B.Sc. in Computer Science and Engineering Thesis

Optimizing Divide and Conquer Based Algorithms: the Case of Heapsort

Submitted by

Abu Sa-Adat Mohammed Moinul Hasan Lutfor
201014009

Md. Shahidul Islam
201014015

Arup Kanti Dey
201014052

Supervised by

Dr. M. Kaykobad
Professor, Department of CSE
Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering
Military Institute of Science and Technology

CERTIFICATION

This thesis with title **"Optimizing Divide and Conquer Based Algorithms: the Case of Heapsort"**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering on December 2013.

Group Members:

Abu Sa-Adat Mohammed Moinul Hasan Lutfor
Md. Shahidul Islam
Arup Kanti Dey

Supervisor:

_____-

Dr. M. Kaykobad
Professor, Department of CSE
Bangladesh University of Engineering and Technology

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis is the outcome of the investigation and research carried out by the following students under the supervision of Dr. M. Kaykobad, Professor, Department of CSE, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

It is also declared that neither this thesis nor any part there of has been submitted anywhere else for the award of any degree, diploma or other qualifications.

_____

Abu Sa-Adat Mohammed Moinul Hasan Lutfor
201014009

_____

Md. Shahidul Islam
201014015

_____

Arup Kanti Dey
201014052

ACKNOWLEDGEMENT

We are thankful to Almighty Allah for His blessings for the successful completion of our thesis. Our heartiest gratitude, profound indebtedness and deep respect go to our supervisor Dr. M Kaykobad, Professor, Department of CSE, Bangladesh University of Engineering and Technology, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study was of great help in completing our thesis.

We are especially grateful to the Department of Computer Science and Engineering (CSE) of Military Institute of Science and Technology (MIST) for providing their all out support during the thesis work.

Finally, we would like to thank our families and our course mates for their appreciable assistance, patience and suggestions during the course of our thesis.

Dhaka
December 2013

Abu Sa-Adat Mohammed Moinul Hasan Lutfor
Md. Shahidul Islam
Arup Kanti Dey

ABSTRACT

The divide and conquer approach is very useful in computer science applications. With this approach a large problem is broken down into small and manageable subproblems and each is solved separately. Then these solutions are combined to give the final solution to the problem. This paper presents a discussion on the divide and conquer based techniques for various sorting algorithm with special emphasis on heap sort. Most divide and conquer approach divides a problem into two subproblems recursively. The thesis shows that ternary systems are more promising than the more traditional binary systems used in the divide and conquer approach. In particular, heap on ternary tree does indicate some theoretical advantages over the more established binary systems. The paper also makes an endeavour to present a set of simulations with random numbers for both integers and floating point numbers to support our theoretically proven claim. For doing so, at first algorithm for conventional binary heapsort has been modified for three and four child heapsort. Thereafter, the paper finds out number of comparisons required, number of movements required and time taken for all the three systems. Experimental results clearly state that in most cases three child heasort demonstrates better performance than both binary and four child version of heapsort. The experimental data with graphical representation have also been presented to support the analysis. This experiment can surely be a guide to establish more promising ternary system in applications related to divide and conquer technique with the motivation- 'A little advantage in optimization can be proven very useful in implementation of large systems'.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATION

BC : Before Christ

FFT : Fast Fourier Transform

VLSI : Very Large Scale Integration

CPU : Central Processing Unit

RAM : Random Access Memory

CH : Convex Hull

D&C : Divide and Conquer

CHAPTER 1

INTRODUCTION

1.1 Literature Review

In computer science, divide and conquer (D&C) [3], [25] is an important algorithm design paradigm based on multi-branched recursion. A D & C algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. This technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g. quicksort, merge sort), multiplying large numbers (e.g. Karatsuba [10]), syntactic analysis (e.g. top-down parsers), and computing the discrete Fourier transform. On the other hand, the ability to understand and design D & C algorithms is a skill that takes time to master. It is because when proving a theorem by induction, it is often necessary to replace the original problem by a more general or complicated problem in order to get the recursion going and there is no systematic method for finding the proper generalization.

The name "divide and conquer" is sometimes applied to algorithms that reduce each problem to only one subproblem, such as the binary search[26] algorithm for finding a record in a sorted list (or its analog in numerical computing, the bisection algorithm for root finding). These algorithms can be implemented more efficiently than general divide-and-conquer algorithms. Under this broad definition, however, every algorithm that uses recursion or loops could be regarded as a "divide and conquer algorithm". Therefore, some authors consider that the name "divide and conquer" should be used only when each problem may generate two or more subproblems. The name decrease and conquer has been proposed instead for the single-subproblem class. The correctness of a D & C algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

1.2 Early Historic Examples

Early examples of these algorithms are primarily decrease and conquer - the original problem is successively broken down into single sub problems, and indeed can be solved iteratively. Binary search, a decrease and conquer algorithm where the sub problems are of roughly half the original size, has a long history. While a clear description of the algorithm on computers appeared in 1946 in an article by John Mauchly, the idea of using a sorted list of items to facilitate searching dates back at least as far as Babylonia in 200 BC. Another ancient D & C algorithm is the Euclidean algorithm to compute the greatest common divisor of two numbers (by reducing the numbers to smaller and smaller equivalent subproblems), which dates to several centuries BC.

An early example of a divide-and-conquer algorithm with multiple subproblems is Gauss's 1805 description of what is now called the Cooley-Tukey fast Fourier transform (FFT) algorithm, although he did not analyze its operation count quantitatively and FFTs did not become widespread until they were rediscovered over a century later. An early two-subproblem D & C algorithm that was specifically developed for computers and properly analyzed is the merge sort algorithm, invented by John von Neumann in 1945. Another notable example is the algorithm invented by Anatolii A. Karatsuba in 1960 that could multiply two n-digit numbers in $O(n^{log_2 3})$ operations. This algorithm disproved Andrey Kolmogorov's 1956 conjecture that $\Omega(n^2)$ operations would be required for that task.

As another example of a D & C algorithm that did not originally involve computers, Knuth[17] gives the method a post office typically uses to route mail: letters are sorted into separate bags for different geographical areas, each of these bags is itself sorted into batches for smaller sub-regions, and so on until they are delivered. This is related to a radix sort, described for punch-card sorting machines as early as 1929.

1.3 Scopes and Objectives

With the invention of computers, two parametric algebra, number system, and graphs among other systems started to flourish with accelerated speed. Boolean algebra got, its important, applications in computer technology, binary number system has occupied the core of computer arithmetic, and binary trees have become inseparable in mathematical analysis of

complexity of algorithms and in the development of efficient algorithms. Since 2 has been being used as a parameter having significant, influence in the efficiency of the concerned algorithms, the claim of its supremacy over other values should be subject to rigorous verification.

Megiddo[20] has placed an objection to the standard translation of problems into languages via the binary coding.Extensive works have already been done on optimality of ternary trees[6] and their VLSI embedding[24]. In this paper we have made simplified theoretical analyses on several problems, where 2 is being used as an algorithmic parameter, to see whether some other values are more promising. We have achieved some positive results in favor of 3 as an algorithmic parameter and these results have been supported by the experimental data.

This paper discusses about the widely used D & Cs based algorithms including their complexity. The scope of this paper is limited to theoretical and experimental analysis of Heap sort by varying the number of child. From the analysis it is observed that in many cases 3 child Heap sort is exhibiting better result than 2 or 4 child Heap sort.

CHAPTER 2

DIVIDE AND CONQUER TECHNIQUES

2.1 Divide and Conquer

Divide and conquer is an established technique for designing effective algorithm for uniprocessors. The basic D & C scheme can be stated as "Given a problem, Divide it into independent subproblems of the same type, once these subproblems been solved (using the D & C scheme recursively), combine their solutions into a solution to the original problem." For example, a D & C approach to summing numbers in a list of is to halve the list, sum the halves and then add the two results. Divide and conquer is called perhaps the most important and most widely applicable technique for designing efficient algorithms.

The divide-and-conquer strategy solves a problem by:
1. Breaking the problem into subproblems which are smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.
The real work is done piecemeal, in three different places: in the partitioning of problems into sub problems; at the end of the recursion, when the sub problems are so small that they are solved outright; and in the gluing together of partial answers. These are held together and coordinated by the algorithm's core recursive structure.

From a theoretical point of view, D & C is an attractive algorithm abstraction. It is computed by solving the following recurrence equation:

T(n)= branchingFactor(N)* T(subproblemSize(N))+ combineCost(N)

2.1.1 Advantages

Solving Difficult Problems

Divide and conquer is a powerful tool for solving conceptually difficult problems: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem. Similarly, D & C only requires reducing the problem to a single smaller problem, such as the classic Tower of Hanoi puzzle, which reduces moving a tower of height n to moving a tower of height $n - 1$.

Algorithm Efficiency

The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsuba's fast multiplication method, the quicksort and mergesort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms. In all these examples, the divide-and-conquer approach led to an improvement in the asymptotic cost of the solution. For example, if the base cases have constant-bounded size, the work of splitting the problem and combining the partial solutions is proportional to the problem's size n, and there are a bounded number p of sub problems of size n/p at each stage, then the cost of the divide-and-conquer algorithm will be O(n log n).

Parallelism

Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors. This approach leads to a quicker solution of the problem as solutions of the subproblems become available much faster and almost together than the sequential solution by a single processor.

Memory Access

Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called cache-oblivious, because it does not contain the cache size(s) as an explicit parameter. Moreover, D & C algorithms can be designed for important algorithms (e.g. sorting, FFTs, and matrix multiplication) to be optimal cache-

oblivious algorithms-they use the cache in a provably optimal way, in an asymptotic sense, regardless of the cache size. In contrast, the traditional approach to exploiting the cache is blocking, as in loop nest optimization, where the problem is explicitly divided into chunks of the appropriate size this can also use the cache optimally, but only when the algorithm is tuned for the specific cache size(s) of a particular machine.

2.1.2 Implementation Issues

Recursion

Divide-and-conquer algorithms are naturally implemented as recursive procedures. In that case, the partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack.

Explicit stack

Divide and conquer algorithms can also be implemented by a non-recursive program that stores the partial sub-problems in some explicit data structure, such as a stack, queue, or priority queue. This approach allows more freedom in the choice of the sub-problem that is to be solved next, a feature that is important in some applications - e.g. in breadth-first recursion and the branch and bound method for function optimization. This approach is also the standard solution in programming languages that do not provide support for recursive procedures.

Stack Size

In recursive implementations of D & C algorithms, one must make sure that there is sufficient memory allocated for the recursion stack, otherwise the execution may fail because of stack overflow. Fortunately, D & C algorithms that are time-efficient often have relatively small recursion depth. For example, the quicksort algorithm can be implemented so that it never requires more than nested recursive calls to sort items.

