International Journal of Scientific Engineering and Research (IJSER)
www.ijser.in
ISSN (Online): 2347-3878
Volume 2 Issue 3, March 2014

Analysis and Review of Sorting Algorithms

Gaurav Kocher[1], Nikita Agrawal[2]

[1,2]BE 8th Semester, CSE, Shri Shankaracharya Institute of Professional Management & Technology
Raipur, Chhattisgarh, India

Abstract: ***One of the fundamental issues in computer science is ordering a list of items. Although there is a huge number of sorting algorithms, sorting problem has attracted a great deal of research; because efficient sorting is important to optimize the use of other algorithms. Sorting algorithms have been studied extensively since past three decades. Their uses are found in many applications including real-time systems, operating systems, and discrete event simulations. In most cases, the efficiency of an application itself depends on usage of a sorting algorithm. Lately, the usage of graphic cards for general purpose computing has again revisited sorting algorithms. In this paper we extended our previous work regarding parallel sorting algorithms on GPU, and are presenting an analysis of parallel and sequential bitonic, odd-even and rank-sort algorithms on different GPU and CPU architectures. Their performance for various queue sizes is measured with respect to sorting time and rate and also the speed up of bitonic sort over odd-even sorting algorithms is shown on different GPUs and CPU. The algorithms have been written to exploit task parallelism model as available on multi-core GPUs using the OpenCL specification.***

Keywords: Selection sort, bubble sort, insertion sort, quick sort, merge sort, number of swaps, time complexity

1. Introduction

A *Sorting Algorithm* is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation (reordering) of the input.

Further, the data is often taken to be in an array, which allows random access, rather than a list, which only allows sequential access, though often algorithms can be applied with suitable modification to either type of data.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. A fundamental limit of comparison sorting algorithms is that they require linearithmic time – O(n log n) – in the worst case, though better performance is possible on real-world data (such as almost-sorted data), and algorithms not based on comparison, such as counting sort, can have better performance. Although many consider sorting a solved problem – asymptotically optimal algorithms have been known since the mid-20th century – useful new algorithms are still being invented, with the now widely used Timsort dating to 2002, and the library sort being first published in 2006.

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and upper and lower bounds.

2. Sorting Algorithms

A. Selection Sort

The algorithm works by selecting the smallest unsorted item and then swapping it with the item in the next position to be filled. The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on. Here is an example,

```
void selectionSort(int[] ar){
 for (int i = 0; i ‹ ar.length-1; i++)
 {
 int min = i;
 for (int j = i+1; j ‹ ar.length; j++)
 if (ar[j] ‹ ar[min]) min = j;
 int temp = ar[i];
 ar[i] = ar[min];
 ar[min] = temp;
 } }
```

Example.
29, 64, 73, 34, **20**,
20, **64**, 73, 34, **29**,
20, 29, **73**, **34**, 64
20, 29, 34, **73**, **64**
20, 29, 34, 64, 73
The worst-case runtime complexity is O(n^2).

International Journal of Scientific Engineering and Research (IJSER)
www.ijser.in
ISSN (Online): 2347-3878
Volume 2 Issue 3, March 2014

B. Insertion Sort

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position. The resulting array after *k* iterations has the property where the first *k* + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position.

Example. We color a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving from right to left.

29, 20, 73, 34, 64
29, 20, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 34, 73, 64
20, 29, 34, 64, 73

Let us compute the worst-time complexity of the insertion sort. In sorting the most expensive part is a comparison of two elements. Surely that is a dominant factor in the running time. We will calculate the number of comparisons of an array of N elements:

we need 0 comparisons to insert the first element
we need 1 comparison to insert the second element
we need 2 comparisons to insert the third element
...
we need (N-1) comparisons (at most) to insert the last element
Totally,
$1 + 2 + 3 + ... + (N-1) = O(n^2)$

C. Bubble Sort

In this task, the goal is to sort an array of elements using the bubble sort algorithm. The elements must have a total order and the index of the array can be of any discrete type. For languages where this is not possible, sort an array of integers. The bubble sort is generally considered to be the simplest sorting algorithm. Because of its simplicity and ease of visualization, it is often taught in introductory computer science courses. Because of its abysmal $O(n^2)$ performance, it is not used often for large (or even medium-sized) datasets.

```
void BubbleSort(int a[])
{
 int i,j;

 for (i=MAXLENGTH; --i >=0;) {
 swapped = 0;
 for (j=0; j<i;j++) {
 if (a[j]>a[j+1]) {
 Swap[a[j],a[j+1]);
 swapped=1;
 }
 }
 if (!swapped) return;
 }
}
```

