Analysing the Performance of Divide-and-Conquer Algorithms on Multicore Processors

A Thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology in Computer Science and Engineering

BY

Sibani Prasad Dash(109CS0137)

K Pradeep Kumar Dora(109CS0511)

Under the Guidance of

Prof. BIBHUDATTA SAHOO

Department of Computer Science and Engineering
National Institute of Technology,
Rourkela - 769008
May - 2013

Acknowledgements

We wish to show our sincere gratitude to our guide Prof. Bibhudatta Sahoo, Computer Science Engineering Department, for his guidance, helpfulness and continuous motivation in all regards during the total duration of our final year project. We would also like to thank all the professors of the department of Computer Science and Engineering, National Institute of Technology, Rourkela, for their constant support, motivation and encouragement. Last but not the least, a sincere thanks to our batch mates for their help and cooperation.

Sibani Prasad Dash
109CS0137

K Pradeep Kumar Dora
109CS0511

B.Tech (Computer Science and Engineering)

...

ABSTRACT

Multicore systems are widely gaining popularity because of the significant availability and performance increase over the single core systems. Multicore systems have a lesser power consumption and heat generation than that of the multiple single core systems. The different compiler support provided by different vendors also make multicore programming one of the main area of research. The multicore programming utilises the power of multiple cores to parallelise a task. The widely used algorithm paradigms for multicore programming are the Divide and Conquer algorithms. The divide and conquer algorithms are candidate problem for the multicore programming because divide and conquer algorithm divides a problem into sub- problems which can be solved by distributing the sub-problems among the different cores and parallel solve them. A wide range of divide and conquer algorithm has been parallelized. In this paper, we have taken two of the widely used divide and conquer algorithms, quick sort and convex hull, parallely implemented them to analyse their performance gain in compared to the sequential version of the algorithm. The parallel implementations distribute the load onto the multiple cores, parallely work upon the loads and finally merge individual results of the each core. We have also proposed a scheme for efficient merging of the parallely sorted sub-arrays in the quick sort. We have taken the mean and standard deviation theory for efficient merging of the sorted sub-arrays. The OpenMP programming model has been used for the implementation of the programs. The processor architecture used for analysing the behaviour of the algorithm is a shared memory based processor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today in this world, time is a major factor for everyone. Everyone needs to do a work in as minimum time as possible. This problem led to different type of programming paradigm to invent. Thus the parallel computing was introduced to the world. The parallel computing is the programming paradigm which deals with solving the problem not in the conventional sequential manner; but in a parallel way. The conventional processors were not capable of parallel computation. This resulted in in the invention of a new genre of processors, popularly called as multicore processors.

The multicore processors are the computing components consisting of two or more than two computing CPUs, called cores. These cores can parallel work and execute CPU instructions. The multicore processors have several advantages over the single core processors such as less heat generation, less energy consumption etc. A wide range of compilers support the multicore programming. Some of programming models that support multicore programming are TPL (Task Processing Library for DOT.NET), OpenMP, MPI and Clik++ etc. This wide range of programming support also makes the multicore programming easier for the programmers.

The idea here is to efficiently implement the quick sort method on the multicore systems. So the first question that strikes to the mind is why multicore systems and how do they increase the speed up? The multiple independent cores can read and execute program instructions which are ordinary CPU instructions such as add, move data etc. The performance can be gained by distributing the load among the various cores which can run them in parallel so as to decrease the overall execution time of the overall process. The next question that is coming into

mind is that by using multicore systems, do all the processes have performance gain? The answer is simply NO! This is because the processes take advantage of the multiple cores; but the multiple cores do not distribute the work among themselves. The implementation of the algorithm should be done in such a way that the process could effectively take advantage of the multiple cores. The main candidate algorithms that can take advantage of the multicore architectures are Divide-n-Conquer algorithms.

The quick sort and convex hull algorithms are Divide-n-Conquer algorithms. So these become candidate problems for the multicore programming. The idea in implementing the quick sort on multicore platform is to distribute the dataset element among each core, sorting them using quick sort and finally merging them into a single dataset. This quick sort is otherwise known as Hybrid Quick Sort because it seems to be a hybrid of sequential quick sort and merge sort. The experimental results show a performance increment of the quick sort technique.
The idea in implementing the convex hull on multicore platform is same as that of quicksort. The convex hull internally uses the quicksort for sorting the points. So the quicksort inside the convex hull program can be parallelized. There is a merge process inside the convex hull which can be also done in parallel among the cores. The experimental results of the parallel convex hull technique show a result in the decrease of execution time.

1.1 Literature review

Divide-n-Conquer algorithm paradigm is one of the bases for the multicore programming. In the Cormen book of Introduction to Algorithms[1], a wide variety of sequential Divide and Conquer algorithms has been published.The multicore programming models proposed mainly are of three types: Master/Slave, dataflow and OpenMP model[16].Prof. B.D. Sahoo,Prof. A.K. Turuk and Ram Prasad Mohanty[13] published a paper on the analysis of the different multicore architecture simulated under Multi2Sim simulator. The parallelization of quicksort proposed by DeWitt, Naughton and Schneider[4] uses an approach called probabilistic splitting proposed by Iyer, Ricard and Varman. Moh,Yu and Han[11] proposed another type of quicksort based on weighted partition.Yuran Lan and

Magdi Mohamed[12] proposed the parallel quicksort in hypercubes. Nakagawa et all[9] proposed the parallel convex hull based on the sorted points which was the reason behind the parallel merging. Miller and Stout also proposed a parallel convex hull algorithm which is a worst case hypercube algorithm[14].The opimized convex hull[8] proposed by Kendre and Kulkarni uses both OpenMP and MPI for better performance.

1.2 Motivation

Multicore processors are the dominant type of processors in the market as compared to their counterpart single core processors. By this we mean that these are highly available in the market. The reason behind this is the downfall of the single core processors performance. The clock frequency cant be increased at a high rate and the single core processors have different heating and design issues.
This readily availability of the multicore processors is one of the primary motivation of this paper. The divide-n-conquer algorithms were chosen for the parallel computing because these algorithms have a tendency of having multiple independent part which can be solved in parallel. A wide range of divide and conquer algorithms are proposed. Two of the most important algorithms are quicksort and convex hull which have large application such as ray tracing, video games and path finding etc.

1.3 Problem Statement

The multicore computing is slowly gaining speed with its wide range of advantages. The quicksort and convex hull have the number of applications in the technical field. There is a good scope of research in the field for increasing the efficiency of the algorithms. The algorithms proposed for this is implemented and the performance parameters are compared with their sequential version. The sequential version needs to have higher cost in terms of the execution time whereas the parallel version implementation tends to have lower execution time.
We have also proposed an enhancement to the implemented algorithms which uses the mean and standard deviation of the population to gain some amount of speed

up. The parallel sorting which needs merging have not considered the sequence of merging of the sorted subsets. The sequence of merging can play an important role in case of inputs where these inputs vary from each other in a wide manner. The sequence can be determined by using the mean of the distribution.
The speed can be further improved by implementing any other kind of distribution technique.

1.4 Organization of the thesis

Organization of the thesis The thesis is organized as follows. Chapter 2 describes the theory behind multicore programming and multicore architecture. Some of the standard architectures and programming models are discussed. Chapter 3 gives the implementation details of the quick sort algorithm and section 3.5 gives the results and performance analysis with the implementation of the enhanced quick sort. Chapter 4 describes another divide and conquer algorithm i.e. convex hull and in the section 4.5, the results are shown. Finally with Chapter 6, we conclude this paper with the future work to be done.

Chapter 2

Multicore Architecture and Multicore Programming

2.1 Introduction

The multicore platforms are the primary requirement for the parallel computing. These multicore systems are primarily of MIMD type. The different cores can execute different threads operating on different parts of the memory i.e. Multiple Instruction Multiple Data(MIMD). The advantage of the multicore systems led to design a wide variety of the different architectures. These architectures vary from each other in terms of sharing memory and symmetry between the cores.
The speed up of a particular task is given by:

$$SpeedUp = \frac{1}{1 - F + \frac{F}{N}}$$

This is called Amdahls law where F is the parallelizable portion of the task and N is the number of processing units. This law puts a barrier on the maximum speed up of a parallelized profwhich depends on the serial portion of the program.

2.2 Multicore Architecture

The multicore architecture replicates multiple processing cores on a single integrated circuit die known as chip multiprocessor or CMP. In this case, the operating system perceives each core as a separate processing unit. The OS scheduler then schedules the processes into different cores. There exists different type of multicore architecture as explained below.

2.2.1 Symmetry

The multiple cores on the single chip can have same design or different designs. According to these , the architecture is of following two types:

Homogenous Multicore:
All the cores on the die have same properties having a shared memory architecture. These cores have the same computation speed up and all the cores run the threads in the same way. Ex: AMD and Intel dual- and quad-core processors.
Heterogeneous Multicore:
There exist different processing core types on a die. Each core is optimized for a different role. Different cores have different computation speed up and each core will be having a special quality to solve a particular problem. Ex: IBM Cell Broadband Engine.

