Parallel Merge Sort with Double Merging

Ahmet Uyar
Department of Computer Engineering
Meliksah University, Kayseri, Turkey
auyar@meliksah.edu.tr

Abstract— Sorting is one of the fundamental problems in computer science. With the proliferation of multi core processors, parallel algorithms for sorting have become very important. In this study, we propose a new parallel merge sort algorithm in which two threads perform the merge operation simultaneously. We have implemented the new merge sort algorithm in Java. We compared the results of the new algorithm with the parallel merge sort implemented in Java library. The results showed that the new algorithm provides between %20-%30 percent speed increase in a quad core system when sorting 10M to 50M double numbers.

***Index Terms*—sorting, parallel merge sort, parallel algorithms**

I. Introduction

Sorting is one of the fundamental problems in computer science. Over the years, researchers have developed many algorithms to solve this problem. Many of these algorithms have been developed to work on single CPU machines. Some of these single CPU sorting algorithms are Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort and Radix Sort.

However, in recent years computer systems have been using more and more cores in processors. Nowadays, even many types of smartphones have quad core processors. Since the year 2004, the trend in processor technology has been to put more cores instead of increasing the clock speed. This trend requires us to develop parallel algorithms for the important problems in computer science. Therefore, we need to develop more parallel algorithms for the sorting problem.

There are some parallel algorithms for sorting. However, they are much fewer compared to the single CPU sorting algorithms. In this study, we propose an improvement for the parallel merge sort algorithm. We implemented the improved merge sort algorithm in Java and compared the results with the parallel merge sort algorithm implemented in Java Library. The results indicate that the new algorithm can perform much faster and utilizes the system resources more efficiently.

II. Parallel Sorting Algorithms

Sorting is a difficult problem to parallelize. Many single CPU sorting algorithms require passing through the whole unsorted data set multiple times. For example, selection sort requires finding the minimum/maximum of the unsorted dataset repeatedly. Since the size of unsorted set is decreasing with every iteration, it is difficult to parallelize it. Similarly, bubble sort, insertion sort and heap sort are all difficult to parallelize.

The recursive sorting algorithms are better suited for parallelization. They divide the unsorted data set into multiple segments and work on them independently. Quick sort divides the unsorted data set into two partitions based on a chosen pivot element. Then, it may divide each partition into two partitions again with chosen pivot elements. This partitioning may continue until the desired partition size is reached. When the partitioning process is completed, each core may sort its partition in parallel to others. The main difficulty with quicksort algorithm is to partition the unsorted dataset into equal partitions. The size of partitions may fluctuate a lot and the workload among the cores may be distributed unevenly. In addition, the process of partitioning should also be parallelized among multiple cores.

Tsigas et al. proposed in [1] a parallel quick sort algorithm in which they employ multiple threads when partitioning the array into two. Edahiro's Mapsort [2] algorithm divides the unsorted array with multiple pivots into multiple partitions. Then each partition is sorted with a thread sequentially.

Merge sort is another recursive algorithm that is suitable for parallelization. The parallel version of the merge sort is shown at Figure 1 for four cores as implemented in parallelSort method of java.utils.Arrays class of Java Library. It first divides the unsorted dataset recursively into two. This process continues until the number of unsorted subsets reaches to the number of cores in the system. Then, each core sorts one unsorted subset independently in parallel. They may use any single CPU sorting algorithm to sort their segments. Once, each thread is done sorting their parts, the process of merging starts. Each parent thread merges the two sorted subsections from its children threads. As the final step, the root thread merges two subsections from its children threads and produces the sorted dataset.

In this algorithm, all four cores are utilized fully when sorting their subsections. However, when merging is performed, system utilization is reduced significantly. Only two cores are used at the first round of merge operations and the other two cores sit idle. In the final stage of the merge operation, only one core is used and the other three cores sit idle. As the number of cores increases in a system, the utilization of cores is reduced even more during the merge operations. Therefore, the primary objective of parallel merge sort algorithms has been to try to distribute the load of merging among more cores.

