COMP 4905 HONOURS PROJECT

Solving 2D Convex Hull With Parallel Algorithms

In partial fulfillment of the requirement for the degree

BACHELOR of COMPUTER SCIENCE

by

HSIN-YI CHIANG

Under the supervision of

Dr. FRANK DEHNE
School of Computer Science

April 2010

Acknowledgments

Special thanks to Frank Dehne, Hamidreza Zaboli, and Nikola Sabo in the writing of this report and their assistance in all aspects of this project.

Abstract

In this report, we implemented a program using divide-and-conquer algorithms to solve *2D convex hull* problems in parallel with either CPUs or GPUs. By measuring the running time of the program and comparing it with sequential programs which solves the same problem, we have shown that solving a problem with multiple processors will yield a better performance in terms of running time.

Table of Contents

List of Figures

List of Tables

Introduction

Before multiple-processor systems were developed, people were trying to speed up the performance of the algorithms by increasing processor speed; however, faster processing units will require a faster memory access speed. In addition, higher CPU speed will increase the heat produced which in effect creates other system problems. In order to reduce the overall heat generated by the CPU, systems with multiple processors were created. With multiple processors, systems were able to achieve higher performance with slower processing speed. In addition to that, systems designed with multiple processors have better scalability.

When looking at the issue of processing speed of a single core system, the best remedy to use instead of creating a faster CPU, is to increase the number of processors in that systems. In the end, the same result, which is the overall increase in performance, is achieved. In addition, creating a faster CPU will subsequently create more heat in the system while having multiple CPUs will effect generate less heat. With multiple-core architecture, it is possible for programs to run multiple threads at the same time to gain higher performance . On the other hand, although multiple threads running on multiple processors can increase the performance of the algorithm, there are overheads when creating, managing, and destroying threads which can affect the overall operation of the programs.

Currently, in computer systems, not only central processing units but also graphics processing units (GPUs) are designed to carry multiple cores in order to handle computing environments which require intensive processing such as modern operating systems or 3D games. The difference between CPUs and GPUs are usually the number of cores in the units and the architecture of each core. GPUs are designed specifically for processing graphics while CPUs are designed to handle multiple requests

from the whole system. As the result, GPUs are given more cores with shared instruction decoders and CPUs are equipped with less cores each having its own instruction decoder. With independent instruction decoders, each CPU core is able to execute different set of instructions simultaneously while GPUs cores are divided into groups upon which cores in the same group can only execute the same set of instructions. In addition, since there are more cores in a GPU than a CPU, GPU is generally running at a lower speed because it is designed specifically for a single purpose, graphics.

In order to take advantage of multiple processors, in most parallel algorithms, the set of problems are divided into smaller sub-problems and fed to multiple processors to perform the same calculations. This is usually done in systems with multiple CPUs; however, as GPU functionality has been improved, a new computing strategy was proposed: general-purpose computing on graphics processing units (GPGPU) which takes the advantage of large amount of cores in a single graphics processing unit and its independent high speed graphic memory. With so many cores, new algorithms are being developed to solve different problems which leads to new fields of study.

In this report, we have chosen the *two dimensional convex hull* problem and tried to solve it in both parallel and in sequential using either graphics processing units or central processing units. Finally, we compare the performance by measuring the running time of different algorithms under different computing environment.

Technology used to compute *2D Convex Hull* problem

Compute Unified Device Architecture(CUDA)

Developed by NVidia and released in 2006, CUDA technology is a software-hardware computing architecture based on the extension of the C programming language. CUDA provides access to GPU

instructions and video memory control to perform parallel computations. With CUDA, programmers are able to use commercial NVidia video cards to execute custom codes. In addition, CUDA also provides a set of APIs including time measurement, thread synchronization and memory copying between device and host to help programmers measure the performance of the algorithm and prevent possible errors in a concurrent environment. One of the latest feature provided by CUDA is the hardware support of double precision. Although it decreases the performance of the execution ability, double precision is important when solving geometric problems[1].

Cilk

Cilk is an extension to C++ programming language designed for multithreaded parallel programming based on ANSI C. With Cilk, programmers are able to create multiple thread which are executed in parallel with multiple-processor systems. In addition, Cilk also offers a set of tools to help the users to verify the correctness of parallel programs and prevent possible errors such as race conditions. Cilk is also designed with built-in scalability such that Cilk codes are not bounded by the number of cores within the system[2].

The problem: Convex Hull

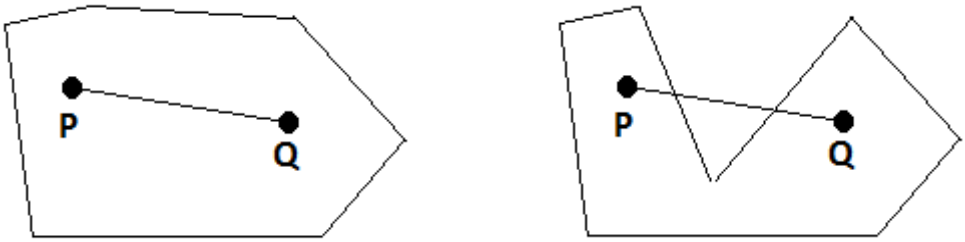
The convex hull of a set of points in a plane is defined as the smallest subset (convex set) containing all points in the set (fig. 1) [3].

set of points *convex hull of the set*

figure 1. Convex Hull of a point set in 2D

There are many properties of a convex set S. The following is a list of important properties[4]:

Property 1. A set ***S*** is the convex if the whole line segment $\overline{PQ}$ of any two arbitrary points *P* and *Q* inside of S is also in S (fig. 2).



convex *non-convex*

figure 2. the line formed by any two points within the hull is also in the hull

Property 2. A set S is the convex if it is exactly equal to the intersection of all the half planes containing it.

Property 3. The convex set, S, of a finite point set *P* has the smallest area and the smallest perimeter of all polygons containing the set *P*.

There are many different applications for convex hulls such as collision avoidance, hidden object determination, path finding, visual pattern matching or Ray Tracing. [5]

Convex Hull Algorithms

Many different convex hull algorithms were proposed and each algorithm has different complexities(table 1).

Algorithm	**Complexity**	**Proposed By**
Brute Force	$O(n^4)$	[Anon, the dark ages]
Gift Wrapping	$O(nh)$	[Chand & Kapur, 1970]
Graham Scan	$O(n \log n)$	[Graham, 1972]
Jarvis March	$O(nh)$	[Jarvis, 1973]
Quick Hull	$O(nh)$	[Eddy, 1977] [Bykat, 1978]
Divide-and-Conquer	$O(n \log n)$	[Preparata & Hong, 1977]
Monotone Chain	$O(n \log n)$	[Andrew, 1979]
Incremental	$O(n \log n)$	[Kallay, 1984]
Marriage-before-Conquest	$O(n \log n)$	[Kirkpatrick & Seidel, 1986]
n = number of points, h = number of vertices in the output hull		

Table 1. list of well-known 2d convex hull algorithms

In order to keep the performance of algorithms independent from the distribution (h) of the output points, we use only $O(n \log n)$ algorithms but not the ones with $O(n*h)$. In this experiment, three different algorithms are selected which are: *Monotone Chaining, Graham Scan* and *Divide-and-Conquer*. All this algorithms are based on one important concept: “left turn” , “right turn”, and “no turn”of the given 3 points. In the following, we will explain the concept of turning before we further explaining the algorithms.

Turning of three points

Given three distinct points A , B and O we can define the following:

Left turn: We call the line formed by connecting A , B and O a left turn if and only if $\overrightarrow{abo}$ is traversing in the counterclockwise direction(fig. 3).

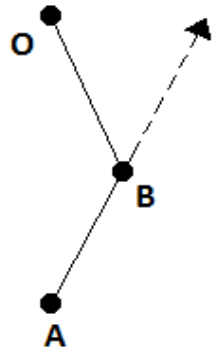


figure 3. a "left turn" formed by $\overrightarrow{abo}$

Right turn: We call the line formed by connecting A , B and O a right turn if and only if $\overrightarrow{abo}$ is traversing in the clockwise direction(fig. 4).

