www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 6 Issue 6 June 2017, Page No. 21764-21773
Index Copernicus value (2015): 58.10 DOI: 10.18535/ijecs/v6i6.36

Analytical and Experimental Performance Evaluation of Parallel Merge sort on Multicore System

Soha S. Zaghloul, PhD[1], Laila M. AlShehri[2], Maram F. AlJouie[3], Nojood E. AlEissa[4], Nourah A. AlMogheerah[5]

[1]Department of Computer Science, King Saud University,
Riyadh, Saudi Arabia
smekki@ksu.edu.sa

[2]Department of Computer Science, King Saud University,
Riyadh, Saudi Arabia
435203401@student.ksu.edu.sa

[3]Department of Computer Science, King Saud University,
Riyadh, Saudi Arabia
435203327@student.ksu.edu.sa

[4]Department of Computer Science, King Saud University,
Riyadh, Saudi Arabia
435203206@student.ksu.edu.sa

[5]Department of Computer Science, King Saud University,
Riyadh, Saudi Arabia
435203977@student.ksu.edu.sa

Abstract: ***Parallel programming has evolved due to the availability of fast and inexpensive processors. This technique allows us to determine which portions of an algorithm may be executed simultaneously by exploiting different processors. Recently, the focus has shifted to implementing parallel algorithms on multicore systems in order to increase performance. One of the most common operations performed by computers is sorting, which is a permutation on elements. Merge sort is an effective divide-and conquer sorting algorithm that is easy to understand relative to other sorting strategies. The aim of this paper is to describe and evaluate the performance of the parallel merge sort algorithm over its sequential version using the Java threading application program interface (API) environment, which allows programmers to directly manipulate threads in Java programs. The main idea of the proposed algorithm is to distribute the input data elements into several sub arrays according to the number of threads in each level. The experiments were conducted on a multi-core processor and examined the running time, speedup, efficiency, and scalability. The experiments also tested against different array sizes and different numbers of processors. The experimental results on a multi-core processor show that the proposed parallel algorithm achieves a good performance compared to the sequential algorithm.***

Keywords: Parallel programming, Multi-core, Merge sort, Sorting algorithms, Parallel merge sort

1. Introduction

When With the growing number of areas in which computers are being used, the need for more computing power machines has increased. Today's processors can compute at incredible speeds, having the ability to process thousands of operations every second. Many algorithms, however, are not optimized for use with modern processors, and so one way to increase performance is to utilize parallel programming. The time it takes to solve larger problems could be reduced radically if parallel computing power were to be fully utilized. Parallel computing refers to the execution of a program by splitting larger problems into smaller ones, each computed on its own processor and all executed simultaneously. One of the advantages of this form of computing is its use of non-local resources and its ability to overcome memory constraints, which lead to increases in the performance of the system.

Sorting is one of the most common operations performed by computers [1]. Basically, sorting is a permutation function that operates on elements [2]. Basic sorting algorithms arrange the elements of a list in a certain order. Many sorting algorithms exist, and they differ in their functions, performance, applications, and resources utilization [3]. Merge sort is an efficient sorting algorithm that is classified as a divide-and-conquer algorithm. Divide-and-conquer algorithms separate the original data into smaller sets to solve a larger problem. This algorithm is fast, stable, easy to understand, and easy to implement. Using this algorithm via sequential computing in a multi-core environment is inefficient when compared to parallel computing. Therefore, this paper will deal with the implementation of a parallel merge sort algorithm for use within a multicore environment using a threading concept and focusing on analyzing the code region that can be executed concurrently. The overview of the proposed algorithm is shown in Figure 1.

DOI: 10.18535/ijecs/v6i6.36

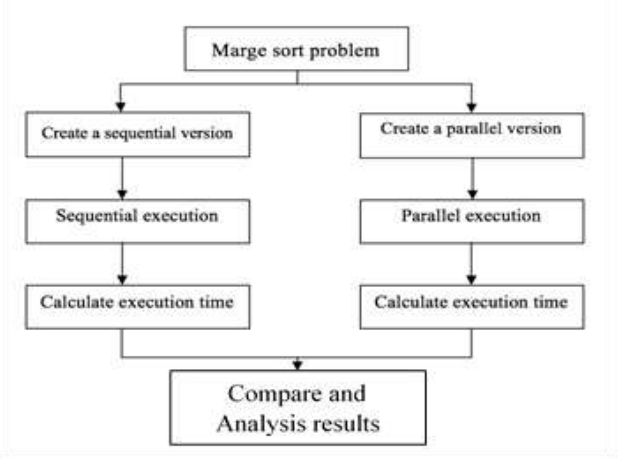

Figure 1: Overview of the proposed work

The rest of this paper is organized into five subsequent sections. In section 2, we give a brief description of the sequential merge sort algorithm along with common parallel models. In section 3, the latest related work is introduced and in Section 4, we demonstrate the proposed parallel merge sort algorithm. Section 5 presents the performance analysis and results. Finally, in Section 6, we draw some conclusions and introduce some suggestions for future work.

2. Background

Set Merge sort is a sorting process that can benefit from using a divide-and-conquer algorithm, which are useful due to their timeless nature [4]. Merge sort arranges the elements of an array in ascending order by recursively dividing the array into two sub arrays until one element left. Then, all arranged sub arrays will be merged into a single, final, sorted array. To sort N elements, a complexity of running time $O(N \log N)$ is fulfilled and known as the finest running time [5]. The divide-and-conquer algorithm used by merge sort can be described as follows: first, it divides the array into two sub arrays, repeating the process until one element is left, which, in fact, is already sorted. Then, the algorithm starts merging the already sorted sub arrays into one array (see Figure 2) [5].

