A Review on Sorting in Multi-Core architectures

Samyak Kumar (B17CS046), Rati Kumari (M20CS012) and

Samanyu A Saji (M20CS013)

Indian Institute of Technology, Jodhpur, India

{kumar.52, kumari.11, saji.2}@iitj.ac.in

June 8, 2021

Abstract

The increase in data and data analysis at an explosive rate essentially meant that the traditional computing paradigms were literally going out of scope. The advent of multiprocessing and parallel processing brought about a drastic change in computing point of view. It had such a huge impact that we possibly cannot think of going back. They have been instrumental in the overall improvement of a lot of applications. One such important application in the field of sorting. Sorting has always been a topic with lots of research works going on in parallel, thanks to its immense set of applications in various fields. Effective utilization of multi-core architectures clubbed with sorting techniques are of great interest and there have been notable works that try to bring about parallelism so that both time and space complexities are minimized according to the need. This paper aims to do a thorough analysis concentrated on the recent works in sorting in multi-core architectures.

Keywords - Sorting, Multi-core, multithreading, parallelism, time and space complexity.

1 Introduction

Sorting is an elementary operation in a numerous applications related to the field of Computer Science. Popularity for this operation is very high, obviously because of the role it plays in enhancing a number of applications; like searching. Hence, finding and working on the right sorting method for different situations has always been necessary. Knuth[4], along with Hopcroft and Aho[5], were instrumental in the upbringing of classical sorting algorithms. Sorting has evolved a lot since then and is still relevant and essential in innumerable occasions.

Sorting algorithms are generally divided into two classes based on the comparison. In comparison-based sorting, the order of an element is decided by comparing each pair of elements. The best time complexity achieved in comparison-based sort is $O(NlogN)$. Quicksort and Mergesort are mostly used in different applications. Quicksort often performs better than merge sort in real datasets. But implementing comparison based sort on SIMD (single instruction multiple data) is challenging. Bitonic sort gives good performance on a SIMD processor where numerous comparisons are made simultaneously. It works well on modern multi-core architectures. In non-comparison-based sorting, elements are distributed into finite categories. If the categories are less than the number of elements, then sorting is possible in linear time. Radix sort is an example of a non-comparison based sort.

The advent of multiprocessing and parallel processing boosted the computing industry as a whole by improving efficiency by a mile. This is true in the case of sorting as well. Even though this is the case, proper tweaking and modification of the original algorithms are always necessary to make sure that parallelism is achieved in multi-core architectures, thereby achieving better performance than before.

1.1 Multi-core Processors

A multi-core processor architecture has the ability to support multiple units for processing with the help of a single IC. The huge support for multi-core over single core is due to its ability to instructions in parallel on different cores at the same time, thereby improving performance and hence, the overall speed. Each processing unit interacts with the operating system (OS) as individual entities and even the OS recognizes them as separate processors. The scheduler of the OS is assigned with the task of flexibly dividing the works evenly between different cores. However it is to be specifically noted that not every program can be parallelized easily. Possible dependency issues can slow up parallel executions. However, we can see that most of the programs now are developed to cater parallel execution demands, obviously due to the immense popularity of multi-core systems.

1.2 Merge Sort

Merge sort is one of the most popular comparison-oriented sorting techniques. It is known for its divide-and-conquer, bottom up approach. John Von Neumann came up with this method in 1945. The classical algorithm has been an extensively studied method, which has a lot of modified versions. The efficiency of the algorithm when it comes to sorting very large sets of numbers made it a favourite in big data analytics and even in external sorting techniques. The basic idea of the algorithm is as follows. The given unsorted list of N data elements are initially divided into smaller lists with an element each. Next, every adjacent pair of these smaller lists are merged by comparing and rearranging within those adjacent sublists itself, leading to N/2 sorted sublists at the end of the first merging. This process is iteratively followed until we get a single sorted list. It has a time complexity of N log N. Even though external memory is commonly used as a temporary storage in the classical approach, there are inplace implementations too.

1.3 Radix Sort

Radix sort is an example of non-comparison sorting that divides the elements into buckets, according to radix. In Radix sort, the bucketing process is replicated for each digit while maintaining the order. As the bins are sorted independently they are passed to the available processor and sorting is performed on each digit. It is also known as bucket sort or digital sort.

1.4 Bitonic Sort

Bitonic sort is an example of comparison sort with the complexity of $O(N(logN)^2)$. In bitonic sort we compare elements in a defined sequence and are data independent so it can be easily implemented in parallel processing. Input has to be in order of 2N, to run bitonic sort. The only requirement is the elements have to be sorted in bitonic sequence; the first half of elements are in increasing order and second in decreasing order. The comparison is made between the same index element of each half. Elements are exchanged if the element of the first half is smaller than the second half. We compare and exchange elements and obtain two bitonic sequences. The same process is repeated with two bitonic sequences and gets four bitonic sequences and in each sequence the left sequence is smaller than the right elements.The process is repeated till the sorted sequence is obtained.