2.2.2 Shared Memory Multicore

Multicore processor is a special kind of multiprocessor in which all the processors are placed on a single die or chip. The multicore processors are MIMD (Multiple Instruction Multiple Data) i.e. different cores can execute different threads (Multiple Instruction) when operating on the different parts of the memory (Multiple Data). The multicore architecture uses shared memory concept. The multicore processor based upon cache sharing can be divided into the following types:

Multicore with shared cache:

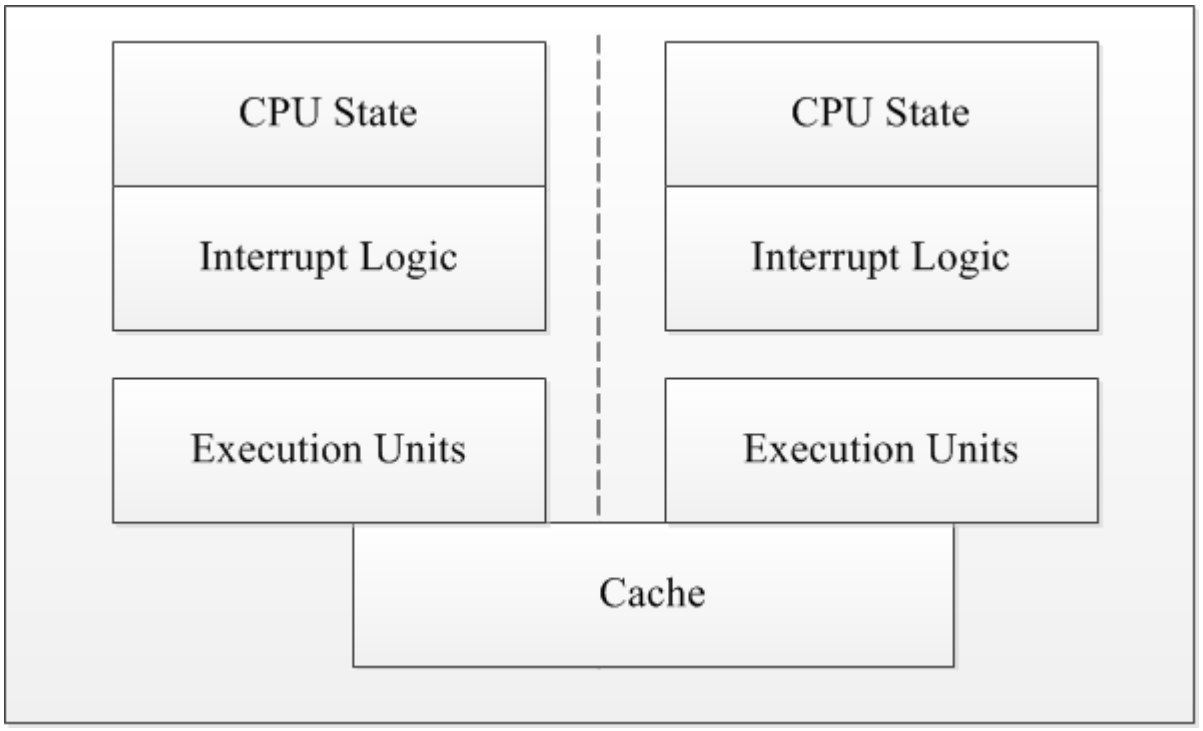

FIGURE 2.1: Multicore with shared cache

These types of multicore systems have a cache present on the chip which is shared among all the cores present on the die. The advantage of this is sharing of instruction/data among multiple cores so that there wont be any communication cost in accessing the shared variable. The disadvantage is the destructive interference among the cores when trying to access a modified shared variable.

Multicore without shared cache:

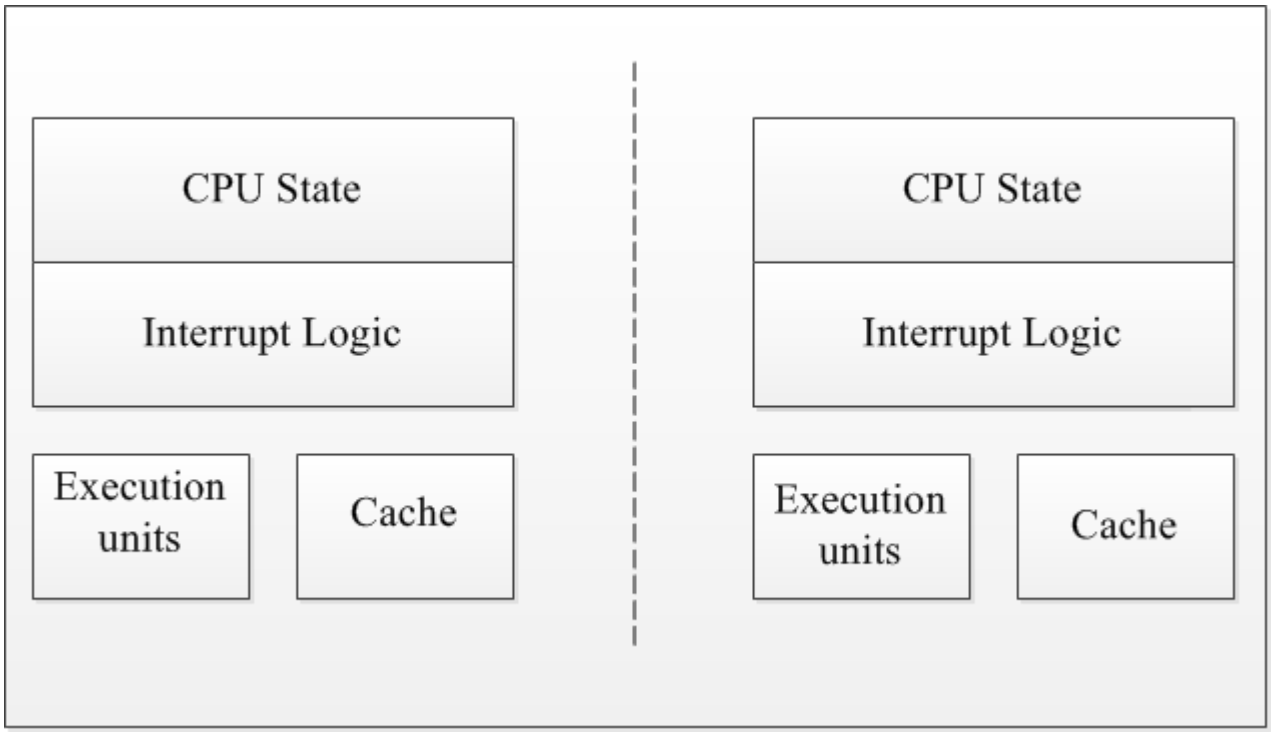

FIGURE 2.2: Multicore without shared cache

These types of multicore systems have different caches for each core on the die. The advantage is being that each core will have its own working area that is not affected by the other cores. The disadvantage is the cache coherency. This happens when a shared variable is updated is one cache; but not modified in the other caches accordingly.

Multicore with Hyper Threading:

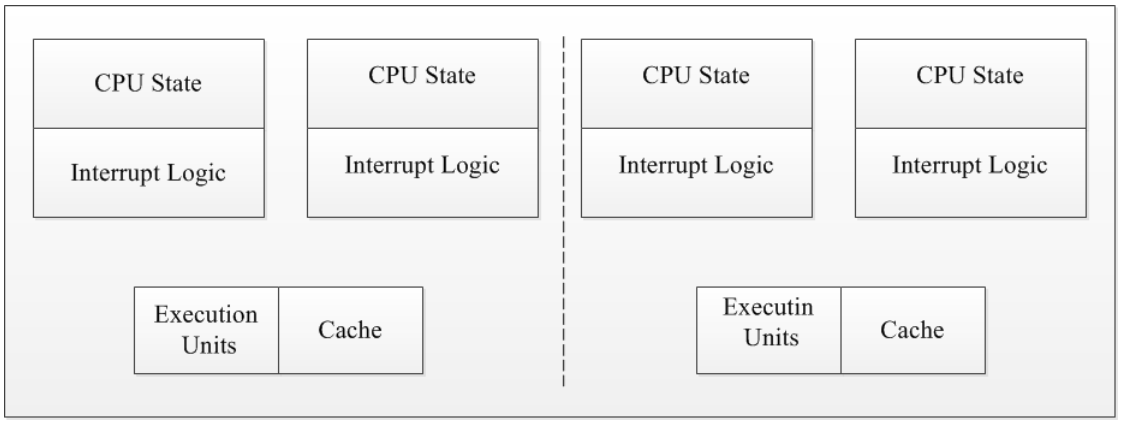

FIGURE 2.3: Multicore with Hyper Threading

Hyper-threading is Intels SMT (Simultaneous Multithreading) implementation used to improve the parallel computing capabilities of the processing cores. It uses shared memory architecture. When this technology is implemented, each processing core will run two threads simultaneously on the cores. So each core will appear as two logical cores to the OS. So the OS can schedule two different processes on the same physical cores. The two processes can use the same resources. The advantage is improved support for the multithreaded code and allowing multiple threads to run simultaneously. The disadvantage, as claimed by AMD, is higher power consumption and increased cache thrashing.