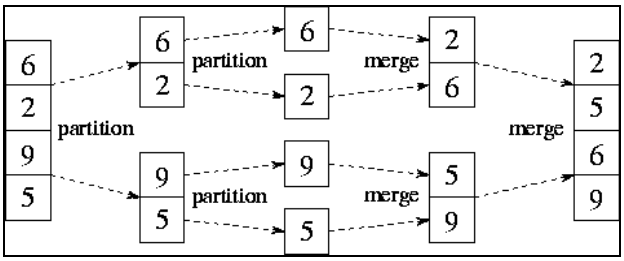

Figure 2: An illustrating merge sort example [6]

The listing in Figure 3 presents the pseudo code of the sequential merge sort with inputs: an array R of size 1 to N and index variables p, q, r where r > q > = p. Furthermore, the array is divided from index p (the first element) until index r (the last element). The output is in ascending order [5].

```
1 MERGESORT(array,p,q,r)
2 if p < r
3  q = (r + p)/2
4  MERGESORT(array, p, q )
5  MERGESORT(array,q+1,r)
6  MERGE(array, p, q, r)
7 MERGE(R, p, q, r)
8  n1 = q-p+1
9  n2 = r-q
10 create arrays L[1...n1+1] and
R[1...n2+1]
11 for i=1 to n1
12    do L[i] = Array[p+i-1]
13   for j=1 to n2
14    do R[j] = array[q+j]
15    L[N1+1] = ∞
16    R[N2+1] = ∞
17 i = 1
18 j = 1
19 for k= p to r
20  do if L[i] ≤ R[j]
21    then array[k] = L[i]
22       i = i+1
23  else array[k] ← R[j]
24       j = j+1
```

Figure 3: The pseudo code of sequential merge sort

In the sequential merge sort algorithm, the MergeSort() method will divide a given array into halves. Then, recursively, the same method is called to divide each half in half again, and so on until each half is a single element. After that, each two halves will be merged into a sorted array by the Merge() method. A number of halves are sorted and merged into a new array for each recursion call of the Merge() method.

Different parallel programming models include the Shared Memory Model (with or without threads) and the Distributed Memory Model (message passing). These models are not specific to a particular memory architecture or machine, and so theoretically, they can be implemented on any underlying hardware [7]. The Shared Memory Model (without threads) is probably the simplest parallel programming model (see Figure 4). The processes and tasks will read and write asynchronously to a mutual shared address space. To avoid race conditions and deadlocks, determining contentions and managing the shared memory's access are achieved using different mechanisms such as locks/semaphores [7].

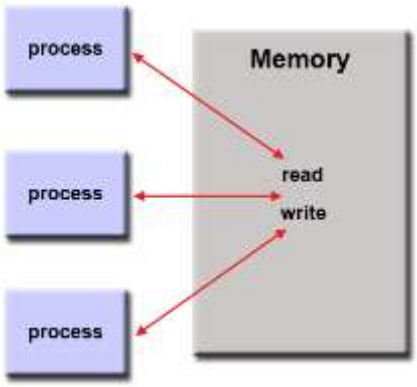

Figure 4: Shared Memory Model (without threads) [7]

From a developer's perspective, the Shared Memory Model can often be simplified. Due to a lack of data "ownership," there is no need to indicate exactly how tasks are manipulating

the data's communication. Thus, accessing and conceiving the shared memory are all the same between processes. Understanding and managing data locality, however, is difficult in terms of performance, which is a disadvantage of this model. Only expert users are typically able to control and understand data locality in this model [7]. In parallel programming using the Shared Memory Model (with threads), threads are usually coupled with shared memory architecture and operating systems. Each process can have several parallel execution paths. The programmer is responsible for shaping all parallelism, and from a programming viewpoint, threads often can be implemented by calling subroutine libraries within the parallel code and by using a group of environment variables and a group of implanted compiler directives in sequential/parallel code [7].

During computation using the Distributed Memory Model (message passing), a group of tasks use their local memory and several tasks can be located across one or more machines. The tasks can swap data over connections by sending and receiving messages (see Figure 5). Additionally, each process performs supportive actions for data transfer, including matching sent actions with received actions.

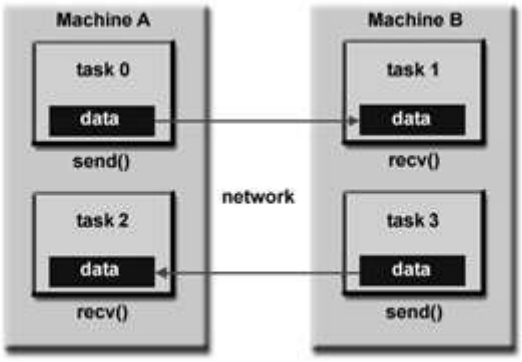

Figure 5: Distributed Memory Model (message passing) [7]

Since the 1980s, many message passing libraries have been available. For this model, the developer is responsible for shaping parallelism. Thus, the implementation of parallel programming in this model is highly difference variable, which makes developing portable applications a challenge for programmers. Usually, implementations include the subroutines of a library, and implanted in the source code are calls to those subroutines.

In 1992, a goal was made to establish a standard interface for message passing implementations (MPI) that would, essentially, replace all other message passing implementations. The first standard interface was released in 1994 as part 1 (MPI-1). Then, in 1996, was the release of part 2 (MPI-2) was released. Finally, in 2012, there was the release of part 3 (MPI-3). MPI implementations are available for almost all known parallel computing platforms. Thus, not all developments include everything from MPI-1, MPI-2, or MPI-3 [7].

3. Related Work

When In [8], only one model of parallel programming is used that explicates threads with shared memory. One thread is similar to a running sequential program, but it can also generate new threads, which are part of the same program. And those generated threads can, in turn, create other threads, and so on. Additionally, they share memory by communicating with the writing and reading fields of the same objects. Theoretically, if all threads have been started but not yet terminated, they are considered to be "running at the same time" in a program. In practice, however, not all threads may be running at any particular moment. Each thread has its own program counter (PC) and call-stack. Also, the threads all share static fields and objects as one collection. According to [8], shared memory is frequently known to be easy and convenient for users because it uses the usual read and write fields for objects to communicate. The negative side of this model, however, is that it is considered error prone because it requires the programmer to know which memory access spaces are engaging in inter-thread communication, and which are not by deeply understanding the code and documentation. Furthermore, for message passing in their model, they are explicating threads without allowing them to share objects. To communicate, the sent message also sends a copy of some data to its receiver. If a thread mistakenly updates a field, other threads will not be affected because each thread has its own object. If processors are occasional, however, we should keep track of the different copies being produced by messages. In Java, threads have a clear scheme regarding how to apply divide-and-conquer parallelism. Each thread creates two assistant threads, left and right, and then waits for them to finish. Critically, the calls to left.start() and right.start() precede the calls to left.join() and right.join(). Thus, we will not have effective parallelism if left.join() comes before right.start(), even if a correct result is produced. Also, it is important to know that the main method calls the run method directly.