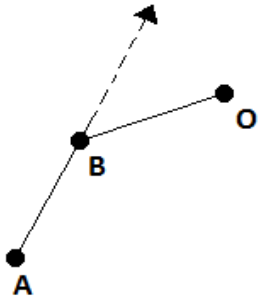


figure 4. a "right turn" formed by $\overrightarrow{abo}$

No turn: We call the line formed by connecting A , B and O "no turn" if and only if $\overrightarrow{abo}$ is traversing in a straight line(fig. 5).

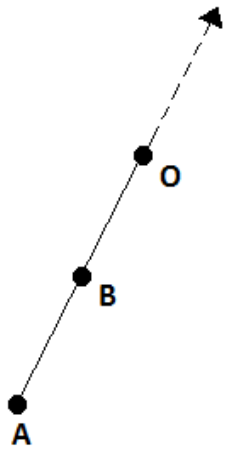


figure 5. a "no turn" formed by $\overrightarrow{abo}$

To determine the direction of the turn, we use the following function:

```
//if the function returns positive values, then it is a "left turn"
//if the function returns 0, then it is a "no turn"
//if the function returns negative values, then it is a "right turn"
function leftTurn(a, b, o):
          return (b.x – a.x)*(o.y - a.y)-(b.y - a.y)*(o.x - a.x)
```

Now, with the concept of turning in mind, we can now start explaining the algorithms used in this experiment.

Graham Scan

Proposed by Ronald Graham in 1972, the Graham scan algorithm is often referred to as the very first "computational geometry" algorithm. The algorithm computes the convex hull of a set of points S by scanning through the array which is sorted according to the polar angle relative to the *pivot point* only once.

The algorithm starts by scanning through the array containing all points to find the point with maximum *y* value, which we will call the *pivot point* and move that point to the beginning of the array.

With maximum *y* value, the point is guaranteed to be in the convex hull which we will use that as the starting point of our calculation. The next step is that the algorithm will start calculating the polar angle of each point in the array relative to the pivot point and sort the array by the polar angle in an increasing order where the pivot point has an angle of 0 degree.

Finally, the algorithm will start computing the convex hull first by initializing a stack and then by performing the following operations starting from the beginning of the array:

- if the size of the stack is less than 2, then the algorithm will push the point onto the stack and move on to the next point in the array
- if `leftTurn(StackTop, stackTop-1, current)` returns a "right turn" then we pop one element off the stack and check again
- otherwise we push the point onto the stack and move on to next point

once the whole array has been scanned, the points that are remaining in the stack will be the convex hull of the point set. In the following diagram, we demonstrate the trace of scanning of the algorithm while computing the convex hull (fig. 6).

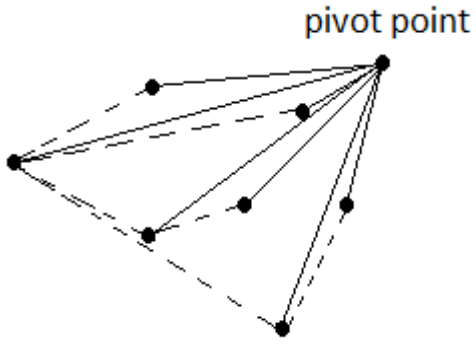


figure 6. forming convex hull with Graham scan algorithm

The pseudo-code for Graham scan:

```
Input: a list P of points in the plane.

Scan the list P for the maximum y-coordinate (in case of a tie, sort
by x-coordinate) and shift it to the front of the list (pivot point)

Calculate the polar angle with the pivot point for all points in the
list

Sort the points according to the value of polar angle

Initialize H as an empty stack.
The stacks will hold the vertices of the convex hull

for i := 1 to n do
    while H contains at least two points and the sequence of last
        two points in H and P[i] does not make a "left turn",
        pop the last point from H
    push P[i] to L

the vertices in the stack H gives the convex hull of P.
```

Andrew's Monotone Chain Algorithm

Published in 1979 by A.M. Andrew [6], the monotone chain algorithm computes the convex hull of a set of points S by forming its upper hull and a lower hull (fig. 7).

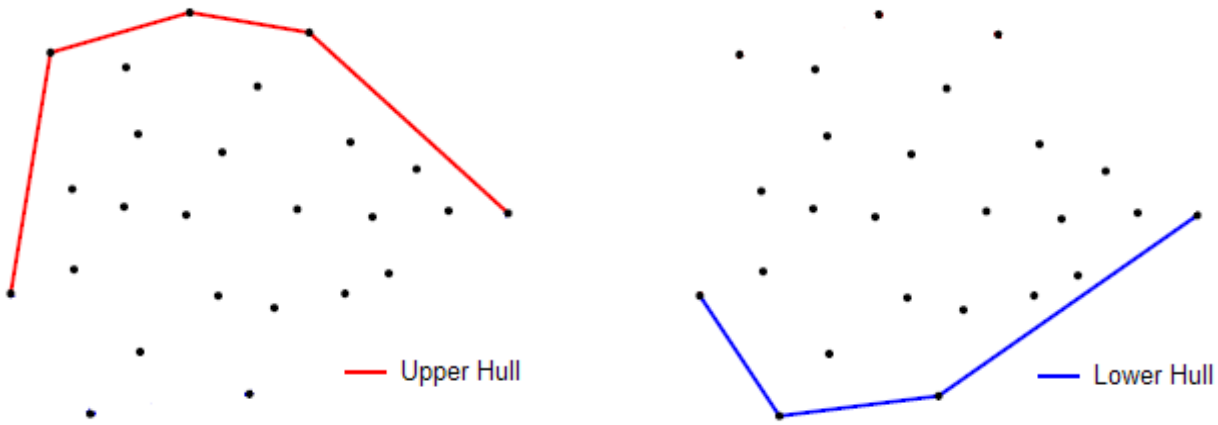


figure 7. the upper and lower hull of a point set

The algorithm first sorts the point set $S = \{P_0, P_1, \ldots P_{n-1}\}$ by increasing order of *x* coordinate. In the case when two points shares the same *x* value, the algorithm sorts them by their *y* value. The result is stored in an array. Once the points are sorted, the algorithm then initialize two stacks which will be used to store the upper hull and the lower hull. Starting from the beginning (head) of the array towards to the end (tail) of the array, the algorithm does the following things:

- if the size of the stack is less than 2, then the algorithm will push the point onto the stack and move on to next point
- if `leftTurn(StackTop, stackTop-1, current)` returns a "right turn" then we pop one element off the stack and check again
- otherwise we push the point onto the stack and move on to next point

Once the sorted array has been scanned once, we will use the second stack and repeat the same steps described above to calculate the lower hull; however, this time the algorithm scans the array from the end (tail) of the array towards to the beginning (head) of the array. With both upper and lower hulls of

monotone chains of points computed, the algorithm can join both hull chains together and return the final result.

The following is the pseudo-code of Andrew's Monotone Chain Algorithm:

```
Input: a list P of points in the plane.

Sort the points of P by x-coordinate (in case of a tie, sort by y-
coordinate).
Remove points, which have same coordinates.

Initialize U and L as empty stacks.
The stacks will hold the vertices of upper and lower hulls
respectively.

for i := 1 to n do
    while L contains at least two points and the sequence of last
        two points in L and P[i] does not make a "left turn",
        pop the last point from L
    push P[i] to L

for i := n down to 1 do
    while U contains at least two points and the sequence of last
        two points in U and P[i] does not make a "right turn",
        pop the last point from U
    push P[i] to U

Remove the first point from U and the last point from L
Concatenation of L and U gives the convex hull of P.
```

Divide-and-Conquer algorithm

Proposed in 1977 by Franco Preparata and S.J. Hung [7], the Divide-and-Conquer algorithm is a parallel algorithm designed to merge existing hulls by searching for its upper tangent and lower tangent lines. A tangent line between two convex hulls is defined as :

definition 1. If $\overline{ab}$ is the upper tangent line between two convex hulls A and B such that A is located at left of B, then for any other points b' in B, $\overrightarrow{abb'}$ forms a right turn and for any

other points a' in A, $\overrightarrow{baa'}$ forms a left turn (fig. 8).