2.3 Multicore Programming

There are different types of multicore parallel programming model. Task parallelism achieved by concurrent execution of the independent tasks in a software or process. On a single core system, the tasks must share the same processing unit; but in the multicore system, each task can independently run on the multiple cores.

To maximize the speed up, we need to select one of the processing models that best suits for the problem under consideration. The three types of the parallel programming models[16] are:

- *Master/SlaveModel*
- *DataFlowModel*
- *OpenMPModel*

2.3.1 Master/Slave Model

The Master/Slave model represents a centralized control with distributed execution. There exists one master core which schedules various threads that can be executed on any of the available slave cores. It is also responsible for supplying data required by each thread. Processes that can be executed using this model have different smaller parts that can be run by a single thread. The software has a lot of control codes and the memory access is random. Each memory access has a little computation and the code base is usually very high.
The real-time load balancing is one of the significant challenges faced by this model. Each thread has different throughput requirements. So the master core has to maintain a list of free cores and should able to optimize the loads among each processing cores so that maximal parallelism could be achieved.
One or more execution threads are mapped to each core. The task assignment is done by message passing between cores. The message provides the control information between cores and contains a pointer to the data that is going to be used by the core. Each core has at least one task whose job is to receive messages containing job assignments. The task is suspended until a message arrives triggering the thread of execution. The data is accessed in a uniform manner unlike the Master/Slave model where data was accessed randomly. The below figure depicts the Master/Slave parallel processing model.

Master/Slave Model

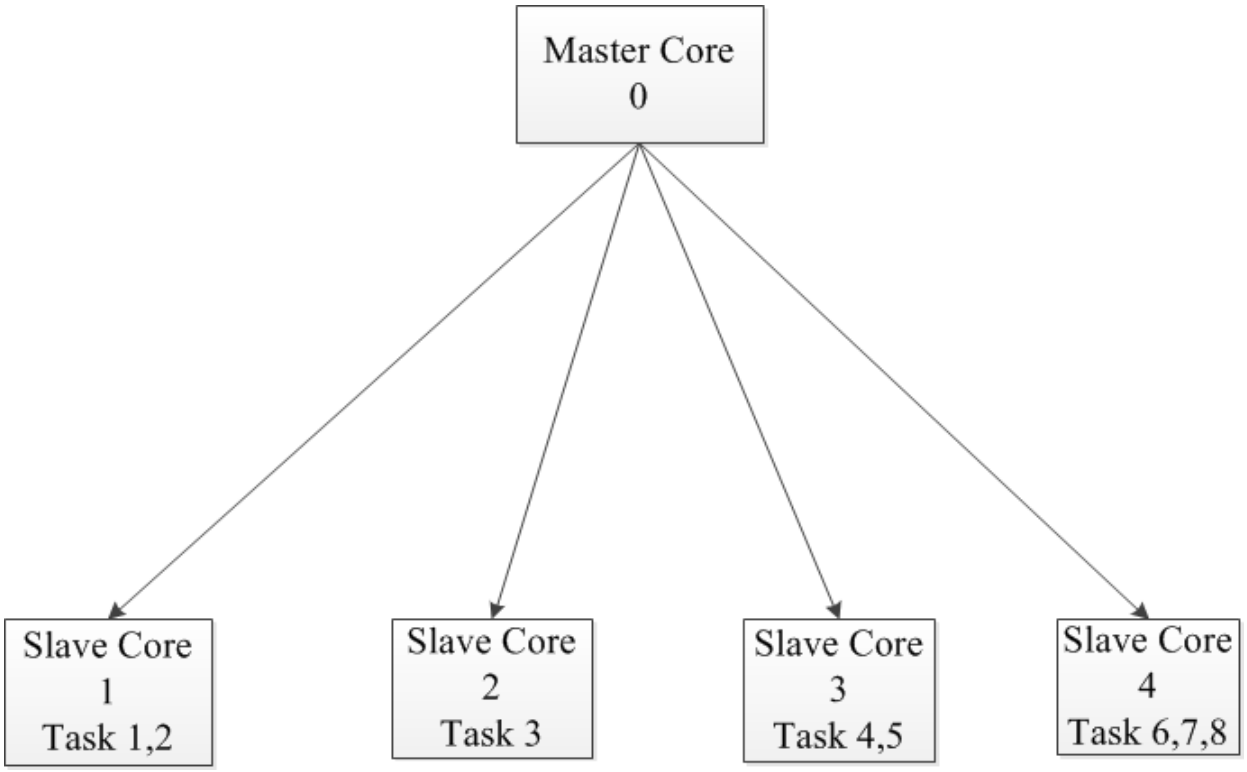

FIGURE 2.4: Master/Slave Model

2.3.2 Data Flow Model

The data flow model represents a distributed control with distributed execution. Each individual core processes some blocks of that using some algorithms and then it passes the processed data to the next core. The application which generally uses this kind of processing model usually has large and computationally complex components which are dependent upon each other and a single processing unit is not enough for the execution.
The challenge faced by the processes or applications using this model is the partitioning of the components across the cores and the high data flow-rate of the system. The components must be mapped to the multiple cores in order to have the pipeline flow regular.
The processing is required to be mapped in such a way that each core executes one or more task. The synchronization between the cores is achieved by having a message passing mechanism. The data flow is done usually having a shared memory or by the DMA controller. The below figure depicts the Data Flow parallel processing model.

Data Flow Model

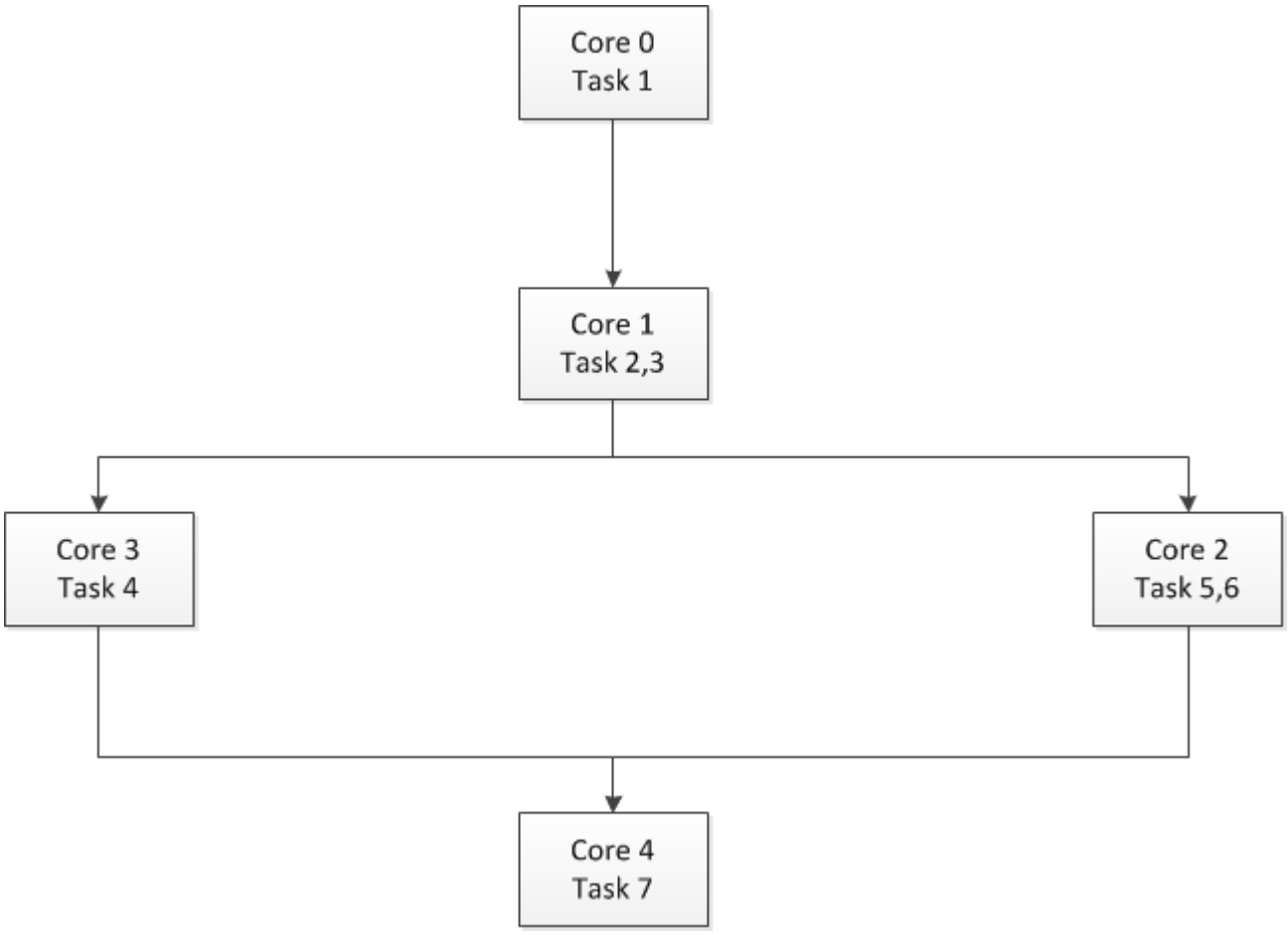

FIGURE 2.5: Data Flow Model

2.3.3 OpenMP Model

OpenMP is API (Application Programming Interface) for developing multi thread applications on multicore architecture. The programming languages that can be used are C, C++ and Fortran. The API is programmer-friendly, easy to use and quick in implementation. The main task for the programmer is to find the parallel task regions in the program. Then insert an relevant OpenMP construct. After this the compiler and the run time system will have the burden to distribute the task among the different threads spread across different cores.
The OpenMP API consists of compiler directives, different library routines and a set of environmental variables associated with the run time.