Stack overflow may be difficult to avoid when using recursive procedures, since many compilers assume that the recursion stack is a contiguous area of memory, and some allocate a fixed amount of space for it. Thus, the risk of stack overflow can be reduced by minimizing the parameters and internal variables of the recursive procedure, and/or by using an explicit stack structure.

Choosing the Base Cases

In any recursive algorithm, there is considerable freedom in the choice of the base cases, the small sub problems that are solved directly in order to terminate the recursion. Choosing the smallest or simplest possible base cases is more elegant and usually leads to simpler programs, because there are fewer cases to consider and they are easier to solve. For example, an FFT algorithm could stop the recursion when the input is a single sample, and the quick sort list-sorting algorithm could stop when the input is the empty list; in both examples there is only one base case to consider, and it requires no processing.

On the other hand, efficiency often improves if the recursion is stopped at relatively large base cases, and these are solved non-recursively, resulting in a hybrid algorithm. This strategy avoids the overhead of recursive calls that do little or no work, and may also allow the use of specialized non-recursive algorithms that, for those base cases, are more efficient than explicit recursion. A general procedure for a simple hybrid recursive algorithm is short-circuiting the base case, also known as arm's-length recursion. In this case whether the next step will result in the base case is checked before the function call, avoiding an unnecessary function call. For example, in a tree, rather than recursing to a child node and then checking if it is null, checking null before recursing; this avoids half the function calls in some algorithms on binary trees. Since a D & C algorithm eventually reduces each problem or sub-problem instance to a large number of base instances, these often dominate the overall cost of the algorithm, especially when the splitting/joining overhead is low.

2.2 Quicksort

Quick sort is a divide-and-conquer sorting algorithm in which division is dynamically carried out (as opposed to static division in Merge sort). The three steps of Quick sort are as follows:

Divide: Rearrange the elements and split the array into two sub arrays and an element in between such that so that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element.

Conquer: Recursively sort the two sub arrays.

Combine: None.

Algorithm 1 Quick Sort

$Quicksort(A,n)$

$Quicksort'(A,l,n)$

$Quicksort'(A,p,r)$

if $p \geq r$ **then**

    return

end if

$q = Partition(A,p,r)$

$Quicksort'(A,p,q-1)$

$Quicksort'(A,q+1,r)$

The Subroutine Partition

Given a sub array A [p r] such that p $\leq$ r - 1, This subroutine rearranges the input sub array into two sub arrays, A [p.. q - 1] and A[q +1 .. r], so that each element in A[p..q - 1] is less than or equal to A[q] and each element in A[q +1.. r] is greater than or equal to A[q] Then the subroutine outputs the value of q.

Use the initial value of A[r] as the pivot, in the sense that the keys are compared against it. Scan the keys A [p... r - 1] from left to right and flush to the left all the keys that are greater than or equal to the pivot.

Algorithm 2 Partition

$Partition(A,p,r)$

$x = A[r]$

$i \leftarrow p-1$

for $j \leftarrow p$ to r-1 **do**

if $A[j] \Leftarrow x$ **then**

    $i \leftarrow i+1$

    $Exchange\ A[i] \leftrightarrow A[j]$

end if

$Exchange\ A[i+1] \leftrightarrow A[r]$

return $i+1$

During the for loop i+1 is the position at which the next key that is greater than or equal to the pivot should go to.

An Example:

q

17 9 22 31 7 12 10 21 13 29 18 20 11

9 17 22 31 7 12 10 21 13 29 18 20 11

9 7 22 31 17 12 10 21 13 29 18 20 11

9 7 10 31 17 12 22 21 13 29 18 20 11

pivot=11 r

9 7 10 11 17 12 22 21 13 29 18 20 31

Another Example:

17 9 22 31 7 12 10 21 13 29 18 20 23

17 9 22 31 7 12 10 21 13 29 18 20 23

pivot=23 r

17 9 22 31 7 12 10 21 13 29 18 20 23

17 9 22 31 7 12 10 21 13 29 18 20 23

17 9 22 7 31 12 10 21 13 29 18 20 23

17 9 22 7 12 31 10 21 13 29 18 20 23

17 9 22 7 12 10 31 21 13 29 18 20 23

2.2.1 Proving Correctness of Partition

Let (A, p, r) be any input to Partition and let q be the output of Partition on this input. Suppose $1 \leq p < r$. Let x = A[r]. We will prove the correctness using loop invariant. The loop invariant we use is: at the beginning of the for-loop, for all k, $p \leq k \leq r$, the following properties hold:

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i+1 \leq k \leq j - 1$, then $A[k] > x$.
3. If k = r, then A[k] = x.

Initialization

The initial value of i is p - 1 and the initial value of j is p. So, there is no k such $p \leq k \leq I$ and there is no k such that $i+1 \leq k \leq j - 1$. Thus, the First conditions are met. The initial value of A[r] = x, is so the last one is met.

Maintenance

Suppose that the three conditions are met at the beginning and that j $\leq$ r - 1. Suppose that A[j] $>$ x. The value of i will not be changed, so (1) holds. The value of j becomes j +1. Since A[j] $>$ x, (2) will for the new value of j. Also, A[r] is unchanged so (3) holds. Suppose that A[j] $\leq$ x. Then A [i+1] and A[j] will be exchanged. By (2), A [i+1] $>$ x. So, after exchange A[i+1] $\leq$ x and A[j] $>$ x. Both i and j will be incremented by 1, so (1) and (2) will be preserved. Again (3) still holds.

Termination

At the end, j = r. So, for all k, 1 $\leq$ k $\leq$ i, A[k] $\leq$ x and for all k, i+1 $\leq$ k $\leq$ r - 1, A[k] $>$ x.

2.2.2 Running Time Analysis

The running time of quick sort is a linear function of the array size, r - p+1, and the distance of q from p, q - p. This is Θ(r - p+1). What are the worst cases of this algorithm?

Worst-case analysis Let T be the worst-case running time of Quick sort. Then
T(n) = T(1)+T(n - 1)+ Ω(n):

2.3 Mergesort

In computer science, a mergesort (also commonly spelled merge sort) is an O (n log n) comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Merge sort is a D & C algorithm that was invented by John von Neumann in 1945. Conceptually, a merge sort works as follows:
1. Divide the unsorted list into n sub lists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sub lists to produce new sorted sub lists until there is only 1 sub list remaining. This will be the sorted list.

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub problems, we state each sub problem as sorting a sub arrayA[p .. r]. Initially, p = 1 and r = n, but these values change as we recurse through sub problems. To sort A[p .. r]:

Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A [p .. r] into two subarrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

Conquer Step

Conquer by recursively sorting the two subarraysA[p .. q] and A[q + 1 .. r].

Combine Step

Combine the elements back in A[p .. r] by merging the two sorted subarrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, a procedure is defined as MERGE (A, p, q, r).

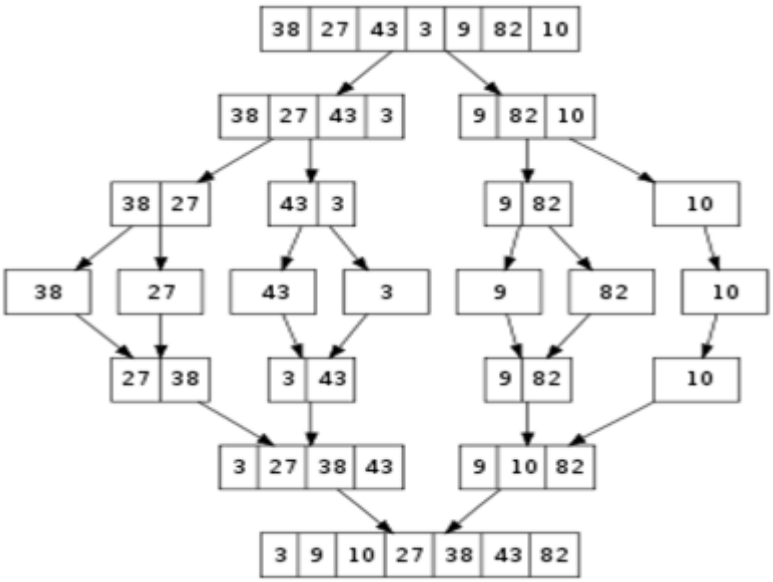

Figure 2.1: Recursive Mergesort Algorithm Used to Sort an Array of 7 Integer Values

2.3.1 Algorithm: Merge Sort

Algorithm 3 Mergesort

$MERGESORT(A, p, r)$

if $p < r$ **then**

$q = \lfloor \frac{p+r}{2} \rfloor$

$MERGE(A, p, q)$

$MERGE(A, q+1, r)$

$MERGE(A, p, q, r)$

$MERGE(A, p, q, r)$

$n1 \leftarrow q - p + 1$

$n2 \leftarrow r - q$

Create arrays $L[1....n1+1]$ *and* $R[1....n2+1]$

for $i \leftarrow 1$ *to* $n1$ **do**

$L[i] \leftarrow A[p+i-1]$

for $j \leftarrow 1$ *to* $n2$ **do**

$R[j] \leftarrow A[q+j]$

$L[n1+1] \leftarrow \infty$

$R[n2+1] \leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

for $k \leftarrow p$ *to* r **do**

if $L[i] \leq R[j]$ **then**

$A[k] \leftarrow L[i]$

$i \leftarrow i+1$

else $A[k] \leftarrow R[j]$

$j \leftarrow j+1$

Analysis

In sorting n objects, merge sort has an average and worst-case performance of O (n log n). If the running time of merge sort for a list of length n is T(n), then the recurrence T(n) = 2T(n/2) + n follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem. In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than $(n\lceil logn\rceil - 2^{\lceil logn\rceil} + 1)$, which is between (n log n - n + 1) and (n log n + n + O (log n)). For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches αn fewer than the worst case where

$$\alpha = -1 + \sum_{k=0}^{k=\infty} \frac{1}{2k+1} \approx 0.2645$$

In the worst case, merge sort does about 39% fewer comparisons than quick sort does in the average case. In terms of moves, merge sort's worst case complexity is O(n log n)-the same complexity as quick sort's best case, and merge sort's best case takes about half as many iterations as the worst case. Merge sort is more efficient than quick sort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quick sort, merge sort is a stable sort as long as the merge operation is implemented properly. Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only n/2 extra spaces). Merge sort also has some demerits. One is its use of 2n locations; the additional n locations are commonly used because merging two sorted sets in place is more complicated and would need more comparisons and move operations. But despite the use of this space the algorithm still does a lot of work: The contents of m are first copied into left and right and later into the list result on each invocation of merge sort.

2.3.2 Variants

Primary Concern: Reducing the Space Complexity and the Cost of Copying

A simple alternative for reducing the space overhead to n/2 is to maintain left and right as a combined structure, copy only the left part of m into temporary space, and to direct the merge routine to place the merged output into m. With this version it is better to allocate the temporary space outside the merge routine, so that only one allocation is needed. The excessive copying mentioned previously is also mitigated, since the last pair of lines before the return result statement (function merges in the pseudo code above) become superfluous.