In [9], a parallel merge sort algorithm with multithreading is introduced. They chose to focus on merge sort rather than the quick sort as it better parallelizes. In terms of divide and conquer, merge sort can be defined as follows: divide a given array in half to have two sub arrays (an array of size zero or one is already sorted). Then, for each sub array, apply sorting recursively. Finally, arrange the sorted sub arrays into one final sorted array. The parallel processing can be done by sorting the two sub arrays and if this operation is over, proceeding to the merge step. The results show that with two cores, the parallel merge sort is 100% faster than the sequential quick sort and 25% faster than the parallel version of the quick sort with O(n) extra cost of space.

In [10], three versions of the parallel merge sort are studied: first with shared memory (OpenMP), second with message passing (MPI), and third through a hybrid (MPI and OpenMP). OpenMP is selected for the implementation of a shared memory merge sort on SMPs because it combines with freely available compilers such as C/C++. OpenMP is a parallel programming language that allows programmers to create multiple threaded applications. Additionally, OpenMP supports a high level parallel programming model that makes it easier to use than different thread libraries. On the other hand, the MPI is chosen to develop merge sort's message passing on computer's clusters because it is well documented, it can be implemented for different architectures, and it has freely available implementations. The final performance experiments show that with OpenMP, the merge sort's shared memory, achieves the highest speedup compared to the merge sort's message passing with MPI. OpenMP, however, was not as efficient at deciding the best relays for a problem's nature, cluster nodes hardware and software, and the cluster network.

In the case of problems with a fast network, the message passing with MPI can be quicker than hybrid solutions and OpenMP shared memory.

In [11], the authors propose a parallel merge sort using OpenMP, which improves the performance, effectiveness, and expansion of the existing program by adding compiler directives to the code, allowing parallel processing while exerting little effort and minimizing changes to the system. OpenMP can only be used in SMP. After a few years, however, most CPUs provide at least a dual core, so using OpenMP poses no real problem. The experimental performance results of the merge sort in this paper show a speedup of 80% on a 2-processor core and a speed up of 180% on a 4-processor core.

In [12], GPUs are considered highly parallel systems that need hundreds (or even thousands) of threads to achieve bandwidth saturation. The authors designed an elevated performance parallel merge sort for these kind of systems. They use GPUs in order to exploit register communication and to avoid shared memory communication as much as possible to decrease the number of binary searches. To avoid load imbalance, the authors process bigger partitions at a time and exploit more register communication. Four main techniques allow their merge sort to achieve high performance: initially, within each thread, they sort 8 elements, which controls register bandwidth. Second, within a thread block, they apply a binary then linear search approach. Then, they avoid the over segmentation of the register windows and moving shared memory. Finally, by applying these three steps to their merge sort, it becomes suitable for the memory hierarchy and computational strength of GPUs. This process achieves 150% faster performance than a thrust merge sort and is quicker than GPU merge sorts by 70%. Applying these techniques, the authors achieved a 250 MKeys/sec sorting rate.

The proposed load-balanced merge sort used in [13] involves several processors in the sorting process. This model divides the input data between all available processors in each period. This makes processors work as much as possible. The main idea in [13] is to initiate a parallel execution of the sort by distributing each part of the input list onto different processors. For instance, every processor saves an identical number of key elements. Thus, the processors together will be merging between as throughout the execution. This method employs more parallelism and thus reduces the running time of as sort algorithm. The proposed algorithm in this paper determines the minimum number of keys by creating a histogram. All processors together are sorting one list, which will be distributing between the available processors. Each processor will compute the histogram of its keys. After that, the histogram is used to merge these processes. Significant performance is achieved by enhancing the speed up to the number of processors minus one divided by the number of processors. After using this algorithm, the running time is reduced to a highest speedup of 860% on a Cray with 32 processors. The results on an 8-node PC cluster achieved a speedup of an added 130% with a 180% upper bound at sorting integers of 4M 32-bit.

In [14], a state-of-the-art, scalable parallel merge tree is proposed. The main difference in this model is that instead of assumes a constant sorting rate, it allows a variable sorting rate in the parallel merge-tree. This assumption deletes the nasty memory bandwidth requirement by proposing a solution that involves optimizing the parallel merge-tree at different data sets collected in a random way. Also, the authors offer three clear positive comparisons to the sequential implementation of sort the sorters: an increase in the sorting rate by one for each cycle, a reduction in the number of times the algorithm is run relative to the sequential version, and a different means of calculating the size of the sorted data that does not involve the available space on FPGA memory, which is used only as communication buffers for the main memory.

Nanjesh et al. [15] propose a parallel merge sort algorithm. A single node (desktop PC) paradigm that involves only two cores is used to evaluate the performance on different RAM sizes using MPI and PVM, which are software tools for the parallel networking of computers. One of these cores acts as master, which handles requests from the user and assigns the problem to a slave, while another core acts as a slave, which accepts problems from the master and sends back solutions. The communication between master and slave is accomplished using MPICH-2 which is a new implementation of MPI provided to assist MPI-1 and MPI-2. The routines in MPICH2 have very low communications overhead and are significantly faster compared to the "classic" MPI. The authors compare the time taken to solve the parallel algorithm using MPI and PVM with its sequential version. The results show that the sequential version is faster than the parallel version due to the overhead in communication involved in the parallel execution, which could be overcome by increasing the number of nodes. Also, the authors show that the MPI is faster than PVM in terms of performance.

Zhang et al. [16] propose an improved two-way merge sort algorithm based on OpenMP. The concept of a two-way merge sort is to repeatedly merge two sorted subsets of data into a new one until the data is sorted. During the execution of the program, parallel threads are initiated for each merge sort operation via parallelizing the operation of the merge, then they are divided among multiple cores to be processed. Their algorithm has three main functions: Merge() is used to merge the sorted subset into new subset; MergePass() is implemented for each trip and pairwise merge on the subsets; Mergesort() is mainly used as an interface that can be initiated by another module. To produce the final ordered list, the MergePass() function will be called many times. The only modification necessary for parallel processing is to run the MergePass() function while the Mergesort() and Merge() functions are implemented sequentially. The performance of this model was tested using randomly produced data with different lengths of unsorted elements and numbers of threads. These experiments show that when compared to the traditional algorithm, this model is more efficient.

Khan and Rajesh [17] propose an adaptive framework to analyze the parallel merge sort using MPI. Their methodology uses the master and slave model in tree form, where a list of elements received by each process from its predecessor is portioned to two halves. One half will be kept for itself and the second half is given to its successor. When the job is completed, the sorted data is sent back to the predecessor. This process will keep going up to the root node. Rank calculation is used to address the predecessor and successor concept. As a result, the parallelized version is faster compared to the sequential version.