The implementation of the OpenMP is based upon the Fork-Join Parallelism. This parallelism tells that a larger task can be divided into smaller tasks whose solutions can then be combined; as long as the smaller tasks are independent, they can be executed in parallel. The initial program contains a single thread known as master thread starting in a sequential region. When the parallel region is encountered, the scheduler invokes a group of threads that execute the parallel region concurrently. After the execution of the parallel region, all the threads terminate and a single thread of execution is again started in the sequential region. The below figure depicts the OpenMP parallel processing model.

OpenMP Model

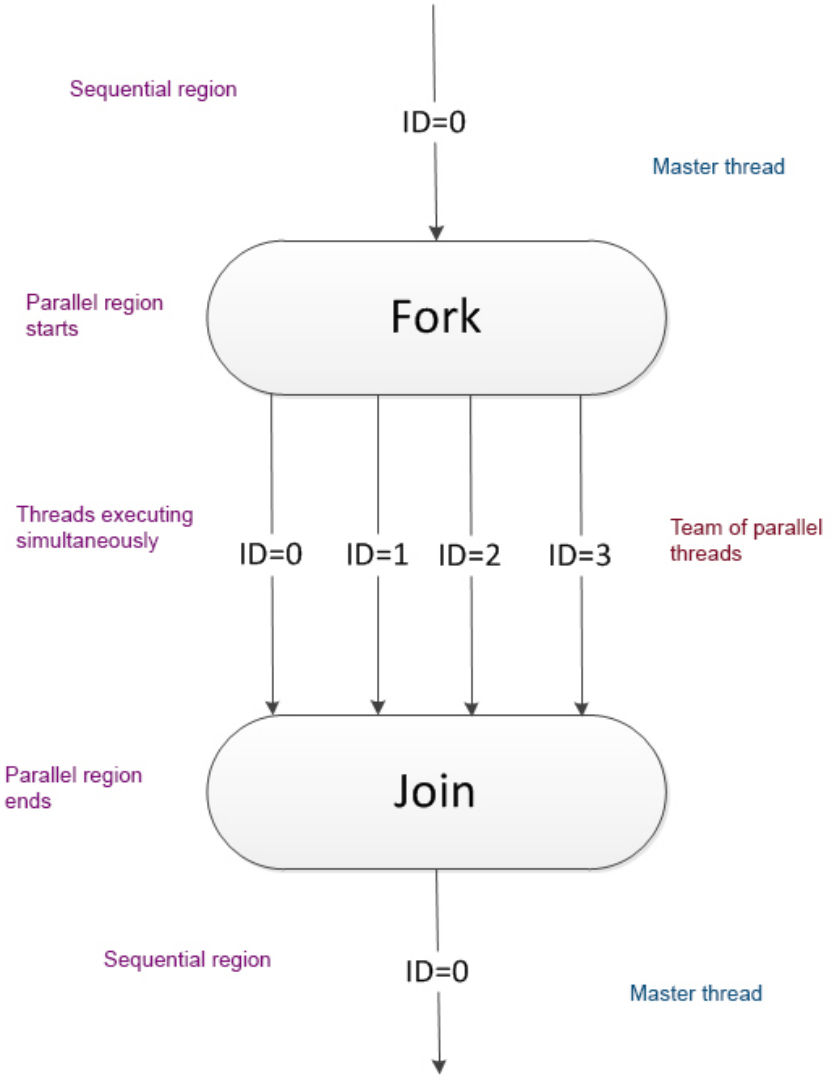

FIGURE 2.6: OpenMP Model

2.4 Conclusion

With this chapter we discussed the different architecture of multicore processors and the different types of programming model. The programming model gives us a hint about which types of algorithms can be a candidate problem for parallelization. The most common computing paradigm that comes into picture is the divide and conquer algorithm paradigm. Because the divide and conquer tell that to divide the work and solve them and finally merge the solution. So when implementing the parallel version, the divided works are solved by the different computing cores and later they are merged. With these result, we take two widely used divide and conquer algorithm to analyse the performance: quicksort and convex hull. We have used the OpenMP programming for the implementation of the programs. Because this model also divides the loads among the independent working threads, solves the problem and finally merges them.

Chapter 3

Analysing a Divide and Conquer Algorithm: Quicksort

3.1 Introduction

Quicksort is one of the most used sorting algorithms. The performance quicksort variedly depends upon the nature of the input. The worst case happens when the input are reversely sorted. The applications where quicksort is used are the routing algorithm, the scheduling processes etc. It uses a partitioning method which places the pivot values at its correct position in each iteration.. The quicksort can be parallelized by parallel sorting the array and then merging the sorted array parallel.

Quicksort (A, p, r)

```
if p < r
    q = Partition(A, p, r);
    Quicksort(A, p, q);
    Quicksort(A, q+1, r;)
```

Partition(A, p, r)

```
    x = A[p] ;
    i = p  1;
    j = r + 1;
    while(1)
    j = j  1;
while A[j] <= x
    j = j  1;
     i = i + 1;
while A[i] >= x
    j = j  1;
if i < j
    swap (A[i], A[j]);
else
    return j;
```

Quicksort of 10 elements

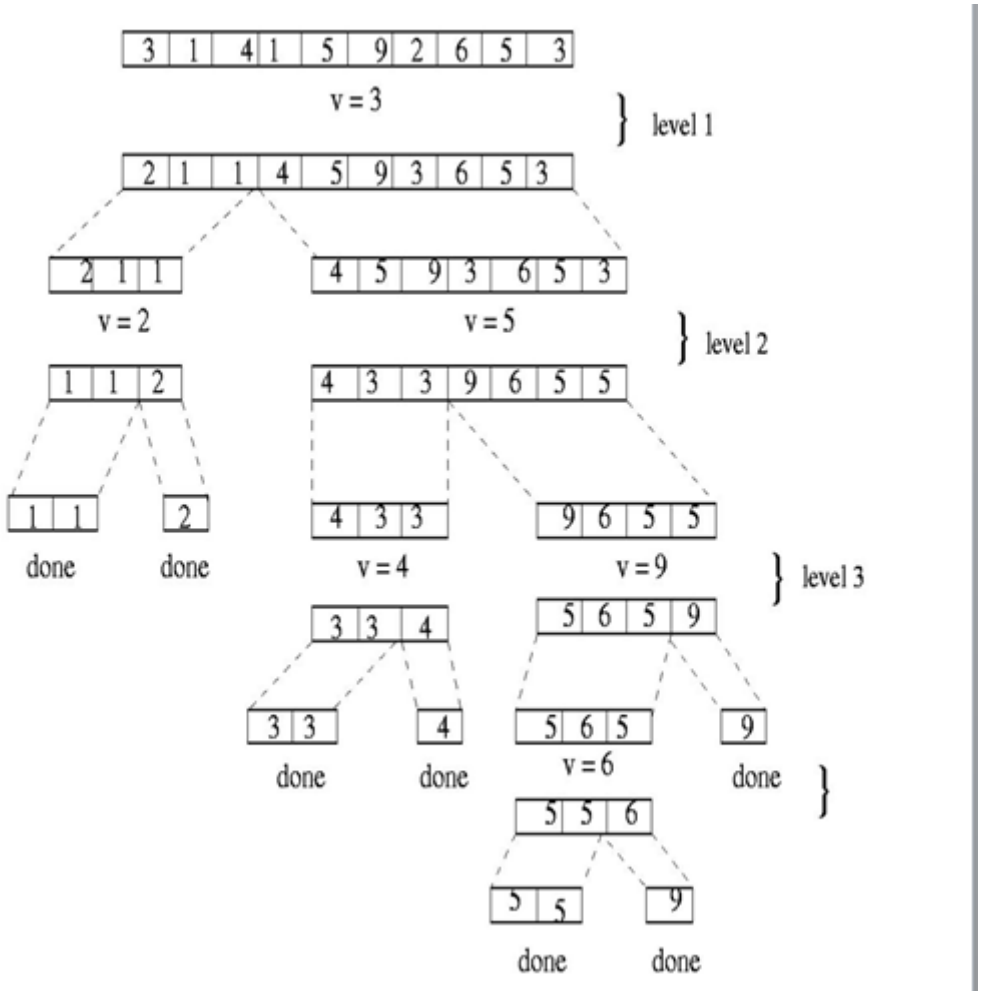

FIGURE 3.1: Quicksort of 10 elements[15]

3.2 Parallel Implementation of Quicksort

For the parallel implementation of quicksort, the algorithm is to divide the array among the different cores, parallel sort them and finally merge the sorted array into one. The algorithm for the parallel quicksort is as follows:

Partition(A, p, r)

```
k=getparallelthreads();
while k>=0
#doparallel
Quicksort(A,p,size%k);
k--;
k=getparallelthreads();
while k>=1
#doparallel
Merge(A,p,size%k);
```

Here the merge function is similar to that of the merge function in merge sort. The parallel sorting and merging reduces the time complexity which is O(n log n) in case of the sequential quicksort; but it is O(n/k log n/k) in case of parallel quicksort.

