Parallel Merge Sort

Anthony-Claret Onwutalobi

Dajie Wang

Nikolay Vasilev

Helsinki, 29 April 2011

UNIVERSITY OF HELSINKI

Department of Computer Science

Table of Contents

1 Team Members 3
2 Introduction And Problem 3
3 Solution Method..4
4 User Instructions 7
5 Evaluation of Solution (correctness, performance, scalability) 8
5.1 Correctness 8
5.1.1 Mutex 9
5.1.2 Deadlock..9
5.1.3 Starvation 9
5.2 Performance...9
6. Performance Tests..10
7. Summary...11
8. References 12

1 Team Members

This Concurrent Programming project was completed with the joint forces of Anthony-Claret Onwu-talobi, Dajie Wang, and Nikolay Vasilev, While contribution to the project was spread fairly equally among the team members,

2 Introduction And Problem

The aim of our project was to implement a classic sorting algorithm utilising concurrency. We decided on Merge Sort because of its inherent parallelism. Sorting in general has been hailed as a cornerstone in Computer Science [SHG09], as many computer algorithms rely heavily upon it. Such an algorithm is for example the classic Kruskal algorithm for constructing a minimum spanning tree of an undirected graph [Kru56]. Merge Sort is based on the popular Divide-and-conquer approach [CLR02], where the problem at hand is broken into smaller problems of similar nature, which are in turn solved recursively, and the solutions are then combined.

The logic of Merge Sort itself is very straightforward. It consists in dividing the array into two parts of approximately the same length (the difference between the lengths of the two subarrays is at most 1). The picture below illustrates the working principle behind the sorting algorithm.

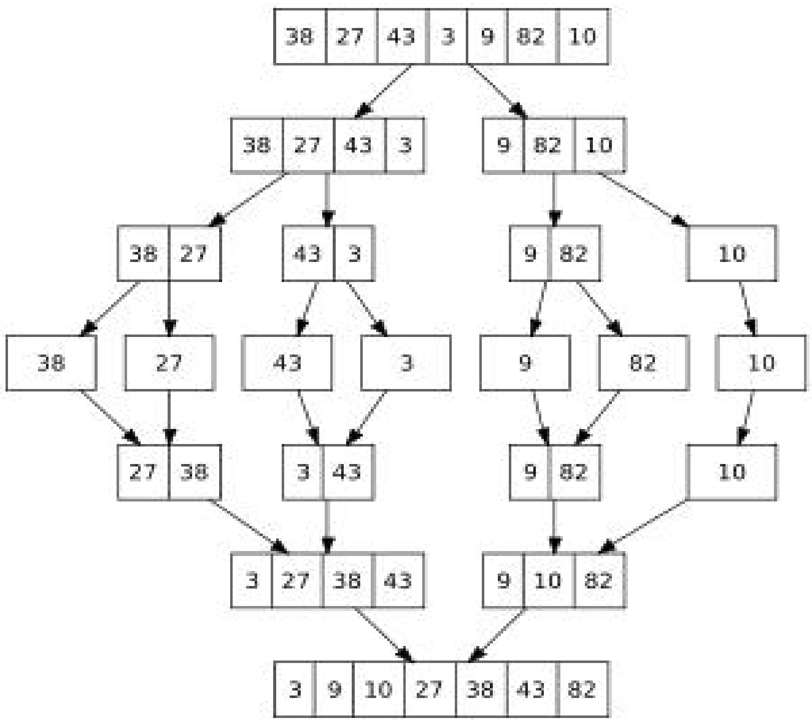


FIGURE 1: MERGE SORT [OUL11]

The algorithm calls itself recursively every time, unless the array that has to be sorted is of length 1, in which case the array is in order, so the algorithm just returns. Merging two sorted arrays is a key procedure in Merge Sort. Thomas Cormen et al. [CLR02] compare it to merging two sorted packs of cards, where we compare the two cards on top of each pack, pick the smaller one and place it face down on top of the cards sorted so far, thus leaving the one underneath it as the new top of the old pack (unless, of course, the chosen card was the last one in its pack, in which case we discontinue the comparisons and place the entire pack of remaining cards face down on the pack of sorted cards). The time complexity of Merge Sort can be easily calculated to be $O(n \log n)$, where n is the number of elements in the array.