In [18], the authors present the implementation of different

commutation algorithms in parallel using OpenMP application program interface (API). Using these algorithms, code is executed in parallel, taking advantage of multi-core CPUs that can perform multiple tasks simultaneously. On the other hand, writing parallel code is more complex than writing serial code. The OpenMP API allows serial code to run simultaneously on multi core processors. In this paper, the authors indicate the sections of code they want executed simultaneously using OpenMP. The paper's aim is to present the performance of the parallel programming model over the sequential programming model using OpenMP. They describe the merge sort and Floyd's algorithm parallel by using OpenMP to reduce the running time on multiple cores, achieving good performance results. The algorithm's execution time is tested on a dual core processor, and a quad core to measure performance. Moreover, the performance demonstrated in this paper is tested for both merge sort and Floyd's algorithm. The results show that the parallel implementation is about 100% quicker than the sequential, and the speedup is linear.

Some large, complex computational numeric formulas can make running a problem sequentially take a long time since sequential models use just one processor at a time. In [19], the authors describe two numerical problems. They present the solutions of the matrix multiplication algorithm and the Floyd-Warshall algorithm for parallel programming, which takes the smallest times on multiprocessor machines. Here, they use a shared memory strategy to implement the simultaneous execution of computational instruction. For this paper, they use the OpenMP API to clearly express multi-threading. On the other hand, they represent the size of a matrix as the number of threads in the matrix multiplication algorithm. Also, for the Floyd-Warshall algorithm, they give each thread a chunk size that specifies the number of iterations necessary to achieve the shortest path between two vertices on the graph. To test the performance of these algorithms, they use a sequential model. After that, they test these algorithms using a parallel model by running them in a multi-core machine. In both methods, the execution time for sequential processing takes a longer time than did the parallel program. Execution time is recorded based on the size of the dataset. For the largest data set used in the experiments, they achieved the largest speedup. They used eight hundred elements for the matrix multiplication and eighty nodes for the Floyd-Warshall algorithm. The parallel models of these algorithms achieved speedups of nearly 50%.

In [20], the authors show two experimental algorithms, which are used to calculate the value of Pi (π) and gauss elimination in order to examine the performance of parallel processing versus sequential execution. They test parallelism by calculating a number of linear equations and then implementing both sequential execution and parallel execution. Finally, they compare and analyze the results. In this paper, they implement a parallel algorithm using OpenMP. The parallel execution performance is based on the number of available cores in the machine, the memory hierarchy, and the synchronization costs. In this paper, the authors describe the algorithms that achieve the fastest running time through OpenMP parallelization on multi-core processors compared to sequential execution. The result of their experiments show that the parallel implementation is about 100% faster than the sequential method and that the speedup is linear.

4. Parallel Merge Sort

When The design of our parallel merge sort algorithm benefits from applying the recursive method that divides the problem size by two between a number of threads, which will increase the speed of sorting the elements, especially with larger problems. The implementation of the merge sort has complexity of $O(N \log N)$, which means more increase in the speed of sorting the elements as the problem size increases compared to other sorting algorithms [5]. The design of our parallel algorithms takes advantage of multi-core hardware capabilities as much as possible. In this paper, we have implemented the sequential and the parallel versions of merge sort algorithm. Using the parallel version, we employ a threading concept. The parallel merge sort algorithm divides an input array between two threads at every runtime. Thus, each thread is executed independently from the other threads. These threads executed by the divide and sort method then merge the input array simultaneously. As a result, the running time of the merge sort program is reduced.

The parallel merge sort algorithm is implemented with Java thread API. Java thread is an API [21] used explicitly for multi-threaded programming using a thread library, which is shared memory parallelism. Figure 6 shows a brief overview of our implementation of the shared memory parallelism for the merge sort algorithm. At runtime, threads will run the parallel parts of the code and execute them in different processors at the same time sharing the same memory and address spaces. This increases the performance of the program's execution time by taking advantage of multiple cores. The threads use the cores to indicate thread level parallelism in the program.

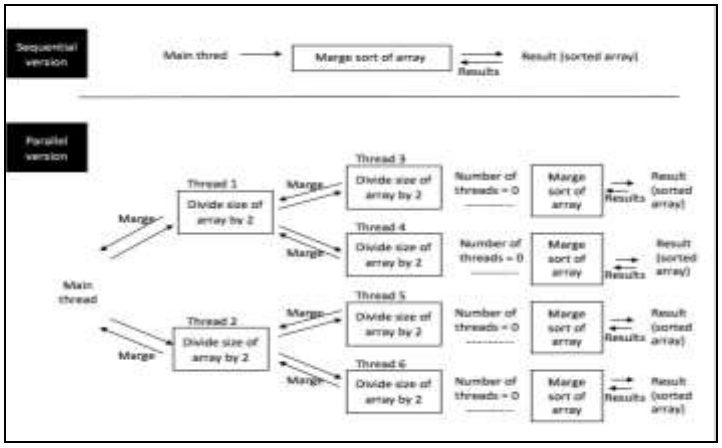

Figure 6: Parallelism of merge sort in Java threads API

In this model, the tasks are executed in parallel by threads using the multiple cores of a processor. More specifically, we will have a main thread that will assign tasks to worker threads (child threads). Private data will be visible only to the single thread and public data will be visible to all threads [21]. Java threads API is supported on many programming shared memory multiprocessing platforms. The Java programming language can use the thread library to compile directives and environment variables to implement multi-threaded processing completely managed by Java threads. A natural way to redesign the merge sort algorithm to run on a parallel computing platform is to split the work that can be run "in parallel" to do the work required at each level of the tree simultaneously.

Note that our parallel solution for each iteration will go one level up starting from the bottom of the tree, which is called an iterative approach. Simply stated, if we have an array with a given number of values, the merge can be executed on them using each individual thread. Furthermore, every thread in each level will execute in parallel and read an independent sub data of the original array. Then, these threads will write in memory the output array result. Overall, the number of the tree's leaf nodes is the same as the number of thread *K*. The number of *K* can be thought of as $2^{\#\text{tree levels}}$. Another parent thread will make use of the newly sorted list, then merge it with another child's sorted list. This process is continued until the first main thread is reached. Figure 7 shows an example of what we described for an array with 8,000 elements and *K* equals 8.

