TDDD56

Lab 2

Parallel Sorting

Sajad Khosravi

November 2025

Introduction

Sorting is one of the most important routines, used in many programs and run frequently on a computer to perform everyday tasks. Because of the very intensive use of sorting, its performance has a great chance to influence the performance of other programs using it and the overall performance of a complete system. Consequently it is a good idea to take profit of multicore computation capabilities to accelerate sorting operations through parallel sorting algorithms.

Compared to many other calculations, sorting requires much fewer calculations, almost comparisons only, but a lot of memory operations to swap values. Because of this property, one can expect sorting algorithms and their parallel equivalent to be very sensitive to data locality issues. In this lab, we investigate the parallelization of the *Merge Sort* algorithm.

Merge Sort

Merge sort is a well-known sequential algorithm that follows the divide-and-conquer strategy. It repeatedly splits an unsorted sequence into two parts, sorts each one, and then merges the sorted parts into a single sorted sequence. The base case occurs when the sequence has only one element, which is already sorted. In the merging step, two sorted sequences of length k are combined to form one sorted sequence of length $2k$.

```
1 void mergeSort(int arr[], int left, int right) {
2     if (left < right) {
3         int mid = (left + right) / 2;
4         mergeSort(arr, left, mid);
5         mergeSort(arr, mid + 1, right);
6         merge(arr, left, mid, right);
7     }
8 }
```

Listing 1: Sequential Merge Sort

TASK 1: Simple parallel merge sort

In this task, a sequential implementation of merge sort has been provided (Listing 1). Your goal is to parallelize this implementation using POSIX Threads (Pthreads) in order to leverage multi-core CPU architectures[1]. The objective is to explore how the divide-and-conquer nature of merge sort can be exploited for parallel execution, and to evaluate how effectively the workload can be distributed among multiple threads. After implementing the parallel version, you will measure and compare its performance against the sequential baseline under different experimental conditions.

Questions:

- How did you distribute the tasks among the available threads?
- Are all tasks working in parallel? Are all threads actively processing data simultaneously, or do some spend time waiting?
- What is the performance bottleneck of your parallel algorithm?
- Do you need any synchronization between these threads? When? How are you handling that?
- How much speedup did you get by making the code parallel? Measure execution time for both sequential and parallel versions across different configurations: Number of threads = {1, 2, 4, 8}, Number of elements = {1000000, 10000000, 100000000}.
- At what recursion depth did you stop creating new threads? How did you decide that threshold? Explain the trade-off between parallel overhead and computational gain.

TASK 2: Fully parallel merge sort

In the previous task, you implemented a simple parallel merge sort algorithm where only a subset of the available threads was used to perform merge operations concurrently on different data chunks. While this approach introduced parallelism into certain stages of the sorting process, it did not fully utilize the computational resources.

In this lab, you will extend that implementation by leveraging all available threads to create a fully parallel version of merge sort. In other words, each recursive call in the merge sort algorithm—both during the divide and merge phases—should be executed in parallel whenever possible, ensuring that the workload is efficiently distributed across all threads[2]. (**Hint**: Please note that it **is not efficient** to create **many more threads than the number of physical cores**, which is 8 on the lab systems. For the implementation of this task, you may refer to the *parallel sample sort* discussed in the lecture to get an idea for how to use a limited number of threads for this purpose.)

Questions:

- Are all tasks working in parallel? Is there a risk for major idle phases of some cores?
- How many elements are processed by each thread? Analyze how the workload distribution changed in your optimized design.

[1] This is called *simple parallel mergesort* in our lecture on parallel sorting.
[2] This is called *fully parallel mergesort* in our lecture on parallel sorting.

- How much speedup did you get compared to the other two approaches? Present a detailed performance analysis showing the relative speedups of the sequential, basic parallel, and improved parallel versions.

Execution

The skeleton can be compiled in different modes you can enable or disable by passing variables when calling `make`:

MEASURE: When this mode is enabled, the program records the execution time (in seconds) before and after the computation and prints these values to the terminal after completion. This can be used by batch and plotting scripts to generate multiple measurements for analysis and plotting. You can also use these values for your own evaluation. Enable this mode by running `make MEASURE=1`. If MEASURE is not set, the program will instead execute a validator at the end to verify the correctness of the implementation.

NB_THREADS: Call `make` with NB_THREADS $= n$ where n is a non-negative integer. It instructs make to compile a multi-threaded version running n threads. If $n = 0$ then it compiles a sequential version. If $n = 1$, then it compiles a parallel version in which only one thread runs.

ALGORITHM: Call make with ALGORITHM $= a$ where $a \in \{0, 1, 2\}$. This selects and compiles one of the merge sort algorithms: sequential (0), simple parallel (1), or fully parallel (2). To execute your implementation, please use the following commands.

```
# ------------------------------
# Build the Merge Sort program
# MEASURE=1 enables performance measurement
#   without it, the program will run and validate output only
# NB_THREADS=[NUMBER] sets the number of threads
# ALGORITHM=[0|1|2] selects the algorithm variant
make MEASURE=1 NB_THREADS=<NB_THREADS> ALGORITHM=<ALGORITHM>

# Run the program
./mergesort_<NB_THREADS>_<ALGORITHM> --size <SIZE_OF_ARRAY>

# ------------------------------
# Example:
# Compile using algorithm 1 with 4 threads
# For validation:
make NB_THREADS=4 ALGORITHM=1

# For measurement
make MEASURE=1 NB_THREADS=4 ALGORITHM=1

# Execute the compiled program with an array of 1000 elements
./mergesort_4_1 --size 1000
```

Listing 2: Compile and Execute Merge Sort