3 Solution Method

A question arises whether we can harness concurrency to sort the array even faster. There does not seem to be any obstacle in general to subarrays being sorted at the same time and simply merged at the end. One can be easily tricked into believing that this might be orders of magnitude faster or at least faster by a constant factor. The truth is, however, very different, because while the initial array might be split into *m* parts, each sorted by a different thread, executing the final merge ”all at once” will pay through the nose. And the bigger *m*, the slower the final **Merge** will complete, thus outweighing any efficiency gained through concurrency. In our solution we strove to preserve and even increase the concurrency benefit by enhancing the **Merge** operation and at the same time keep the solution as simple as possible.

Our programme consists of 3 classes – **ArrayObject**, **MergeSort** and **FifthAttemptAtConcurrency** (programme structure in Figure 2).

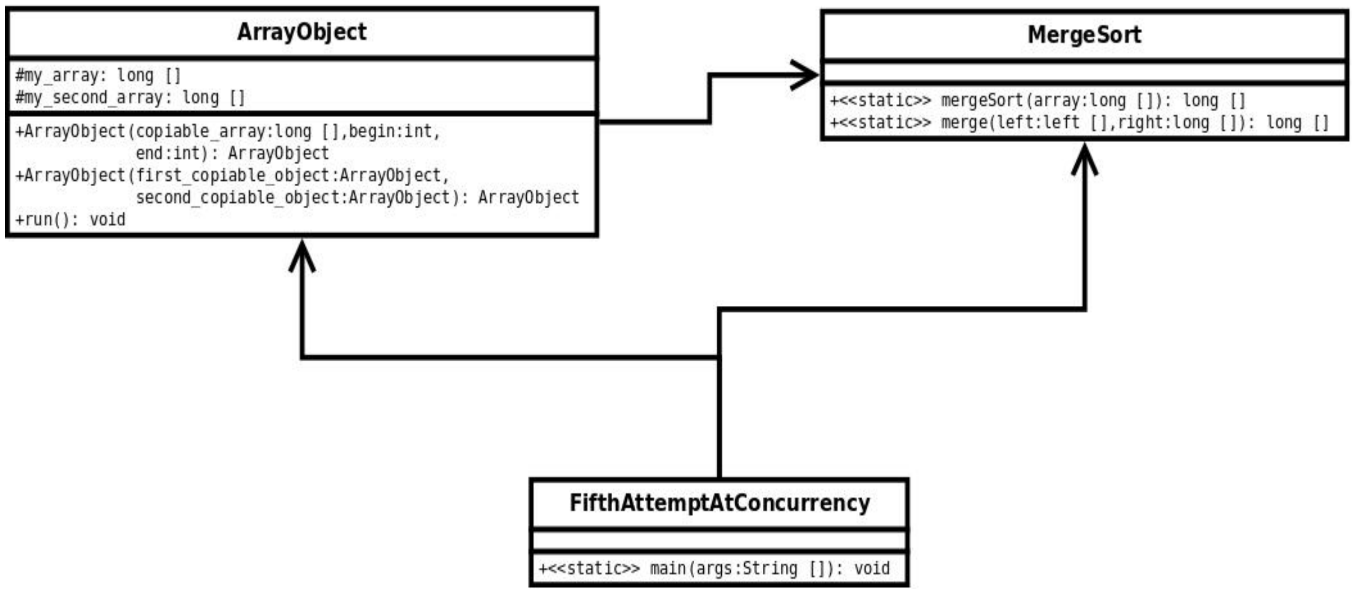

ArrayObject
#my_array: long []
#my_second_array: long []
+ArrayObject(copiable_array:long [],begin:int, end:int): ArrayObject
+ArrayObject(first_copiable_object:ArrayObject, second_copiable_object:ArrayObject): ArrayObject
+run(): void
MergeSort
+<<static>> mergeSort(array:long []): long []
+<<static>> merge(left:left [],right:long []): long []
FifthAttemptAtConcurrency
+<<static>> main(args:String []): void