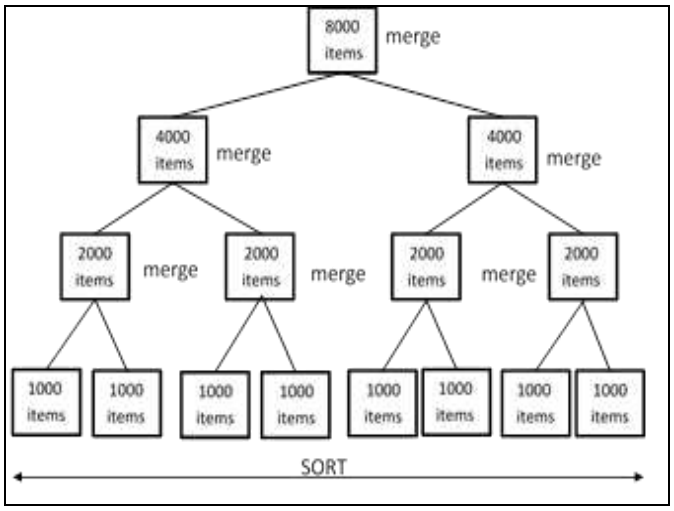

Figure 7: An illustration of the proposed parallel merge sort algorithm with 8,000 elements

Furthermore, the parallel version of the merge sort is implemented by converting the sequential merge sort algorithm. Implementation of the parallel merge sort, however, has an additional parallel section in the code. Using the Java threads API, we create the parallel section of code using a class that implements a runnable interface. In this class is an overriding method named run(). This method is implemented every time an instance of an object from the parallel merge sort class is created. Figure 8 shows the parallel version of the merge sort that includes the added parallelism section of code. Mainly, the parallelism section of the merge sort algorithm is used to divide the array into halves, to assign each half to a thread, to recursively sort the halves, and finally to combine the sorted halves into one array all accomplished in parallel. Thus, Each thread has a unique piece of the original array.

The main method will run the parallel merge sort version by calling the parallel section and creating an instance of the parallel merge sort class that includes a parallel section. A parallel merge sort creates an object that divides the array elements between threads, which are generated automatically. Figure 9 shows the main method of the merge sort. By default, the main input array is communal to all threads. As a result, the recursive call is applied from the parallel version.

In this implementation, parallelization occurs at the merge operation. This is done by a conversion on the creating of threads. Apparently, this method can make parallel merge sort run faster than the parallel quick sort. This is most likely because parallelizing the merge process is simpler than the quick sort's partition process [10]. We cannot, however, execute the merge method of the merge sort algorithm in parallel using the OpenMP API [19].

```
public class ParallelMergeSort implements Runnable
{
    //Declarations of array and thred level

    //constructor
    public ParallelMergeSort(long[] a, int threadlevel)
    {
        this.array = a;
        this.threadlevel = threadlevel;
    }//end of constructor

    //this method invoked by start() to start the process.
    public void run()
    {
        parallelMergeSort(array, threadlevel);
    }

    // this method include the parallel part of the parallel marge sort algorithm
    private static void parallelMergeSort(long[] array, int threadlevel)
    {
        if (array.length < 2) // length of the array that want to be sorted
        { return;}

        if (threadlevel == 0)
        {
            mergeSort(a); // as the method in the sequential
            return;
        }

        // split array in half
        long[] left ; half of the array from start index  until to the middle index.
        long[] right; half of the array from middle index  until to the last index.

        // sort each half (in parallel)
        Thread lThread = new Thread(new ParallelMergeSort(left, threadlevel - 1));
        Thread rThread = new Thread(new ParallelMergeSort(right, threadlevel - 1));
        lThread.start();
        rThread.start();
        lThread.join();
        rThread.join();
        // merge them back together in parallel
        merge(left, right, array);
        return array;
    }


 }//end the parallel marge sort class
```

Figure 8: The code of the parallel section of the merge sort algorithm

```
public class SharedMemory { //start of implementation the parallel marge sort
 void main
    {
        long[] array; // array to store random numbers that needed to be sorted
        //instantiate the main thread and pass the array and number of threadslevel
        Thread thrd = new Thread(new ParallelMergeSort(array, Threadlevel));
        //Threadlevel to specify the number of threads you want to created
        thrd.start();// method invokes the run() method.
        thrd.join();//join() waits for a thread to complete

    }//end main

}//end class
```

Figure 9: The code of the main class for the parallel merge sort version

5. Performance Analysis and Results

To analyze the performance of the algorithm, we ran the sequential and parallel versions of the merge sort algorithm on an intel Core i7-3630QM CPU @ 2.40GHz. In both experiments, the results are discussed and evaluated in terms of execution time, speedup, efficiency, and scalability performance metrics. The algorithms are evaluated against different sizes of array (*N*), which is generated randomly, and different number of processors (*K*). For each input of length *N*, the smallest running time of multiple runs is considered to record the results.

5.1 Evaluation of the execution time

Table 1 in the appendix presents the execution time of both the sequential and parallel algorithms in seconds according to the different problem sizes *N* and 8 processors. As illustrated in Figure 10, for the sequential algorithm, when increasing the data size, the execution time increases because of the increased

time needed for division, merging, and a number of comparisons. For the parallel algorithm, however, the performance is sensitive to the available number of system cores (*K*) where, for all *N* values, the more cores we have, the less running time is required due to the better work distribution among available processors. As a result, the parallel version of the merge sort is better and faster in terms of execution time with different sizes of data, both small and large.

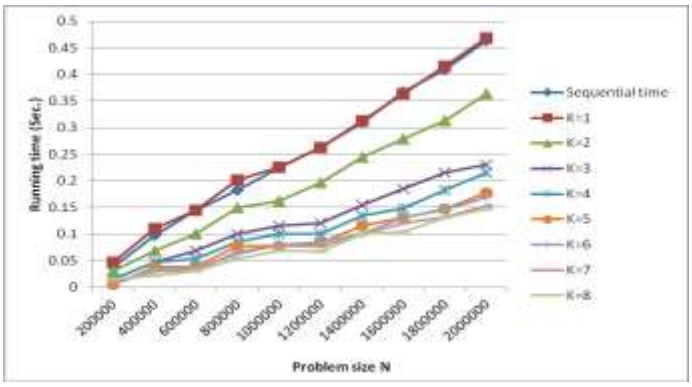

Figure 10: Running time against problem size in sequential and parallel merge sort algorithms

