Study of parallelization of sorting algorithms
Final Project, CSC 746, Fall 2024

Fabian F. Weiland*
SFSU

ABSTRACT

This paper explores the optimization of sorting algorithms through parallelization, focusing on how runtime can be reduced by increasing the level of parallelization. Specifically, it compares the performance of Mergesort and Quicksort using the OpenMP API to introduce varying levels of concurrency. The study demonstrates that both algorithms can achieve significant speedups when parallelized, with Quicksort showing better performance than Mergesort. However, the presence of sequential components in both algorithms limits the overall speedup, as higher concurrency levels yield diminishing returns. This research highlights the potential benefits and challenges of parallelizing sorting algorithms.

1 INTRODUCTION

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 $O(nlogn)$. 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. 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.

The findings show that both algorithms can be effectively parallelized, leading to significant performance improvements. Quicksort outperforms Mergesort in terms of speedup compared to its sequential version. However, the sequential components of both algorithms limit the full potential of parallelization, resulting in diminishing returns as concurrency levels increase.

2 IMPLEMENTATION

The following section describes the implementations of the Quicksort and Mergesort algorithms, both in their sequential and parallel versions. These algorithms are designed to sort an array in ascending order. To parallelize the algorithms, the task construct from OpenMP was used. OpenMP (Open Multi-Processing) is an API for parallel programming in C++, providing tools for efficient multi-threaded execution [3]. The task construct in OpenMP allows for task parallelism, where a unit of work is defined as a task that can be executed concurrently with other tasks. This construct enables asynchronous execution of tasks, with OpenMP runtime handling the scheduling and management of tasks [1]. This is particularly beneficial for recursive sorting algorithms such as Quicksort and Mergesort, where tasks can be executed independently and in parallel, improving overall performance.

Both sorting algorithms are divide-and-conquer algorithms, but they differ in their approach to partitioning and merging elements. Quicksort is implemented using a recursive strategy where an array is partitioned into two subarrays based on a pivot element, and the process is repeated on each subarray until the entire array is sorted. Although there are various strategies for selecting the pivot, for this project, the last element of the array is chosen as the pivot. This choice typically yields good average-case performance. On the other hand, Mergesort follows a recursive approach where the array is divided into two halves, each of which is sorted independently before being merged together in a sorted manner. The merging process is carefully implemented to combine the sorted subarrays while maintaining their order. The complexity and performance characteristics of these algorithms, such as time and space complexity, are taken into consideration during their implementation to ensure efficient sorting even for large input sizes.

To measure performance, I instrumented key sections of the code—call to the sorting algorithms—using C++ chrono timers to track the elapsed runtime of each algorithm [2]. For clarity and conciseness, some subsections are described in pseudocode.

2.1 Overall Code Harness

The benchmark.cpp file functions as a driver for this project. It is responsible for making smaller test runs, initializing the different problem sizes, denoted as N and run each algorithm for every problem size and the given concurrency level. The set up of the array with problem size N, is encapsulated in the setUpArray.cpp file, where the array elements are initialized with random values between 0 and 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.

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.

2.2 Sequential Quicksort algorithm

The C++ source code provided in Listing 1 shows the pseudocode for the Quicksort algorithm. The input for this algorithm are the upper bound (last index of array), the lower bound (first index of array) and a uint64_t pointer to the array, which has to get sorted. The method 'partition' selects a pivot element from the array and partitions the

*email:fweiland@sfsu.edu

other elements into two subarrays: one containing elements smaller than the pivot and the other containing elements greater than the pivot. The pivot is then placed in its correct position in the sorted array. Finally, the 'partition' method returns the new index of the pivot. This process is recursively applied to the subarrays in the recursive 'quickSort' method, until the entire array is sorted. This algorithm sorts the elements in place. No additional space is needed.

```
// up = upper bound
// lb = lower bound
// pivot = last element in the array

int partition(int64_t lb, int64_t ub, uint64_t*
    Array)
{
    int64_t pivot = Array[ub];
    int64_t i = lb - 1;

    for(int64_t j = lb; j < ub; ++j){
        if(Array[j] < pivot){
            swap arr[i+1] and arr[j]
        }
    }

    swap arr[i+1] and arr[ub] (pivot)
    return i;
}


void quickSort(int64_t lb, int64_t ub, uint64_t*
    Array)
{
    if(lb < ub){
        int64_t pivot_index = partition(lb, ub,
    Array);

        // left part of array
        quickSort(lb, pivot_index - 1, Array);

        // right part of array
        quickSort(pivot_index + 1, ub, Array);
    }
}
```