Parallel quick sort merging technique

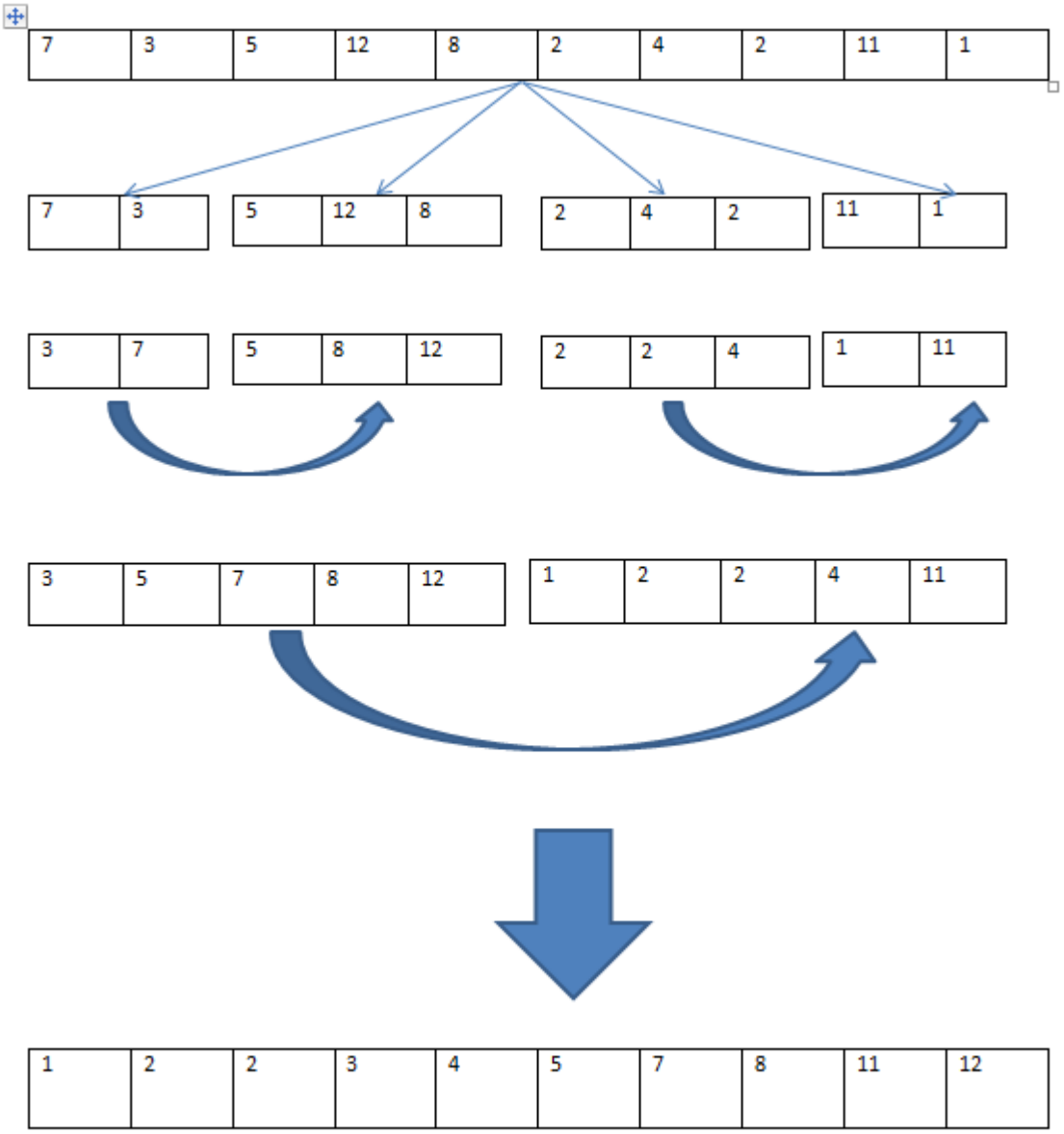

FIGURE 3.2: Parallel quick sort merging technique

3.2.1 Proposed Enhancement

The parallelization of quick sort here takes help of the merging method. The merging method is the same as that of the merge sort. So we propose a theory that will make the merging technique more efficient.

The merging in parallel quicksort is done on the sequential ordering of the indices. But this can lead to some performance loosing because we dont care about the elements behaviour when merging two arrays. To efficiently merge the arrays, we use two theories: mean and standard deviation of the distribution.
Mean of a distribution is defined as the average of the values inside a distribution where as standard deviation of a distribution gives how much each value differs from the mean in a distribution.
When merging two arrays, the mean of the sub-arrays are taken into consideration. The sub arrays are merged in the order from lowest to the highest mean. If there is a tie, then the standard deviation of the population is taken into account. The arrays with lowest standard deviation are merged first than highest standard deviation. The reason behind this is that when comparing the elements in the merging process, the tow arrays having nearer means will make lesser comparison than two arrays having mean more means. The process is illustrated as below.

Enhanced Parallel quick sort merging technique

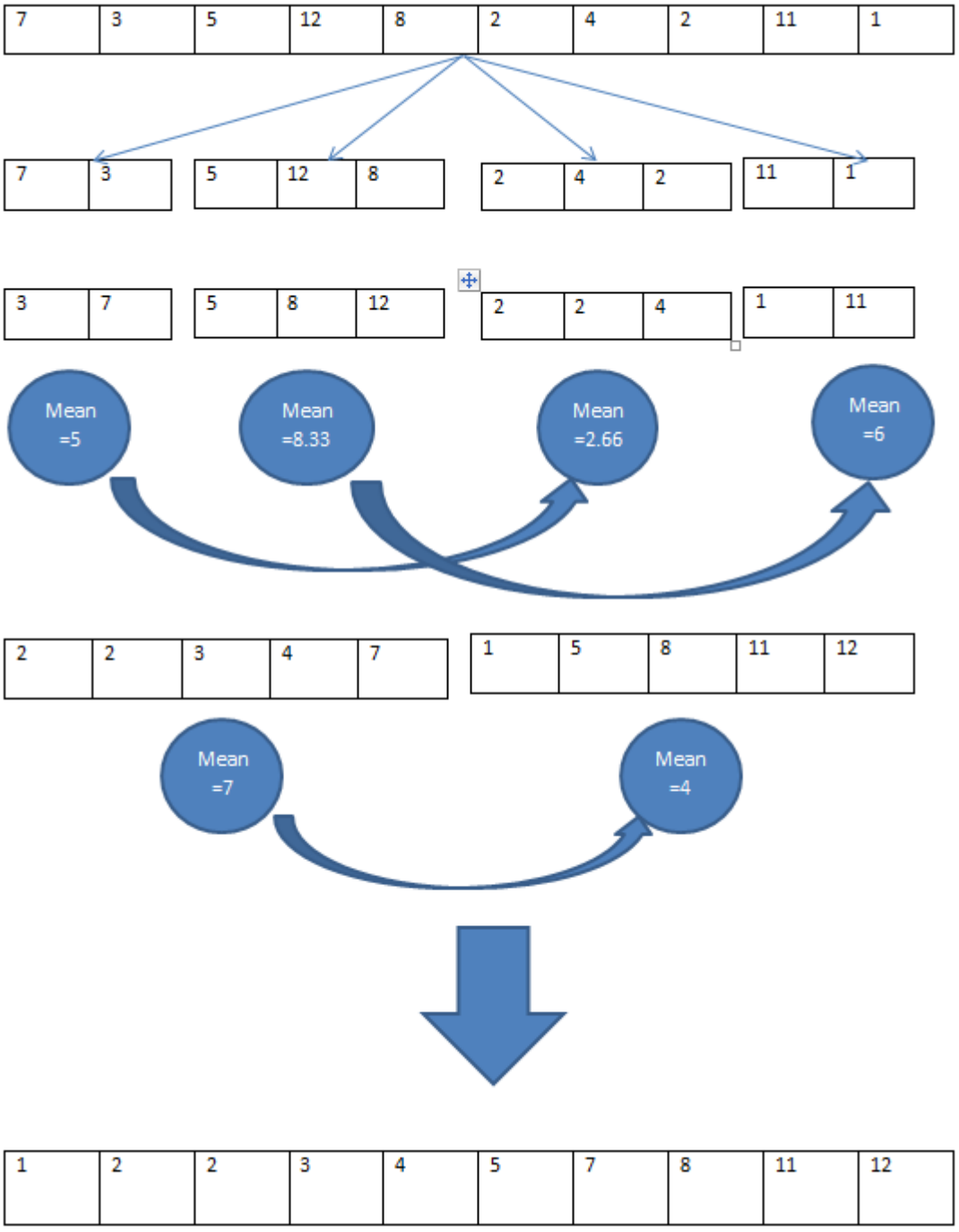

FIGURE 3.3: Enhanced Parallel quick sort merging technique

3.3 Simulation Setup

The total number of elements used for sorting were generated randomly and the number of elements varied from 1000 to 50000000.The simulation is done under the given processor architecture specification:

Processor

- Manufacturer : Intel
- Model : Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz
- Speed : 3.09GHz
- Cores per Processor : 2 Unit(s)
- Threads per Core : 2 Unit(s)

L1D (1st Level) Data Cache

- Size : 2x 32kB, 8-Way, 64bytes Line Size, 2 Thread(s)
- TLB - Translation Lookaside Buffer : 4kB : 64 4-Way (256kB); 2MB : 32 4-Way (64MB)

L1I (1st Level) Code Cache

- Size : 2x 32kB, 8-Way, 64bytes Line Size, 2 Thread(s)
- TLB - Translation Lookaside Buffer : 4kB : 64 4-Way (256kB); 2MB : 8 255-Way (16MB)

L2 (2nd Level) Data/Unified Cache

- Size : 2x 256kB, ECC, 8-Way, 64bytes Line Size, 2 Thread(s)
- TLB - Translation Lookaside Buffer : 4kB : 512 4-Way (2MB)

L3 (3rd Level) Data/Unified Cache

- Size : 3MB, ECC, 12-Way, Fully Inclusive, 64bytes Line Size, 16 Thread(s)

3.4 Performance Metric

The performance metric for this program is the total running time of the program. The run time of the program is defined as total time required to complete the process. This is taken as a performance metric because most of the divide and conquer algorithm are analysed on the basis of their run time complexities.

3.5 Simulation Results for parallelized Quick Sort

The run time of both sequential and parallel version of the quicksort are listed in a table as follows:

Table 3.1: Runtimes of sequential and parallel version of the quick sort

Total no. of elements	Runtime of sequential quicksort(in seconds)	Runtime of parallel (in seconds) quicksort(in seconds)
1000	0.000107596	0.00219961
10000	0.00133652	0.00912619
100000	0.0137406	0.009544
1000000	0.163225	0.0966222
10000000	1.85102	0.839037
20000000	3.86831	1.74822
50000000	10.1471	4.50473

Graph of Runtime analysis of sequential and parallel Quick Sort:

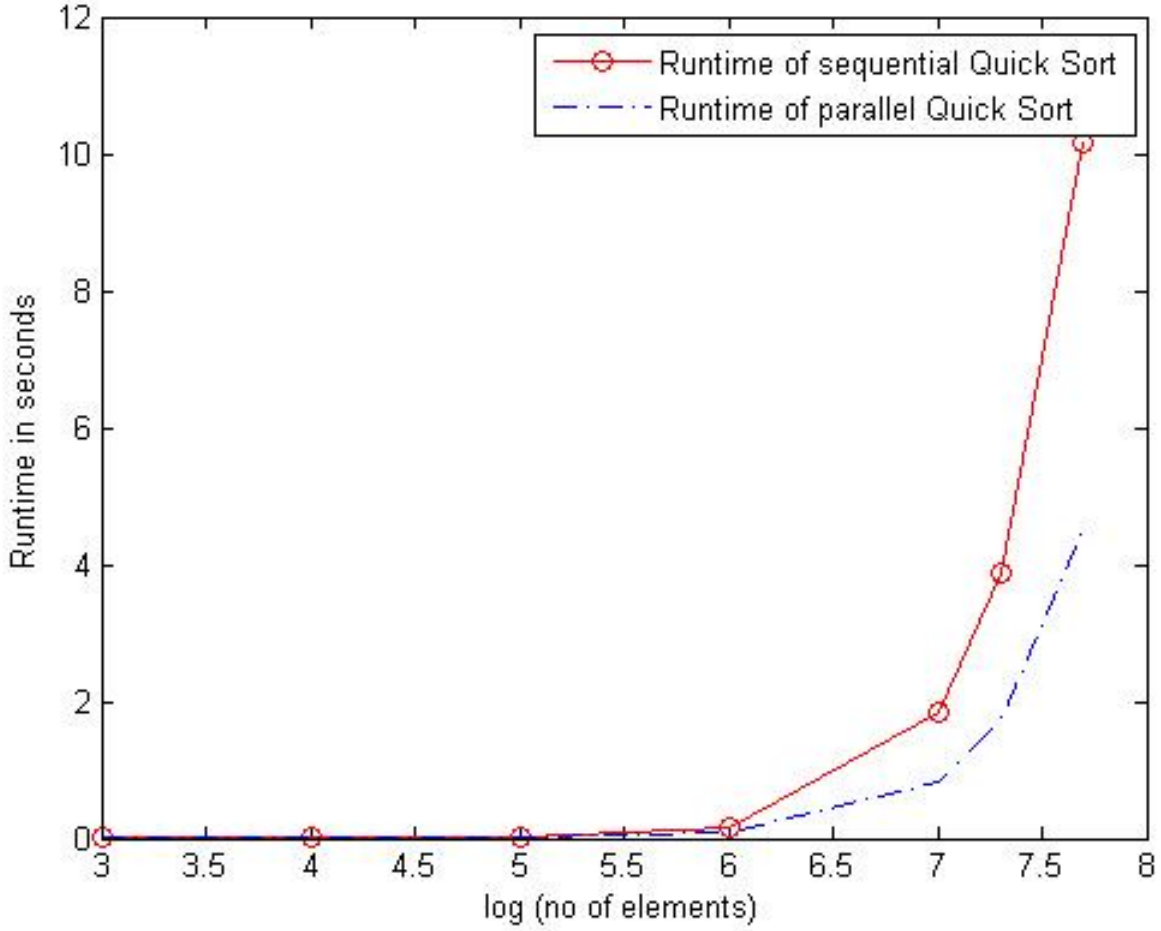

FIGURE 3.4: Graph of Runtime analysis of sequential and parallel Quick Sort

3.5.1 Simulation results for the Proposed Enhancement

The simulations were carried out under same processor architecture as given in the section 3.3. The performance metric for the simulation is taken as the no of comparisons in the merging phase. The following results were obtained in the process.

TABLE 3.2: No of comparisons for parallel and enhanced parallel version of the quick sort

Total no. of elements	No.of comparisons in quicksort(in seconds)	No.of comparisons in (in seconds) enhanced quicksort(in seconds)
1000	0.000107596	0.00219961
1000	1995	1995
10000	19988	19988
100000	199985	199486
1000000	1999996	19999886
10000000	19999994	19998563
20000000	39999995	39984562
50000000	99999997	99925866

3.6 Conclusion

The results from the above simulations tell the runtime of a parallel quick sort is dependent upon the total no of elements. The sequential version of the quicksort runs faster for less number of elements because the thread creation time is higher than the sorting of the elements in this case. The graphical results were drawn with the help of MATLAB.As the no of elements increases, the parallel version takes the advantage of multicore and the running time decreases. At 50 million elements the run time is reduced to as much as less than the half that of the sequential quicksort. In the case of the enhanced parallel quicksort, the total no of comparisons remain same for the lower no of inputs; but as the number of elements increases , the no of comparison decreases for the enhanced parallel quicksort. At 50 million elements , 0.07percent lesser comparisons required for the sorting method. The performance can be increased by increasing the no of core up to a certain limit after which the performance will remain constant.

Chapter 4

Analysing a Divide and Conquer Algorithm: Convex Hull

4.1 Introduction

Convex Hull is one of the most popular and widely used divide and conquer algorithm in the field of computing. The algorithm is mainly used for number plate detection, path routing and face recognition. As this is a divide and conquer algorithm, it becomes a candidate problem for the multicore programming.

ConvexHull(A,n)

1. Sort the n points according to x coordinate.
2. Divide the n points into two halves.
3. Find convex hull of each subset
4. Combine the two hulls into overall convex hull.

The sorting of the n points is done with the process of quicksort. So when implementing the parallel version of the quicksort, the parallel quick sort is used in this case. This divide and conquer convex hull algorithm have a limitation upon the input point set which tells that no three points in the point set can be collinear.

The merging process requires to find two tangents which are known as upper and lower tangents. These tangents are repetitively found and all the point occurring between these two tangents are discarded.