An alternative to reduce the copying into multiple lists is to associate a new field of information with each key (the elements in m are called keys). This field will be used to link the keys and any associated information together in a sorted list (a key and its related information is called a record). Then the merging of the sorted lists proceeds by changing the link values; no records need to be moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used. This is a standard sorting technique, not restricted to merge sort.

Optimizing merge sort

On modern computers, locality of reference can be of paramount importance in software optimization, because multilevel memory hierarchies are used. Cache-aware versions of the merge sort algorithm, whose operations have been specifically chosen to minimize the movement of pages in and out of a machine's memory cache, have been proposed. For example, the tiled merge sort algorithm stops partitioning sub arrays when sub arrays of size S are reached, where S is the number of data items fitting into a CPU's cache. Each of these sub arrays is sorted with an in-place sorting algorithm such as insertion sort, to discourage memory swaps, and normal merge sort is then completed in the standard recursive fashion. Also, many applications of external sorting use a form of merge sorting where the input get split up to a higher number of subsists, ideally to a number for which merging them still makes the currently processed set of pages fit into main memory.

Comparison with other sort algorithms

Although heap sort has the same time bounds as merge sort, it requires only $\Theta(1)$ auxiliary space instead of merge sort's $\Theta(n)$, and is often faster in practical implementations. On typical modern architectures, efficient quick sort implementations generally outperform merge sort for sorting RAM-based arrays. On the other hand, merge sort is a stable sort and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for

sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

Utility in online sorting

Merge sort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning. In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list.

2.4 Convex Hull

2.4.1 Convexity

A convex is a polygon in which any line joining two points within the polygon lies within the polygon. Alex-Ander Kalashnikov[18] gives similar definitions of convexity, the first is that, a subset set S of the plane is called convex if and only if for any pair of points P, $Q \in S$ the line segment PQ is completely contained in S. The second goes as, a set S is convex if it is exactly equal to the inter-section of all the half planes containing it. The figures below show convex and non-convex shapes.

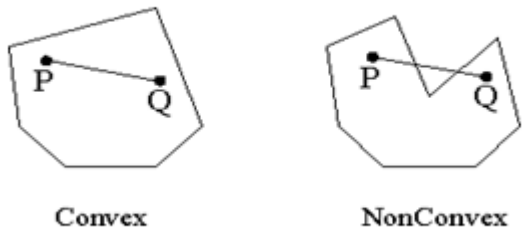

Figure 2.2: Convex and Non-Convex Shapes

Convex Hull

Convex Hull of a set Q of points is the smallest convex polygon P, for which each point in

Q is either on the boundary of P or in its interior. In line with the above[21], asserts that the convex hull of a set S of points, denoted hull(S) is the smallest polygon P for which each point of S is either on the boundary or in the interior of P. Similarly, establishes that the convex hull CH(P) of a finite point set P is the smallest convex polygon that contains P. Figure 2.3 shows a convex hull P.

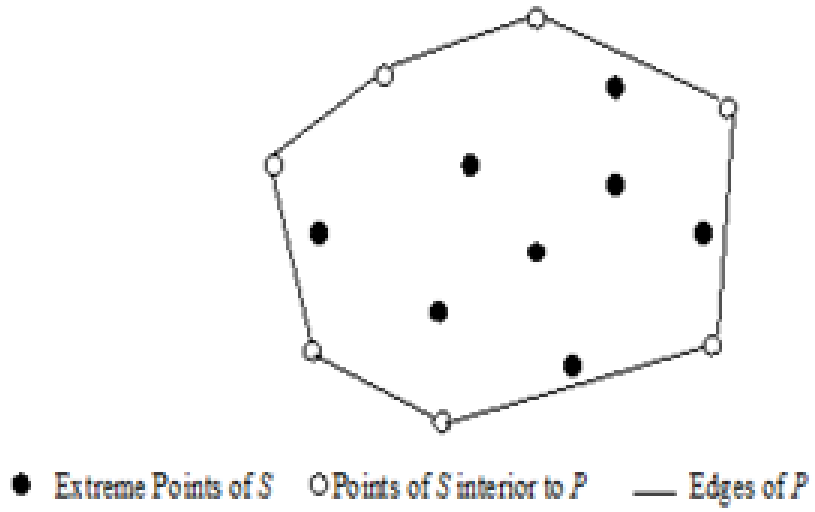

Figure 2.3: Convex Hull P

Intuition

If each point is represented by a nail sticking out from a board and you take a rubber band and lower it over the nails, so as to completely encompass the set of nails, and then let the rubber band naturally contract, the rubber band will give you the edges of the convex hull of the set of points, and those nails that correspond to a change in slope of the rubber band represent the extreme points of the convex hull.

2.4.2 Convex Hull Problem

In a convex hull problem a set of points or coordinates are given and one is asked to come up with the smallest polygon in which all points are either inside or on the edge of the polygon.

Finding the convex hull using divide and conquer

Souvaine[28] establishes that in order to find the convex hull using a divide-and-conquer

approach, one has to follow these steps: sort points (p1, p2, . . . , pn) by their x-coordinate recursively find the convex hull of p1 through pn/2 recursively find the convex hull of pn/2 +1 through pn merge the two convex hulls A more detailed algorithm for finding a Convex Hull is presented in [22]. The algorithm is as follows: Hull(S): If —S— ≤ 3, then compute the convex hull by brute force in O(1) time and return. Otherwise, partition the point set S into two sets A and B, where A consists of half the points with the lowest x co-ordinates and B consists of half of the points with the highest x co-ordinates. Recursively compute HA = Hull (A) and HB = Hull (B). Merge the two hulls into a common convex hull, H, by computing the upper and lower tangents for HA and HB and discarding all the points lying between these two tangents.

2.4.3 Running Times of Convex Hull Divide and Conquer Algorithm

Sequential Algorithm

Sorting requires O (n log n) time. However, it is critical to note that the sort only needs to be performed once at the beginning of the algorithm as a pre-processing step and not every time through the recursion. The stitch step requires $\Theta(logn)$ time to determine the lines of support (i.e., the common tangent lines) and $\Theta(n)$ time to reorder (compress) the data. So, there is $\Theta(nlogn)$ pre-processing time and T (n) = 2T (n/2) + O (n) = O (nlog n) running time. Therefore, the running time for the algorithm is optimal at $\Theta(nlogn)$.

Mesh Algorithm

Given n points, arbitrarily distributed one point per processor on a mesh of size n, the convex hull of the set S of points can be determined in $\Theta(n^{\frac{1}{2}})$ time. First, sort the points into shuffled row major order so that the following holds (geometric points on left side of figure mapped to mesh quadrants as shown in right side of figure 2.4):

Binary Search

Binary search from S1 to S2 and from S2 to S1, where after each iteration, the remaining data is compressed together so that the running time of the binary search is linear in the edge length of the sub-mesh that the points initially reside in. Broadcast the 4 tangent points. Eliminate points inside of quadrilateral and renumber. Running time of binary search is B(n) = B(n / 2) +$\Theta(n^{\frac{1}{2}})$ = $\Theta(n^{\frac{1}{2}})$. So, the running time of the divide-and-conquer convex

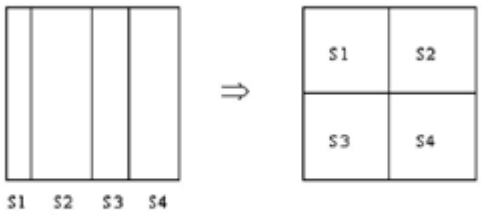

Figure 2.4: Mesh Algorithm

hull algorithm on a mesh of size n is T (n) = T (n / 4) + B(n) = T (n / 4) +$\Theta(n^{\frac{1}{2}})$ = $\Theta(n^{\frac{1}{2}})$, plus $\Theta(n^{\frac{1}{2}})$ for the initial sort, which is $\Theta(n^{\frac{1}{2}})$.

2.4.4 Advantages of the Divide and Conquer Approach

Algorithm efficiency

The Quick Hull recursively subdivides point set S, and assembles the convex hull H(S) by "merging" the sub- problem results or partial solutions. The base case requires O(1), constant-bounded time. Thus the D & C algorithm will have O (n log n) complexity.

Subdivision allows Parallelism

Divide-and-conquer algorithms are adapted for execution in multi-processor machines, especially shared memory systems as in the testing of robots using convex hulls; where the communication of data between processors does not need to be planned in advance. Thus distinct sub-problems can be executed on different processors.

Ideal for solving difficult and complex problems

Divide and conquer is a powerful tool for solving conceptually difficult problems, such as the classic Tower of Hanoi puzzle: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem.

Memory access

Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in princi-

ple, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called cache oblivious, because it does not contain the cache size(s) as an explicit parameter.

2.4.5 Disadvantages of the Divide and Conquer Approach

Recursion which is the basis of D & C is slow, the overhead of the repeated subroutine calls, along with that of storing the call stack. Inability to control or guarantee sub-problem size results in sub-optimum worst case time performance. Requires a lot of memory for storing intermediate results of sub-convex hulls to be combined to form the complete convex hull. The use of D & C is not ideal if the points to be considered are too close to each other such that other approaches to convex hull will be ideal Is complicated if a large base cases are to implemented for performance reasons.

2.4.6 Algorithm Performances for Convex Hull Problem

Table 2.1: Algorithms for convex hull problem

Algorithm	**Speed**	**Discovered by**
Brute Force	O(n)	[Anon,the dark]
Gift Wrapping	O(nh)	[Chan & Kapur,1970]
Graham Scan	O(n log n)	[Graham, 1972]
jarvis March	O(nh)	[Jarvis, 1973]
Quick Hull	O(nh)	[Eddy, 1977]
Divide-and-Conqure	O(n log n)	[Preparata]
Monotone	O(n log n)	[Andrew, 1979]
Incremental	O(n log n)	[Kallay, 1984]
Marriage	O(n log n)	[Kirkpatrick]

2.5 Closest Pair

The closest pair of points problem or closest pair problem is a problem of computational geometry[16]: given n points in metric space, find a pair of points with the smallest distance between them. The closest pair problem for points in the Euclidean plane was among the first geometric problems which were treated at the origins of the systematic study of the computational complexity of geometric algorithms. A naive algorithm of finding distances between all pairs of points and selecting the minimum requires $O(dn^2)$ time. It turns out that the problem may be solved in O(n log n) time in a Euclidean space or space of fixed dimension d. In the algebraic decision tree model of computation, the O(n log n) algorithm is optimal. In the computational model which assumes that the floor function is computable in constant time the problem can be solved in O(n log log n) time. If we allow randomization to be used together with the floor function, the problem can be solved in O(n) time[12].