A parallel merge sort algorithm proposed by Varman et al. [3] and popularized by the developers of the GNU Multi-Core Standard Template Library (MCSTL) [4]. In this algorithm, first the unsorted array is divided by the number of threads and each partition is sorted by one thread. Then all threads take part in merging the sorted partitions. Parallel merging is a complex process.

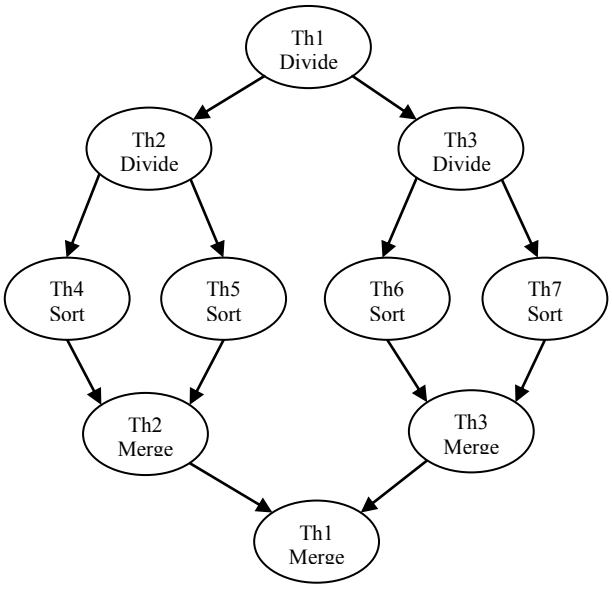

Figure 1 Parallel merge sort with 4 threads

III. Parallel Merge Sort with Double Merging

We propose the merge sort algorithm as shown in Figure 2 for a quad core system. We improve the parallelization of merge operations by performing every merge operation by two simultaneous threads. With this improvement, in the first round of merging, all cores are used. No core sits idle. In the second round of merging, half of the cores are used. The other half sits idle. In the final round of merging, two cores are used instead of one. This improved algorithm speeds up the merge process by two fold. Every merge operation is performed by two threads simultaneously, instead of one. In the next section, we explain how the process of merging is performed by two threads simultaneously.

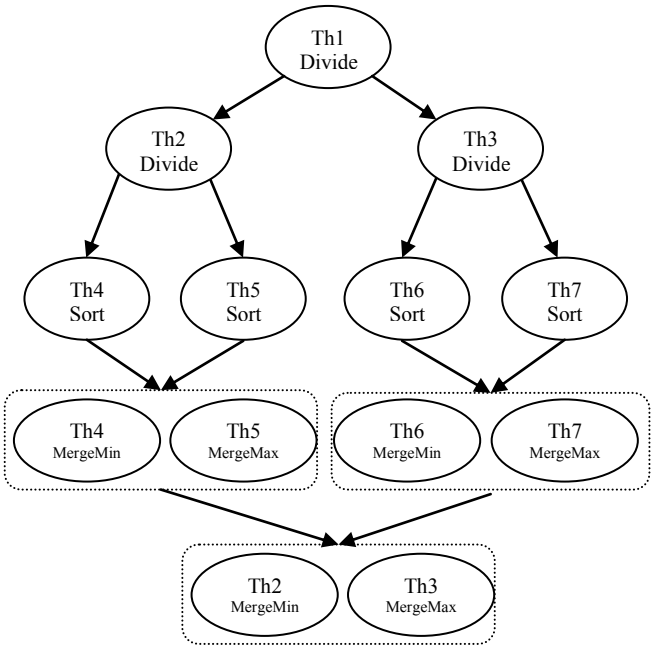

Figure 2 Parallel merge sort with double merging

IV. Merging with Two Threads

Merge algorithm processes two sorted subsets and produces one sorted data set. The algorithm picks the minimum or maximum of the two sorted subsets in every iteration and saves it to an auxiliary array. This iteration continues until all the elements in the sorted subsets are transferred to the auxiliary array. As the last step of the merging, all the data is transferred back to the original array from the auxiliary array. When there are n elements in total in two sorted subsets, the process of merging takes O(n) time.