D. Quick Sort

Quick sort is a comparison sort developed by Tony Hoare. Also, like merge sort, it is a divide and conquer algorithm, and just like merge sort, it uses recursion to sort the lists. It uses a pivot chosen by the programmer, and passes through the sorting list and on a certain condition, it sorts the data set. Quick sort algorithm can be depicted as follows:

```
QUICKSORT (A)
    1: step←m;
    2: while step > 0
    3: for (i←0 to n with increment 1)
    4: do temp←0;
    5: do j←i;
    6: for (k←j+step to n with increment step)
    7: do temp←A[k];
    8: do j←k-step;
    9: while (j>=0 && A[j]>temp)
    10: do A[j+step]=A[j];
    11: do j←j-step;
    12: do Array[j]+step←temp;
    13: do step←step/2;
```

E. Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merg() function** is used for merging two halves. The merge (arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

Complexity of Mergesort
Suppose T (n) is the number of comparisons needed to sort an array of n elements by the Merge Sort algorithm. By splitting an array in two parts we reduced a problem to sorting two parts but smaller sizes, namely n/2. Each part can be sort in T(n/2). Finally, on the last step we perform n-1 comparisons to merge these two parts in one. All together, we have the following equation

International Journal of Scientific Engineering and Research (IJSER)
www.ijser.in
ISSN (Online): 2347-3878
Volume 2 Issue 3, March 2014

$$T(n) = 2*T(n/2) + n - 1$$

The solution to this equation is beyond the scope of this course. However I will give you a reasoning using a binary tree. We visualize the merge sort dividing process as a tree.

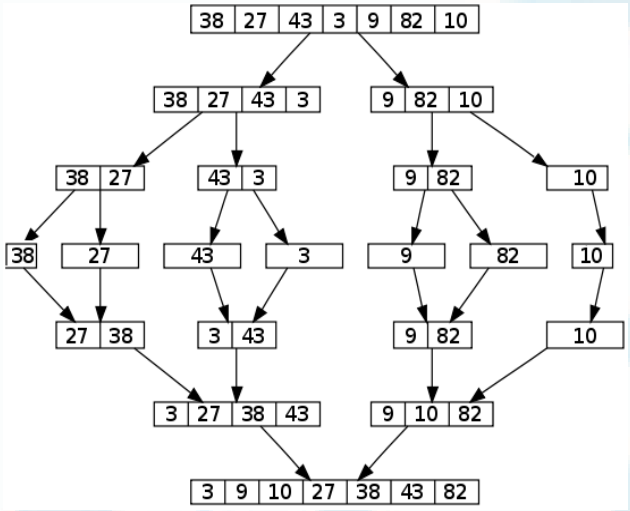

Figure1: Example of Merge Sort

F. Heap Sort

Heapsort is a comparison-based sorting algorithm. Heapsort is part of the selection sort family; it improves on the basic selection sort by using a logarithmic-time priority queue rather than a linear-time search. Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case O(*n* log *n*) runtime. Heapsort is an in-place algorithm, but it is not a stable sort. It was invented by J. W. J. Williams in 1964.

	Worst Case	Average Case	Best Case
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Heap Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Figure 2: Comparison of sorting algorithms

3. Literature Survey

Robert Sedgewick is the author of a well-known book series *Algorithms*, published by Addison-Wesley. The first edition of the book was published in 1983 and contained code in Pascal. Subsequent editions used C, C++, Modula-3, and Java. With Philippe Flajolet he wrote several books and preprints which promoted analytic combinatorics, a discipline which relies on the use of generating functions and complex analysis in order to enumerate combinatorial structures, and to study their asymptotic properties. As explained by Knuth in *The Art of Computer Programming*, this is the key to perform average case analysis of algorithms. [1][3].