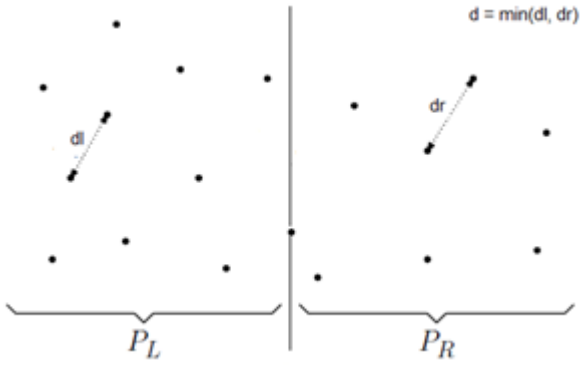

Figure 2.5: Closest Pair in Separate Partition

2.5.1 Divide and Conquer for Closest Pair of Points

Given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, it may be required to monitor planes that come too close together, since this may indicate a possible collision. We know the following formula for distance between two points p and q.

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

The Brute force solution is $O(n^2)$, compute the distance between each pair and return the smallest. The smallest distance can be calculated in O(n log n) time using D & C strategy. Here a $O(n(logn)^2)$ approach is discussed.

2.5.2 Algorithmic Steps

Following are the detailed steps of a $O(n(logn)^2)$ algorithm.

Input: An array of n points P [].

Output: The smallest distance between two points in the given array.

As a pre-processing step, input array is sorted according to x coordinates.
1) Find the middle point in the sorted array, we can take P [n/2] as middle point.

2) Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P [n/2+1] to P [n-1].

3) Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d.

4) From above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from left half and other is from right half. Consider the vertical line passing through passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip [] of all such points.

5) Sort the array strip [] according to y coordinates. This step is O(nlogn). It can be optimized to O (n) by recursively sorting and merging.

6) Find the smallest distance in strip []. This is tricky. From first look, it seems to be a $O(n^2)$ step, but it is actually O(n). It can be proved geometrically that for every point in strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate).

7) Finally return the minimum of d and distance calculated in above step (step 6).

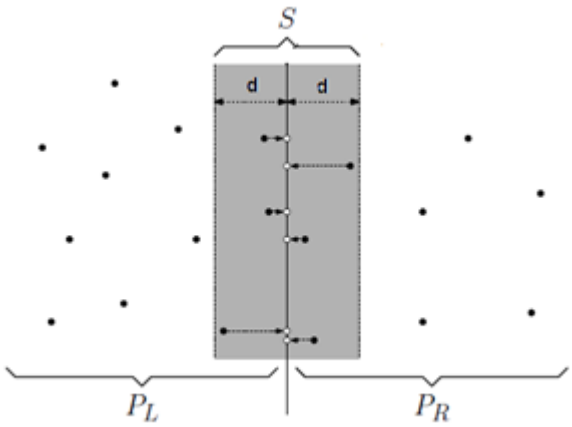

Figure 2.6: Closest Pair Across the Partitions

Algorithm 4 Closest pair

$closestPair\ of\ (xP, yP)$

$where\ xP\ is\ P(1).....P(N)\ sorted\ by\ x\ Coordinate,\ and$

$yP\ is\ P(1)....P(N)\ sorted\ by\ y\ Coodinate$

if $N \leq 3$ **then**

return $closest\ points\ of\ xP\ using\ brute-force\ algorithm$

else

$xL \leftarrow points\ of\ xP\ from\ 1\ \text{to}\ [\frac{N}{2}]$

$xR \leftarrow points\ of\ xP\ from\ 1\ \text{to}\ [\frac{N}{2}]+1\ to\ N$

$xm \leftarrow xP(\frac{N}{2})_x$

$yL \leftarrow \{p \in yP : p_x \leq xm\}$

$yR \leftarrow \{p \in yP : p_x > xm\}$

$(dL, pairL) \leftarrow closestPair\ of(xL, yL)$

$(dR, pairR) \leftarrow closestPair\ of(xR, yR)$

$(dmin, pairMin) \leftarrow (dR, pairR)$

if $dL < dR$ **then**

$(dMin, pairMin) \leftarrow (dL, pairL)$

end if

$yS \leftarrow \{p \in yP : |xm - p_x| < dmin\}$

$nS \leftarrow number\ of\ points\ in\ yS$

$(closest, closestPair) \leftarrow (dmin, pairMin)$

for $i\ from\ 1\ \text{to}\ nS-1$

$k \leftarrow i+1$

while $k < leqnS$ and $yS(k)_y - yS(i)_y < dmin$

if $|yS - yS| < closest$ **then**

$(closest, closestPair) \leftarrow (|yS(k) - yS(i)|, \{yS(k)yS(i)\})$

endif

$k \leftarrow k+1$

endwhile

endfor

return $closest, closestPair$

end if

CHAPTER 3

HEAPSORT AND ITS OPTIMIZATION WITH THREE CHILD ANALYSIS

3.1 Heapsort

Heap sort is a comparison-based sorting algorithm to create a sorted array (or list), and is part of the selection sort family. Although somewhat slower in practice on most machines than a well-implemented quick sort, it has the advantage of a more favorable worst-case O(n log n) runtime. The heap sort algorithm can be divided into two parts. In the first step, a heap is built out of the data. In the second step, a sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. Once all objects have been removed from the heap, we have a sorted array. The direction of the sorted elements can be varied by choosing a min-heap or max-heap in step one[3]. Heap sort can be performed in place. The array can be split into two parts, the sorted array and the heap. The storage of heaps as arrays is diagrammed here. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.

3.1.1 Heap Property

In a heap, for every node i other than the root, the value of a node is greater than or equal (at most) to the value of its parent.

$$A[\text{PARENT } (i)] \geq A[i]$$

Thus, the largest element in a heap is stored at the root[7].

By the definition of a heap, all the tree levels are completely filled except possibly for the lowest level, which is filled from the left up to a point. Clearly a heap of height h has the

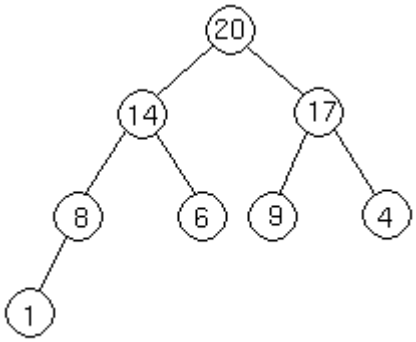

Figure 3.1: Heapsort Property

minimum number of elements when it has just one node at the lowest level. The levels above the lowest level form a complete binary tree of height h -1 and 2^{h-1} nodes. Hence the minimum number of nodes possible in a heap of height h is 2^h. Clearly a heap of height h, has the maximum number of elements when its lowest level is completely filled. In this case the heap is a complete binary tree of height h and hence has $2^{h+1} - 1$ nodes[1].

Height of a node

We define the height of a node in a tree to be a number of edges on the longest simple downward path from a node to a leaf.

Height of a tree

The number of edges on a simple downward path from a root to a leaf is the height of a tree. Note that the height of a tree with n node is $log_2 n$ which is $O(log_2 n)$. This implies that an n-element heap has height $log_2 n$. In order to show this let the height of the n-element heap be h. From the bounds obtained on maximum and minimum number of elements in a heap,

$$2^h \leq n \leq 2^{h+1} - 1$$

Where n is the number of elements in a heap[4].

$$2^h \leq n \leq 2^{h+1}$$

Taking logarithms to the base 2

$$h \leq log_2 n \leq h + 1$$

It follows that $h = \lceil log_2 n \rceil$.

Maintaining the Heap Property

Heapify is a procedure for manipulating heap data structures. It is given an array A and index i into the array. The subtree rooted at the children of A[i] are heap but node A[i] itself may possibly violate the heap property i.e., A[i] $<$ A[2i] or A[i] $<$ A[2i +1]. The procedure 'Heapify' manipulates the tree rooted at A[i] so it becomes a heap. In other words, 'Heapify' is let the value at A[i] "float down" in a heap so that subtree rooted at index i becomes a heap[23].

3.2 An Example on Heapsort

Let 6, 5, 3, 1, 8, 7, 2, 4 be the list that we want to sort from the smallest to the largest.

Table 3.1: Build heap

Heap	**Newly added element**	**swap elements**
Nil	6	
6	5	
6,5	3	
6,5,3	1	
6,5,3,1	8	
6,5,3,1,8		5,8
6,5,3,1,5		6,8
8,6,3,1,5	7	
8,6,3,1,5,7		3,7
8,6,7,1,5,3	2	
8,6,7,1,5,3,2	4	
8,6,7,1,5,3,2,4		1,4
8,6,7,4,5,3,2,1		

Now, with the given data the heap is built. Data need to be sorted and for that following procedure is followed. Each step is described in a row with explanation in the last column.

Table 3.2: Sorting

Heap	**Swap**	**Delete**	**Sorted array**	**Details**
8,6,7,4,5,3,2,1	8,1			swap 8 and 1 in order to delete 8 from heap
1,6,7,4,5,3,2,8		8		delete 8 from heap and add to sorted array
1,6,7,4,5,3,2	1,7		8	swap 1 and 7 to correct order in the heap
7,6,1,4,5,3,2	1,3		8	swap 1 and 3 to correct order in the heap
7,6,3,4,5,1,2	7,2		8	swap 7 and 2 in order to delete 7 from heap
2,6,3,4,5,1,7		7	8	delete 7 from heap and add to sorted array
2,6,3,4,5,1	2,6		7,8	swap 2 and 6 to correct order in the heap
6,2,3,4,5,1	2,5		7,8	swap 2 and 5 to correct order in the heap
6,5,3,4,2,1	6,1		7,8	swap 6 and 1 in order to delete 6 from heap
1,5,3,4,2,6		6	7,8	delete 6 from heap and add to sorted array
1,5,3,4,2	1,5		6,7,8	swap 1 and 5 to correct order in the heap
5,1,3,4,2	1,4		6,7,8	swap 1 and 4 to correct order in the heap
5,4,3,1,2	5,2		6,7,8	swap 5 and 2 in order to delete 5 from heap
2,4,3,1,5		5	6,7,8	delete 5 from heap and add to sorted array
2,4,3,1	2,4		5,6,7,8	swap 2 and 4 to correct order in the heap
4,2,3,1	4,1		5,6,7,8	swap 4 and 1 in order to delete 4 from heap
1,2,3,4		4	5,6,7,8	delete 4 from heap and add to sorted array
1,2,3	1,3		4,5,6,7,8	swap 1 and 3 to correct order in the heap
3,2,1	3,1		4,5,6,7,8	swap 3 and 1 in order to delete 3 from heap
1,2,3		3	4,5,6,7,8	delete 3 from heap and add to sorted array
1,2	1,2		3,4,5,6,7,8	swap 1 and 2 to correct order in the heap
2,1	2,1		3,4,5,6,7,8	swap 2 and 1 in order to delete 2 from heap
1,2		2	3,4,5,6,7,8	delete 2 from heap and add to sorted array
1		1	2,3,4,5,6,7,8	delete 1 from heap and add to sorted array
			1,2,3,4,5,6,7,8	completed

3.3 Theoretical Complexity of Heapsort

In HeapSort the algorithm operated by first building a heap in a bottom-up manner, and then repeatedly extracting the maximum element from the heap and moving it to the end of the array. One clever aspect of the data structure is that it resides inside thearray to be sorted. We argued that the basic heap operation of Heapify runs in O(log n) time, because the heap has O(log n) levels, and the element being sifted moves down one level of the tree after a constant amount of work.