FIGURE 2 PROGRAMME STRUCTURE

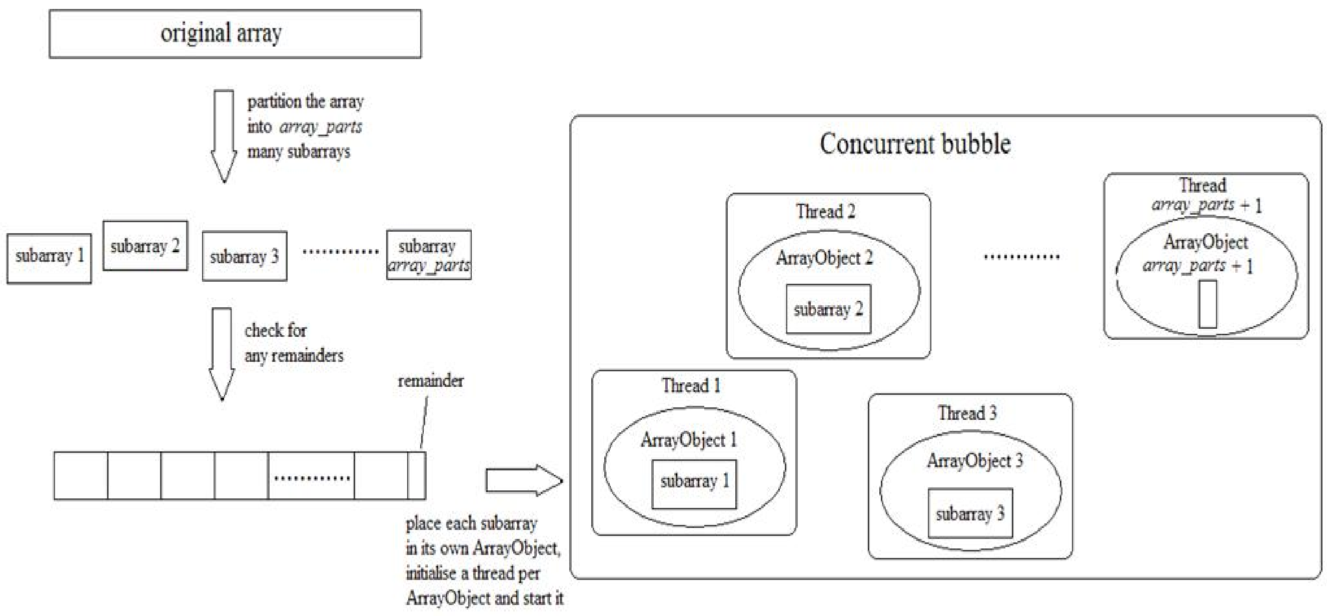
The class **ArrayObject** defines a simple class that has only two private instance variables, both of which are arrays of the type **long** – *my_array* and *my_second_array*. In a way this class resembles Java's wrapper classes, such as **Integer**, **Boolean**, etc. The **ArrayObject** class provides two constructors depending on whether the user wants both arrays instantiated or just the first one. The former constructor is similar to the *copy constructors* known in C++, as its parameters are also **Ar-**

rayObjects. The class also implements the Java interface *Runnable*, which will allow for instantiating threads with **ArrayObject**s. The method *run* checks if both arrays have been instantiated or just the first one. In the former case it assumes that both arrays are in order, so it just merges them, preserving the order, and sets the first array (variable *my_array*) to point to the newly merged array and the second one is set to **null**. Otherwise it just sorts *my_array* sequentially.

MergeSort contains two class methods *mergeSort* and *merge*. As their names suggest, these are the methods needed to implement the usual sequential Merge Sort.