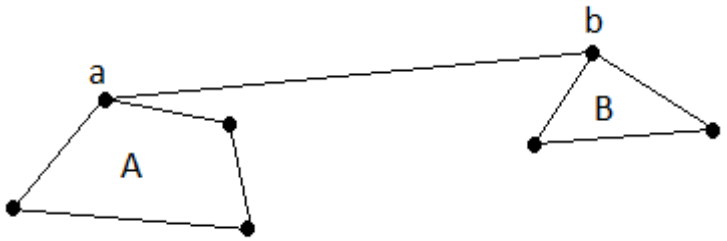


figure 8. the upper tangent of hulls A and B

definition 2. If $\overline{ab}$ is the lower tangent line between two convex hulls A and B such that A is located at left of B, then for any other points b' in B, $\overrightarrow{abb'}$ forms a left turn and for any other points a' in A, $\overrightarrow{baa'}$ forms a right turn (fig. 9).

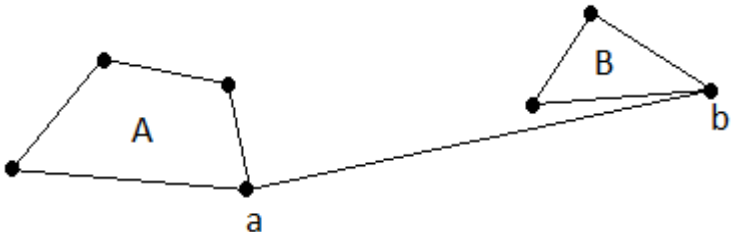


figure 9. the lower tangent of hulls A and B

definition 3. If $\overline{ab}$ is not the tangent line between between two convex hulls A and B, then there exists at least two points b' and b'' in B such that $\overrightarrow{abb'}$ forms a left turn and $\overrightarrow{abb''}$ forms a right turn (fig. 10).

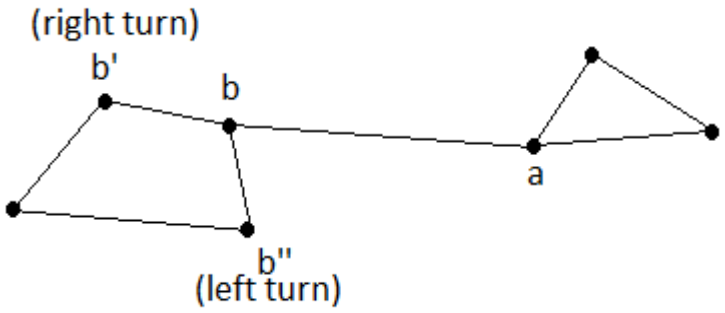


figure 10. a line which is not a tangent

The first two definitions of a tangent line is pretty intuitive since the idea is similar to a tangent line for a circle. However, the third definition requires some more explanations. The basic idea behind the third definition is this: if a line is not a tangent of two convex hulls, then then the line must cut through one of the hull which separates the hull into upper and lower halves resulting resulting the difference in turning directions.

Based on the definitions of tangent lines, the divide-and-conquer algorithm can merge two convex hulls by calculating the tangent lines. Assuming that two convex hulls P and Q, P is left to the Q, are stored in two different cyclic linked lists A and B such that the hulls are in a counter clockwise direction. The algorithm starts by scanning the lists A and B and looks for the right most point in A, a, and the left most point in B, b. To find the upper tangent of the hulls, the algorithm checks for the directions of turning, $\overrightarrow{abb}_{prev}$ and $\overrightarrow{abb}_{next}$. Based on *definition 3*, if both sets of points do not make right turns, then set $b = b_{prev}$. We do the same check for $\overrightarrow{baa}_{prev}$ and $\overrightarrow{baa}_{next}$ and look for left turns, if not, set $a = a_{next}$. The algorithms repeat this checking process until both sides satisfy the conditions, then the upper tangent is found. For the lower tangent, the same process repeats except the directions are mirrored. First we check for the directions of

turning, $\overrightarrow{abb_{prev}}$ and $\overrightarrow{abb_{next}}$. If both sets of points do not make left turns, then set $b = b_{next}$. The similar check is performed for the other convex hulls, $\overrightarrow{baa_{prev}}$ and $\overrightarrow{baa_{next}}$. If both sets of points do not make right turns, then set $a = a_{prev}$. The check repeats until both sides satisfied the conditions. With both upper and lower tangents computed, the two convex hulls can be combined into a single larger hull (fig. 11).

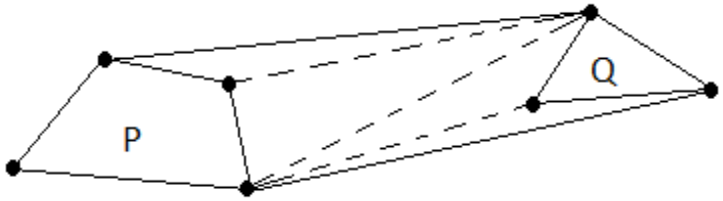


figure 11. two hulls P and Q are merged by tangent lines

The pseudo-code for divide-and-conquer algorithm:

```
Input: a list P of points in the plane.

Sort the points of P by x-coordinate (in case of a tie, sort by y-
coordinate).
Remove points, which have same coordinates.

for i := 1 to n do
    for every integral of points, form a convex hull. The number of
    of convex hulls formed are stored in the variable H

for i := 1 to log(H) do
    merge every odd convex hulls with even ones and remove all
    useless points

the points remain in the list is the convex hull of P
```

Time complexity of the algorithms

In this section, we will take the algorithms described previously and analyze their time complexities. In

this report, we will not discuss the total work done by each algorithm since our main purpose is to compare the time difference between parallel and sequential algorithms.

Graham scan algorithm

Based on the algorithm, the complexity can be separated into four different parts:

- Scan the array for pivot point: $O(n)$
  - To scan the list of size n for the point with the largest y-coordinate value will take $O(n)$ time since the algorithm will have to scan from the beginning to the end of the list exactly once.
- Calculate the polar angles for all points based on the pivot point: $O(n)$
  - To calculate the polar angles for all points will take $O(n)$ time since the algorithm will scan through the whole list once.
- Sorting: $O(n*\log_2(n))$
  - As we know, the optimal comparison based sorting algorithm will take $O(n*\log_2(n))$ to sort. In this experiment, we choose *Merge Sort* as our sorting algorithm to sort the list based on the polar angles of the points.
- Compute Convex Hull: $O(n)$
  - With the sorted list, the algorithm scans through the array once and takes away the points which is not in the hull.

As described above, we calculate that the Graham scan algorithm for computing convex hull has a time

complexity of

$$O(n)+O(n)+O(nlog_2(n))+O(n)=O(3n+n*\log_2(n))=O(n*\log_2(n))$$

Andrew's monotone chaining algorithm

Andrew's monotone chaining algorithm consists of three parts:

- Sorting: $O(n*\log_2(n))$
  - Similar to the previous algorithm, the optimal comparison based sorting algorithm can not exceed $O(n*\log_2(n))$ time. In this case, *Merge Sort* is selected as the sorting method.
- Compute convex hull: $O(2n)$
  - The algorithm builds the convex hull by combining the upper hull and lower hull. To compute the upper hull, it takes $O(n)$ to scan the list from left to right and another $O(n)$ to scan the list from right to left to build the lower hull.
- To merge upper hull and the lower hull: $O(1)$
  - To combine the upper and the lower hull, it takes a constant time to merge by appending one list to another.

When adding all parts together, we will get:

$$O(nlog(n))+O(n)+O(n)+O(1)=O(2n+n*\log(n))=O(n*\log_2(n))$$

although this algorithm has the same complexity as the Graham scan, monotone chaining algorithm has less hidden costs than the Graham scan algorithm($O(2n)$ instead of $O(3n)$).