Based on this we can see that (1) that it takes O(nlogn) time to build a heap, because we need to apply Heapify roughly n=2 times (to each of the internal nodes), and (2) that it takes O(nlogn) time to extract each of the maximum elements, since we need to extract roughly n elements and each extraction involves a constant amount of work and one Heapify. Therefore the total running time of HeapSort is O(nlogn).

In fact, it can be found out that it is not possible to sort faster than $\Omega(nlogn)$ time, assuming that comparisons are used, which HeapSort does. However, it turns out that the first part of the analysis is not tight. In particular, the BuildHeap procedure that we presented actually runs in (n) time. Although in the wider context of the HeapSort algorithm this is not significant (because the running time is dominated by the (nlogn) extraction phase). Nonetheless there are situations where it might not be needed to sort all of the elements. For example, it is common to extract some unknown number of the smallest elements until some criterion (depending on the particular application) is met. For this reason it is nice to be able to build the heap quickly since it may not be needed to extract all the elements.

3.3.1 Build Heap Analysis

For BuildHeap analysis the most convenient assumption is that n is of the form $n = 2^{h+1} - 1$, where h is the height of the tree. The reason is that a left-complete tree with this number of nodes is a complete tree, that is, its bottommost level is full. This assumption nullify the requirement of using floors and ceilings. With this assumption, level 0 of the tree has 1 node, level 1 has 2 nodes, and up to level h, which has 2^h nodes. All the leaves reside on level h.

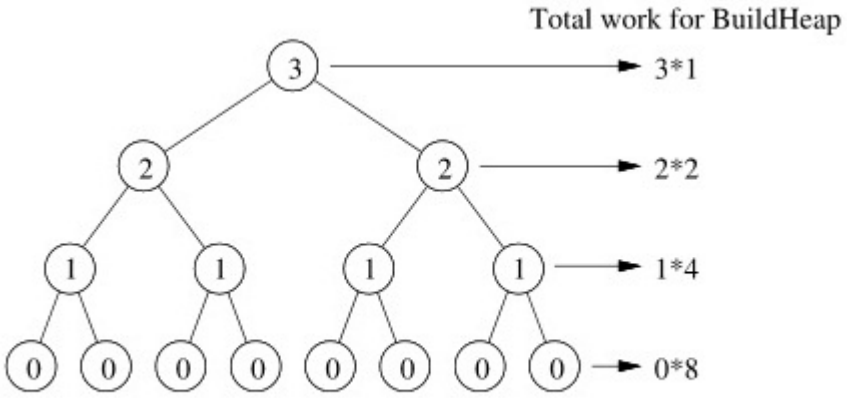

Figure 3.2: Heapsort Tree

When Heapify is called, the running time depends on how far an element might shift down before the process terminates. In the worst case the element might shift down all the way to the leaf level. Let us count the work done level by level. At the bottommost level there are 2^h nodes, but we do not call Heapify on any of these so the work is 0. At the next to bottommost level there are 2^{h-1} nodes, and each might sift down 1 level. At the 3rd level from the bottom there are 2^{h-2} nodes, and each might sift down 2 levels. In general, at level j from the bottom there are 2^{h-j} nodes, and each might shift down j levels. So, if we count from bottom to top, level-by-level, we see that the total time is proportional to

$$T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j}$$

If we factor out the 2^h term, we have

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

This is a sum that we have never seen before. We could try to approximate it by an integral, which would involve integration by parts, but it turns out that there is a very cute solution to this particular sum. We'll digress for a moment to work it out. First, write down the infinite general geometric series, for any constant x$<$ 1.

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$$

Then taking the derivative of both sides with respect to x, and multiply by x giving:

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2} \qquad \sum_{j=0}^{\infty} jx^{j} = \frac{x}{(1-x)^2}$$

and putting x = 1/2, we have the desired formula:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{\frac{1}{2}}{1-(\frac{1}{2})^2} = \frac{\frac{1}{2}}{\frac{1}{4}} = 2$$

In our case we have a bounded sum, but since the infinite series is bounded, we can use it instead as an easy approximation. Using this we have

$$T(n) = 2^h \ \sum_{j=0}^{h} \frac{j}{2^j} \ \leq \ \sum_{j=0}^{\infty} \frac{j}{2^j} \ \leq \ 2^h.2 = 2^{h+1}$$

Now we know that $n = 2^{h+1} - 1$, so we have $T(n) \leq n + 1 \in O(n)$. Clearly the algorithm takes at least $\Omega(n)$ time (since it must access every element of the array at least once) so the total running time for BuildHeap is $\Theta(n)$.

3.4 Comparison with Other Sorting Algorithm

Heapsort primarily competes with quicksort, another very efficient general purpose nearly-in-place comparison-based sort algorithm. Quicksort is typically somewhat faster due to some factors, but the worst-case running time for quicksort is $O(n^2)$, which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk. Quicksort represrents a detailed discussion of this problem and possible solutions[11]. Thus, because of the O(n log n) upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort[5]. Heapsort also competes with merge sort, which has the same time bounds. Merge sort requires $\Omega(n)$ auxiliary space, but heapsort requires only a constant amount. Heapsort typically runs faster in practice on machines with small or slow data caches. On the other hand, merge sort has several advantages over heapsort: Merge sort on arrays has considerably better data cache performance, often outperforming heapsort on modern desktop computers because merge sort frequently accesses contiguous memory locations (good locality of reference); heapsort

references are spread throughout the heap[19]. Heapsort is not a stable sort; merge sort is stable[26]. Merge sort parallelizes well and can achieve close to linear speedup with a trivial implementation; heapsort is not an obvious candidate for a parallel algorithm[2]. Merge sort can be adapted to operate on linked lists with O(1) extra space.[4] Heapsort can be adapted to operate on doubly linked lists with only O(1) extra space overhead[27]. Merge sort is used in external sorting; heapsort is not. Locality of reference is the issue[8]. Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort[9].

3.5 Optimality of Ternary Heapsort

Ternary heapsort uses a ternary heap instead of a binary heap; that is, each element in the heap has three children. It is more complicated to program, but does a constant number of times fewer swap and comparison operations. This is because each step in the shift operation of a ternary heap requires three comparisons and one swap, whereas in a binary heap two comparisons and one swap are required. The ternary heap does two steps in less time than the binary heap requires for three steps, which multiplies the index by a factor of 9 instead of the factor 8 of three binary steps. Ternary heapsort is about 12% faster than the simple variant of binary heapsort[9].

CHAPTER 4

EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Experimental Setup

With the invention of computer the use of two parametric algebra, number system, graph, tree is started rapidly. Binary system in particular binary tree established its supremacy in D & C method and in other system. In sorting D & C method is most popular way. Here we divide the given list of data and again merge them in sorted way. Binary system is used here when to divide the data and when to merge them. Megiddo has placed an objection to the standard translation of problems into languages via the binary coding. Extensive works have already been done on optima and of ternary trees and their VLSI embedding. In this paper we show that ternary system is more promising than traditional binary system applying on heap sort.

Let us consider a tree consist of n node and each of node have d child and the total height of the tree is h. for our theory we can consider following picture

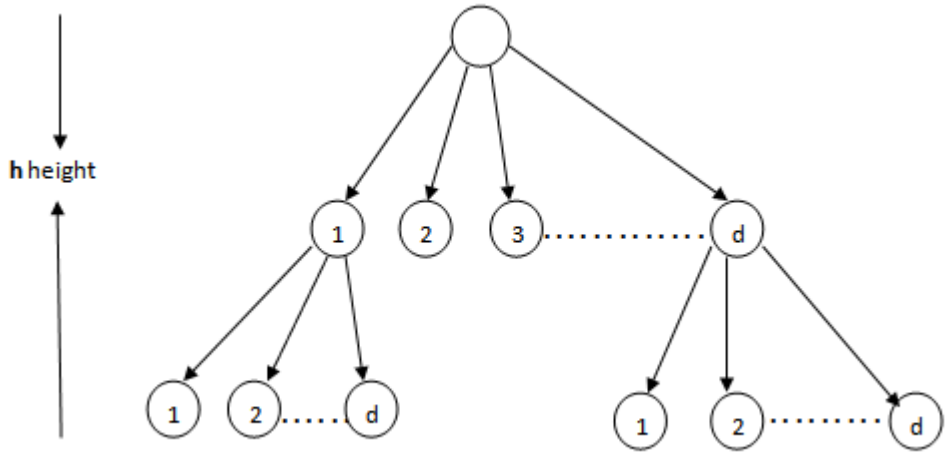

Figure 4.1: A Tree with Height h and Branching Factor d

$$n \leq d^0 + d^1 + ... + d^{h-1}$$
$$or, n \leq \frac{d^h - 1}{d-1}$$
$$or, d^h - 1 \geq n(d-1)$$
$$or, h = \lceil log_d\{n(d-1)+1\}\rceil$$

now, at each level there will be at most d-1 comparison to find the elder son and 1 comparison to decide whether to swap or not. So there can be d Comparisons per level. So, total comparisons for this tree will be dh.

$$dh = d * (ln_d n + ln_d d - 1) \approx d\frac{ln\, n}{ln\, d}$$

So cost= (dln n)/lnd where ln n is constant[13]. Therefore cost will vary depending on value of d/ln d. In this equation we can put several values and try to find the optimal number of child to reduce the cost. We have found that the cost value increases for larger value of d except for 3 which is less than both d=2 and d=4. So, we get the following relation which supports our claim.

$$\frac{2}{ln2} > \frac{3}{ln3} < \frac{4}{ln4}$$

This relation clearly indicates that 3 child or ternary heapsort theoritically is more optimal than other variants of heapsort. So, attempt is made to support this claim by experimental data.

4.2 Algorithms for Varying Child

We have the algorithm of heapsort for two child . For our experimental purpose we have modified it for three child and four child . In two child heap sort algorithm we compare left child and right child with their parent. In three child we get three child as left, middle and right child and for four child we have got four child named as left, left middle, right middle and right child. The algorithm we used for our experiment is given below.

