Study of parallelization of sorting algorithms. Project Proposal, CSC 746, Fall 2024

Fabian F. Weiland*
SFSU

1 Problem Statement

This paper delves into the intricate problem of optimizing sorting algorithms through the exploration of parallelization techniques. The primary focus will be on investigating how the runtime of these algorithms can be significantly reduced as the level of parallelization increases, thereby enhancing overall performance.

Sorting algorithms form a cornerstone of computer science, playing a crucial role in various applications ranging from data processing to algorithm optimization. Traditional, naive approaches to sorting exhibit time complexities that can be as inefficient as $O(n^2)$ for basic algorithms like bubble sort or selection sort, while more optimized methods, such as merge sort and quicksort, achieve complexities of $O(n * logn)$. However, many non-optimized implementations rely on executing repetitive and independent operations in a serial manner, which presents a substantial opportunity for performance improvement through the application of parallelization techniques. True parallelization effectively leverages multiple computational cores, allowing for the equitable division of sorting tasks across these cores, thus improving efficiency and reducing runtime.

In this study, I will conduct a comprehensive comparison between two prominent sorting algorithms: merge sort and quicksort. If time permits, I will also extend our analysis to include other sorting algorithms. The primary focus will be on employing varying degrees of concurrency through the Open Multi-Processing API (OpenMP) and benchmarking these parallelized algorithms against their basic, non-parallel counterparts. This comparative analysis aims to highlight the advantages and potential speedups achievable through parallel processing.

2 Background and previous work

The organizational style and structure for my project, including the creation of files such as 'benchmark.cpp', were inspired by templates provided by Professor Bethel. [1] I will create the serial implementations of both merge sort and quicksort independently.

For their parallel versions, I will utilize documentation on merge sort parallelization, including sources such as [4] and [3]. Additionally, I will refer to resources like [2] to assist with the implementation of the parallel quicksort algorithm.

3 Approach

To systematically implement this analysis, I will begin by creating a `benchmark.cpp` file, which will be responsible for initializing arrays of varying problem sizes, denoted as N. Each sorting algorithm—merge sort and quicksort—will be encapsulated within its own dedicated `.cpp` file. Furthermore, the parallelized versions of these algorithms will each be implemented in two additional, separate `.cpp` files. The `benchmark.cpp` file will orchestrate the execution of these implementations by calling the appropriate functions for each specified problem size N.

To accurately measure the runtime of these algorithms, I will introduce instrumentation code that will execute before and after the function calls, ensuring that I capture the precise execution time associated with the sorting operations. The number of concurrent threads utilized during the sorting process can be specified conveniently from the command line before executing the program. This flexibility will allow me to test different concurrency levels effectively.

4 Expected Results

Each implementation of the sorting algorithm will be structured as a function that accepts the problem size N, which represents the number of elements in the input array and a pointer to the array as parameters. Within this function, the sorting process will arrange the elements in ascending order. During the execution of the sorting function, I will measure the time taken to ensure that I am capturing only the performance of the core sorting algorithm itself. Both sorting

*email:fweiland@sfsu.edu

algorithms, namely merge sort and quicksort, will be implemented in two distinct versions: a basic, serial implementation (which does not utilize OpenMP) and an OpenMP version that allows for the parallelization of the sorting process.

To control the level of concurrency, I will employ the command line argument 'export OMP_NUM_THREADS=¡concurrency_level¿', which allows us to dynamically set the number of threads used during execution. I will conduct performance tests at various concurrency levels, specifically at 4, 16, and 64 threads. The problem sizes, or the number of elements in the arrays being sorted, will vary significantly, ranging from 16, 32, 64, 128, 256 -million. For each combination of problem size N and concurrency level, I will meticulously measure the runtime and conduct comparisons across all test cases. The results will be visually represented in figures that illustrate the runtime of each algorithm, highlighting both the basic and parallelized implementations across the various problem sizes and concurrency levels.

As we anticipate, the division of the sorting task among multiple threads should yield shorter runtimes for the parallelized versions compared to their serial counterparts. Furthermore, as the level of concurrency increases, I expect to observe a corresponding decrease in runtime due to the enhanced ability to distribute workloads across more threads. Additionally, for larger problem sizes N, I should observe a more pronounced reduction in runtime, particularly when utilizing the higher concurrency levels. This is because larger datasets can be divided more efficiently among multiple threads, allowing each thread to handle a more substantial portion of the sorting task. Therefore, I hypothesize that at concurrency levels of 16 and 64, I will witness the most optimized performance, particularly for larger problem sizes such as 128 million and 256 million elements.

All experiments will be conducted on CPU nodes of the Perlmutter supercomputer, also known as NERSC-9, which is operated by the National Energy Research Scientific Computing Center (NERSC) within the U.S. Department of Energy. Each CPU node in Perlmutter features 2 AMD EPYC 7763 (Milan) CPUs, each with a base clock speed of 2.45 GHz and 256 MB of L3 Cache. Each core of the AMD EPYC 7763 (Milan) CPU has a 32KiB L1 write-back data cache and a 512KiB unifed (instruction/data) L2 cache. Additionally, each node is equipped with a total of 512 GB of DDR4 memory.

The primary metrics I will assess include the runtime of the sorting methods under various conditions. The elapsed runtime will be measured by placing instrumentation code around the main computational segments of our code. For this purpose, I will utilize the `std::chrono` library, a powerful feature of C++, which allows for precise measurement of time intervals.

References

[1] Bethel. *CSC 746 Homework Template*, 2023. Accessed: 2024-10-25.

[2] McBeukman. *Parallel Quicksort Using OpenMP*, 2023. Accessed: 2024-10-25.

[3] I. Mouth. *Parallel Bubble Sort and Merge Sort Using OpenMP*, 2023. Accessed: 2024-10-25.

[4] R. Vasudeva. *Parallel Merge Sort Algorithm*, 2023. Accessed: 2024-10-25.