 1973.

[3] G. T. T. Ed., *Computational Geometry*. Elsevier Science Publishers, 1985.

[4] J.-P. Laumond, "Obstacle growing in a non-polygonal world," *Information Processing Letters*, pp. 41–50, 1987.

[5] T. Lozano-Perez, "Spatial planning: a configurational space approach," *IEEE Transactions on Computers*, pp. 108–119, 1983.

[6] D. H. Ballard and C. M. Brown, *Computer Vision*. Prentice-Hall, 1982.

[7] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

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

[9] R. Miller and Q. F. Stout, "Efficient parallel convex hull algorithms," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1605–1618, 1988.

[10] T. Hayashi, K. Nakano, and S. Olariu, "An $o((loglogn)^2)$ time convex hull algorithm on reconfigurable meshes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 12, pp. 1167–1179, 1998.

[11] K. Nakano, "Computing the convex hull of a sorted set of points on a reconfigurable mesh," *Parallel Algorithms and Applications*, pp. 243–250, 1996.

[12] R. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Information Processing Letters*, pp. 132–133, 1972.

[13] O. Berkman, B. Schieber, and U. Vishkin, "Optimal double logarithmic parallel algorithms based on finding all nearest smaller values," *Journal of Algorithms*, vol. 14, no. 3, 1993.

[14] W. Chen, K. Nakano, T. Masuzawa, and N. Tokura, "Optimal parallel algorithms for computing convex hulls," *IEICE Transactions*, vol. J75-D1, no. 9, pp. 809–820, 1992.

[15] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[16] K. Nishihata, D. Man, Y. Ito, and K. Nakano, "Parallel sampling sorting on the multicore procesors," in *Proc. of the International Conference on Applications and Principles of Informatin Science(APIS)*, 2009, pp. 233–236.

[17] I. Corporation, "Intel xeon processor 5000 sequence." [Online]. Available: http://www.intel.com/products/processor/xeon5000/

[18] D. J. Kuck, B. Chapman, G. Jost, and R. V. der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.