The class **FifthAttemptAtConcurrency** is where the real job is done. One of its global variables is *array_parts*, which determines in how many subarrays the original array should be split. The value of *array_parts* is given by the user. First, the programme reads the input file and saves its contents into the array *array_of_integers* (the name is a bit misleading since it is actually an array of type **long**). To minimise reading time we used the Java classes **BufferedInputStream**, **DataInputStream** and **FileInputStream.** Next the programme divides the *array_of_integers* into *array_parts* many subarrays and apportions each subarray to an **ArrayObject**. Note that here by " dividing " and " apportioning " we mean actually copying the relevant parts of the *array_of_integers* into a new array. All **ArrayObjects** are then saved in an array *array_of_objects* of the type **ArrayList<ArrayObject>**. Next, the programme creates a thread per each **ArrayObject** and, in turn, saves all threads in an array *array_of_threads* of the type **ArrayList<Thread>**. In case the size of the *array_of_integers* is not divisible by *array_parts*, the programme instantiates another **ArrayObject** to hold the remainder, a thread with it, and adds each to the respective array. The programme proceeds to sort the original array sequentially, measuring the time sequential sorting lasted, and then ensures the array is in order.

FIGURE 3: PARTITIONING THE ARRAY AND STARTING EACH THREAD



Having done that, the programme traverses the *array_of_threads*, starting each thread and then enters a busy wait loop (see Figure 3 above). The *Concurrent bubble* in the picture denotes the pool of threads executing concurrently.

In the loop the main thread waits for two threads to die, removes them and their **ArrayObjects** from the arrays *array_of_threads* and *array_of_objects* respectively and then instantiates one **ArrayObject** to hold both old objects' first arrays (recall that upon completion of an **ArrayObject**'s *run* method, only *my_array* contains something and *my_second_array* is **null**). The new **ArrayObject** is used to create a new thread, which is then started. When started, the new thread calls the *run* method of its **ArrayObject**. The *run* method sees that both arrays have been instantiated, so it merges them. Both the new thread and its **ArrayObject** are saved in the arrays *array_of_threads* and *array_of_objects* respectively. The loop exits when the number of threads reaches one (see Figure 4 below), whereupon the main thread waits for the last thread in *array_of_threads* to die and then compares the values of the first array of its **ArrayObject** with the values of the original sorted array. It is easy to observe that the number of threads and **ArrayObjects** constantly decreases (for two dead threads only one is instantiated), so we can be sure that the loop eventually exits. Finally, the programme prints the time elapsed from starting all the threads to the death of the last thread and exits, adding the time overhead incurred copying the values of the array into the **ArrayObjects** at the beginning.