Listing 1: Source code of the sequential Quicksort algorithm

2.3 Parallelized Quicksort algorithm

The C++ source code in Listing 2 defines three methods. The 'partition' method is identical to the one used in the sequential implementation. However, the recursive 'quickSort' method includes some additional logic. A new variable, taskLimit, is introduced to set an upper bound on the size of the subarrays. When the size of a subarray falls below a certain threshold (specified by taskLimit), the sequential Quicksort algorithm is invoked to handle that smaller subarray. This mechanism helps avoid the overhead of parallelizing very small arrays, which could be inefficient due to the costs of task creation and thread management.

The main idea of parallelizing this algorithm is to have one thread run the recursive 'quickSort' method, so it can write all tasks down, but then have all available threads execute the individual tasks in parallel fashion.

In the 'quickSort' method, two tasks are created for each recursive call. This is achieved by placing the #pragma omp task shared(Array) directive above the recursive call, ensuring that the Array is shared across all tasks. Note that the partitioning step remains sequential.

The third method 'quickSort_openMP' creates the parallel region with "#pragma omp parallel". It's crucial that only one thread runs the recursive 'quickSort' method to avoid the creation of excessive tasks. This is achieved using "#pragma omp single" , which ensures that only one thread creates tasks. Finally, "#pragma omp taskwait" is used to synchronize the tasks, waiting until all threads have completed their work.

```
// up = upper bound
// lb = lower bound
// pivot = last element in the array

int partition(int64_t lb, int64_t ub, uint64_t*
    Array)
{
    // same as sequential version
}

void quickSort(int64_t lb, int64_t ub, uint64_t*
    Array, int64_t taskLimit)
{
    if(lb < ub) {return;}

    if ((ub - lb) < taskLimit )
    {
        //small array, therefore sort sequentially
        return sequential_quickSort(lb, ub, Array)
    ;
    }

    int64_t pivot_index = partition(lb, ub, Array)
    ;

    // left part of array
    #pragma omp task shared(Array)
    quickSort(lb, pivot_index - 1, Array,
    taskLimit);

    // right part of array
    #pragma omp task shared(Array)
    quickSort(pivot_index + 1, ub, Array,
    taskLimit);
}


void quickSort_openMP(int64_t lb, int64_t ub,
    uint64_t* Array)
{
    #pragma omp parallel
    {
        #pragma omp single
        quickSort(lb, ub, Array, 300);
        #pragma omp taskwait
    }
}
```

Listing 2: Source code of the parallelized Quicksort algorithm with openMP and the task construct

2.4 Sequential Mergesort algorithm

The C++ source code, along with pseudocode provided in Listing 3, defines the sequential Mergesort algorithm. The input for this algorithm are the size of the array (n), a uint64_t pointer to a temporary array (used during merge step to facilitate the merging of two sorted subarrays into a single sorted array) and a uint64_t pointer to the original array. The 'mergeSort' method recursively splits the array

into two halves until each subarray contains a single element. These individual elements are then merged back together in sorted order inside the 'merge' method. The merging process involves comparing the smallest unmerged elements from each subarray and combining them into a sorted sequence. This continues until all subarrays are merged into a single sorted array. Despite its favorable runtime, Mergesort requires the creation of new arrays at each step, resulting in a higher space complexity compared to Quicksort, which sorts the array in place.

```
79 void merge(int64_t n, uint64_t* tmp, uint64_t*
       Array)
80 {
81     int64_t i,k = 0;
82     int64_t j = n/2;
83
84     while(i<n/2 && j<n){
85         if(Array[i] < Array[j]){
86             tmp[k] = Array[i];
87             k++; i++;
88         } else{
89             tmp[k] = Array[j];
90             k++; j++;
91         }
92     }
93     while(i<n/2){
94         // lower half
95         tmp[k] = Array[i];
96         k++; i++;
97     }
98     while (j<n){
99         // upper half
100        tmp[k] = Array[j];
101        k++; j++;
102    }
103    memcpy(Array, tmp, n*sizeof(uint64_t));
104 }
105
106 void mergeSort(int64_t n, uint64_t* tmp, uint64_t*
       Array){
107    if (n < 2) return;
108
109    mergeSort(n/2, tmp, Array); //left array
110
111    mergeSort(n- n/2, tmp+n/2, Array+n/2); //right
       array
112
113    // merge two sub arrays back together
114    merge(n, tmp, Array);
115 }
```