Divide-and-conquer algorithm

This algorithm is designed to merge the existing convex hulls. Therefore it is required for the programmer to utilize an alternative way to construct convex halls as the base case. In the following, the number of cores in the system is denoted as p .

The first one is with fixed size base-case-hulls:

- Sort the array in parallel: $O(\frac{n}{p}\log_2(\frac{n}{p}))$
    - With multiple cores, the work is evenly distributed to every core and merge the results together. As the result, each core will take $O(\frac{n}{p}\log_2(\frac{n}{p}))$ to process as stated in [8].
- Forming base convex hulls: $O(\frac{n}{p}\log_2(\frac{n}{p}))$
    - As list in *table 1*, the fastest known convex hull algorithm in sequential runs in $O(n*\log_2(n))$ time. Since we have p cores, the running time for forming base-case-hulls can be done in $O(\frac{n}{p}\log_2(\frac{n}{p}))$ time. In our implementation, both Graham scan and Andrew's monotone chaining algorithms are used for this section of the code.
- Merging the convex hulls: $O(\frac{n}{p}*\log_2(\frac{n}{number\,of\ hulls}))$
    - To merge the existing convex hulls into 1 convex hull, $\log_2(\frac{n}{number\,of\ hulls})$ stages are required. At each stage, all n points are scanned. With p cores, each core will

have to scan only at most $\frac{n}{p}$ points. As the result,

$O(\frac{n}{p}) * O(\log_2(\frac{n}{number\ of\ hulls})) = O(\frac{n}{p} * \log_2(\frac{n}{number\ of\ hulls}))$ time is required to complete the task.

with all different components, the divide-and-conquer algorithm has an overall time complexity of

$$O(\frac{n}{p}\log_2(\frac{n}{p})) + O(\frac{n}{p}\log_2(\frac{n}{p})) + O(\frac{n}{p} * \log_2(\frac{n}{number\ of\ hulls}))$$

$$= O(\frac{n}{p} * (\log_2(\frac{n}{p}) + \log_2(\frac{n}{p}) + \log_2(\frac{n}{number\ of\ hulls})))$$

$$= O(\frac{n}{p} * (2 * \log_2(\frac{n}{p}) + \log_2(\frac{n}{number\ of\ hulls})))$$

Although the time complexity in divide-and-conquer algorithm seems to be affected by the number of convex hulls which the algorithm has to merge when the size of the problem remains the same, there is a cost if we want to decrease the number hulls which need to be merged. Reducing the number of base convex hulls constructed, the size of each base convex hull will increase and we have to pay the price for the hidden cost to construct them. On the other hand, if we want to reduce the size of base convex hulls by increasing the number of it, there is different cost which we will have to pay: overheads for thread switching. With greater numbers of convex hulls, to merge them, the system will have to perform thread switching more times.

Actual running time

As mentioned previously in this report, two types of parallel computing technology are used: Cilk from Massachusetts Institute of Technology designed for general purpose multi-core central processing units and CUDA from NVidia designed specific with multi-core graphic processing units. Since both technologies are based on different processors that have different architectures, memory types and speeds, the three algorithm will be tested separately on different systems for differences in running time between parallel and sequential algorithms.

The running times under Cilk environment

We will first start with the Cilk version of the implementations. The running time test consists of 11 different sizes of problem ranging from $n=2^{14}$ to $n=2^{24}$ and the time is measured in milliseconds. For each data size, 10 runs are measured and the average running time is calculated. Due to the lack of precision timers under normal C++ environment, the maximum resolution of running time for Graham scan algorithm and Monotone chaining algorithm is at 10 milliseconds instead of 1 millisecond. The following table shows the running time for Graham scan algorithm (*Table 2).*

Graham Scan	
Size of the data	*Average running time in milliseconds*
2^{14}	20
2^{15}	30
2^{16}	80
2^{17}	150
2^{18}	340
2^{19}	710
2^{20}	140

2^{21}	3140
2^{22}	6630
2^{23}	13920
2^{24}	32710

Table 2. the running time for Graham scan under Cilk

To plot the data to a graph(fig. 12).

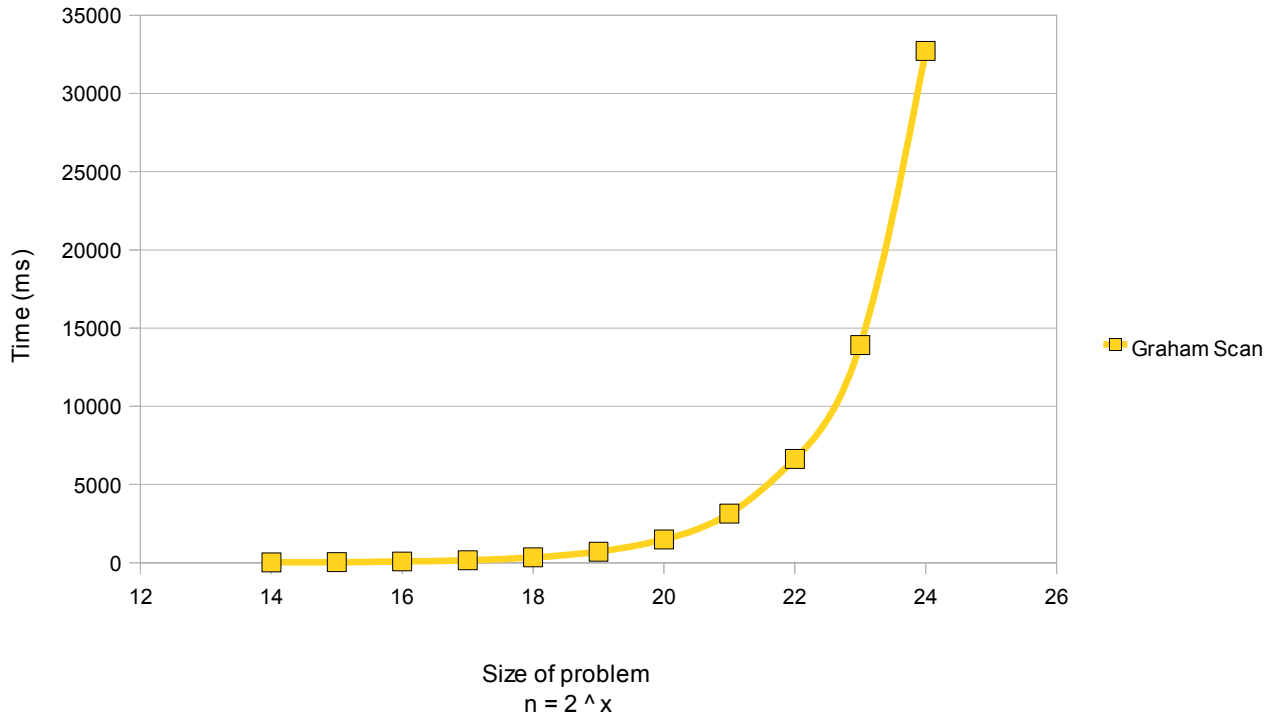


figure 12. the running time for Graham scan algorithm under Cilk

The next table presented will show the measured running time when using Andrew's monotone chaining algorithm (table 3) and the graph plotted (fig. 13).