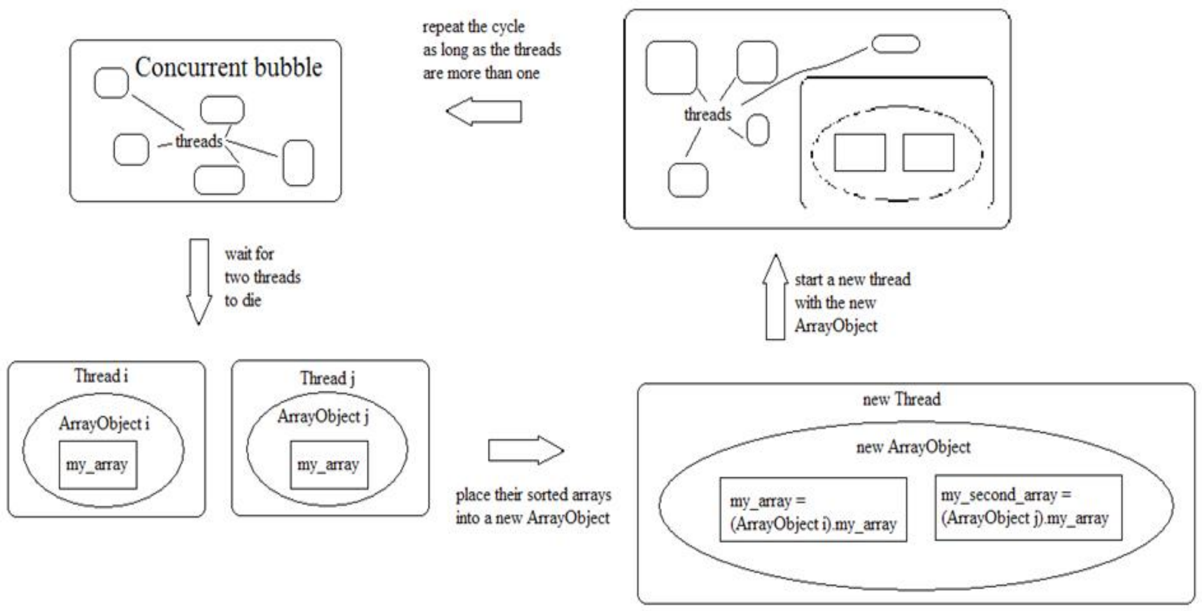


FIGURE 4: THE BUSY-WAIT LOOP

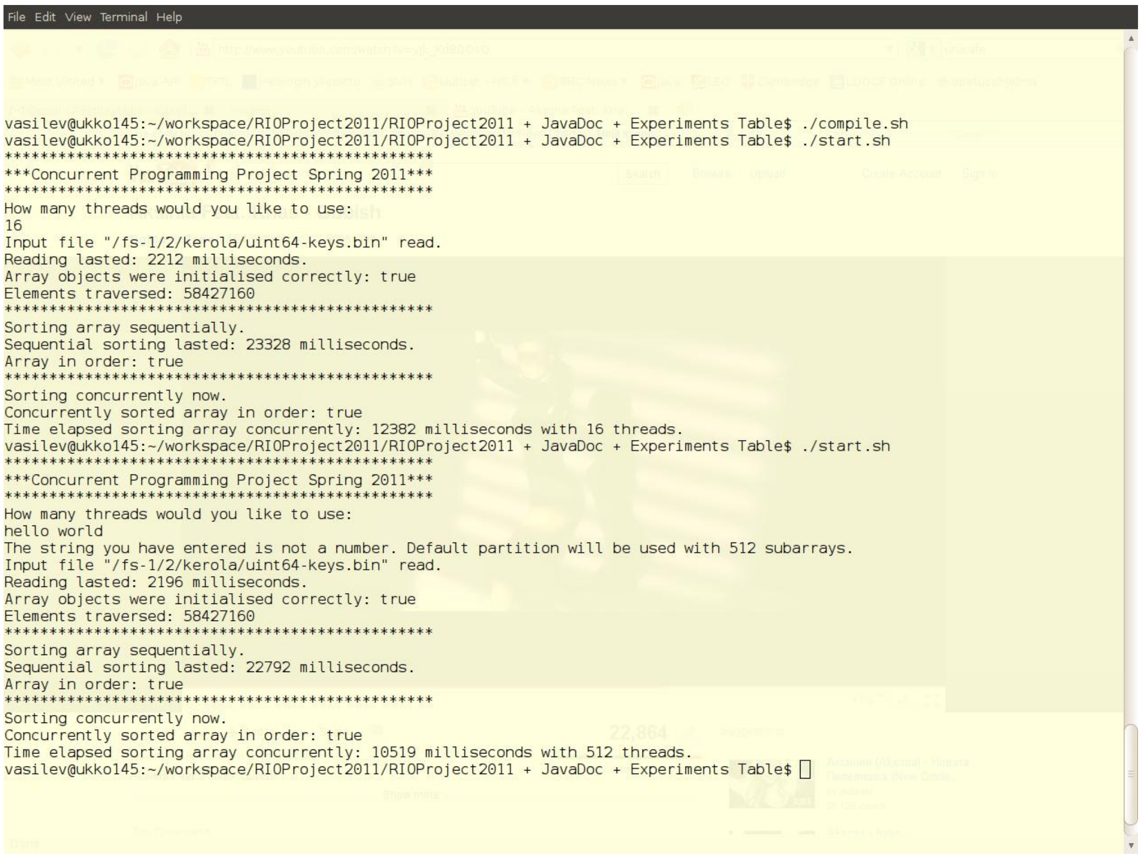
In our solution the usual problems associated with critical sections, namely *mutual exclusion*, *deadlock* and *starvation* were not cause for great concern, as there was little to no contention for shared resources. Mutual exclusion was indeed vital lest a thread started merging two arrays that had not yet been sorted/merged by two others. The method *isAlive()* of each thread precluded the possibility of anything like this happening. While *deadlock* is in fact impossible, since each thread operates on different arrays, a question arises whether *starvation* is. Again, threads do not share any common resources, and a thread is instantiated to merge two sorted arrays only when it is needed, i.e. after the arrays have been sorted, so *starvation* is indeed impossible.