Algorithm 5 HEAPSORT FOR TWO CHILD

$Parent(i)$

return $\lfloor \frac{i}{2} \rfloor$

$Left(i)$

return $2i$

$Right(i)$

return $2i+1$

BUILD _ HEAP (A)

$heap-size(A) \leftarrow length[A]$

for $i \leftarrow \left\lfloor \frac{length[A]}{2} \right\rfloor$ downto 1 **do**

$HEAPIFY(A,i)$

HEAPIFY(A, i)

$l \leftarrow Left(i)$

$r \leftarrow Right(i)$

if $l \leq heap-size[A]$ *and* $A[l] > A[i]$ **then**

$largest \leftarrow l$

else

$largest \leftarrow i$

end if

if $r \leq heap-size[A]$ *and* $A[r] > A[i]$ **then**

$largest \leftarrow r$

end if

if $largest \neq i$ **then**

exchange $A[i] \leftrightarrow A[largest]$

end if

$HEAPIFY(A,i)$

HEAPSORT(A)

$BUILD\_HEAP(A)$

for $i = length[A]$ down to 2 **do**

exchange $A \leftrightarrow A[i]$

$heap-size[A] \leftarrow heap-size[A]-1$

$Heapify(A,1)$

For three child version of the algorithm, we need to define an extra child which is named 'middle'here. In this model height of the tree is reduced which contributes in reducing cost. At the same time in the heapify phase each time one extra comparison is required. After the modification the algorithm looks like the one presented below.

Algorithm 6 HEAPSORT FOR THREE CHILD

$Parent(i)$

return $\lfloor \frac{i+1}{3} \rfloor$

$Left(i)$

return $3 * i - 1$

$Middle(i)$

return $3 * i$

$Right(i)$

return $3i + 1$

BUILD _ HEAP (A)

$heap - size(A) \leftarrow length[A] - 1$

for $i \leftarrow \left\lfloor \frac{length[A]}{3} + 1 \right\rfloor$ downto 1 **do**

$HEAPIFY(A, i)$

HEAPIFY(A, i)

$l \leftarrow Left(i)$

$m \leftarrow Middle(i)$

$r \leftarrow Right(i)$

if $l \leq heap - size[A]$ *and* $A[l] > A[i]$ **then**

$largest \leftarrow l$

else

$largest \leftarrow i$

end if

if $m \leq heap-size[A]$ *and* $A[m] > A[i]$ **then**

$largest \leftarrow m$

end if

if $r \leq heap-size[A]$ *and* $A[r] > A[i]$ **then**

$largest \leftarrow r$

end if

if $largest \neq i$ **then**

exchange $A[i] \leftrightarrow A[largest]$

end if

$HEAPIFY(A,i)$

HEAPSORT(A)

$BUILD\_HEAP(A)$

for $i = length[A]$ down to 2 **do**

$exchange\ A[1] \leftrightarrow A[i]$

$heap-size[A] \leftarrow heap-size[A]-1$

$Heapify(A,1)$

Four child version of the algorithm is modified in the same manner. Here we have defined extra 2 child which are named 'leftmiddle'and 'rightmiddle'here. In this model height of the tree is significantly reduced which contributes in reducing cost. At the same time in the heapify phase number of comparison required is increased each time. After the modification the algorithm looks like the one presented below.

Algorithm 7 HEAPSORT FOR FOUR CHILD

$Parent(i)$

return $\lfloor \frac{i+2}{4} \rfloor$

$Left(i)$

return $4 * i - 2$

$LMiddle(i)$

return $4 * i - 1$

$RMiddle(i)$

return $4 * i$

$Right(i)$

return $4i + 1$

BUILD _ HEAP (A)

$heap - size(A) \leftarrow length[A] - 1$

for $i \leftarrow \left\lfloor \frac{length[A]}{4} + 1 \right\rfloor$ downto 1 **do**

$HEAPIFY(A, i)$

HEAPIFY(A, i)

$l \leftarrow Left(i)$

$lm \leftarrow LeftMiddle(i)$

$rm \leftarrow RightMiddle(i)$

$r \leftarrow Right(i)$

if $l \leq heap - size[A]$ *and* $A[l] > A[i]$ **then**

$largest \leftarrow l$

else

$largest \leftarrow i$

end if

if $lm \leq heap - size[A]$ *and* $A[lm] > A[i]$ **then**

$largest \leftarrow m$

end if

if $r \leq heap-size[A]$ *and* $A[rm] > A[i]$ **then**

$largest \leftarrow rm$

end if

if $largest \neq i$ **then**

exchange $A[i] \leftrightarrow A[largest]$

end if

$HEAPIFY(A,i)$

HEAPSORT(A)

$BUILD\_HEAP(A)$

for $i = length[A]$ down to 2 **do**

exchange $A[1] \leftrightarrow A[i]$

$heap-size[A] \leftarrow heap-size[A]-1$

$Heapify(A,1)$

4.3 Experimental Result

Experimental results are obtained using the above stated algorithms. For each test we have generated experimental data using random number generation. Effort were made to use data that are as realistis as possible. We have started with 100 sample data and tested upto 20000000 data for all three algorithms. In one test, same set of data is used for 2 child, 3 child and four child so that we can compare the results. We have done the test for both integer values and floating point numbers.

4.3.1 Number of Move for Heapsort

In heapsort, data need to be moved to appropriate places before they are sorted. These movement require considerable amount of CPU time and contribute to complexity. For a specific dataset, the less number of move will be required the more optimal will be the algorithm. So, we have determined number of move required for heapsort with both integer and float values.

Table 4.1: Number of Move Required for Heapsort (Integer Value)

value	child	Test-01	Test-02	Test-03
100	2	669	675	678
	3	512	517	517
	4	450	459	453
500	2	4497	4514	4525
	3	3314	3310	3337
	4	2873	2874	2870
5000	2	62092	62064	62058
	3	43767	43737	43761
	4	36994	36967	36948
10000	2	134170	134249	134101
	3	93608	93576	93466
	4	79396	79298	79200
20000	2	288439	288370	288400
	3	200383	200321	200343
	4	167966	167992	167906
50000	2	787603	787392	787538
	3	542832	542661	542781
	4	454691	454418	454641
100000	2	1675061	1674889	1674691
	3	1145507	1145636	1145590
	4	955176	955263	955259
500000	2	9524679	9523635	9524174
	3	6476719	6476314	6476781
	4	5365855	5365321	5365616
1000000	2	20047812	20047704	20049802
	3	13561477	13561884	13563274
	4	11244303	11244592	11245391
2000000	2	42095600	42095887	42096235
	3	28397367	28397267	28395869
	4	23461780	23461687	23462297

Table 4.2: Number of Move Required for Heapsort (Float Value)

value	child	Test-01	Test-02
100	2	673	692
	3	509	526
	4	509	464
500	2	4534	4537
	3	3330	3331
	4	2896	2877
5000	2	62211	62068
	3	43773	43790
	4	36999	36970
10000	2	134224	134285
	3	93484	93556
	4	79191	79234
20000	2	288558	288253
	3	200426	200495
	4	167938	167935
50000	2	787716	787327
	3	543067	542528
	4	454750	454525
100000	2	1674765	1674843
	3	1145753	1145839
	4	954709	955087
500000	2	9523563	9523829
	3	6477326	6477090
	4	5365117	5365058
1000000	2	10347431	20048029
	3	7300528	13562091
	4	6172534	11245873
2000000	2	42096843	42096441
	3	28399178	28396791
	4	23462206	23462078

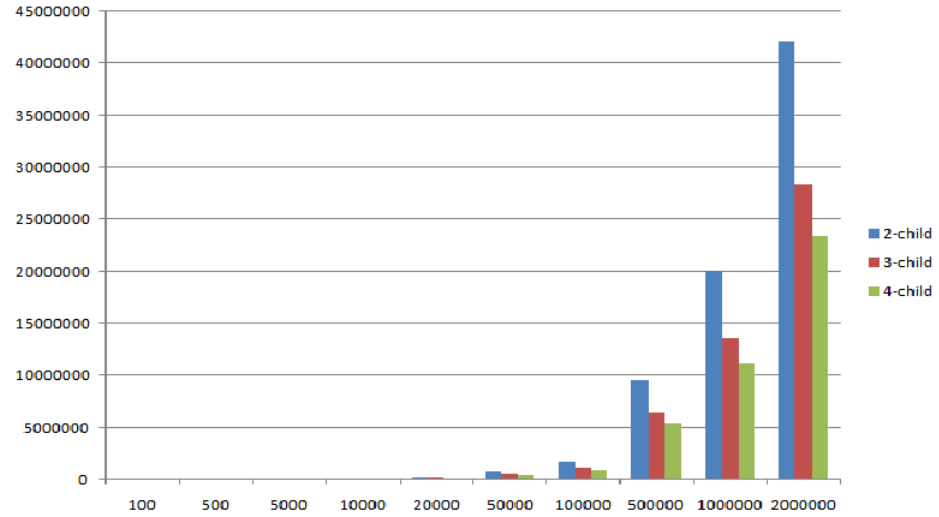

Figure 4.2: Graph for Number of Move(Integer Value)

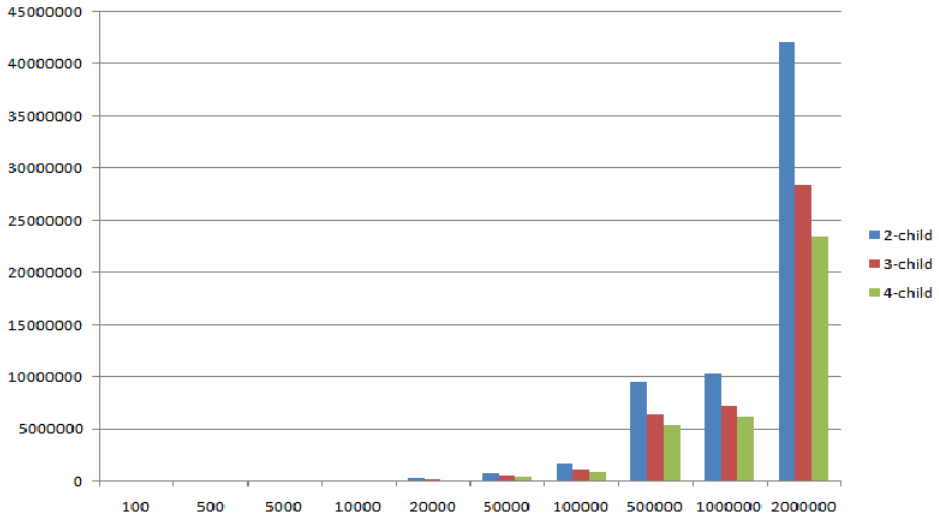

Figure 4.3: Graph for Number of Move(Float Value)

4.3.2 Number of Comparison for Heapsort

A value whether need to be moved or not is determined by number of comparisons. These comparisons also contribute to complexity. For a specific dataset, the less number of comparison will be required the more optimal will be the algorithm. Number of comparisons for integer and float values are tabulated below.