Andrew's Monotone Chaining	
Size of the data	*Average running time in milliseconds*
2^{14}	10
2^{15}	20
2^{16}	50
2^{17}	90
2^{18}	190
2^{19}	400
2^{20}	880
2^{21}	1740
2^{22}	3680
2^{23}	7670
2^{24}	16150

Table 3. the running time for Andrew's monotone chaining under Cilk

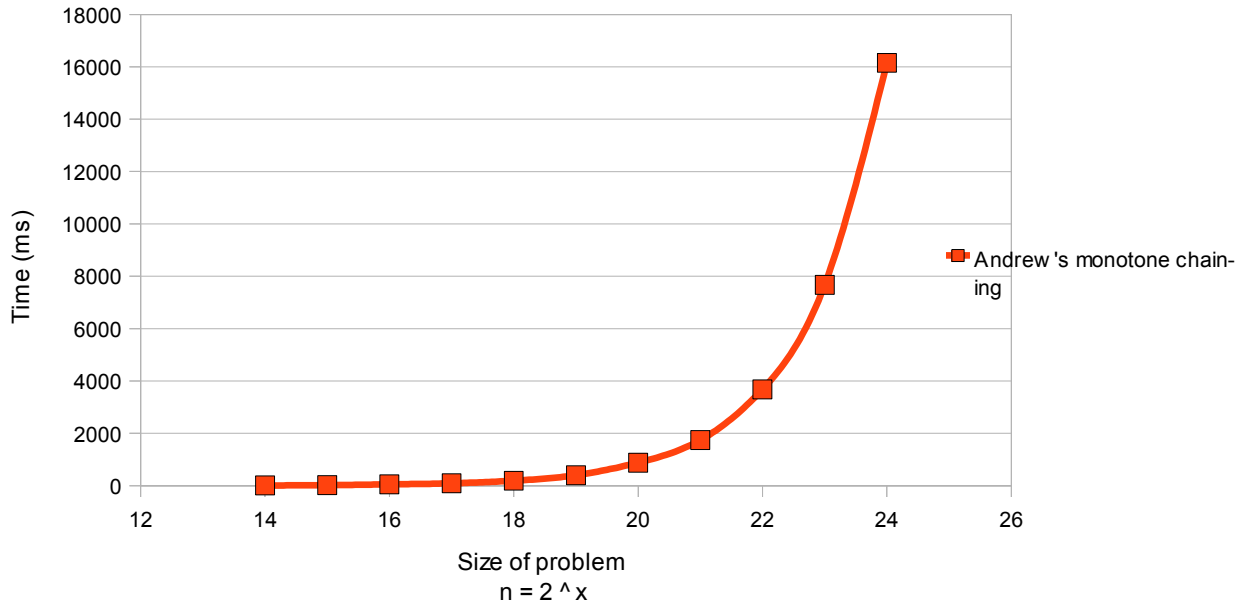


figure 13. the running time for monotone chaining algorithm under Cilk

Lastly, the divide and conquer algorithms for Cilk is tested (table 4) and the results are plotted (fig. 14).

Divide-and-Conquer	
Size of the data	*Average running time in milliseconds*
2^{14}	8
2^{15}	17
2^{16}	38
2^{17}	81
2^{18}	173
2^{19}	367
2^{20}	777
2^{21}	1643
2^{22}	3471
2^{23}	7321
2^{24}	15329

Table 4. the running time for divide-and-conquer algorithm under Cilk

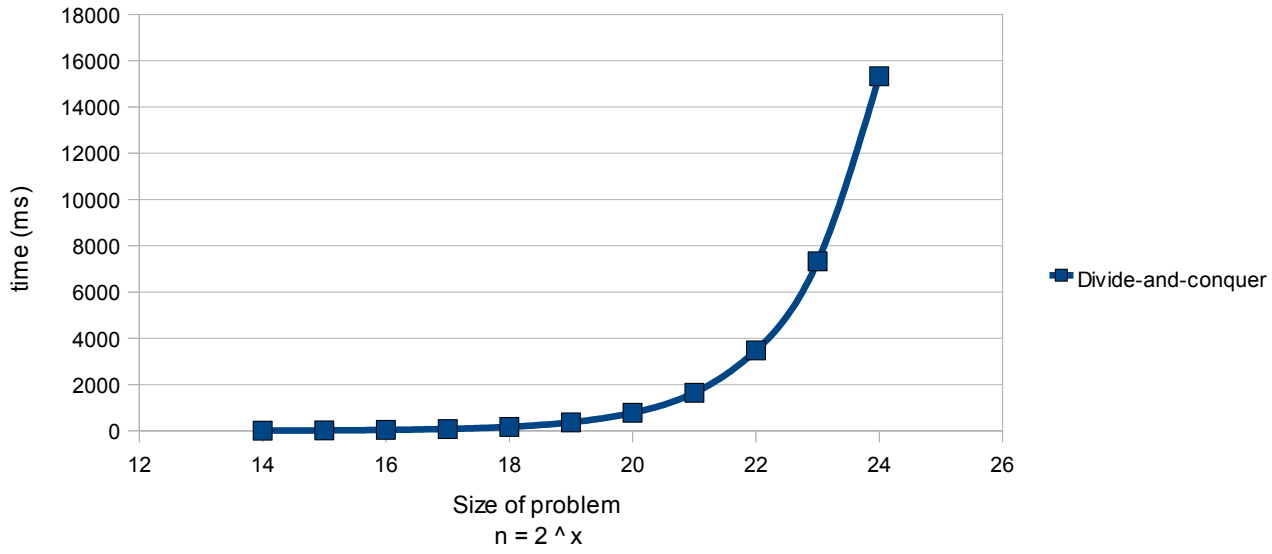


figure 14. the running time for divide-and-conquer algorithm under Cilk

Finally, to compare the difference of running time, we combine the results together (fig. 15)

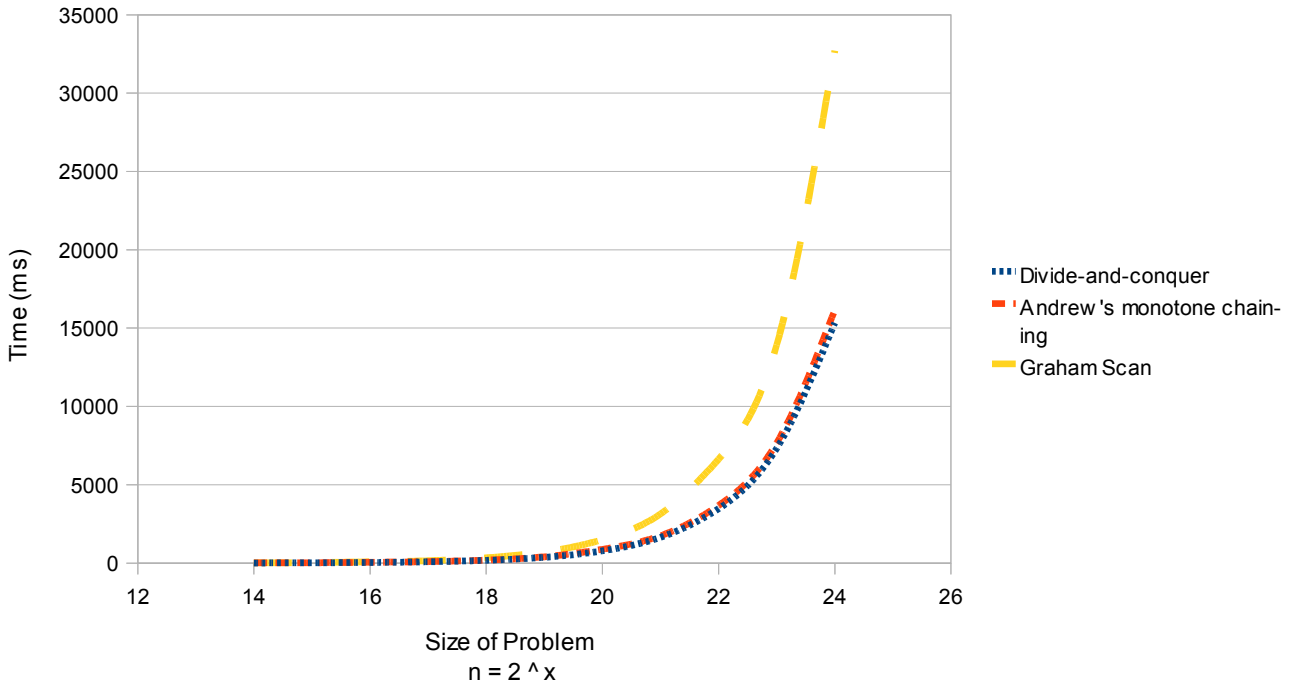


figure 15. the comparison of running times under Cilk

From the graph above, we can observe the following points:

- The parallel divide-and-conquer algorithm has the best execution time in all sizes of data
- Andrew's monotone chaining has a very similar growth rate compare to divide-and-conquer algorithm but it is slightly higher than the parallel code
- Graham scan algorithm has the highest growth rate, the difference can be observed easily starting from $n=2^{19}$
- All three algorithms have similar curves which grows in a steady rate in which the performance does to change with larger set of points

- As the size of input data increases, the difference in terms of running time increases

The running times under CUDA environment

The next part of the testing is to take the same algorithms and execute them under a CUDA environment. Similar to the previous test, 12 different sizes of problem ranging from $n=2^6$ to $n=2^{17}$ is used and the time is measured in milliseconds. For each data size, 10 runs are measured and the average running time is calculated. Compared to the previous test, the data size of for CUDA environment is smaller because of the design of the CUDA processing units. In order to compare the difference in performance between sequential and parallel execution methods, data with same input size is required. While a single core in CUDA computing device is designed with limited amount of memory, which is usually much smaller than a standard desktop computer, the input data size can not be too large. However, CUDA API does provide a timer with great precisions so the results gathered are still sufficient for measuring the performance of the algorithms.