4 User Instructions

The programme is supposed to be run on the university Ukko server. To that end the zipped package is saved in the directory of choice on one of the Department of Computer Science machines and its contents extracted there (right-click on package and select "Extract Here" from the drop-down menu). A new folder, named **RIOProject2011**, will appear on the screen. Next establish an **ssh** connexion with the desired cluster of the Ukko server and access the directory where you unzipped the programme package. Open a Linux Terminal and type **ssh ukko*xxx*.hpc.cs.helsinki.fi**. Replace *xxx* with a number between 002 and 240. A full list of Ukko's nodes and current load can be found here. The server will ask for a user name and password. Enter your Department of Computer Science user name and password. Next, open the directory where you unzipped the package using the Unix command **cd**. The path of the directory is the same as the Department's **melkki** server, i.e. it should look something like ***user***@ukko*xxx*:~/directory/RIOProject2011$. The scripts required on the course page are written so that the whole programme with all its pertaining classes should be compiled with the command **./compile.sh** and run with the command **./start.sh**.

The user need nod provide the name/address of the data set to the programme, as it automatically opens the input file designated on the course page and prints on the command line. However, the programme does ask for the number of subarrays it should partition the original array into. In the event of an illegitimate input on the part of the user (e.g. a string "hello world" instead of a number), the programme uses the default setting of 512 threads.

The number of threads constitutes the only difference between different test cases, since the array to be sorted, i.e. the so called data set, has been predetermined on the project page. Figure 5 shows two sample runs of the programme with the script **start.sh**, where the user enters a correct number in the first run and the string " hello world " in the second when asked about the number of threads the programme should use.



```
vasilev@ukko145:~/workspace/RIOProject2011/RIOProject2011 + JavaDoc + Experiments Table$ ./compile.sh
vasilev@ukko145:~/workspace/RIOProject2011/RIOProject2011 + JavaDoc + Experiments Table$ ./start.sh
************************************************
***Concurrent Programming Project Spring 2011***
************************************************
How many threads would you like to use:
16
Input file "/fs-1/2/kerola/uint64-keys.bin" read.
Reading lasted: 2212 milliseconds.
Array objects were initialised correctly: true
Elements traversed: 58427160
************************************************
Sorting array sequentially.
Sequential sorting lasted: 23328 milliseconds.
Array in order: true
************************************************
Sorting concurrently now.
Concurrently sorted array in order: true
Time elapsed sorting array concurrently: 12382 milliseconds with 16 threads.
vasilev@ukko145:~/workspace/RIOProject2011/RIOProject2011 + JavaDoc + Experiments Table$ ./start.sh
************************************************
***Concurrent Programming Project Spring 2011***
************************************************
How many threads would you like to use:
hello world
The string you have entered is not a number. Default partition will be used with 512 subarrays.
Input file "/fs-1/2/kerola/uint64-keys.bin" read.
Reading lasted: 2196 milliseconds.
Array objects were initialised correctly: true
Elements traversed: 58427160
************************************************
Sorting array sequentially.
Sequential sorting lasted: 22792 milliseconds.
Array in order: true
************************************************
Sorting concurrently now.
Concurrently sorted array in order: true
Time elapsed sorting array concurrently: 10519 milliseconds with 512 threads.
vasilev@ukko145:~/workspace/RIOProject2011/RIOProject2011 + JavaDoc + Experiments Table$ 
```

FIGURE 5: TWO SAMPLE RUNS OF THE PROGRAMME

5 Evaluation of Solution (correctness, performance, scalability)

5.1 Correctness

After sorting the array, first sequentially and then concurrently, the programme performs an inspection of the computed results. It ensures that both arrays are in ascending order and that their corresponding elements coincide. For proof of the correctness of MergeSort itself the reader is referred to Thomas Cormen's textbook [CLR02]. For our purposes it suffices to show that the following statement is an invariant of our concurrent algorithm: *”After a thread dies, its **ArrayObject** holds a sorted array in its variable my_array and my_second_array is **null**”*.