5.2 Speedup Evaluation

In terms of speedup, the performance of the parallel algorithm is measured in Tables 2 and 4. The speedup is used to demonstrate the gain of parallelizing a program against running the program sequentially. The actual speedup can be calculated as given in the formula (1). Also, Amdahl's law can be calculated using formula (2).

$$Speedup(N,K) = \frac{T_{seq}(N,1)}{T_{par}(N,K)} \quad (1)$$

$$Speedup(N,K) = \frac{1}{\frac{F+(1-F)}{K}} \quad (2)$$

Such that *F* represents the sequential fraction that can be calculated as (3). Table 3 shows the results of *F*.

$$F = \frac{K*T(N,K)-T(N,1)}{K*T(N,1)-T(N,1)} \quad (3)$$

Graphical representations in Figure 11 illustrate the calculated speedup against *K* number of processors using Amdahl's law on the parallel algorithm with different problem sizes. The analytical results show that Amdahl's speedup and actual speedup are the same. If the number of processors (*K*) are increased, the speedup increases as well, which is shown in Figure 11. In one case, when *N* is equal to 200,000, the speedup decreases if the number of processors is higher than 5.

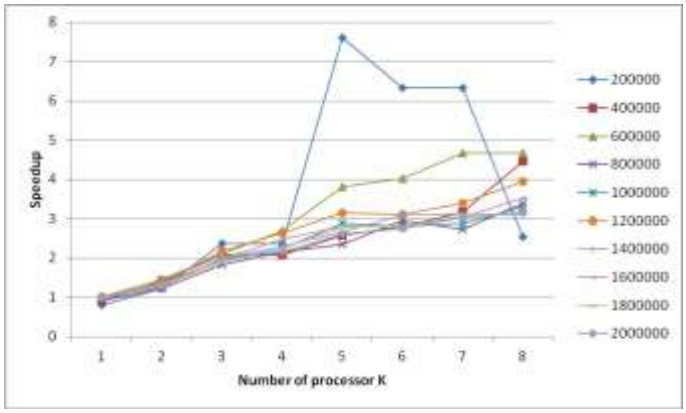

Figure 11: Speedup against number of processors

5.3 Efficiency Evaluation

Parallel efficiency is the ratio between speedup and the number of processors (*K*) as shown in formula (4). Formula (5) calculates Amdahl's law in terms of efficiency. Tables 5 and 6 present the results we obtained for each *N* problem and each *K* value when formulas 4 and 5 are applied. We obtained the same results for both formulas.

$$E(N,K) = \frac{Speedup(N,K)}{K} \quad (4)$$

$$E(N,K) = \frac{1}{F*K+(1-F)} \quad (5)$$

Figure 12 demonstrates the efficiency results against the number of *K* ranging from 1 to 8 and for all *N* sizes. We can conclude that the efficiency decreases when the number of processors (*K*) increases. With *N* equal to 200,000, however, and with 5 processors, the efficiency increased enormously compared to the other values due the high speedup value.

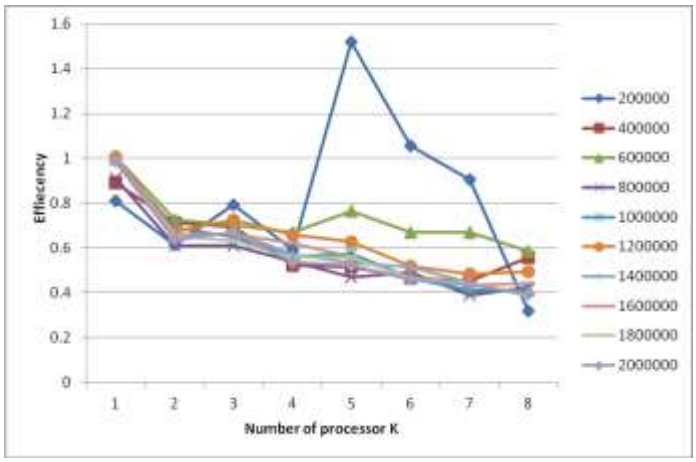

Figure 12: Efficiency against number of processors

5.4 Scalability Evaluation

If we add more processors, we should be able to increase the size of a problem that we can solve in a given amount of time. The scalability of our parallel merge sort can be calculated using formula (6).

$$Sizeup(T,K) = \frac{N_{par}(T,K)}{N_{seq}(T,1)} \quad (6)$$

Figure 13 illustrates the experimental results we obtained. The results proved that an increased number of processors leads to an increase in the problem sizes that can be computed at the same given time by the parallel merge sort algorithm.

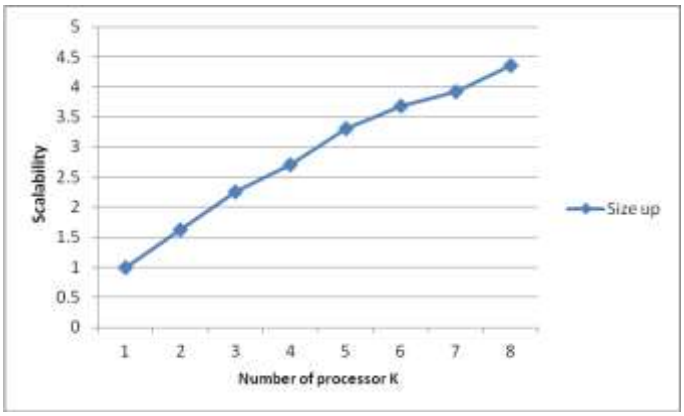

Figure 13: Scalability against number of processors

6. Conclusion

In this paper, we present the performance evaluation achieved through the implementation of the parallel merge sort algorithm in Java threads API using a multi-core system. The evaluation of the algorithm calculated the execution time, speedup, efficiency, and scalability. Furthermore, the algorithm was tested against different array sizes with *N* elements, which were generated randomly, and on multiple processors. The performance of the parallel model was compared with the use of the sequential merge sort. In general, our implementation of the parallel merge sort algorithm achieves a good performance result by reducing the execution time. The results achieved by applying the sequential algorithm with small amounts of data gives a good performance, but as the data set size increases, the performance falls. The parallel algorithm gives a better result with increased data set size relative to the increased number of processors. The speedup is achieved by dividing the execution time of the sequential version over the execution time of the parallel version. Also, the efficiency is calculated by the ratio between speedup and the number of processors. Finally, the results demonstrate that increasing the number of threads leads to an increase in the problem sizes that can be handled, which reflects the scalability of the parallel algorithm.

