Computational Geometry (2025)
Project One: Parallel Merge Sort

Peyman Afshani

September 10, 2025

1 Introduction

In this project, we will practice a bit with the idea of coding parallel algorithms. You can use any programming language you like (although python might not give you a good enough performance to get good quality tests).

After implementing the algorithms and making sure that they run correctly, you must evaluate their performance on some "test cases". You should try to follow sound algorithm engineering principles in doing so (have a plan, design your experiments with respect to your plans, and hypothesis, evaluate the results and potentially go back to repeat the process). For this project, and to reduce your workload, you can run all your experiments on the same machine. Recall that the goal is to learn more about the algorithms rather than finding a very optimized algorithm.

Important. For simplicity, we will assume that we are dealing with distinct elements. All the algorithms in this project can be made to work without this assumption but this assumption will make some of the description as well as implementations easier.

1.1 Part 1: Basic Parallel Merge Sort

Task 1 (Basic Merge Sort): Implement parallel merge sort with a *sequential* merge step. The pseudocode for this merge could be something like this. This implementation only uses a one-time allocated scratch space, instead of allocating scratch space at every recursion level.

Algorithm 1 A basic merge sort. It takes as input a pair, (I, S), of arrays of size n each, where I is the input array and S is the scratch space reserved to contain the output. It returns the pair (I, S) where S contains the correctly output sorted array.

procedure BasicMergeSort(I,S) ▷ I: Input array of n values. S: output.
  Split I into two equal parts I_ℓ and I_r of size $\frac{n}{2}$
  Split S into two equal parts S_ℓ and S_r of size $\frac{n}{2}$
  for each I_ℓ and I_r **in parallel do**
    $(I_\ell, S_\ell) \leftarrow$ BasicMergeSort(I_ℓ, S_ℓ) ▷ The returned S_ℓ is the sorted input, I_ℓ the scratch space
    $(I_r, S_r) \leftarrow$ BasicMergeSort(I_r, S_r) ▷ The returned S_r is the sorted input, I_r the scratch space
  Sequentially merge S_ℓ and S_r into I ▷ Merge into I
  return (S, I) ▷ I now contains the output

A Remark. There is one detail that is ignored in the above pseudocode so you might not be able to just copy/paste it in your favorite programming language. The issue is that different sub-problems might reach

their base case at different levels and also the role of scratch space and input array is swapped after each recursion step.

1.1.1 The Report and the Submission.

In your report, first, briefly explain this algorithm. Next, after implementing it, try to run some tests using a different number of threads. Come up with some hypotheses for the behavior of the merge sort in practice. Mention the theoretical runtime of the algorithm in the PRAM model and explain if the behavior is expected or not. **You must submit your code together with the PDF report!**

It is very important that you think about the possible results that you can get. Your report should also include a brief description of your hardware, e.g., the number of cores. Based on this, explain what should be your hypothesis when it comes to the running time of the algorithm and then explain whether the results that you see in your experiments match up with your expectations.

You don't have to implement the above algorithm but constantly allocating memory can cause the test a bit unreliable, specially in Java.

1.2 Part 2: Parallel Merge

In this part, you will implement a fully parallel merging algorithm. The details of the algorithm will be given below; however, there will be some parts that you will need to figure out on your own. The algorithm has one main component: a selection algorithm that is outlined below.

1.2.1 Selection

The input to our selection problem is the following: two lists sorted in increasing order, A and B, and a value k. The goal is to find two values a and b that are defined as follows. Let C be the result of merging A and B, increasingly. Recall that we have assumed that C contains distinct elements. Consider the set of k smallest values of C and denote it with C_k; the index a is defined as the number elements of A contained in C_k (i.e., $a = |C_k \cap A|$) and b is defined similarly (i.e., $b = |C_k \cap B|$).

Task 2 (Solving selection.): Given A, B, the value of k, and an index i, show that in $O(1)$ time (using a single processor) we can check whether $a < i$, $a = i$ or $a > i$ (without having to compute C). Based on this, show that selection can be solved in $O(\log n)$ time using a single processor.

1.2.2 Sub-problems

Now consider the problem of merging two sorted lists of A and B into another list C. Let n be the total input size, i.e., $n = |A| + |B| = |C|$. Assume that we have P processors. Define $P - 1$ indices, $k_1, \cdots, k_{P-1}$ where $k_i = \frac{in}{P}$. Define the values a_i and b_i to be the values obtained by solving the selection problem with the value k_i. Define $a_0 = b_0 = k_0 = 0$ and $a_P = b_P = k_P = n$.

In the parallel version of the merging problem, the i-th CPU will have to merge the values in A between indices a_{i-1} and a_i with the values in B between indices b_{i-1} and b_i and place them in the array C between indices k_{i-1} and k_i.

Task 3 (Fully Parallel Merging.): Implement the fully parallel merging algorithm as above and also analyze it (assuming that we have P processors).

Task 4 (Experiments.): Run some experiments and try to understand the behavior of this algorithm in practice.

Your report should also discuss how you have generated the two input arrays. Note that the input generation in this case is a bit more involved. First, the two input arrays must be sorted so generating random numbers do not work. Second, there are many different ways two arrays could potentially overlap.

If you generate two random arrays A and B and then sort them, then the arrays will likely overlay somewhat evenly, where as in general many other distributions are possible and for example it is possible that all the values of the first array are smaller than the smallest element of the second array. This is one example of a problem where input generation is not straightforward and needs some care. Discuss how you generated inputs for this algorithm.