We propose a merging algorithm that are performed by two simultaneous threads. The working of the algorithm is shown on Figure 3. One thread starts from the minimums of two sorted subsets and picks the minimum of two sorted subsets on every iteration. It generates the first half of the whole sorted dataset. Meanwhile, the second thread starts from the maximums of two sorted subsets and picks the maximum of two sorted subsets on every iteration. It generates the second half of the whole sorted dataset.

The proposed algorithm speeds up the merging process by two fold. Instead of one thread, each merge operation is performed by two simultaneous threads, However, the new algorithms does not change the time complexity of the merge algorithm. It still takes O(n) time.

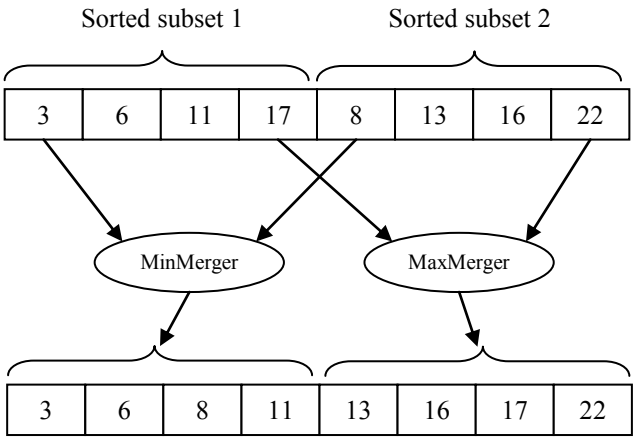

Figure 3 Merging by two simultaneous threads

A. Synchronization of Two Merging Threads

Since the two threads do not change the elements of the original array with two sorted subsets, they can simultaneously work to generate the merged datasets without interfering with one another. However, both threads have to finish generating the merged subsets before the process of copying back to the original array starts.

Copying back of the sorted data from the auxiliary array to the original array can also be performed by two threads simultaneously. Each thread can copy back their half to the original array.

Two merging threads need to be synchronized at two points when they are performing the merging. Synchronization algorithm is given for two merging threads at Figure 3.

```
Merging starts{
   Merge:
      Thread 1: merge mins
      Thread 2: merge maxes
   Synchronize:
      Two threads wait each other
   Copy back:
      Thread 1: copy back first half
      Thread 2: copy back second half
   Synchronize:
      Two threads wait each other
} Merging ends
```

Figure 4 Synchronization algorithm for two merging threads

V. Implementation of Parallel Merge Sort with Double Merging

There are two main alternatives to implement the parallel merge sort algorithm in Java. First one is to use the Fork Join Framework [5]. The second one is to use the CyclicBarrier synchronization object in Java library [6].

A. Parallel Merge Sort with Fork Join

parallelSort method of Arrays class of Java Library uses Fork Join framework to implement the parallel merge sort algorithm. It works as shown in Figure 1. First, a single thread is started (Th1). This thread divides the problem into two and starts two new threads to solve each half of the problem (Th2 and Th3). Then, as needed each thread may divide its portion of the problem into two and starts two new threads to solve each half. Usually, when the number of threads reaches to the number of cores in a system, the problem is not divided into smaller pieces anymore and each thread solves its portion of the problem sequentially. Then, the process of combining the results begin. Each thread combines the results from its two children and passes it to the parent thread. The root thread computes the overall result.

Implementation of the new merge sort algorithm with Fork Join Framework can be done as shown in Figure 2. The process of merging can be performed by two children threads. In the implementation of parallel merge sort algorithm in Java library, parent threads perform the merge operations. We propose that the merge operations should be performed by two children threads.