The first algorithm tested was the Graham scan (table 5)

Graham scan	
Size of the data	*Average running time in milliseconds*
2^6	1.7883
2^7	3.9960
2^8	9.0147
2^9	9.8324
2^{10}	43.4051
2^{11}	94.2198
2^{12}	203.2733

2^{13}	436.3441
2^{14}	931.9713
2^{15}	1982.4707
2^{16}	4201.7957
2^{17}	8878.6211

Table 5. the running time for Graham scan algorithm under CUDA

To plot the data to a graph (fig. 16)

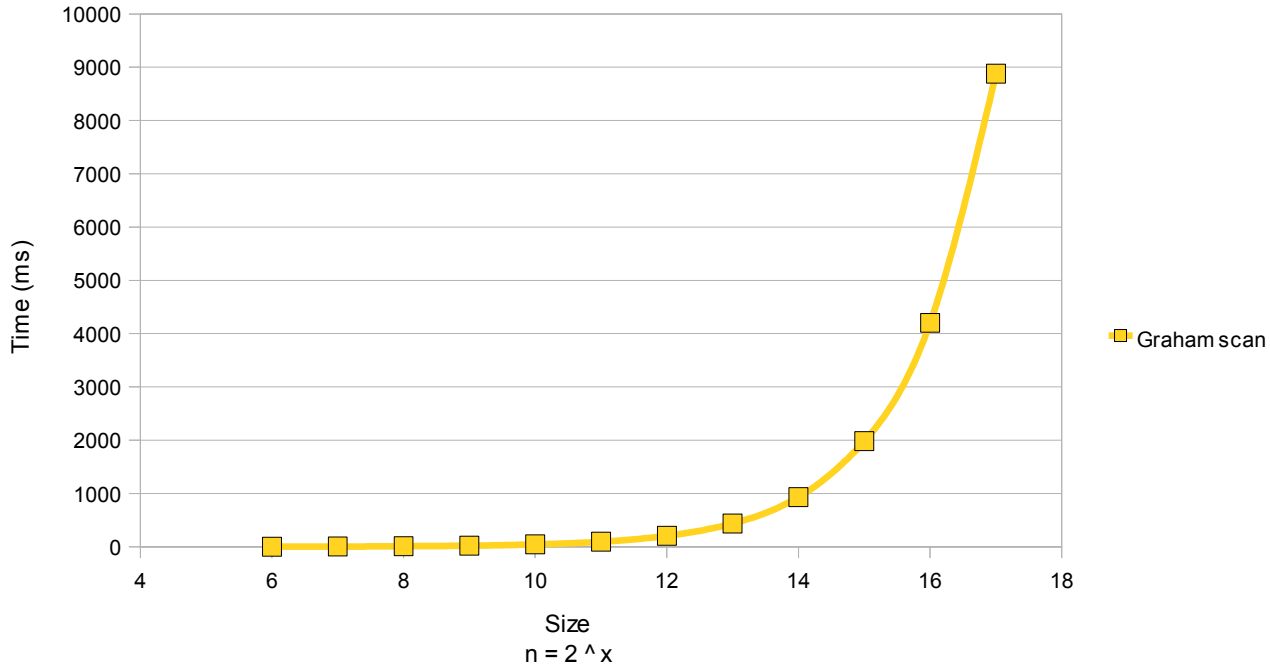


figure 16. the running time for Graham scan algorithm under CUDA

The second algorithm tested is Andrew's monotone chaining (table 6):

Andrew's monotone chaining	
Size of the data	*Average running time in milliseconds*
2^{6}	1.0816

2^7	2.3824
2^8	5.3113
2^9	11.7522
2^{10}	25.5513
2^{11}	55.3772
2^{12}	119.1765
2^{13}	255.2938
2^{14}	544.2289
2^{15}	1155.8436
2^{16}	2447.1861
2^{17}	5168.1157

Table 6. the running time for Andrew's monotone chaining algorithm under CUDA

The graph plotted with the data (fig. 17)

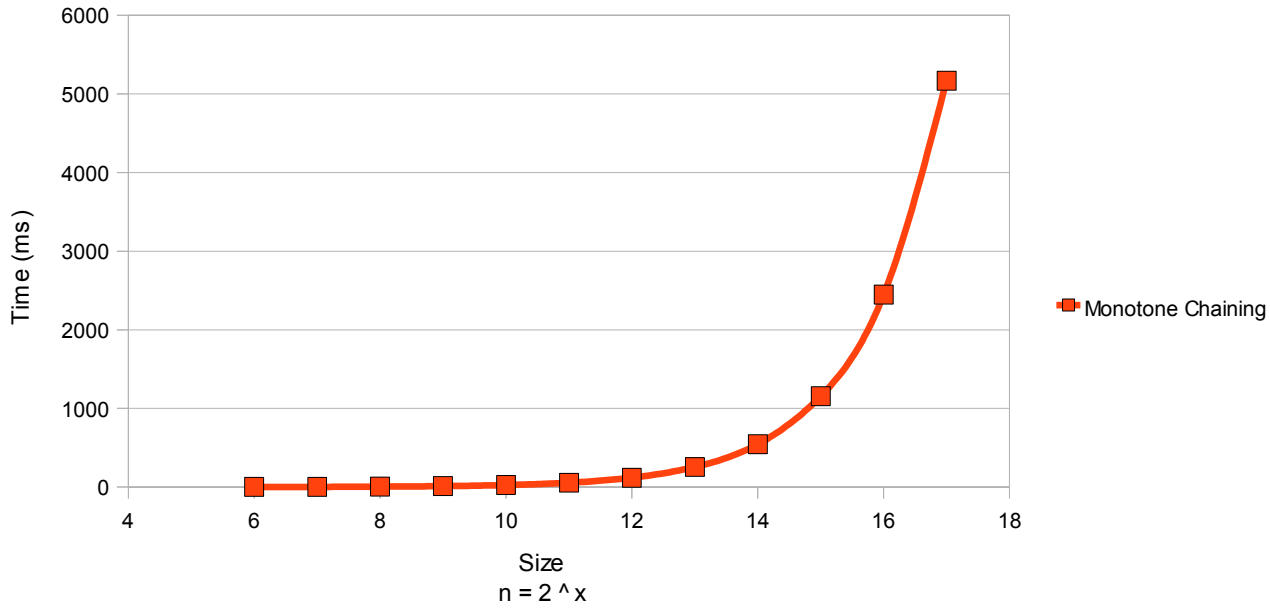


figure 17. the running time for Andrew's monotone chaining algorithm under CUDA

The last algorithm tested under the CUDA environment is the divide-and-conquer algorithms (table 7)

Divide-and-conquer	
Size of the data	*Average running time in milliseconds*
2^6	0.4405
2^7	0.7651
2^8	1.2717
2^9	2.3745
2^{10}	4.1151
2^{11}	7.7329
2^{12}	14.2726
2^{13}	29.2878
2^{14}	55.8384
2^{15}	110.6774
2^{16}	223.0235
2^{17}	452.4411