Remarks. There are probably very many possible strategies to generate inputs for this. You don't have to go overboard with input generation and you don't have to spend a lot of time on it.

1.3 Part 3: Fully Parallel Merge Sort

In this part, plug in your parallel merging step instead of the sequential merging that you did initially. As before, run some tests to on the resulting algorithm, inline with your hypotheses.

Remarks. If you want more algorithms to throw in for comparison, you can also use Java's own parallel sorting, in `java.util.Arrays` via `parallelSort` methods. Refer to the API for more information.

A General Remarks

You are not actually forced to measure the runtime and you can come up with your own experiments and tests, according to your hypotheses. Recall that you can also use software counters if you think they can be useful.

This is not necessary but it is possible to use software counters to simulate running the algorithms in a machine with very many processors. One way to do it is to use a limited variant of the *fork-join* model. The basic setup is as follows: your program creates p child threads (for some parameter p) that work on independent parts of the input (i.e., they do not write to same memory location or they do not read a memory location written by another thread). Each thread keeps track of the number of comparisons and data movement operations it does via software counters. Then, your main thread joins with the p child threads and retrieves the counters done by each thread. The parent thread then keeps the maximum of the counters as its depth (plus any additional data movements or comparisons that it does). This process can repeat for as many times as possible. Technically, it is possible to create n threads in this way, although realistically, most operating systems have a limit on how many threads a process can spawn and it is often much smaller than n for any reasonable size input. But nonetheless, it is possible to simulate having hundreds of cores in your system via such software counters.

Your report should contain some plots that depict the running times or whatever else you decide to measure. Make sure these plots are of good quality and that you have avoided the common mistakes that people make while plotting. Make good choices when it comes to whether an axis must be in logarithmic scale or not. Also, note that the input size ideally should grow exponentially.

Finally, pay attention to the details and the system you are using for testing. Mention your operating system, your set up and your hardware that you use for testing. Make sure to take necessary steps to increase the reliability of your experiments. For example, if you are using a laptop for testing, make sure that it is always plugged in as sometimes power management policies can throttle mobile CPUs down and add more noise to the experiments. Another common issue is the concept of hyperthreading, where one physical core can appear as two logical cores and this is used in some modern hardware. The problem is that the two cores do not have the performance of two physical cores. You can often disable hyperthreading in the setup, if your machine has it.

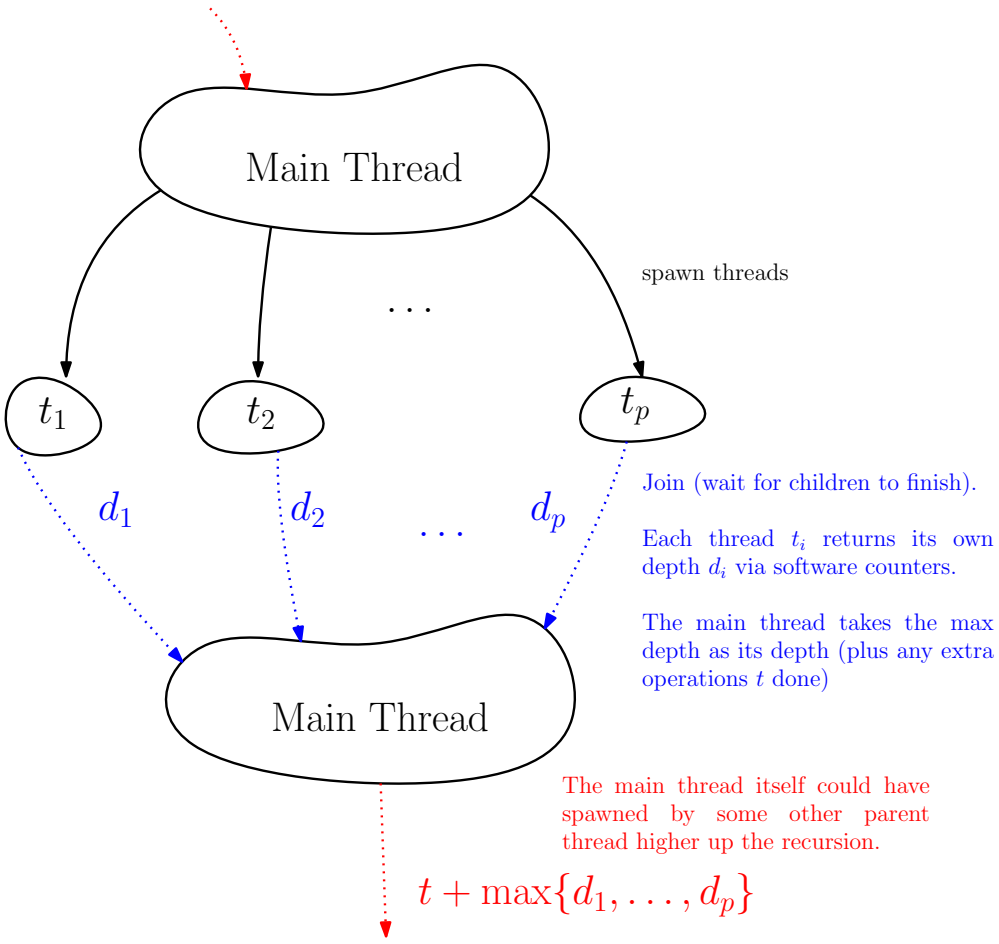

Figure 1: An illustration of how to use software counters to simulate having more cores.