Set of points

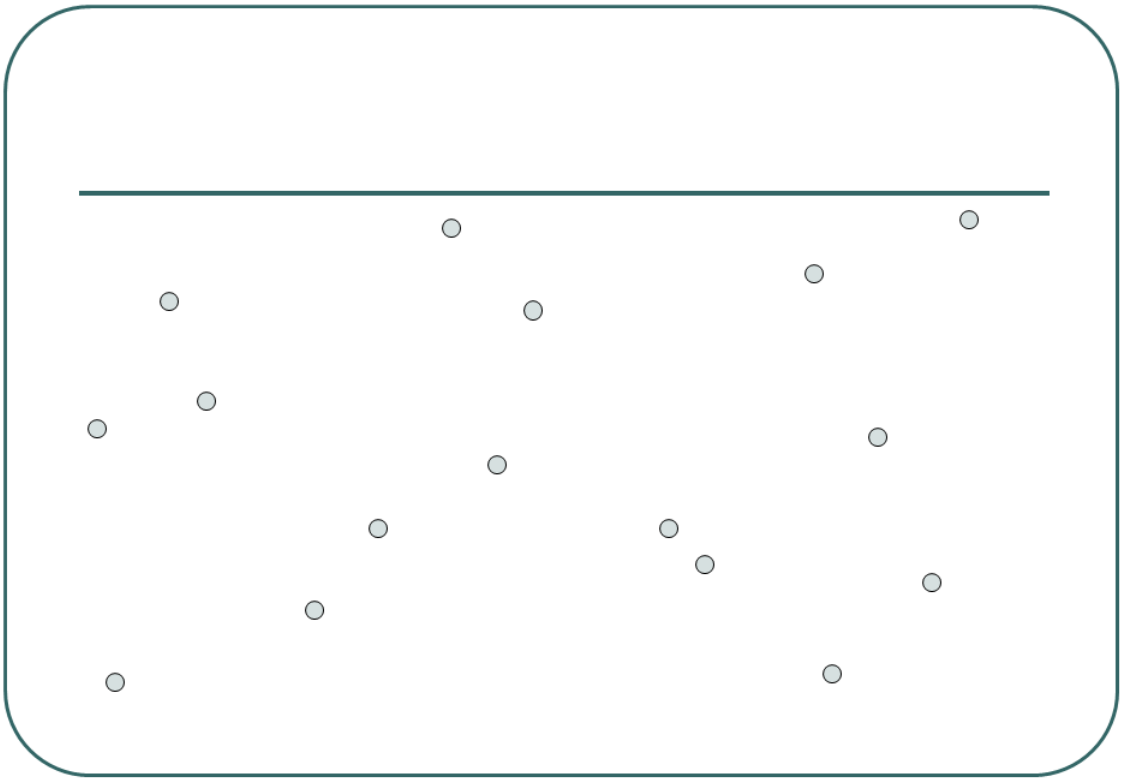

FIGURE 4.1: Set of points

Two subsets of for merging

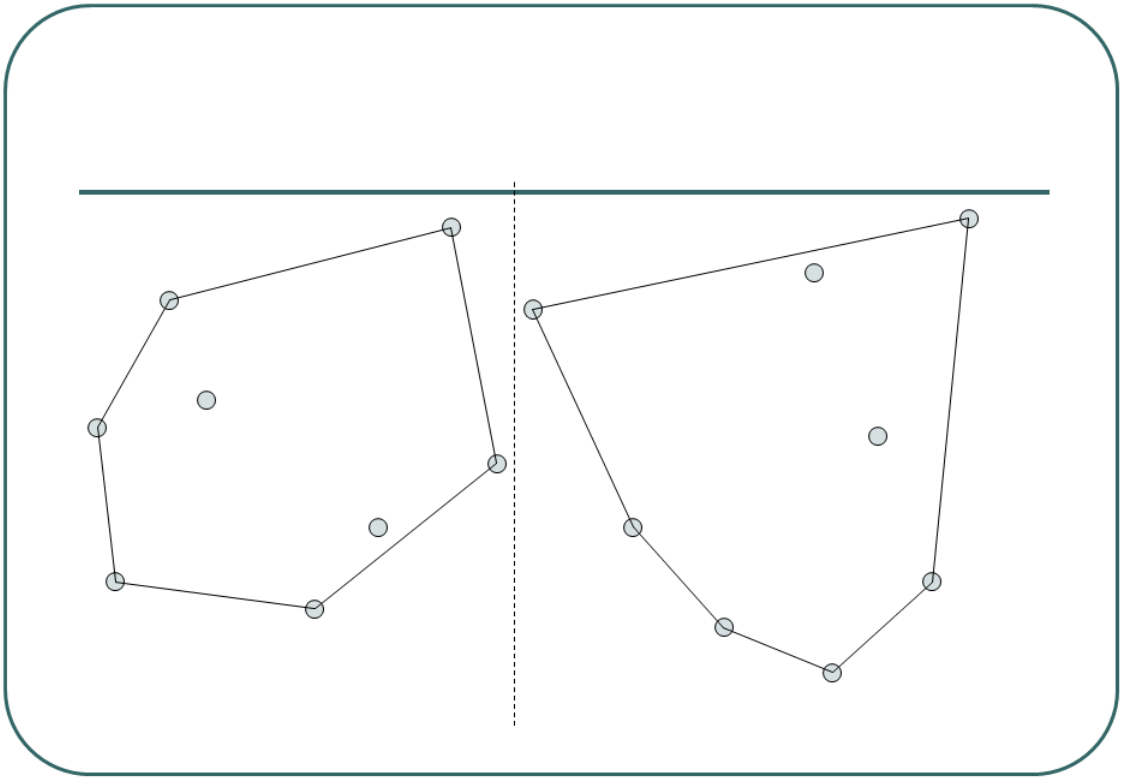

FIGURE 4.2: Two subsets of for merging

Upper and Lower tangents for the subsets

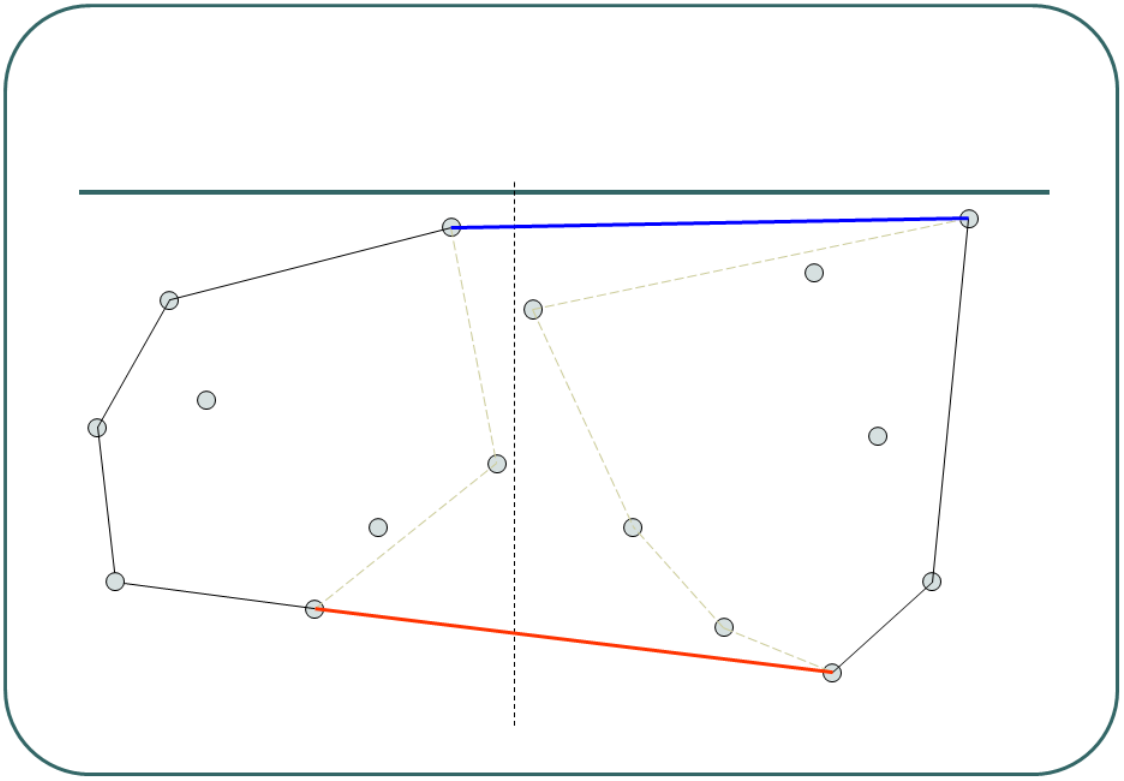

FIGURE 4.3: Upper and Lower tangents for the subsets

Resultant convex hull of the points

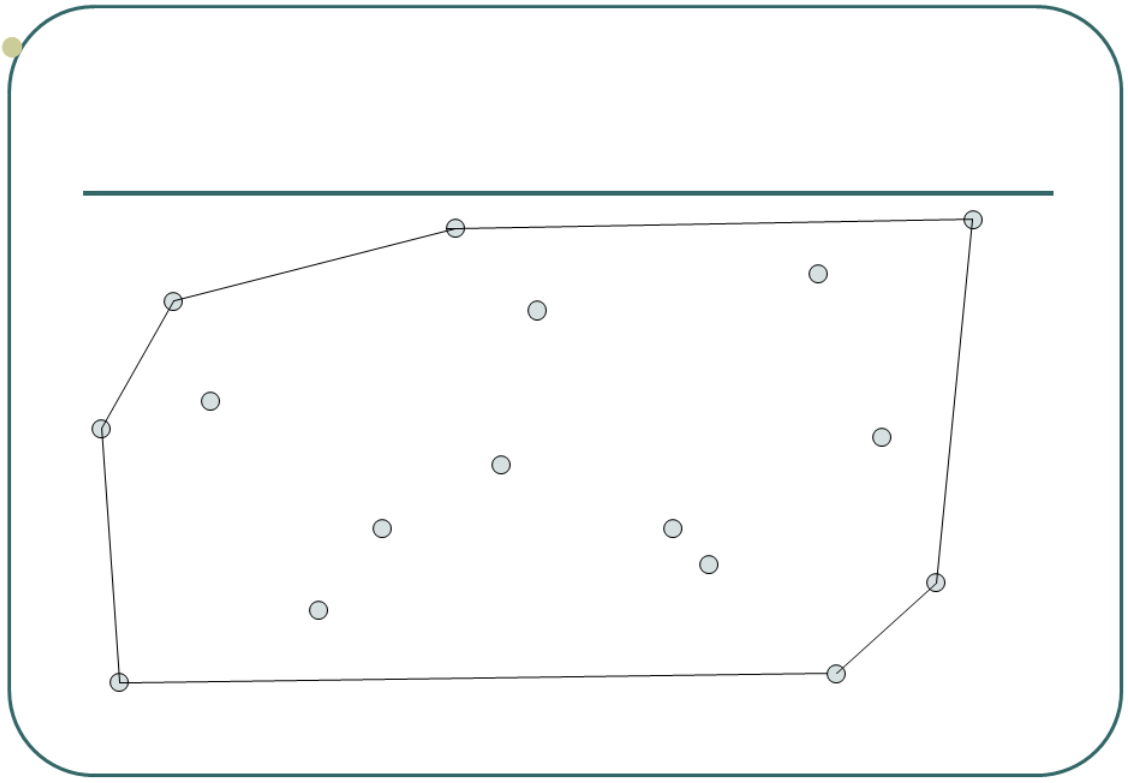

FIGURE 4.4: Resultant convex hull of the points

4.2 Parallel implementation of Convex Hull

For the parallel implementation of the convex hull, the points are sorted in the ascending order of the x- coordinate. This sorting is parallelized using the parallel quicksort as given in the chapter 3.Then the merging of the convex hulls are also done in parallel. The parallel version of convex hull algorithm is as follows:
ParallelConvexHull(A,n)

1. Sort the n points parallel according to x coordinate.
2. Distribute the divided sets among the cores.

3. Repeat steps 4 to 6 while all the cores have not found their convex hulls.
4. Divide the subsets into two halves.
5. Find convex hull of each subset
6. Combine the two hulls into overall convex hull.
7. Parallel merge the found convex hull by all the cores.

4.3 Simulation Setup

The total number of points used for convex hull were generated using a random generator from 1000 to 50 million and collinearity of ppoints were taken proper care.The simulation is done under the given processor architecture specification:
Processor

- Manufacturer : Intel
- Model : Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz
- Speed : 3.09GHz
- Cores per Processor : 2 Unit(s)
- Threads per Core : 2 Unit(s)

L1D (1st Level) Data Cache

- Size : 2x 32kB, 8-Way, 64bytes Line Size, 2 Thread(s)
- TLB - Translation Lookaside Buffer : 4kB : 64 4-Way (256kB); 2MB : 32 4-Way (64MB)

L1I (1st Level) Code Cache

- Size : 2x 32kB, 8-Way, 64bytes Line Size, 2 Thread(s)

- TLB - Translation Lookaside Buffer : 4kB : 64 4-Way (256kB); 2MB : 8 255-Way (16MB)

L2 (2nd Level) Data/Unified Cache

- Size : 2x 256kB, ECC, 8-Way, 64bytes Line Size, 2 Thread(s)
- TLB - Translation Lookaside Buffer : 4kB : 512 4-Way (2MB)

L3 (3rd Level) Data/Unified Cache

- Size : 3MB, ECC, 12-Way, Fully Inclusive, 64bytes Line Size, 16 Thread(s)

4.4 Performance Metric

The performance metric for this program is the total running time of the program. The run time of the program is defined as total time required to complete the process. This is taken as a performance metric because most of the divide and conquer algorithm are analysed on the basis of their run time complexities.

4.5 Simulation Results

The run time of both sequential and parallel version of the convex hull are listed in a table as follows:

TABLE 4.1: Runtimes of sequential and parallel convex hull

Total no. of points	Runtime of sequential convex hull(in seconds)	Runtime of parallel (in seconds) convex hull(in seconds)
1000	0.0007257	0.0032488
10000	0.0018876	0.0091677
100000	0.0157423	0.0125
1000000	0.36325986	0.1566432
10000000	2.05102432	1.0390328
20000000	4.1631765	1.9463653
50000000	11.7471532	4.9653865

Graph of Runtime analysis of sequential and parallel D-n-C Convex Hull:

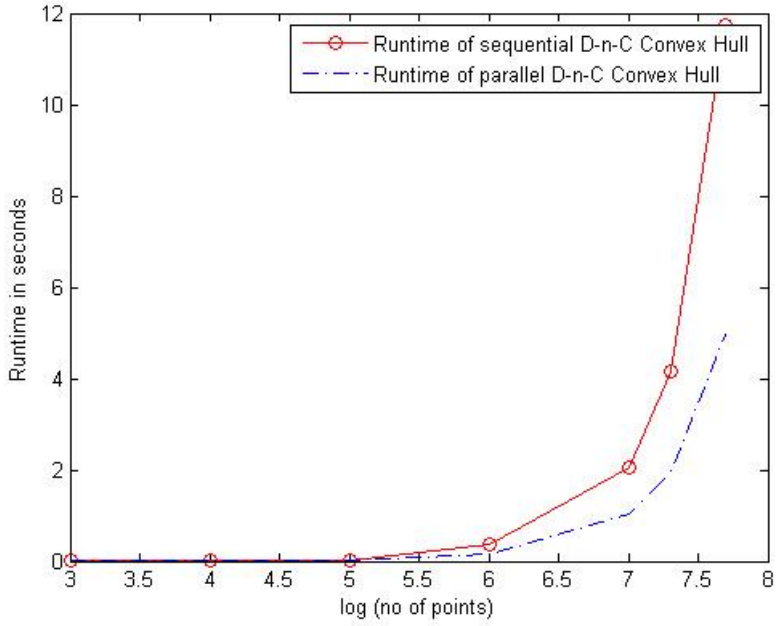

FIGURE 4.5: Graph of Runtime analysis of sequential and parallel D-n-C Convex Hull

4.6 Conclusion