Listing 3: Source code of the sequential Mergesort algorithm

2.5 Parallelized Mergesort algorithm

The code in Listing 4 defines three methods. The 'merge' method is identical to the one used in the sequential implementation. However, the recursive 'mergeSort' method includes some additional logic. A new variable, taskLimit, is introduced to set an upper bound on the size of the subarrays. When the size of a subarray falls below a certain threshold (specified by taskLimit), the sequential Mergesort algorithm is invoked to handle that smaller subarray. This mechanism helps avoid the overhead of parallelizing very small arrays, which could be inefficient due to the costs of task creation and thread management.

The main idea of parallelizing this algorithm is very similar to the idea of Quicksort. We want to have one thread run the recursive 'mergeSort' method, so it can write all tasks down, but then have all available threads execute the individual tasks in parallel fashion.

In the 'mergeSort' method, two tasks are created for each recursive call. This is achieved by placing the #pragma omp task shared(Array) directive above the recursive call, ensuring that the Array is shared across all tasks. Finally, "#pragma omp taskwait" ensures that the merging step "merge(n, tmp, Array)" only occurs after both recursive calls to "mergeSort" have completed. Note that the merge step remains sequential.

The third method 'mergeSort_openMP' creates the parallel region with "#pragma omp parallel". It's crucial that only one thread runs the recursive 'mergeSort' method to avoid the creation of excessive tasks. This is achieved using "#pragma omp single" , which ensures that only one thread creates tasks.

```
118 void merge(int64_t n, uint64_t* tmp, uint64_t*
       Array)
119 {
120  // same as sequential version
121 }
122
123 void mergeSort(int64_t n, uint64_t* tmp, uint64_t*
       Array, int64_t taskLimit){
124     if (n < 2) return;
125
126     if (n < taskLimit ){
127         //small array, therefore sort sequential
128         return sequential_mergeSort(n, tmp, Array)
        ;
129     }
130
131     #pragma omp task shared(Array)
132     mergeSort(n/2, tmp, Array, taskLimit); //left
        array
133
134     #pragma omp task shared(Array)
135     mergeSort(n- n/2, tmp+n/2, Array+n/2,
        taskLimit); //right array
136
137     // merge two sub arrays back together
138     #pragma omp taskwait
139     merge(n, tmp, Array);
140 }
141
142 void mergeSort_openMP(int64_t n, uint64_t* tmp,
       uint64_t* Array)
143 {
144     #pragma omp parallel
145     {
146         #pragma omp single
147         mergeSort(n, tmp, Array, 150);
148     }
149 }
```

Listing 4: Source code of parallelized MergeSort algorithm with openMP and task construct

3 Evaluation

The performance evaluation of the two sorting algorithms and their parallel counterparts is based on two experiments: First, measuring the runtime for each algorithm as the level of concurrency, and/or problem size, increases. Second, measuring the speedup between levels of concurrency, while the problem size, increases.

3.1 Computational platform and Software Environment

All experiments were conducted on a single CPU node 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.

Perlmutter uses the Cray Linux Environment (CLE) as its operating system with the Kernel version 5.14.21-150400.24.111_12.0.91-cray_shasta_c. The source code was compiled with GCC version 11.4.0.

3.2 Methodology

In this study, I used several metrics to assess performance, each of which I computed as follows: The parallel speedup for implementation x was calculated using the formula: $speedup(x) = (sequentialruntime)/(runtimeatconcurrencyx)$. I utilized the C++ Chrono Timer to instrument the code around each sorting algorithm and measure the runtime to sort an array in ascending order in milliseconds.

The tests were conducted over 6 different concurrency levels, ranging from 1, 4, 9, 15, 40, 60 and problem sizes ranging from 16, 32, 64, 128, 265 - million.

