Computational Geometry (2023)
Project One: Parallel Merge Sort

Peyman Afshani

September 6, 2023

1 Introduction

In this project, we will practice a bit with the idea of coding parallel algorithms. I recommend using Java but if you want, you can use other programming languages (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.

1.1 Part 1: Basic Parallel 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 $I_\ell$ and $I_r$ of size $\frac{n}{2}$
    Split $S$ into two equal parts $S_\ell$ and $S_r$ of size $\frac{n}{2}$
    for each $I_\ell$ and $I_r$ in parallel do
        $(I_\ell, S_\ell) \leftarrow$ BasicMergeSort($I_\ell, S_\ell$)   ▷ The returned $S_\ell$ is the sorted input, $I_\ell$ 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 $S_\ell$ and $S_r$ into $I$                        ▷ Merge into $I$
    return $(S, I)$                                            ▷ $I$ now contains the output
```

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.

1.2 Part 2: Parallel Merge

In this part, implement the parallel merging step of two **sorted** arrays, based on the explanations given in the class. As before run some tests on the performance of the merging step and then compare it to the sequential way of merging two arrays. Explain if the algorithm behaves as expected or if there are any unusual behavior that you see 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.

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.

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.