In future work, the performance on different RAM sizes could be evaluated using the same parallel merge sort algorithm to solve problems of larger size. Also, the work could be extended and compared with other parallel merge sort algorithms that are implemented using different model or on different platforms.

References

[1] S. Akl, Parallel sorting algorithms, Orlando: Academic Press, 1985.

[2] R. Rashidy, S. Yousefpour, M. Koohi, "Parallel bubble sort using stream programming paradigm," In 5th International Conference on Application of Information and Communication Technologies (AICT), 2011.

[3] S. Altukhaim, "Bubble Sort Algorithm," Florida Institute of Technology, Melbourne, Florida, USA, 2003.

[4] R. Zadeh, "DAO: Distributed Algorithms and Optimization," Stanford.edu, Apr. 5, 2016. [Online]. Available: https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture12/cme323_lec12.pdf. [Accessed: Jan. 12, 2017].

[5] S. Qin, "Merge sort Algorithm," Florida Institute of Technology, Melbourne, 2008. [Online]. Available: http://cs.fit.edu/~pkc/classes/writing/hw13/song.pdf. [Accessed: Jan. 12, 2017].

[6] I. Foster, "11.4 Mergesort," Mathematics and Computer Science, 1995. [Online]. Available: http://www.mcs.anl.gov/~itf/dbpp/text/node127.html. [Accessed: Jan. 12, 2017].

[7] B. Barney, "Introduction to Parallel Computing," Department of Energy by Lawrence Livermore National Laboratory. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/. [Accessed: Jan. 12, 2017].

[8] D. Grossman, "A sophomoric introduction to shared-memory parallelism and concurrency," Lecture notes, Department of Computer Science & Engineering, University of Washington, 2012. [Online]. Available: http://homes.cs.washington.edu/~djg/teachingMaterials/spac/sophomoricParallelismAndConcurrency.pdf. [Accessed: Dec. 24, 2016].

[9] D. Huba, "Parallel Merge sort," Oct., 2010. [Online]. Available: http://dzmitryhuba.blogspot.com/2010/10/parallel-merge-sort.html. [Accessed: Dec. 24, 2016].

[10] A. Radenski, "Shared memory, message passing, and hybrid Merge sorts for standalone and clustered SMPs," In Proc. PDPTA'11, the 2011 international conference of parallel and distributed processing technique and applications. CSREA press, pp. 367–373, 2011.

[11] J. Park, K. Lee, T. Kim, "Parallel Merge sort Implementation Using OpenMP," School of information communication engineering, Sungkyunkwan University, Suwon, GyeongGi-Do, South Korea, 2011.

[12] A. Davidson, D. Tarjan, M. Garland, J. Owens, "Efficient parallel Merge sort for fixed and variable length keys," In Proceedings of the IEEE Innovative Parallel Computing (InPar), 2010.

[13] M. Jeon, D. Kim, "Parallel Merge Sort with Load Balancing," In International Journal of Parallel Programming, vol. 31, no. 1, pp. 21-33. Springer, 2003.

[14] W. Song, D. Koch, M. Lujan, J. Garside, "Parallel Hardware Merge sorter," In IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016.

[15] B. R. Nanjesh, et al., "Parallel Merge sort based performance evaluation and comparison of MPI and PVM," In Proceedings of 2013 IEEE Conference on Information and Communication Technologies (ICT), 2013.

[16] J. Zhang, et al., "Algorithm Improvement of Two-Way Merge sort Based on OpenMP," Applied Mechanics and Materials, Vols 701-702, pp 24-29, Trans Tech Publications, Switzerland, 2015.

[17] H. U. Khan, R. Tiwari, "An Adaptive Framework towards Analyzing the Parallel Merge sort," In International Journal of Science and Research (IJSR), 2012.

[18] M. Basthikodi, W. Ahmed, "Parallel Algorithm Performance Analysis using OpenMP for Multicore Machines," In International Journal of Advanced

Computer Technology (IJACT), vol. 4, no. 5, pp. 28-32, 2015.

[19] P. Kulkarni, S. Pathare, "Performance Analysis of Parallel Algorithm over Sequential using OpenMP," In IOSR Journal of Computer Engineering (IOSR-JCE), vol. 16, no. 2, pp. 58-62, 2014.

[20] S. K. Sharma, K. Gupta, "Performance Analysis of Parallel Algorithms on Multi-core System using OpenMP," In International Journal of Computer Science, Engineering and Information Technology (IJCSEIT), Vol.2, No.5, Oct. 2012.

[21] M. Oriol, "Java Programming: Concurrent Programming in Java," May 10, 2007. [Online]. Available: se.inf.ethz.ch/old/teaching/ss2007/0284/book/Threads.pdf . [Accessed: Dec. 29, 2016].

Appendix

Table 1: Execution time of sequential and parallel

N	*Sequential time*	*Parallel time at K=1*	*Parallel time at K=2*	*Parallel time at K=3*	*Parallel time at K=4*	*Parallel time at K=5*	*Parallel time at K=6*	*Parallel time at K=7*	*Parallel time at K=8*
200000	0.038	0.047	0.031	0.016	0.016	0.005	0.006	0.006	0.015
400000	0.098	0.11	0.069	0.047	0.047	0.038	0.035	0.031	0.022
600000	0.145	0.144	0.1	0.069	0.054	0.038	0.036	0.031	0.031
800000	0.183	0.201	0.15	0.1	0.085	0.078	0.062	0.067	0.054
1000000	0.224	0.226	0.162	0.115	0.1	0.078	0.08	0.078	0.068
1200000	0.264	0.262	0.196	0.121	0.1	0.084	0.085	0.078	0.067
1400000	0.309	0.313	0.244	0.154	0.135	0.116	0.1	0.099	0.1
1600000	0.367	0.364	0.279	0.185	0.148	0.131	0.131	0.121	0.104
1800000	0.409	0.415	0.313	0.216	0.183	0.147	0.146	0.132	0.131
2000000	0.464	0.469	0.363	0.231	0.216	0.178	0.169	0.153	0.147