Table 7. the running time for divide-and-conquer algorithm under CUDA

and the graph plotted accordingly(fig. 18)

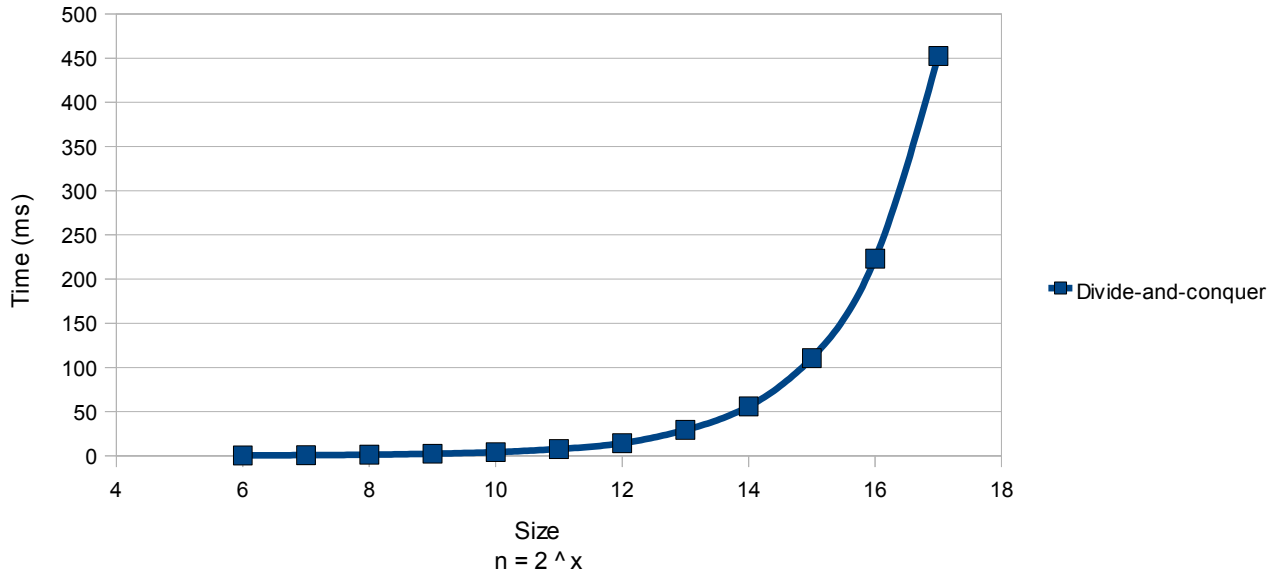


figure 18. the running time for divide-and-conquer algorithm under CUDA

To compare the performance, we plot the running times of all three algorithms together (fig. 19)

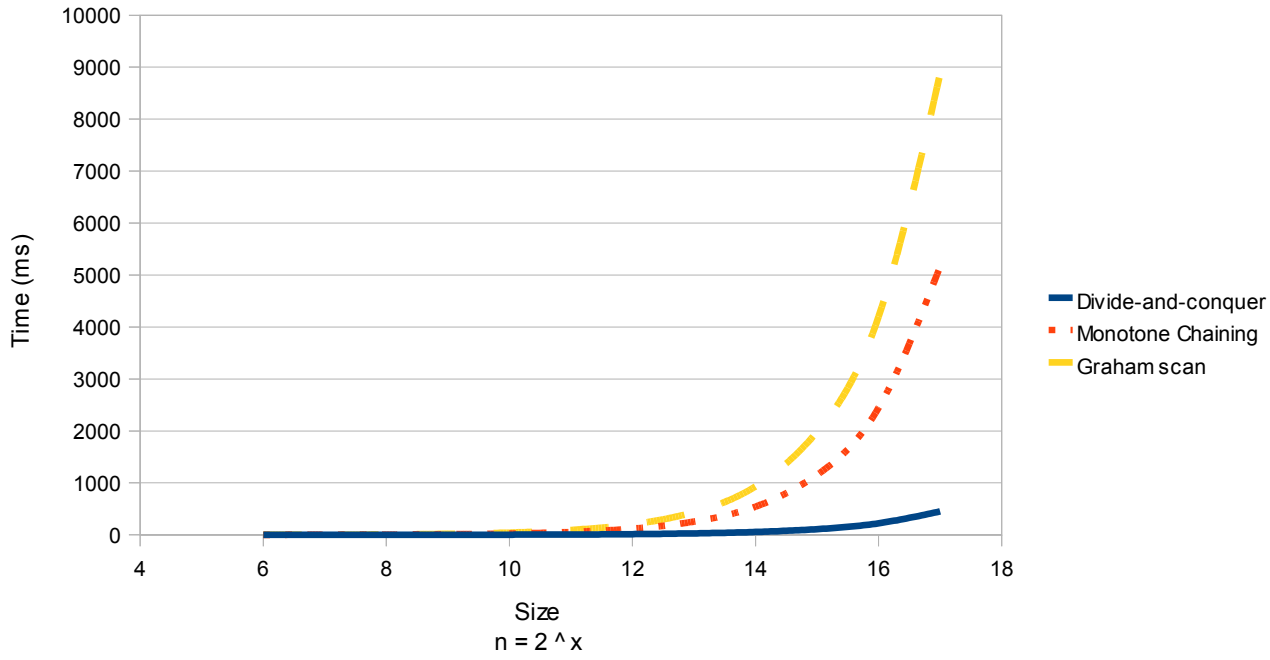


figure 19. the performance difference between algorithms

From the graph above, we can observe the following:

- In terms of time, Divide-and-conquer algorithm has the best performance while Graham scan has the worst performance amount the three of the algorithm
- Graham scan and monotone chaining have a very similar curve; however, Graham scan algorithm has a higher growth rate
- The growth rate of the divide-and-conquer algorithm is very slow compared to the other two methods of computing convex hulls
- As the number of points increases, the result of performance comparison remains the same: divide-and-conquer has the best performance, Graham scan has the worst performance
- As the size of input data increases, the difference in terms of running time increases

Discussion

Based on the output of the tests, the parallel divide-and-conquer algorithm clearly has achieved the best performance amount out of all other algorithms tested in this experiment. To determine the improvement of parallel computing over sequential computing, we need to calculate the speedups by comparing the running time of divide-and-conquer to the other two algorithms(table. 8).

Size of the data	*Speedup comparing to Monotone Chaining*		*Speedup comparing to Graham Scan*	
	CUDA	*Cilk*	*CUDA*	*Cilk*
2^6	2.46		4.06	
2^7	3.11		5.22	
2^8	4.18		7.09	

2^9	4.95		8.35	
2^{10}	6.21		10.55	
2^{11}	7.16		12.18	
2^{12}	8.35		14.24	
2^{13}	8.72		14.90	
2^{14}	9.75	1.25	16.69	2.50
2^{15}	10.44	1.18	17.91	1.76
2^{16}	10.97	1.32	18.84	2.11
2^{17}	11.42	1.11	19.62	1.85
2^{18}		1.10		1.97
2^{19}		1.09		1.93
2^{20}		1.13		1.92
2^{21}		1.06		1.91
2^{22}		1.06		1.91
2^{23}		1.05		1.90
2^{24}		1.05		2.13

Table 8. the speedups of different algorithms

and if we plot the data from the table to a graph (fig. 20)

