Central Washington University

Advanced Algorithms

Winter 2019

Project 2 Report

Author:
Hermann Yepdjio

Professor:
Dr. Razvan Andonie

February 8, 2019

Contents

1 Introduction **2**

2 experimentation process **2**

3 Results **2**

4 Conclusion **3**

1 Introduction

The purpose of this project was to implement and compare different implementations of the merge sort algorithm from the book.

2 experimentation process

- we implemented 4 different versions of the merge-sort algorithm

  -the multi-threaded MERGE_SORT from the textbook

  -the multi-threaded P_MERGE_SORT from the textbook

  -the sequential versions of both algorithms above

- C is the programming language we used along with the POSIX threads
- we generated 4 arrays of sizes 100, 1000, 10000 and 100000 and filled them up with random numbers, to test the implementations and we recorded the processing times for each implementation over each array.

3 Results

Number of Entries	Sequential implementation time taken (in ms)	Multi-threaded Implementation time taken (in ms)
100	8	7058
1000	77	84889
10000	1131	660888
100000	11189	994744

Figure 1: MERGE_SORT(sequential(left) and multithreaded(right)

Number of Entries	Sequential implementation time taken (in ms)	Multi-threaded Implementation time taken (in ms)
100	46	50
1000	379	554
10000	5162	6086
100000	60234	2203846

Figure 2: P_MERGE_SORT(sequential(left) and multithreaded(right)

4 Conclusion

From the tables above, we can see that

- the sequential version of MERGE_SORT performs better than the sequential version of P_MERGE_SORT. This can be explained by the fact that the latter performs more computations than the first.
- the sequential version of MERGE_SORT performs better than its multi-threaded version. This can be explained by the fact that it takes times to create the threads and also many of the threads will be created only to perform really simple operations such as sorting 1 or 2 numbers.
- the sequential version of P_MERGE_SORT performs better probably because of the same reasons described above.
- the multi-threaded version of P_MERGE_SORT is more efficient than the multi-threaded version of MERGE_SORT probably because the merge operation is parallelized in the first. However, we also see that when the array gets bigger(array size = 100000), the multi-threaded MERGE_SORT becomes faster. This is probably due to the fact that the number of threads created increases faster in the P_MERGE_SORT than in the MERGE_SORT.

So we can conclude that making a program multi-threaded does not necessary mean that the program will run faster especially if the threads will have to perform on a shared resource. In this case, a mechanism such as the use of mutex is needed to avoid the problem of race condition. Threads therefore have to wait for the resource to become available before they can do their job, which is time consuming and is not observed in the sequential implementation.