B. Parallel Merge Sort with CyclicBarrier

CyclicBarrier synchronization object helps a group of threads to wait for each other at given barrier points in a multi-threaded program.

When implementing the merge sort algorithm, one thread for each core is started and they are grouped with a CyclicBarrier object for synchronization. The unsorted data set is divided by the number of cores in the system. Each thread is assigned one unsorted segment and each of them sorts their sections sequentially. Then, all threads wait for others to complete. After this step, every two consecutive threads perform the merge operations in the first round of merging. All threads take part in the first round of merge operations. In the second round of merge operations, only half of the threads take part. Others loop idle. In the final round of merge operations, two threads take part.

We implemented the new parallel merge sort algorithm by using CyclicBarrier as explained above. Using Fork Join or CyclicBarrier for synchronization should not make any difference for the performance of the parallel merge sort algorithm. Because, the amount of time for synchronization should be very small compared to the time taken for merging. In addition, the number of threads is small and the amount of computation is divided almost evenly among the computing threads.

VI. Performance Tests

We compare the running times of three sorting algorithms:

- Sequential merge sort by a single thread as implemented in java.utils.Arrays.sort method in Java library.
- Parallel merge sort in Java library (PMSinJL) as implemented in java.utils.Arrays.parallelSort method.

- Parallel merge sort with double merging (PMSwithDM).

We sorted randomly generated double values with each algorithm. We sorted 5 different sets of numbers: 10million, 20M, 30M, 40M and 50M.

We performed the tests on a machine with a quad core Intel i5 3.1 GHz CPU and 8GB of RAM. It was running Windows 7. We used java 1.8.

When measuring the running times, the same program may take different amounts of time at different times. To measure the running times accurately, it is suggested by Kaminsky in his book [7] that all other programs should be shut down as much as possible and the program should be run 7 times. Then, the minimum of the running times should be used as the running time. We followed this procedure when measuring the running times.

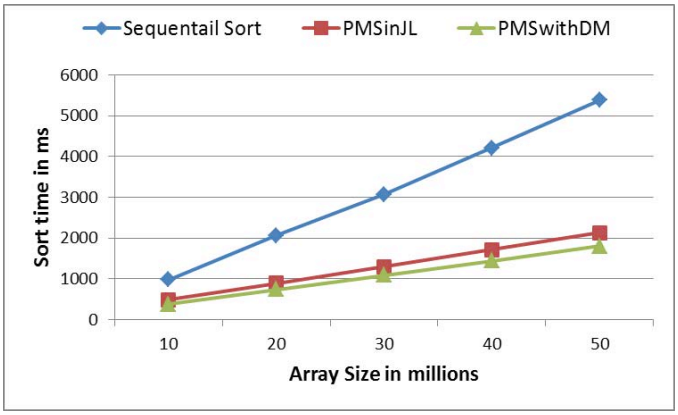

Figure 5 Sorting time comparisons

Figure 5 shows the performance test results for three sorting algorithms. The first observation is that the two parallel merge sort algorithms are much faster compared to the sequential merge sort.

PMSwithDM performs consistently better compared to PMSinJL. When sorting 10M doubles, PMSinJL takes 484ms and PMSwithDM takes 374ms. PMSwithDM is %30 faster than PMSinJL. Similarly, when sorting 50M doubles, PMSwithDM is 20% faster than PMSinJL. As the number of elements to be sorted increases, the difference between the two parallel algorithms decreases. The reason for this is that for larger arrays, initial sequential sorting for each thread takes more time and the process of merging takes comparatively less time. Therefore, the percentage of the difference decreases.

A. Comparison of Merging Times

Parallel merge sort algorithms presented at this study have two separate stages. At first stage, an unsorted array is divided into four parts, since we have running the tests on a quad core machine. Then, each core sorts its section independently. This first stage is the same in the two parallel merge sort algorithms that we compare. They use the same sorting method for this step. Therefore, both algorithms should take almost the same amount of time for this step.