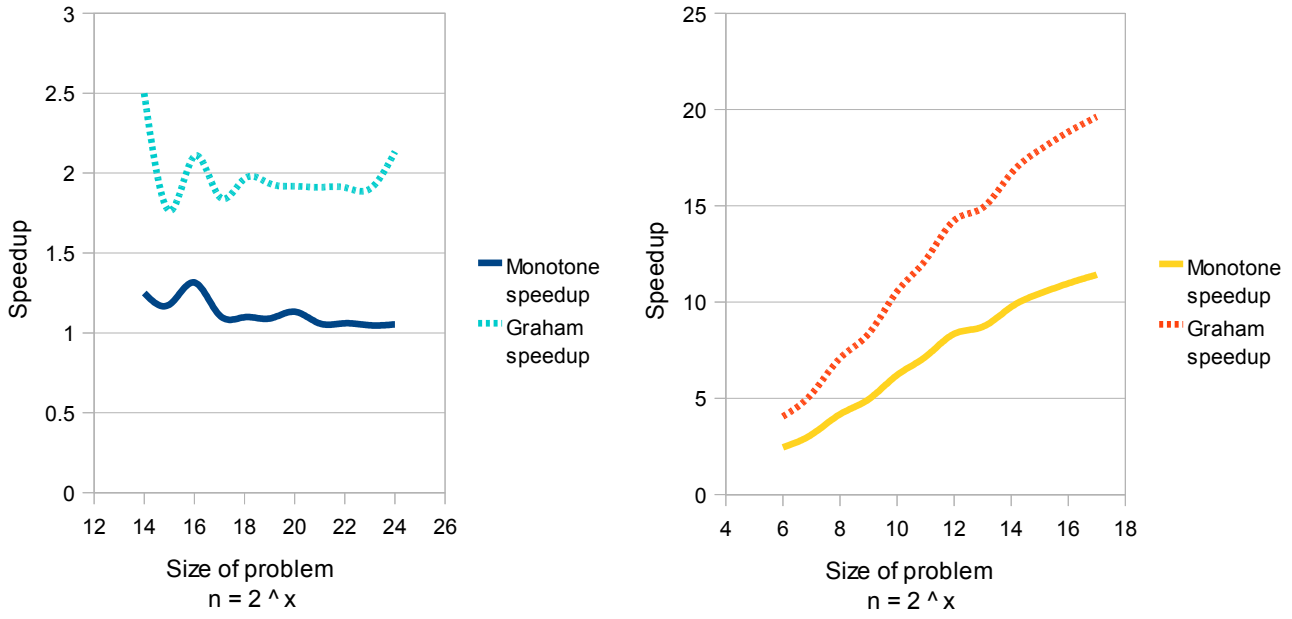


figure 20. the speedups between algorithms

From the graph above, we can see that the Cilk divide-and-conquer algorithm speedups remain the same regardless of the size of the problem. On the other hand though, the CUDA divide-and-conquer algorithm gains more speedup when the number of input points increases. The reasons for this can be explained with Amdahl's law.

According to Gene Amdahl's research in 1967 [9], the speedup of a parallel program is based upon the portion of the algorithms which can be computed in parallel (parallelism) and the number of processor available for the execution. In other words, to achieve high speedups, not only more processors are needed but also a higher portion of parallelism is required. With more than one processor, to maximize the usage of the computing ability, the algorithms must keep all processors as busy as possible during the execution of the code while minimizing the usage for non-essential computing such as synchronization, thread switching, or data communication. In other words, the

number of threads created during the computation should be the same as the number of processors in order to utilize the computing ability. With less threads then processors, the system is not using all its computing power because some processors will be idle. On the other hand, if there are more threads than the processor, the execution time will increase due to the overhead of thread switching. With that said, in the optimal situation where the number of threads is the same as the number of processors, we should get a constant speedup regardless the size of the input data.

In our program while using the Cilk version of the implementation, the number of thread is fixed, which depends on the number of processors of the machine, despite the size of input data. Since the number of threads created is optimal for the number of processors in the system, the speedup for the parallel remains constant which is consistent with Amdahl's theory. In order to calculate the speed up, in the CUDA version of the implementation, the number of input data points is limited to the largest possible data a single CUDA core can handle. Having such a limit, even with the smallest possible base-case-hulls, there is not enough threads for all processors to process. Trying to optimize the performance of the implementation, the size of each base case convex hull is kept minimal in order to generate more threads for the processors. With the design like that, when the size of the input data increases, the number of threads increases too. As the result, more of the processors are used and therefore the performance of the code improves since it achieved higher parallelism.

Also given in the article of Amdahl's research [9], there is a formula for calculating estimated speedup for the algorithms based on the portion of program which can be compute in parallel (P) and the number of processors in the system (S):

$$\text{Speedups} = \frac{1}{(1-P)+\frac{P}{S}}$$

to apply the formula for our experiment, we first need to estimate whats the proportion of the code which is made parallel in our divide-and-conquer algorithm. The algorithm can be break down to three parts:

- Sorting: the program uses parallel merge sort so it is parallel computed
- forming base cases: multiple threads running on multiple processors are used to form the base-case-hulls, so this part is also parallel
- merging convex hulls: multiple threads and processors are also used to merge convex hulls, so it is parallel

Since all three parts of the programming are running in parallel, for the sake of simplicity, we assume that the proportion of the code which were made parallel is 100 percent of the program so $P=1$. In our tests for Cilk, the program was running on the machine with four processors; therefore, $N=4$. To calculate the estimated speed up:

$$\text{Estimated speedup} = \frac{1}{(1-1)+\frac{1}{4}} = \frac{1}{\frac{1}{4}} = 4$$

which is much higher than the actual speed up recorded. The reasons for that is because we did not consider the overhead of creating, monitoring, synchronizing and destruction of the threads. In addition, we also ignore the extra memory access delay caused by multiple scanning of the points during the merging process. As for our test under CUDA environment, the code was running on a CUDA device with 112 cores. With the same formula, we get an estimation of

$$\text{Estimated speedup} = \frac{1}{(1-1)+\frac{1}{112}} = \frac{1}{\frac{1}{112}} = 112$$

although we were unable to test the CUDA programming for optimal speedups, we know for a fact that the speed up can be increased much more if larger sets of data were used based on our estimation.

In conclusion, from this experiment, we have shown that the execution time can be reduced with multiple cores. In addition to that, we have also shown that there are differences between an optimized and a non-optimized parallel code in terms of speedups. Further work remains to be done to assess more quantitatively in order to see how does creating and managing multiple threads affects the performance of the program and its relationship with the number of cores present in the system.

Reference

[1] *What is CUDA?* [Online]. Available: http://www.nvidia.com/object/what_is_cuda_new.html

[2] *Cilk* [Online]. Available: http://en.wikipedia.org/wiki/Cilk

[3] J. Daintith. (2004). *Convex Hull* [Online]. Available: http://www.encyclopedia.com/doc/1O11-convexhull.html

[4] D. Sunday. *The convex Hull of a 2D Point Set or Polygon* [Online]. Available: http://softsurfer.com/Archive/algorithm_0109/algorithm_0109.htm

[5] *Applications* [Online]. Available: http://fleischer.selfip.com/Courses/Algorithms/Alg_ss_07w/Webprojects/Chen_hull/applications.htm

[6] A.M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", Info. Proc. Letters 9, pp. 216-219 (1979)

[7] Franco Preparata and S.J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions", Comm. ACM 20, Vol. 20, pp. 87-93, 1977

[8] D. Bitton, D. DeWitt, D.K. Hsiao, and J. Menon, “A Taxonomy of Parallel Sorting”, ACM Computing Surveys, Vol. 16, pp. 287-318, Sep. 1984

[9] G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," in *AFIPS Joint Computer Conferences*, Atlantic City, 1967, pp. 483-485

Additional Reference

- D. R. Chand and S. S. Kapur, “An Algorithm for Convex Polytopes”, Journal of the ACM, Vol. 17, pp. 78-86, 1970
- R. A. Jarvis, “On the identification of the convex hull of a finite set of points in the plane”, Inform. Process. Lett., Vol. 2, pp. 18-21, 1973
- W.F. Eddy, “Algorithm 523: CONVEX, A New Convex Hull algorithm for Planer Sets [Z]”, ACM Transactions on Mathematical Software, Vol 3, pp. 411-412, 1977
- A. Bykat, “Convex hull of a finite set of points in two dimensions”, Inform. Process. Lett., Vol. 7, pp.296-298, 1978
- M. Kallay, “The Complexity of Incremental Convex Hull Algorithms in R^d , Inform. Process. Lett., Vol. 19, pp. 197, 1984
- D.G. Kirkpatrick and R. Seidel, “The Ultimate Planar Convex Hull algorithm?”, Cornell University, Ithaca, NY, Rep. TR83-577, pp. 287-299, 1986