Table 2: Calculation of the actual speedup

N	*Speed up at K=2*	*Speed up at K=3*	*Speed up at K=4*	*Speed up at K=5*	*Speed up at K=6*	*Speed up at K=7*	*Speed up at K=8*
200000	1.22580645	2.375	2.375	7.6	6.33333333	6.33333333	2.53333333
400000	1.42028986	2.08510638	2.08510638	2.57894737	2.8	3.16129032	4.45454545
600000	1.45	2.10144928	2.68518519	3.81578947	4.02777778	4.67741935	4.67741935
800000	1.22	1.83	2.15294118	2.34615385	2.9516129	2.73134328	3.38888889
1000000	1.38271605	1.94782609	2.24	2.87179487	2.8	2.87179487	3.29411765
1200000	1.34693878	2.18181818	2.64	3.14285714	3.10588235	3.38461538	3.94029851
1400000	1.26639344	2.00649351	2.28888889	2.6637931	3.09	3.12121212	3.09
1600000	1.31541219	1.98378378	2.47972973	2.80152672	2.80152672	3.03305785	3.52884615
1800000	1.30670927	1.89351852	2.23497268	2.78231293	2.80136986	3.09848485	3.1221374
2000000	1.27823691	2.00865801	2.14814815	2.60674157	2.74556213	3.03267974	3.15646259

Table 3: The calculation of the sequential fraction F

N	*F at K=2*	*F at K=3*	*F at K=4*	*F at K=5*	*F at K=6*	*F at K=7*	*F at K=8*
200000	0.631578947	0.131578947	0.228070175	-0.085526316	-0.010526316	0.01754386	0.308270677
400000	0.408163265	0.219387755	0.306122449	0.234693878	0.228571429	0.202380952	0.113702624
600000	0.379310345	0.213793103	0.163218391	0.077586207	0.097931034	0.082758621	0.101477833
800000	0.639344262	0.319672131	0.285974499	0.282786885	0.206557377	0.260473588	0.194379391
1000000	0.446428571	0.270089286	0.261904762	0.185267857	0.228571429	0.239583333	0.204081633
1200000	0.484848485	0.1875	0.171717172	0.147727273	0.186363636	0.178030303	0.147186147
1400000	0.579288026	0.247572816	0.249190939	0.219255663	0.188349515	0.207119741	0.226999538
1600000	0.520435967	0.25613079	0.204359673	0.196185286	0.228337875	0.217983651	0.181004282
1800000	0.530562347	0.292176039	0.263243684	0.199266504	0.228361858	0.209861451	0.223192455
2000000	0.564655172	0.246767241	0.287356322	0.229525862	0.237068966	0.218031609	0.219211823

Table 4: Calculation of the Amdahl's law speedup

N	*Speed up at K=2*	*Speed up at K=3*	*Speed up at K=4*	*Speed up at K=5*	*Speed up at K=6*	*Speed up at K=7*	*Speed up at K=8*
200000	1.22580645	2.375	2.375	7.6	6.33333333	6.33333333	2.53333333
400000	1.42028986	2.08510638	2.08510638	2.57894737	2.8	3.16129032	4.45454545
600000	1.45	2.10144928	2.68518519	3.81578947	4.02777778	4.67741935	4.67741935
800000	1.22	1.83	2.15294118	2.34615385	2.9516129	2.73134328	3.38888889
1000000	1.38271605	1.94782609	2.24	2.87179487	2.8	2.87179487	3.29411765
1200000	1.34693878	2.18181818	2.64	3.14285714	3.10588235	3.38461538	3.94029851
1400000	1.26639344	2.00649351	2.28888889	2.6637931	3.09	3.12121212	3.09
1600000	1.31541219	1.98378378	2.47972973	2.80152672	2.80152672	3.03305785	3.52884615
1800000	1.30670927	1.89351852	2.23497268	2.78231293	2.80136986	3.09848485	3.1221374
2000000	1.27823691	2.00865801	2.14814815	2.60674157	2.74556213	3.03267974	3.15646259

Table 5: The Efficiency results for each N value and K

N	*K=1*	*K=2*	*K=3*	*K=4*	*K=5*	*K=6*	*K=7*	*K=8*
200000	0.808510638	0.612903226	0.791666667	0.59375	1.52	1.055555556	0.904761905	0.316666667
400000	0.890909091	0.710144928	0.695035461	0.521276596	0.515789474	0.466666667	0.451612903	0.556818182
600000	1.006944444	0.725	0.700483092	0.671296296	0.763157895	0.671296296	0.668202765	0.584677419
800000	0.910447761	0.61	0.61	0.538235294	0.469230769	0.491935484	0.390191898	0.423611111
1000000	0.991150442	0.691358025	0.649275362	0.56	0.574358974	0.466666667	0.41025641	0.411764706
1200000	1.007633588	0.673469388	0.727272727	0.66	0.628571429	0.517647059	0.483516484	0.492537313
1400000	0.987220447	0.633196721	0.668831169	0.572222222	0.532758621	0.515	0.445887446	0.38625
1600000	1.008241758	0.657706093	0.661261261	0.619932432	0.560305344	0.46692112	0.433293979	0.441105769
1800000	0.985542169	0.653354633	0.63117284	0.558743169	0.556462585	0.466894977	0.442640693	0.390267176
2000000	0.989339019	0.639118457	0.66955267	0.537037037	0.521348315	0.457593688	0.433239963	0.394557823

Table 6: The Amdahl's law in terms of efficiency results for each N value and K

N	*K=1*	*K=2*	*K=3*	*K=4*	*K=5*	*K=6*	*K=7*	*K=8*
200000	0.808510638	0.612903226	0.791666667	0.59375	1.52	1.055555556	0.904761905	0.316666667
400000	0.890909091	0.710144928	0.695035461	0.521276596	0.515789474	0.466666667	0.451612903	0.556818182
600000	1.006944444	0.725	0.700483092	0.671296296	0.763157895	0.671296296	0.668202765	0.584677419
800000	0.910447761	0.61	0.61	0.538235294	0.469230769	0.491935484	0.390191898	0.423611111
1000000	0.991150442	0.691358025	0.649275362	0.56	0.574358974	0.466666667	0.41025641	0.411764706
1200000	1.007633588	0.673469388	0.727272727	0.66	0.628571429	0.517647059	0.483516484	0.492537313
1400000	0.987220447	0.633196721	0.668831169	0.572222222	0.532758621	0.515	0.445887446	0.38625
1600000	1.008241758	0.657706093	0.661261261	0.619932432	0.560305344	0.46692112	0.433293979	0.441105769
1800000	0.985542169	0.653354633	0.63117284	0.558743169	0.556462585	0.466894977	0.442640693	0.390267176
2000000	0.989339019	0.639118457	0.66955267	0.537037037	0.521348315	0.457593688	0.433239963	0.394557823