Proof. Since the thread merely executes the **ArrayObject**'s *run* method, there are two possible scenarios:

a) the **ArrayObject** had only one array before the thread started executing (i.e. *my_array* and *my_second_array* was set to **null**), in which case its *run* method used the methods in the class **MergeSort** to sort it sequentially and then set its *my_array* variable to point to the sorted array. The variable *my_second_array* remained intact.

b) the **ArrayObject** had both of its array variables pointing to some arrays before the start of the thread. Observe that if this is the case, it means both arrays are already in order because only when a thread has sorted an array and subsequently died, is a new **ArrayObject** instantiated with two arrays, i.e never at the beginning, when the original array is being partitioned. In this case the thread only merges the two sorted arrays, so the result is also a sorted array, to which variable *my_array* points, and *my_second_array* is set to **null**.

We did not use any of the concurrent programming constructs we have acquainted ourselves with during the course, such as *semaphores*, *monitors* etc. Their usage was deemed unnecessary for our purposes, since the primitive busy-wait loop provided the functionality we needed. Note that in our solution we do not have a *critical section* in the formal sense, because living threads are not supposed to share resources. The three key problems associated with concurrent programming are tackled as follows:

5.1.1 Mutex

Mutual exclusion follows from the simple fact that different threads operate on different arrays. Merging two sorted arrays starts only after their threads have died.

5.1.2 Deadlock

Rather loosely interpreted, in this context, deadlock would be possible only if a thread enters an infinite loop, thereby making it impossible to merge the whole sorted array. We must recall here that each thread sorts its own portion of the original array *sequentially* and any array can sooner or later be sorted. The logic of Merge Sort precludes any infinite loops and in our implementation we strove to put this into effect. And even if infinite loops were in fact possible, an infinite loop in one thread cannot prevent another from doing its job. An infinite loop will only result in the main thread waiting indefinitely, i.e. an infinite loop in the main thread in its own right, which does not meet the formal definition of a *deadlock* or *starvation* of the main thread.

5.1.3 Starvation

Starvation is irrelevant in this context, since threads do not share common resources they might compete to get hold of.

5.2 Performance

Improving performance was the ultimate goal of our project. The results we achieved do not share perhaps the glamour of competing teams. However, we did manage to bring down the algorithm's time complexity by a constant factor of 2. The experiments we conducted showed an average of 25 sec. for sorting the array sequentially and an average of 12 sec. for our parallel implementation (more on the experiments we conducted in the next section). The elegance of our solution lies in the virtually unlimited number of threads our programme can utilise (variable *array_parts* in the code). There are no restrictions on the number of threads (provided they are fewer than the length of the array, of course), which, in turn, allows for better scalability. The programme can be set to use e.g. 2,4 5, 16, 31, 128 etc. threads, i.e. The number of threads does not have to be a power of 2 or a divisor of the size of the original array.

6. Performance Tests

We ran our concurrent implementation of Merge Sort using the data set provided on the course page as our input. We managed to minimise reading time (although this is actually extraneous to Merge Sort itself) by dint of the Java classes **BufferedInputStream**, **DataInputStream** and **FileInput-Stream**. The performance tests we conducted consisted in running our Merge Sort algorithms on the same input (i.e. the data set), varying the number of threads used. We ran tests with 2, 4, 8, 16, 128, 256, 512, 1024, and 2048 threads, and repeated the test 10 times per thread variation. We documented the time to sort the array both sequentially and concurrently with every test run, so as to prevent any momentary fluctuations in the load of the cluster from corrupting the results. Afterwards we calculated the arithmetic mean speed-up factor that concurrency had contributed. By a speed-up factor here we mean the quotient of the time it took to sort the array sequentially and the duration of the concurrent sort. For example, if sorting the array sequentially lasted 20 sec. and concurrent sorting took 10 sec., then this would constitute a speed-up factor of 2. The logs we kept are enclosed with this report as attachment 1 and here we present a short summary of the different speed-up fac-