1.5 Multiway Merging

We know problems where sizes are large bandwidth play a crucial role in controlling the time of execution of the overall process. In such cases multiway merging is used address the problem. Once we have computed the individual lists we combine them to form the the final list. This is achieved by using binary tree data structure where the leaves of the tree are the sorted chucks received from step(a) of the algorithm and the root of the tree pointing towards the array containing the sorted output once step(b) has finished.

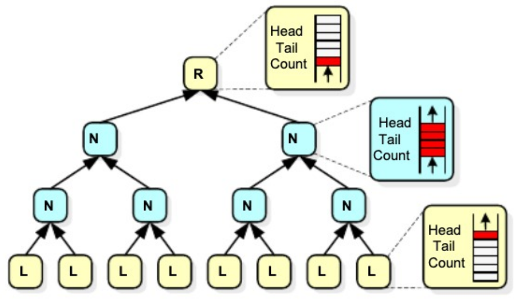

Figure 1: Multiway Merging

Figure 1 shows binary tree for the multiway merging process. Now to store the results from intermediate instruction for each node N we maintain a small FIFO queue. If in any point of time there is an empty slot in the FIFO queue and each of the children nodes has at least one of the elements then smaller of the two can be moved from child's queue to parent nodes queue. When we reach to a scenario where no ready nodes are present the final array will contain the sorted data. In reality we found that this scheme works great for the current multi-core processors.

2 Literature Review

This section tries to uncover the kinds of work that is going on in the field of sorting by exploiting the abilities of multi-core processing. K. Sujatha et al. [2] mainly concentrated on the overall impact that implementation of multi-core systems brought about on their counterparts, the single-core systems. The basic idea of how well multiple core systems could be efficiently utilized has been explained. However, it is also specifically noted that this utilization is entirely dependent on the ability of the running application to be decomposed into sub-modules that could be independently run by different processors. Their findings were backed up by the results they received when they tried to run the selection sort and bubble sort algorithms. The time gap that it took to get executed in a single core architecture and quad-core architecture were compared. Obviously, quad-core showed better performance and it was observed that the time gap got more significant as the number of elements increased. Marcin Woźniak et al. [3] wanted to make sure that all the available resources of a multi-core architecture were made use to the fullest. Here, they proposed a method where the entire sorting process is divided such that each logical core is able to meet separation-of-concerns. This directly results in independence and no chances of disruption. This model is referred to as the Fully Flexible Parallel Merge Sort, or FFPMS. They use the CREW-PRAM model here. Like a typical merge sort, this approach also has a bottom-up build up with the need for temporary space. Every step of the merge procedure has two phases. The first phase combines every pair of strings and sorts them, followed by writing it to the temporary space. Then, these pairs are further combined for merging, and this output is written back from the temporary to the original output space. Each processor works independently on each element such that they compute the index of that element in the other half of an original string, which comes next in sorted order. This output index is used to find the new position of each element in the temporary space. Since we follow the CREW-PRAM model, processors can read at the same time and all their actions are independent in nature, making the whole process interrupt-free. The FFPMS of n such processors produces a time complexity of

$O((log(n))^2)$.

Zekun Yin et al.[7] focused on exploiting both memory bandwidth and computing capacity by cache blocking, multithreading and SMID vectorization. Dataset used for the experiment is single precision floating point numbers (32K to 512M elements) and performance is measured on multicore systems.

Processor	Gold 6148	Xeon Phi(TM) 7210
Core Clock Speed	2.4GHz	1.2GHz
Number of Cores	20	64
Memory Type	DDR4-2666	DDR4-2133 + MCDRAM
Memory Mode	-	FLAT

Table 1: Performance Comparison - Gold 6148 vs Xeon Phi(TM) 7210

Implementation was done in C language and performance is checked on both multicore systems. We can find a scaling behavior on multi-core and many-core architectures. And it was concluded that peak performance achieved is 1.14 billion elements per seconds. D. Abhyankar et al.[8] has conducted experiments on parallel sorting for large input size in Java by exploiting the threading concept. Threading concepts works better if the dataset is huge otherwise the threading overhead exceeds the computation cost. Satish et al.[9] have demonstrated efficient algorithms for radix sort and merge sort on many-core architecture. The experiments have shown that radix sort is the fastest sorting algorithm for modern GPU processors. It is also faster than comparison based sorting techniques.

Jathin Chhugani et. al[1] improved upon the above architectures to bring about a more efficient sorting implementation specifically for the SIMD architecture.