3.3 Runtime performance study - Quicksort

See Sec. 3.2 for computation details. Figure 1 illustrates the runtime of the Quicksort algorithm. It is clear that the sequential version has the longest runtime, taking 2.5 ms for a problem size of 16 million, and rising to 40 ms for the largest problem size of 256 million. As the problem size grows, the runtimes for all versions increase, but the parallel versions exhibit a significantly slower rate of growth compared to the sequential one. The best performance is achieved by the versions with concurrency levels of 15, 40, and 60, which start at around 1 ms and only increase to about 5 ms for the largest problem size. Notably, as the concurrency level increases, performance improves, but beyond a concurrency level of 15, there are no big improvements, with higher levels no longer yielding faster runtimes.

Figure 2 depicts the speedup of the Quicksort algorithm across different levels of concurrency. Concurrency level 4 demonstrates the lowest speedup, maintaining a steady rate of 3.4 for all problem sizes. As the problem size increases, the speedup for all concurrency levels, except 4, also improves. However, for a problem size of 32 million, there is a noticeable dip in speedup for nearly all concurrency levels, before the speedup rises again for larger problem sizes. Concurrency level 15 delivers the best overall speedup, being nearly 7 times faster than the sequential version for the smallest problem size and over 8 times faster for the largest problem size. Higher concurrency levels, such as 40 and 60, yield similar results for larger problem sizes but exhibit less speedup for smaller problem sizes, likely due to the overhead of managing a higher number of concurrent threads.

3.4 Runtime performance study - Mergesort

Refer to Section 3.2 for computation details. Figure4 illustrates the runtime of the Mergesort algorithm. It is clear that the sequential version has the longest runtime, taking 2.8 ms for a problem size of 16 million, and rising to almost 50 ms for the largest problem size of 256 million. As the problem size grows, the runtimes for all versions increase, but the parallel versions exhibit a significantly slower rate of growth compared to the sequential one. The best performance is achieved by the versions with concurrency levels of 9 and 15, which start at around 1 ms and only increase to about 11 ms for the largest problem size. Notably, as the concurrency level

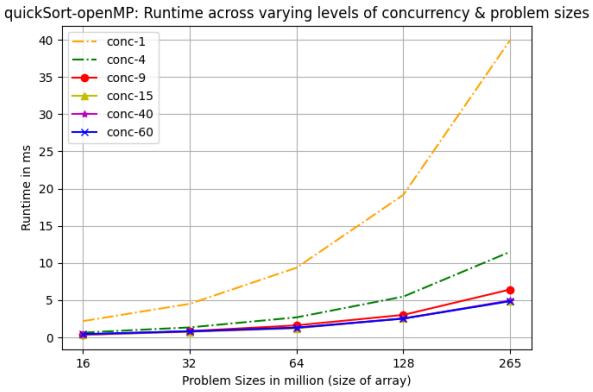

Figure 1: Quicksort algorithm: runtime across problem size in million and concurrency level

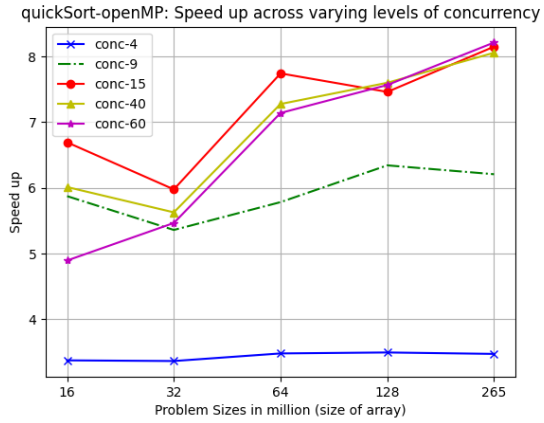

Figure 2: Quicksort algorithm: speed up across varying levels of concurrency

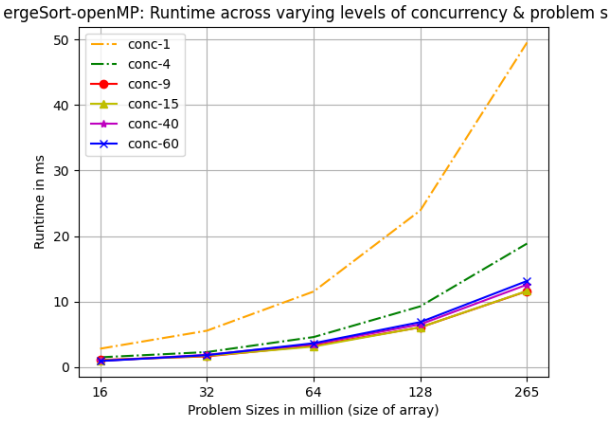

Figure 3: Mergesort algorithm: runtime across problem size in million and concurrency level