Bentley received a B.S. in mathematical sciences from Stanford University in 1974, and M.S. and Ph.D in 1976 from the University of North Carolina at Chapel Hill; while a student, he also held internships at the Xerox Palo Alto Research Center and Stanford Linear Accelerator Center. After receiving his Ph.D., he joined the faculty at Carnegie Mellon University as an assistant professor of computer science and mathematics. At CMU, his students included Brian Reid, John Ousterhout, Jeff Eppinger, Joshua Bloch, and James Gosling, and he was one of Charles Leiserson's advisors. Later, Bentley moved to Bell Laboratories. [2][3].

Dr. Baecker is an expert in human-computer interaction ("HCI") and user interface ("UI") design. His research interests include work on electronic memory aids and other cognitive prostheses; computer applications in education; computer-supported cooperative learning, multimedia and new media; software visualization; groupware and computer-supported cooperative work; computer animation and interactive computer graphics; computer literacy and how computers can help us work better and safer; and entrepreneurship and the management of small business as well as the stimulation of innovation. Baecker is also interested in the social implications of computing, especially the issue of responsibility when humans and computers interact. [7].

4. Conclusion

This paper discusses comparison based sorting algorithms. It analyses the performance of these algorithms for the same number of elements. It then concludes that selection sort shows better performance than Quick sort but being the simple structure selection sort is more preferred and widely used. It is clear that both sorting techniques are not that popular for the large arrays because as the arrays size increases both timing is slower for integer and string, but generally the study indicate that integer array have faster CPU time than string arrays. Both have the upper bound running time O (n2). Bubble Sort is a very simple algorithm that is only suitable for small lists. There are lots of alternative sorting algorithms that are more performant than Bubble Sort. Bubble Sort is widely considered to be the least performant of all the established sorting algorithms.

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of $O(n \log(n))$ Mergesort is optimal. However, selection sort has the property of minimizing the number of swaps. In applications where the cost of swapping items is high, selection sort very well may be the algorithm of choice.

5. Future Scope

An important key to algorithm design is to use sorting as a basic building block, because once a set of items is sorted,

many other problems become easy. Consider the following applications:

- *Frequency distribution* - Given a set of *n* items, which element occurs the largest number of times in the set? If the items are sorted, we can sweep from left to right and count them, since all identical items will be lumped together during sorting. To find out how often an arbitrary element *k* occurs, start by looking up *k* using binary search in a sorted array of keys. By walking to the left of this point until the element is not *k* and then walking to the right, we can find this count in $O(\lg n + c)$ time, where *c* is the number of occurrences of *k*. The number of instances of *k* can be found in $O(\lg n)$ time by using binary search to look for the positions of both $k - \epsilon$ and $k + \epsilon$, where ϵ is arbitrarily small, and then taking the difference of these positions.
- *Selection* - What is the *k*th largest item in the set? If the keys are placed in sorted order in an array, the *k*th largest can be found in constant time by simply looking at the *k*th position of the array. In particular, the median element appears in the (*n*/2)nd position in sorted order.

References

[1] *Algorithms in Java, Parts 1-4, 3rd edition* by Robert Sedgewick. Addison Wesley, 2003.
[2] *Programming Pearls* by Jon Bentley. Addison Wesley, 1986.
[3] *Quicksort is Optimal* by Robert Sedgewick and Jon Bentley, Knuthfest, Stanford University, January, 2002.
[4] Dual Pivot Quicksort: Code and Discussion.
[5] *Bubble-sort with Hungarian ("Csángó") folk dance* YouTube video, created at Sapientia University, Tirgu Mures (Marosvásárhely), Romania.
[6] *Select-sort with Gypsy folk dance* YouTube video, created at Sapientia University, Tirgu Mures (Marosvásárhely), Romania.
[7] Sorting Out Sorting, Ronald M. Baecker with the assistance of David Sherman, 30 minute color sound film, Dynamic Graphics Project, University of Toronto, 1981. Excerpted and reprinted in SIGGRAPH Video Review 7, 1983. Distributed by Morgan Kaufmann, Publishers.

Author Profile

Mr. Gaurav Kocher is perusing Bachelor of Engineering (CSE) from Shri Shankaracharya Institute of Professional Management & Technology, Raipur and presently is in Final Year. His research interests are Data Structure, Web Designing, Analysis and design of algorithms.

Ms. Nikita Agrawal is perusing Bachelor of Engineering (CSE) from Shri Shankaracharya Institute of Professional Management & Technology, Raipur and presently is in Final Year. Her research interests are Data Structure, Web Designing, Analysis and design of algorithms.