Algorithmic details

Before moving onto the algorithm ,consider the set of notations sed here:

P, the total number of processors in action

K, the SIMD width

C, the size of the Cache Size in bytes

M the size of the block

N, the total number of input elements

ϵ, the size of the element

BW, bandwidth of the memory

Algorithm Overview

We can see that the process of merge sort, is nothing but essentially a streaming kernel, as each iteration takes in an element and then puts it back to its correct position after processing it through the merge sort operations. Nowadays, we know that modern computers are actually having a good size of shared caches, which can be taken into consideration to bring down the number of rounds to the memory. The idea proposed by them was to divide the dataset into blocks and then store it into separate caches. If the size of the cache is C then the block size(M) would be C $/2\epsilon$. The algorithm consists of broadly 2 phases -

Phase 1

Dividing the data set into the block size M and then sorting them individually. This step can further be broken down into two further steps -

- Dividing each block among the available processors(P). Each thread or processors sorts the part assigned to it using SIMD implementation of merge sort. So once this step is completed we will get P sorted lists.
- Now merge the sorted lists into a common list of size M. For this step too we need multiple threads to work together simultaneously. This step contains $\log(P)$. This step will finish off generating N/M sorted blocks.

Phase 2

This step consists of $\log N$-$\log M$ iterations. In each of these iterations we merge pairs of the list received from phase 1 to obtain a list of twice the size of the last iteration. The processors again work simultaneously to merge the lists, using similar algorithms as in phase 1.

Complexity of the algorithm

Please note that they are loading the dataset from the memory only once during the step(a). However each step(b) needs to load as well as store the complete dataset from the main memory. So the Total number of instructions for the complete algorithm would be $O(Nlog(N)log(2K)/K)$.

Exploring Multi Core Aspects of the Algorithm

The idea is fairly simple - when considering the case of multicore systems, multiple threads share last level caches. Hence, instead of making use of a block size of M for each thread, a block size of M'=M/P can be chosen. After each block is done with the sorting of its assigned block of size M' we then merge the resultant P sorted lists obtained at step(a), 4th part of the algorithm discussed above. Threads or the processors work together to merge the P sorted lists into one, so this step has $\log(P)$ iterations. Let us understand this by taking an example of two lists X and Y each of size N'. For the threads to work independently we calculate the median of merged list which can be done in $\log 2N'$ steps. The first processor starts its work from the starting of the list and generates N' elements while the second processor starts from the median element and again generates N' elements(so the lists are mutually exclusive). Note there are a total of $\log(P)$ iterations, plus the partitioning is done in a way that each thread is assigned a multiple of K iterations.

Unaligned Loads

The algorithm eliminates unsigned loads during the simultaneous multi threading phase as the number of elements assigned to each thread is a multiple of SIMD width. Try considering the example of the two threads again - up till now, the conclusion can be made that the output of the two threads are always aligned stores. For the first thread, the starting point is the two aligned addresses and for the second thread, the starting point may be unaligned, but the sum of unaligned elements must be equal to SIMD width. So with this approach, the boundary cases can be handled separately, that is, the initial elements outside the loop can be merged separately. Further, the boundary cases can be handled by SIMD and aligned loads as the starting point of the boundary case second thread can be used as boundary case for the first. Hence, this idea can be easily extended to multiple threads working together on merging of two sorted arrays. So a conclusion can be made that there are no unaligned loads in the algorithm stated above.

Multi-thread Implementation

Given - Execution time in cycles for each step of single threaded SIMD-fied merge sort,Other system parameters.

Our Goal - To calculate total time taken by multiple thread implementation. As explained earlier we know that thee algorithm consists of two parts, first we divide the input into N/M blocks. Now let's examine the computation for this block. For using the P threads, the block gets divided into P lists where M/P elements are sorted.

The execution time this phase can be calculated as :-

$T_{parallel-phase1(a)} = Mlog(M/P) * T_{pe}$

Where T_{pe} is the execution time/element/iteration. Now the P threads work together build the one sorted list. This further adds up two new components to the execution time :

- Time for the calculation of starting location in the arrays before merging (T_{extra}).
- Time for the synchronous working of the threads(T_{sync}).