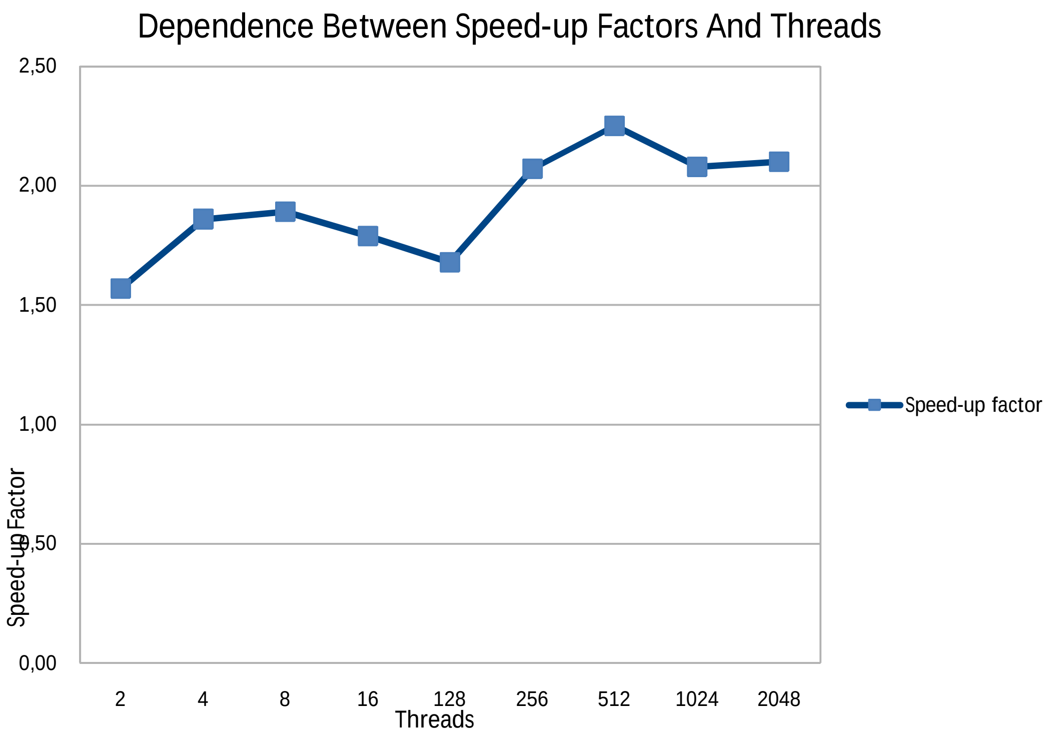


DIAGRAM 1: PERFORMANCE WITH DIFFERENT NUMBERS OF THREADS

tors we achieved and the extent of concurrency that generated them.

As the above line chart shows (a little surprisingly), best performance is achieved with 512 threads. It is this arrangement that broke the 9 seconds barrier and set our all-time record at 8,9 seconds. One should take into account, however, that the current load of the cluster is essential to the actual performance of the algorithm. The chart also suggests that scalability actually fails where it should reach its peak, namely with 16 threads. Performance does improve, however, as the number of threads increases to 512 and then slowly abates again. We are at a loss to explain the reasons behind this phenomenon. Our conjecture is that synchronising the threads outweighs the time gain achieved by concurrency in certain cases.

7. Summary

Our project aimed at utilising concurrency in the implementation of Merge Sort – a classic sorting algorithm. Although inherent to Merge Sort, parallelism did not contribute any asymptotic gain to the overall performance of the algorithm. However, our solution managed to bring down the time needed to sort the same array by a constant factor of at least 2. One must observe that this is by far not the best parallelisation of Merge Sort possible. Scientific literature abounds in concurrent sorting algorithms. Yet, we decided to use our own ideas and not to refer to external resources.

As attachments we have enclosed the experiment report, and the programme code. The programme input would be impracticable to print.

8. References

CLR02 Cormen, T., H., Leiserson, C. E. Rivest, R., L., Stein, C., *Introduction to algorithms (Second Edition)*. The MIT Press, 2002.

Kru56 Kruskal, J., B., Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7,1 (1956), pages 48-50.

Oul11 Lecture slides for the Design of algorithms course. University of Oulu [Also: https://www.raippa.fi/Ohjelmointi/2.%20Luento/140%20Lomituslajittelun%20toiminta periaate 9.4.2011].

SHG09 Satish, N., Harris, M., Garland, M., Designing efficient sorting algorithms for manycore GPUs. *IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy, 2009, pages 1-10.