The main difference is on the second stage of sorting when parallel merging is performed. Therefore, we want to compare the running times of both algorithms for parallel merging step.

We cannot measure the running times for these two stages separately in the parallel merge sort in Java Library. However, we can measure the running times of these two stages separately in parallel merge sort algorithm with double merging. Moreover, we can assume that the running times for the first stages of both algorithms are the same. Then, we calculate the running times for the merging stage of PMSinJL.

Figure 6 shows the running times for parallel merges in these two parallel merge sort algorithms. On the average, the merging times of PMSwithDM is %50 faster than the merging times of PMSinJL. These results clearly indicate the superior performance of the new algorithm. The speed of the merge step is doubled by using two threads for merging.

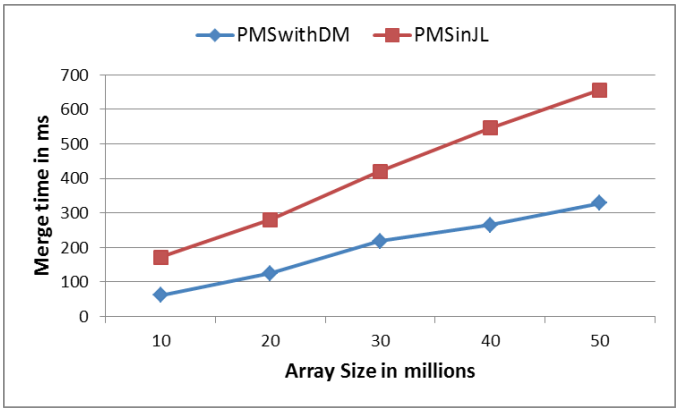

Figure 6 Comparison of merge times

When there are more cores in a system, we can expect the improvement of the new parallel sort algorithm to be more apparent. In this case, the sorting algorithm spends more time merging and we can get more improved speed.

VII. CONCLUSIONS

We have presented a new merge algorithm for parallel merge sort. With this algorithm, two threads can perform one merge operation simultaneously. One thread generates the first half of the sorted values starting from the minimums of the two sorted subsets. The other thread generates the second half of the sorted values starting from the maximums of the two sorted subsets. We have implemented the new parallel merge sort algorithm in Java. We compared the results of the new algorithm with the parallel merge sort implemented in Java library.

The results showed that the new algorithm can provide between %20-%30 percent speed increase in a quad core system when sorting 10M to 50M double numbers. In addition, we separately measured the improvement of merge times in the merge sort algorithm. We have seen that the new algorithm merges two times faster than the parallel merge sort algorithm in Java library.

REFERENCES

[1] P. Tsigas and Y. Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. In Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network based Processing, pages 372–384, 2003.

[2] M. Edahiro. Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA acceleration. In Proceedings of the 2009 Asia and South Pacific Design Automation Conference, pages 230–233, Yokohama, Japan, 2009.

[3] P.J. Varman, S.D. Scheufler, B.R. Iyer, and G.R. Ricard. Merging multiple lists on hierarchical-memory multiprocessors. Journal of Parallel and Distributed Computing, 12(2):171–177, 1991.

[4] J. Singler, P. Sanders, and F. Putze. MCSTL: The Multi-core Standard Template Library. Lecture Notes in Computer Science, 4641:682–694, 2007.

[5] Fork/Join Tutorial by Oracle, http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, Accessed in June 2014.

[6] Java API documentation by Oracle, http://download.java.net/lambda/b78/docs/api/java/util/concurrent/CyclicBarrier.html, Accessed in June 2014.

[7] Alan Kaminsky, Building Parallel Programs: SMPs, Clusters, and Java, Cengage Course Technology, 2010, ISBN 1-4239-0198-3.