Algorithm Engineering
WS 24/25

Patrick Koston
Simon Schwitz

Algorithm Engineering Report 02

Abstract

Abstract is very abstract

1 Introduction

2 Build and Run the project

In order to build the project you must meet all requirements from the `README.md` file in the project root directory. The build script will check whether `gcc` and `g++` commands are present on your system. Note: it will not check the their version so make sure the required minimum version is installed!

In case the build fails, we now also include pre-built binaries in the `bin/` directory.

You can also check the state whether Linux build success or not using the GitlabCI Pipeline. This pipeline uses the latest `gcc` version which as of writing is `14.2.0`. You can verify this on Dockerhub or when looking into the pipeline result.

2.1 On Failed Build

The script then throws an error that the CMake build did fail. Usually it then still continues with executing the experiments using the pre-build binaries. In case this should not work, please modify the experiment script using the instructions below:
On Windows, please remove lines 8 and 9 from the corresponding PowerShell script.
On Linux, please remove lines 9 and 10 from the corresponding Bash script.

2.2 Run Experiments

Working Directory must be the corresponding exercise directory!
On Windows, execute the `experiment_0{1,2}.ps1` scripts in the exercise directory to run the corresponding experiment.
On Linux, execute the `experiment_0{1,2}.sh` scripts in the exercise directory to run the corresponding experiment.

2.3 Source Code / Project Structure

The source to run the experiments can be found in the `src/` directory of the current exercise directory. Reusable code however is not included in the exercises source directory and can be found in the `lib/` directory instead. To keep the project clean, we reserve to export reusable code to separate libraries which are then linked in the header of the exercise source files. Please see the README.md file for information about which shared libraries exist and what they contain.

3 Preliminaries

4 Algorithm & Implementation

4.1 Implementation

4.1.1 Parllell Merge Sort

The code can be found in `lib/Sort/mergeSort.cpp` (see included headers).

Given a variable T indicating the number of threads to use, the algorithm first checks if the available input data can be split into T buckets of at

least 2 elements. If this condition is not met, a classic merge sort will be performed on the data set instead. In the next step, the input data is partitioned into T buckets of size $input.size()/T$. These buckets are then individually sorted using classic merge sort. We then apply a similar bottom-up approach to the external memory merge sort and create multiple tasks that each wait for two tasks lower in the tree structure to finish sorting and then performs the merging step on that level. During the bottom-up merging, we have $T/2$ tasks at most thus we never exceed the thread limit. In the event of T being

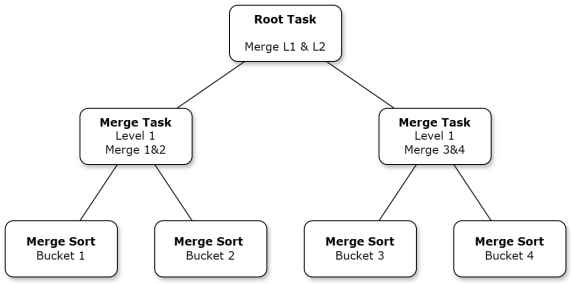

Figure 1: Parallel merge sort using an even number of $T = 4$ buckets

odd, the corresponding merging tasks will wait for a single previous task and then merely pass on the result as there is no second block to merge with. Modern computers usually have an even number of cores/threads so we can argue that this case does not occur under normal circumstances.

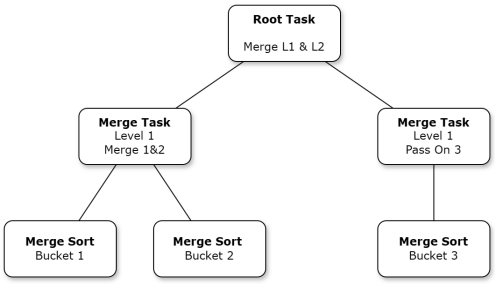

Figure 2: Parallel merge sort using an odd number of $T = 3$ buckets

4.2 Parallel Quick Sort

The code can be found in `lib/Sort/quickSort.cpp` (see included headers).

Given a variable T indicating the number of threads to use, the algorithm first checks if the available input data can be split into T buckets of at least 2 elements. Additionally, a check is performed if the input size is less than a predefined constant of 1000. In this case, classic merge sort will be performed for performance reasons.
If this is not the case we select the number in the middle (rounded down) of the input data as the pivot element. In the next step, the input data is partitioned into T buckets of size $input.size()/T$. Each bucket then sorts the data into a left and right half. The left half contains only elements smaller than our pivot element and the right side only elements greater than our pivot element. To later split the array into a "smaller than" and "larger than" half, we count the number of elements on each side. If the array contains the pivot itself, the pivot is placed in between both halves and is not counted. Once each task has finished splitting, the data is written back into the original array. When writing back, we first move all halves smaller than the pivot in the array, then the pivot itself, and then lastly all halves greater than the pivot. We also have to keep track of the size of the combined "smaller than" and "greater than" half. In each subsequent step, we then apply the same algorithm to both, the smaller and the greater half. We do this until we reach the predefined constant threshold point for input (bucket) size. From this point on further branching is way too expensive. To reduce runtime costs, the predefined constant should have an even higher value (at least 10000).

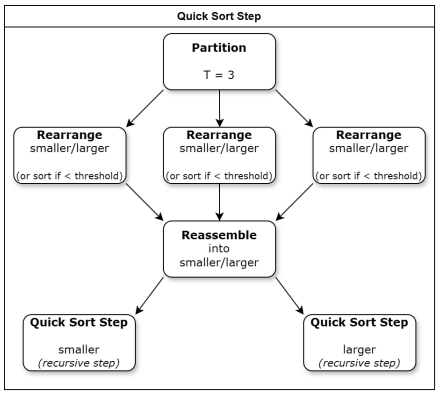

Figure 3: Parallel quick sort using $T = 3$ buckets

5 Experimental Evaluation

5.1 Data and Hardware

5.2 Results

6 Discussion and Conclusion