Although this one is too small and can be ignored. For the next part we have log(P) iterations so for this the equation becomes:

$T_{parallel-phase1(b)} = log(P) * [T_{sync} + MT_{pe}]$ As there were N/M blocks, the total time spent in the first phase would be

$T_{parallel-phase1} = N/M[] * [T_{parallel-phase1(a)} + T_{parallel-phase1(b)}]$

For the second phase we have already discussed that threads work together to merge the sorted block of size M into a final sorted list so time of execution for this phase can be modeled as :

$T_{parallel-phase2} = log(N/M) * NMaxlog(N/M)T_{pe}, 2\epsilon/BW + T_{sync}$

, so the total the of execution comes out to be

$T_{parallel} = T_{parallel-phase1} + T_{parallel-phase2}$

Multi-thread Performance

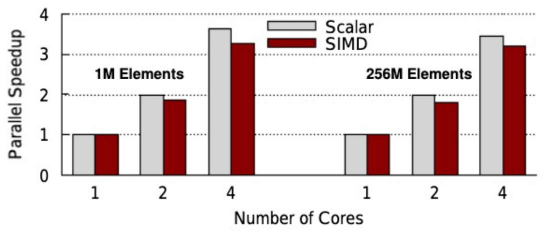

Figure 2: Multi-thread Performance

We have analysed both the scalar and SIMD versions of parallel merge sort algorithm on multi thread systems. Figure 2 shows Multi-thread Performance and is constructed using SIMD versions of merge sort for first 1 million and then 256 million elements. The following insights were drawn from the graph -

- Earlier the SSE version of 256M elements scaled only upto 1.75x; the reason being limitations proposed by the memory bandwidth.

- The algorithm is able to effectively make use of the computational power and its resources.
- It is to be noted that SSE graphs are slightly lower than Scalar code because of much large synchronous overhead.

3 Analysis

We observed that the authors have used different approaches to achieve fast computing on multithreading, multicore architecture. Different sorting techniques are implemented on various systems to achieve greater efficiency for large data. Analysis based on each paper can be observed from this table.

#	Author	Techniques	Area
1	K. Sujatha et. al [2]	Sorting on multicores system	Comparison
2	Marcin Woźniak et al. [3]	FFPMS on CREW-PRAM model	Analysis
3	Zekun Yin et al.[7]	Multicore and parallel computation	Analysis
4	D. Abhyankar et al. [8]	Multithreading	Comparison
5	Chhugani et al.[1]	Sorting on multi-core SIMD	Analysis
6	Satish et al.[9]	Parallel computing	Comparison

Table 2: Analysis and Comparison

4 Conclusion

We have presented an effective way of carrying out merge sort on current CPU architectures. Our study includes analysis of the major architectural features of modern day processors so as to achieve significant benefits on the overall performance. Some of the features to make use of the parallelism include vectorizing SIMD, load balancing and multiway merging. In our analysis we deployed wider sorting or multiple independent networks. We analysed the SIMD and scalar versions of parallel merge sort and finally concluded that the algorithm to be efficient in terms of computation and utilization of system resources. Further we deeply analysed the results we got from single and multi thread architectures and found that sorting on multicore systems was much faster than single core systems. We found that two processors increases the performance upto 2 times while using 4 such processors speeds up the performance anywhere between 2.5x to 4x. The overall analysis indicate that the results would had been much better on switching to a better hardware.

References

[1] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. Proc. VLDB Endow. 1, 2 (August 2008), 1313–1324. DOI:https://doi.org/10.14778/1454159.1454171

[2] K. Sujatha, P. V. N. Rao, A. A. Rao, V. G. Sastry, V. Praneeta and R. K. Bharat,

”Multicore parallel processing concepts for effective sorting and searching,” 2015 International Conference on Signal Processing and Communication Engineering Systems, Guntur, 2015, pp. 162-166, doi: 10.1109/SPACES.2015.7058238.

[3] Zbigniew Marszałek, Marcin Woźniak, Dawid Połap, ”Fully Flexible Parallel Merge Sort for Multicore Architectures”, Complexity, vol. 2018, Article ID 8679579, 19 pages, 2018. https://doi.org/10.1155/2018/8679579

[4] D. E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Professional, 2011, vol. 3.

[5] A. V. Aho and J. E. Hopcroft, The design and analysis of computer algorithms, Pearson Education India, 1974.

[6] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi, “Performance evaluation of a vector supercomputer sx-aurora tsubasa,” in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018, pp. 685–696.

[7] Zekun Yin, Tianyu Zhang, Andre M´ uller, Hui Liu, Yanjie Wei, Bertil Schmidt, Weiguo Liu, ‘Efficient Parallel Sort on AVX-512-based Multi-core and Many-core Architectures’, 2019 IEEE 21st international conference on High Performance Computing and Communications

[8] D.Abhyankar, M.Ingle, “An Efficient Parallel Sorting Algorithm for Multicore Machines ”, (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (5) , 2011, 1995-1998

[9] Satish, Nadathur & Harris, Mark & Garland, Michael. (2009). “Designing efficient sorting algorithms for manycore GPUs. Parallel & Distributed Processing”, 2009 IPDPS 2009 IEEE International Symposium on: 2009. 23. 1-10. 10.1109/IPDPS.2009.5161005.