The simulation results were carried out and found that the lower no of points have a better sequential run. Here 50 million points have nearly half execution time.The graphical results were drawn with the help of MATLAB.Here also the execution time for less number points is less for the sequential version because of the thread creation overhead. The performance can be increased by increasing the no of core upto a certain limit after which the performance will remain constant.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Two of the most commonly used divide and conquer algorithms, quick sort and convex hull, are implemented sequentially and parallel and both were analysed based upon their run time characteristic. An enhancement to the quick sort method is proposed which takes into consideration the total number of comparisons in the merging process. Both the programs showed a better run time for the sequential method for a lower number of inputs where as for the higher number of inputs, the parallel version seemed to be superior. So it is beneficial to use sequential version of the algorithms for a lower number of inputs and for higher number of inputs, the parallel version should be used.The scalability of the multicore platforms have also a significant contribution to the run time characteristic of the programs.

5.2 Future Work

However, the results obtained here can still be improved by efficient distribution of the loads among the multiple cores. The parallel merging can have a performance increment by consider other types of distribution which are better than mean and standard deviation methods. The convex hull algorithm which uses the concept of upper and lower tangent, can be parallelized for finding the upper and lower tangents in parallel.

Refernces

1. Introduction to Algorithms, T. Cormen, C. Leiserson and R. Rivest
2. Improving of Quicksort Algorithm Performance by Sequential Thread or Parallel Algorithms, Abdulrahman, Hamed Almutairi, Abdulrahman Helal Alruwaili,King Saud University,2012
3. A Simple Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000, P. Tsigas, Y. Zhang, 2003
4. Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting, David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider,1991
5. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures ,Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn,2007
6. Software behavior oriented parallelization. In Conference on Programming Language Design and Implementation, C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang,2007.
7. Sequential and parallel approximate Convex Hull algorithms, Chul E. Kim and Ivan Stojmenovic,1995.
8. Optimized Convex Hull With Mixed (MPI and OpenMP) Programming On HPC, Sandip V. Kendre and Dr. D.B. Kulkarni,2010
9. 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,2009.
10. Computing the convex hull of a sorted set of points on reconfigurable mesh, Parallel Algorithms and Applications, K. Nakano, pp. 243-250,1996.
11. Design and Experiment of a Communication-Aware Parallel Quicksort with Weighted Partition of Processors,Sangman Moh,Chansu Yu and Dongsoo Han,2004.
12. Parallel Quicksort in Hypercube,Youran Lan and Magdi A. Mohammed,1992.

13. Analysing the performance of Multicore Architecture,Prof. B.D. Sahoo,Prof. A.K. Turuk and Ram Prasad Mohanty,2012

14. Efficient Parallel Convex Hull Algorithms,R. Miller and Q.F. Stout,1988.

15. http://lcm.csa.iisc.ernet.in/dsa/node201.html

16. Multicore Programming Guide, Multicore Programming and Applications/DSP Systems, Texas Instruments,2012