Table 4.3: Number of Comparison Required for Heapsort (Integer Value)

value	child	Test-01	Test-02	Test-03
100	2	1205	1225	1231
	3	1211	1217	1225
	4	1280	1288	1285
500	2	8379	8391	8429
	3	8274	8262	8276
	4	8804	8813	8801
5000	2	117680	117673	117588
	3	114393	114363	114305
	4	120817	120690	120757
10000	2	255370	255504	255355
	3	246788	246733	246582
	4	263174	262684	262496
20000	2	550710	550674	550701
	3	533200	533182	533397
	4	563312	563271	563253
50000	2	1509888	1509712	1509773
	3	1459062	1458417	1458711
	4	1546959	1546798	1547233
100000	2	3219887	3219493	3219811
	3	3097345	3097288	3097589
	4	3277686	3277948	3278181
500000	2	18397368	18396350	18397203
	3	17732440	17731355	17733816
	4	18748529	18747695	18748716
1000000	2	38792722	38793610	38795929
	3	37291184	37292243	37294495
	4	39551609	39552984	39553839
2000000	2	81586318	81586346	81588021
	3	78403519	78404783	78401507
	4	82992521	82993706	82995315

Table 4.4: Number of Comparison Required for Heapsort (Float Value)

value	child	Test-01	Test-02
100	2	1223	1246
	3	1209	1229
	4	1269	1299
500	2	8416	8423
	3	8263	8259
	4	8828	8792
5000	2	117844	117632
	3	114346	114380
	4	120820	120769
10000	2	255389	255471
	3	246577	246601
	4	262583	262642
20000	2	550976	550709
	3	533537	533491
	4	563092	563142
50000	2	1510031	1509315
	3	1458933	1458358
	4	1547222	1547170
100000	2	3219500	3219291
	3	3097527	3097413
	4	3276650	3277376
500000	2	18396587	18396083
	3	17733491	17733920
	4	18747136	18748637
1000000	2	21550579	38793454
	3	21040889	37292201
	4	22232631	39553927
2000000	2	81587739	81587699
	3	78405481	78401907
	4	82995911	82996570

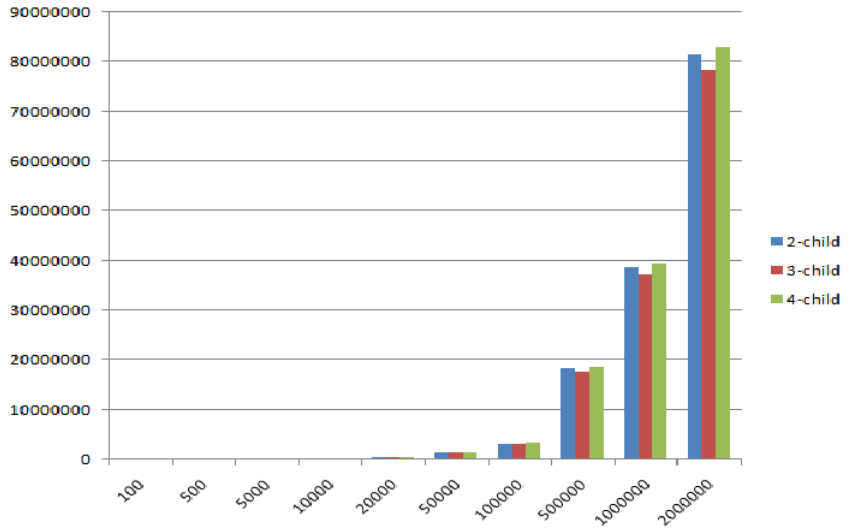

Figure 4.4: Graph for Number of Comparison(Integer Value)

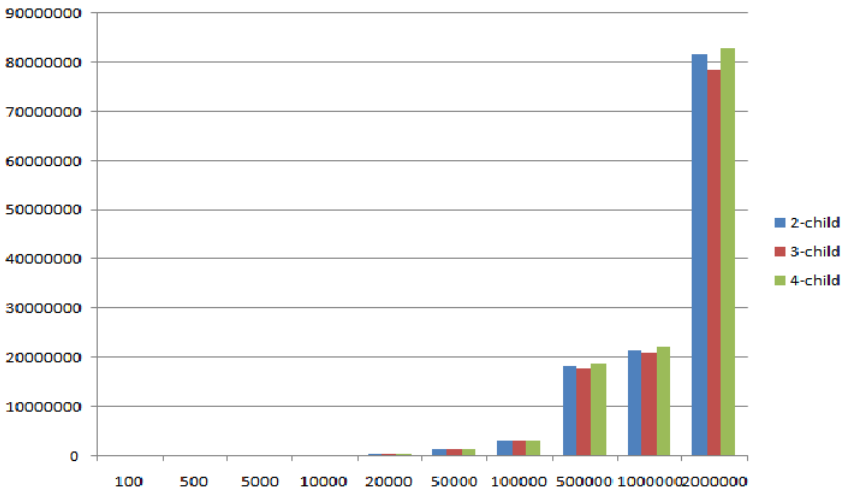

Figure 4.5: Graph for Number of Comparison(Float Value)

4.3.3 Time Required for Heapsort

Time is another important parameter that defines optimality of an algorithm. Therefore, in our simulation we calculated the amount of time required as well for all possible combination. The test is performed in both windows 8 and linux 11.10 environment. We have used a machine with processor Intel(R) core i3, clockspeed 2.93GHz and Ram 1.86 GB.

Table 4.5: Time Required for Heapsort in Linux and Windows Machine(Integer Value)

value	**child**	**Test-01**		**Test-02**	
		Linux	Windows	Linux	Windows
100	2	226021	195981	228952	199480
	3	226021	194931	204550	197380
	4	236393	194230	240422	198081
500	2	570720	1262676	477084	1282974
	3	474178	1262676	442581	1224530
	4	683726	1559797	659793	1391463
5000	2	2477909	1954908	2235097	1993054
	3	2286722	1924461	2027794	1911162
	4	2637165	2089295	2563700	2053598
10000	2	7261443	3345670	6410001	3426163
	3	6820791	3311025	5965735	3286177
	4	6462947	3450311	6898447	3555650
20000	2	5932292	7016110	6253252	6481012
	3	5386643	6386872	5918640	6305679
	4	6125310	6515309	6063398	6527908
50000	2	12529335	15236450	12618249	15345290
	3	11322036	13624160	11476509	13334388
	4	12734967	13853037	13067133	13630459
100000	2	27105518	28781868	27219154	28629633
	3	23150710	26073135	22674991	25786513
	4	26102136	26543488	25669412	25860706
500000	2	157835354	166660251	158427045	166829284
	3	131092896	141575069	131755902	142571421
	4	146649505	143426387	147035389	144985133
1000000	2	403000000	353252440	344000000	356274742
	3	290000000	317192338	287000000	306880253
	4	315000000	320126100	309000000	309514444
2000000	2	760000000	847309675	800000000	813754627
	3	661000000	682810461	668000000	707398692
	4	716000000	683180025	714000000	688840718

Table 4.6: Time Required for Heapsort in Linux and Windows Machine(Float Value)

value	child	Test-01		Test-02	
		Linux	Windows	Linux	Windows
100	2	209448	288372	207272	294322
	3	192561	226429	201835	239027
	4	206018	283473	209108	302021
500	2	1785971	975006	1270728	922860
	3	1201782	961356	1202347	1055847
	4	1256805	1293125	1246164	1322521
5000	2	26167825	2194287	26495260	2126042
	3	24351576	2092097	24491238	2194985
	4	25673879	2245730	25898246	2864819
10000	2	56693780	3638244	58220559	3568250
	3	35960759	3520655	36585359	3512256
	4	38271976	3666591	55469321	3791878
20000	2	16239762	6646199	26224635	6700443
	3	14593914	6558707	19345959	6517061
	4	17981931	6862127	21459762	6962917
50000	2	21113349	16213210	22401178	15579073
	3	19399030	14685960	20867840	14661462
	4	22529897	15475833	21818269	15225607
100000	2	32697145	32175145	32375626	30410968
	3	32306802	29241033	31418026	28968060
	4	35337080	29997309	35019767	30554804
500000	2	137736446	172673419	142711690	180265925
	3	138676708	161111603	134541767	162089057
	4	144169510	189426277	144388177	164800591
1000000	2	208453346	211193793	295747188	379948996
	3	180860672	188088708	296667176	366867629
	4	180607866	195648316	294818265	371138610
2000000	2	667035604	883147157	668785322	934004871
	3	640019781	805433077	645587453	806117260
	4	659041395	833739352	651123489	800463915

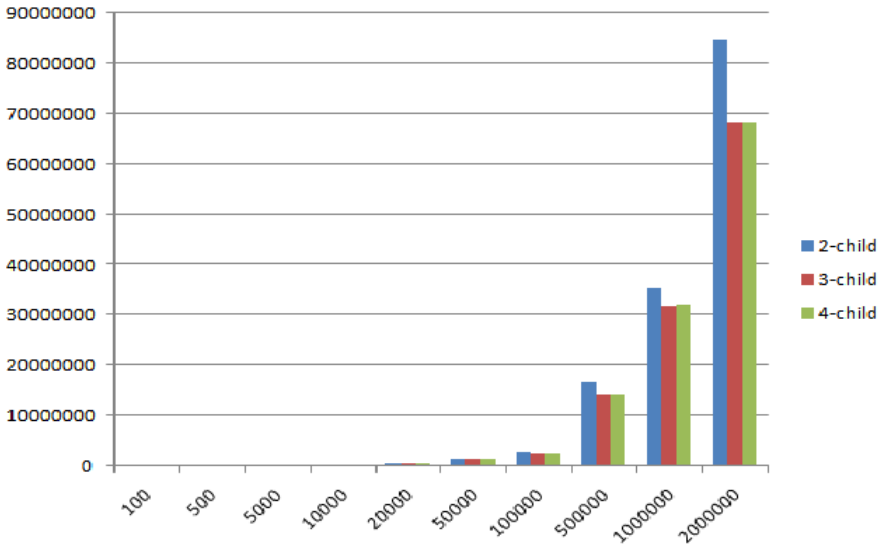

Figure 4.6: Time Taken in Windows Environment(Integer Value)

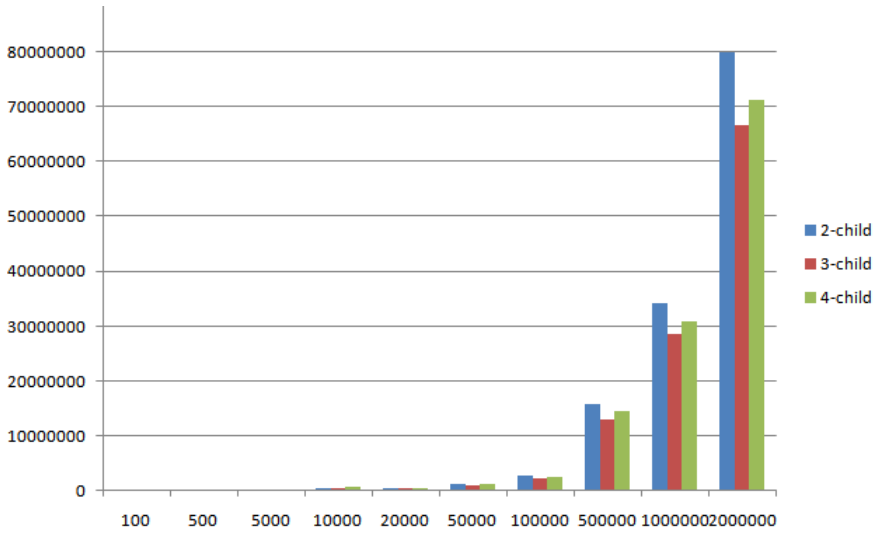

Figure 4.7: Time Taken in Linux Environment(Integer Value)