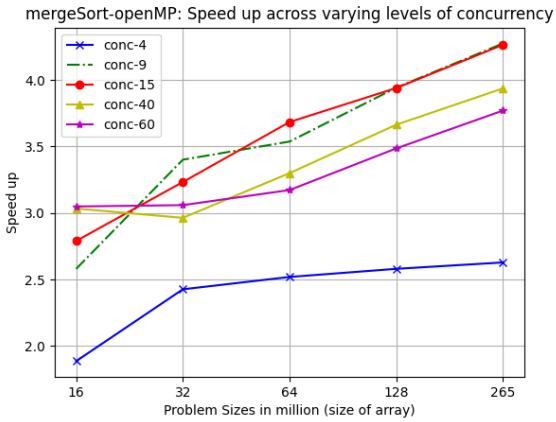

Figure 4: Mergesort algorithm: speed up across varying levels of concurrency

increases, performance improves, but beyond a concurrency level of 9, there are no big improvements, with higher levels no longer yielding faster runtimes.

Figure 2 depicts the speedup of the Mergesort algorithm across different levels of concurrency. Concurrency level 4 demonstrates the lowest speedup, having almost 2 at the beginning but then maintaining a steady rate of 2.5 for all other problem sizes. As the problem size increases, the speedup for all concurrency levels, except 4, also improves. Concurrency level 15 delivers the best overall speedup (with 9 being second best), being nearly 2.75 times faster than the sequential version for the smallest problem size and 4.25 times faster for the largest problem size. Higher concurrency levels, such as 40 and 60, yield worse results, likely due to the overhead of managing a higher number of concurrent threads.

3.5 Findings and Discussion

In this study, parallelizing both the Quicksort and Mergesort algorithms resulted in significant performance improvements. However, the Quicksort algorithm exhibited better performance gains when parallelized, achieving better results than Mergesort.

Both algorithms contain substantial sequential components that cannot be parallelized. For Quicksort, this sequential part is the partitioning step, while for Mergesort, it is the merging step. These sequential portions of the code limit the potential speedup achievable by parallelization. Since Quicksort's partitioning step requires less computational overhead and has a lower space complexity compared to the merging step in Mergesort, Quicksort benefited more from parallelization. Nevertheless, for larger problem sizes, both algorithms achieved impressive speedups—over 8x for Quicksort and over 4x for Mergesort. These results demonstrate that OpenMP's Task construct is an effective method for parallelizing recursive sorting algorithms like Quicksort and Mergesort.

A key observation is that for both algorithms, the speedup levels plateaus after a certain level of concurrency. Increasing the number of threads beyond a specific point no longer yields additional improvements in runtime. This phenomenon aligns with Amdahl's Law, where the performance gains from parallelization are capped by the size of the sequential part of the code.

Another important factor influencing performance is the taskLimit variable, which defines the maximum size of the subarrays that are eligible for parallelization. When the size of a subarray drops below this threshold, the algorithm switches to the sequential sorting version to handle the smaller subarray. This approach prevents the overhead of task creation and thread assignment from outweighing the benefits of parallelizing small subarrays. Multiple test runs indicated that a taskLimit of 300 for Quicksort and 150 for Mergesort provided the best performance across various problem sizes and concurrency levels.

To further optimize the parallel versions of these sorting algorithms, minimizing the sequential portions—partitioning for Quicksort and merging for Mergesort—is essential. By focusing on either reducing the computational cost or parallelizing these sequential steps, further performance improvements can be achieved in future work.

References

[1] O. A. R. Board. Openmp 5.0 specification, 2020. Accessed: 2024-12-10.
[2] cppreference.com. Chrono library in c++, 2024. Accessed: 2024-12-10.
[3] OpenMP Architecture Review Board. About us, 2023. Accessed: 2024-11-01.