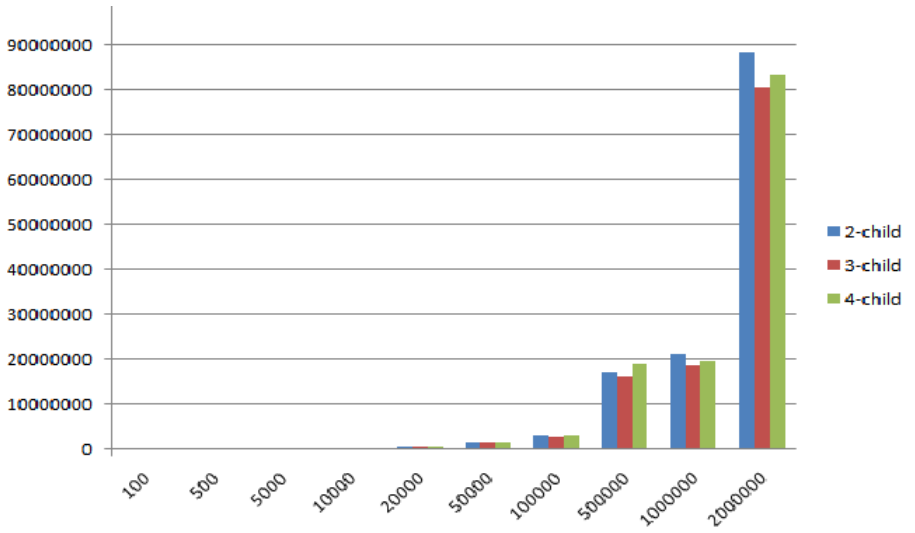

Figure 4.8: Time Taken in Windows Environment(Float Value)

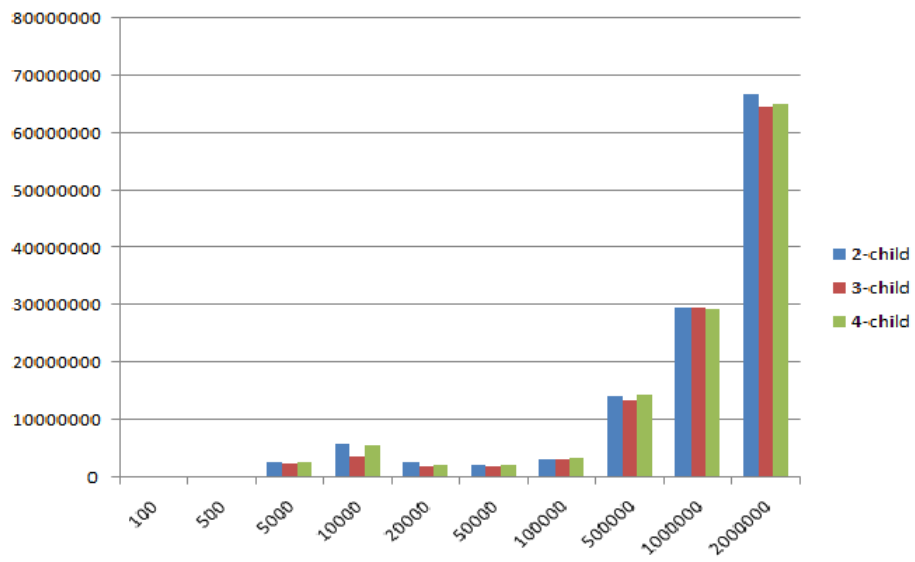

Figure 4.9: Time Taken in Linux Environment(Float Value)

4.3.4 Discussion on Results

From the experimental results some observations can be identified. In case of move, 2 child variant of heapsort requires maximum move, 3 child heapsort requires fewer move and four child heapsort requires fewest move. This observation is true for both integer and float values. But for comparison 3 child variant of heapsort is invariably less than both 2 child and 4 child variant. So, we can comment that number of comparison contributes more in the complexity than number of move.

Time require for heapsort is more in windows than that in linux. It is because windows operating system runs number of background program which increases the time rquire to complete the sort. On the other hand linux operating system exclusively completes one operation. About the variants, two child heapsort requires more time where 3 and 4 child heapsort takes simliar time to complete the heapsort. This observation holds true for both integer and float values.

CHAPTER 5

CONCLUSION

Divide and conquer method is a widely used technology in computer science applications. Some of the applications that use D & C technology can be named as mergesort, quicksort, convex hull, closest pair etcetera. These algorithms have their own complexity which largely depends on the performance of D & C technology. Most of these algorithms have average case complexity O(nlogn) which do not take number of child into consideration.

Heapsort is another widely used algorithm based on D & C technology. To improve the performance or optimality of heapsort we have worked with its number of child. Theoritically it has been shown that if all the node of a tree has d child each then complexity of sort is varied by d/logd. When replaced by value we found 3 to be the optimal value of number of child. Therefore, standard algorithm of binary heapsort has been modified for 3 and 4 child variant. All three algorithms have been tested for optimality taking number of move, number of compare and time as parameters.

Simulation with randomly generated numbers clearly supports our theoretically proven claim. For both integers and floating point numbers we can see that heapsort with three child performs better than 2 child or 4 child heapsort. Therefore we can say that optimal heapsort can be done using the three child variant of heapsort.

5.1 Limitations

Our experiment was limited to heapsort only from the wide variety of applications that use D & C technology. Due to resource and time limitation the thesis has been restricted to datasets with maximum 20000000 values. The experiment was conducted on a single machine in two different environment i.e windows and linux. Overcoming these limitations might display a more decisive and acceptable result.

5.2 Recommendations for Further Research

Basing on our experimental results we recommend to extensively test three child variants of the D & C based algorithms. This can certainly improve the performance of conventionally used or practiced methods. If it is possible to obtain positive results for maximum cases then it will be a impressive breakthrough in the respective field and may open further opportunities to obtain more efficient and optimal algorithms.

REFERENCES

[1] Akepogu R. A., Palagiri R. R., *Data Structures and Algorithms using C++*, Dorling Kindersley Pvt. Ltd., ISBN-978-81-317-5567-9, 1st Edition, page- 220, 2011.

[2] Barry W., *Parallel Programming: Techniques And Applications Using Networked*, Pearson Education Inc. and Dorling Kindersley Publishing Inc., ISBN-81-317-0239-1, 2nd Edition, Page-335, 2007.

[3] Coreman T.H., Leiserso C.E., Rivest R.C., Stein C., *Introduction to Algorithm*, MIT Press and McGraw-Hill, ISBN-0-262-03293-7, 2nd Edition, 2011.

[4] Du D.Z., Zhang X.S., *Algorithms and Computation: 5th International Symposium*, ISAAC '94, Beijing, P.R chaina, Proceedings, ISBN-3-540-58325-4, Page-295, August 25-27, 1994.

[5] Du D. Z., Ko K. I, *Advances in Algorithms, Languages, and Complexity*, Kluwer Academic Publisher, ISBN-0-7923-4396-4, 1st Edition, Page-159, 1997.

[6] Gobel F. and Hoede C., *On an optimality property of ternary trees*, Information and Control 42(1), 10-26 , July 1979.

[7] Gupta P., Agarwal V., Varshney M., *Design and Analysis of Algorithms*, PHI learning Pvt Ltd, ISBN-978-81-203-3421-2, 2nd Edition, Page-38, 2008.

[8] *http://en.wikipedia.org/wiki/External_sorting*, last accessed on November 30, 2013.

[9] *http://en.wikipedia.org/wiki/Heapsort*, last accessed on November 30, 2013.

[10] *http://en.wikipedia.org/wikiKaratsuba_algorithm*, last accessed on December 17, 2013.

[11] *http://pages.cs.wisc.edu/vernon/cs367/notes/14.SORTING*, last accessed on November 29, 2013.

[12] *http://en.wikipedia.org/wiki/Closest_pair_of_points_problem*, last accesssed on December 18, 2013.

[13] Islam M., Kaykobad M., Murshed M.M. and Amyeen E., "3 is a more promising algorithmic parameter than 2", *Computers and Mathematics with Applications*, vol-36, page 19-24, May 1998.

[14] Islam, T.M. and Kaykobad M., "Worst-case analysis of generalized heapsort algorithm revisited", *International Journal of Computer Mathematics*, Vol. 83, No.1, page 59-67, January 2006.

[15] Islam, T.M. and Kaykobad M., "Worst-case analysis of generalized heapsort algorithm revisited" *Proceedings of the International Conference on Computer and Information Technology*, Dhaka,pp. 224-228, 18-20 December.

[16] Karim M.Z., Akter N. *Optimum Partition Parameter of Divide-And-Conquer Algorithm: Solving Closest-Pair Problem*, Publisher: LAP LAMBERT Academic Publishing, ISBN-10: 3848426722 ISBN-13: 978-3848426720, March 20 2012.

[17] Knuth D., *The Art of Computer Programming*, Volume 1, Addison-Wesley, Paris, 1972.

[18] Kolesnikov A., *Design of Spatial Information Systems:Convex Hull*, University of Joensuu, Joensuu, Finland.

[19] Lin X., "Computing theory 98": *proceedings of the 4th Australasian Theory Symposium*, CATS'98, Pert, Page-93, 2-3 February 1998.

[20] Megiddo N., *Theoretical Computer Science 10*, page-337-341, 1982.

[21] Miller R., *Computational Geometry(Divide and Conquer)*, 1996.

[22] Mount D.M., *Computational Geometry*, University of Maryland, College Park, 2000.

[23] Neapolitan R. E., Naimipour K., *Foundations of Algorithms Using Java Pseudo code*, Jones and Bartleet Publishers, ISBN-978-443-5000, 2nd Edition, 2004.

[24] Pinter S.S. and Wolfstahl Y. "Embedding ternary tree in VLSI arrays", *Information Processing Letters* 26, page-187-191, 1987.

[25] Roux S.D. *The structure of Divide and Conquer Algorithms*, 1983.

[26] Sedgewick R., *Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4*, Dorling Kindersley Publishing Inc, ISBN-81-317-1291-5, 3rd Edition, Page-336, 2007.

[27] Sedgewick R., *Algorithms in Java*, Pearson Education Inc., ISBN-0201361205, 4th Edition, Page-348, 2004.

[28] Souvaine D., *Dynamic Convex Hull and Order Decomposable Problems*, Tufts University, 2005.