UPPSALA
UNIVERSITET

Parallel Quicksort Implementation with OpenMP

Yuanjing Yang

July 11, 2025

1 Introduction

Quicksort is one of the fastest sorting algorithms, which is division and conquer-based sorting algorithm that picks an element as a pivot and partitions the given array around the picked pivot, the lower values are on the left of the pivot and higher values on the right of it, and then recursively sort it until the sub_array are too small to be sorted. The worst case scenario for quicksort that the pivot element is either the highest or lowest value in every sub_array is $O(n^2)$, and on average , the time complexity is $O(N \log N)$, the recursive part of quicksort is the reason [1], the good pick of the pivot element, the array will be split in half somewhat evenly each time the algorithm calls itself.
The divide_and_conquer nature of Quicksort makes it a natural candidate for parallization. Parallel computing offers an effective way to reduce execution time and improve performance. By breaking down the sorting problem into smaller, independent sub_problems that can be processed concurrently.

2 Problem description

The primary objective is to sort a large array of integers using parallel computing techniques. The program accept three input arguments as follows: `project n num_threads`
where the input arguments have the following meaning:
`n` is the size of array
`n_threads` is the number of threads to use, and it must be a power of two and greater than zero.
In parallel quicksort, it involves selecting a pivot, partitioning data around pivot, and then recursively sorting the sub-arrays, and in parallel context, these steps are distributed among multiple threads to achieve concurrent execution and accelerate the sorting process [2].

2.1 Solution method

The implemented parallel quicksort is optimized by shared-memory systems using Openmp. The main idea is to distribute the sorting tasks among available threads, pivot-based partitioning, and data exchanging.
Algorithm Overview:
1. Selecting a Global Pivot:
In **select_pivot** function, each thread sort its array and computes the local median, and collect it to **global_medians[group][locid]**, where **group** identifies the current logical thread group(**myid/current_group_size**0, and **locid** is the thread's ID within that group(**myid % current_group_size**0. And only the thread with **locid==0** collects all local medians within its group, sorts them and selects the median of these medians as the global pivot for the current recursion level. The chosen pivot is sorted in **global_pivots[group]** [3].
(2) Local Data partitioning:

In **findsplit** function, it return the **split** index. In this part, for avoiding race conditions and improve temporal locality, each thread has its private copy of the buffer, the elements are written to temporary buffer based on pivot condition, and then copy back to the original array.
(3) Parallel Data exchange:
In **exchange_data** function, I was inspiered by the MPI communication pattern by exchanging boundary information between partitions. For lower half of current group, it keeps the elements less than or equal to the pivot, and sends the elements greater than the pivot, and the upper half is opposite, it keeps the elements greater than the pivot and sends the elements less than or equal to pivot, and then **send_scr** data is copied into **thread_tem_buffers[myid]**, it simliar to the concept of message buffers in MPI. Each thread identified its partner thread for exchanging data. The partner thread repeat the operation above. Here I used double-buffering system [4], which can manage data efficiently and reduce memory allocation overhead during exchange, and the **buffer_track[myid]** array keeps track of the currently active buffer and the **target_buffer_index** determines the inactive buffer that merged data will be wrriten, the **target_buffer** pointer is then set to **thread_buffers_A[myid]** or **thread_buffers_B[myid]**. [5]
(4) Recursive calls:
The **global_sort** function is called recursively, and the **current_gourp_size** is halved in each recursive call, and eventually leading to **current_group_size ¡=1**, and each thread performed a final **sort_array** [6].

2.2 Experiment

1. Correctness verification:
In **main** function, I used **is_sorted** function to confirm that all elements are in ascending order, the example output:

Figure 1: Example output(N=1,000,000)

2. Evaluation of Performance:
As the table showed below, we can see the execution time increases with the array sized for a fixed number of threads, this indicates that the complexity $O(N \log N)$ and this algorithm scales well with increasing input array size.

To evaluate the performance of parallel efficiency, I tested the situation that fixed array size with different number of threads.

$$\text{Speedup} = \frac{T_1}{T_n}$$

Table 1: Execution Time for $P = 4$ Threads

Array Size (N)	**Time (s)**
1,000,000	0.0344
5,000,000	0.2039
10,000,000	0.3985
50,000,000	2.1566
100,000,000	4.4815

If the speedup is ideal, then with n threads, the execution time should be

$$T_n = \frac{T_1}{n}$$

From the figure below, we can see that while speedup reached 5.29x at 16 threads, the efficiency dropped to 0.33, indicating that each additional thread contributed less effectively to overall performance Within the increase of threads, the speedup is achieved, but it is not ideal, and the efficiency is decreasing.

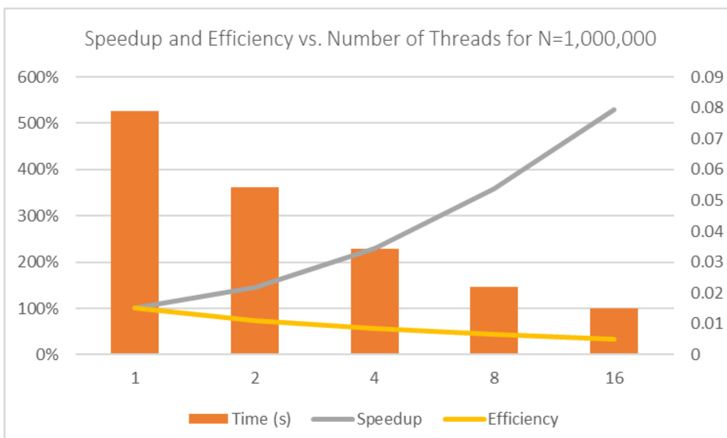

Figure 2: Speedup and Efficiency vs. Number of Threads(N=1,000,000)

There has two reasons for this results, one is communication overhead, even this algorithm used double-buffering system, but the data exchanging between threads possibly leads to cache coherence overhead. One is Synchronization Overhead, even the pivot strategy creates balanced partitions, but with the use of synchronization(**pragma omp barrier**), all threads need to wait until the slowest thread reaches that points.

3 Conclusion

This project implemented and evaluated an OpenMP-based parallel Quicksort algorithm, which is a divided-and-conquer strategy with double-buffering system

and data exchanging, parallelizing the sorting processes on shared-memory. The result shows the algorithm's correctness, and it achieves a speedup, reducing the execution time. However, the speedup didn't fit the expectation perfectly, it can be considered about influence of communication and Synchronization overheads. Here are some ideas for improvement:
1. Using MPI in **exchange_data** function.
2. Using Dynamic scheduling [7] for better load balancing(I have tried this before, but it hard to match with the strategy of data exchanging, and it causes more complex race conditions, leading to a bad performance).

4 Declaration of AI

In my project, I used Chatgpt and deepseek to write some repeated and basic codes, debugging and make a improvement of my code. I write the original code frame and tell AI my the ideas, for example, I studied parallel and distribution programming in this period, and learn the use of MPI [6], so I plan to use it in this project, chatgpt help me Outlines the process of converting from MPI to OpenMP(in exchanging data part). AI help me to find some related materials, I found a way to improve performance(double buffer system) and looked for the bibliographic on online library, it increases the efficiency a lot, I don't need to waste a lot time in searching method. AI also helped me to check the debug information, especially when I used gdb and valgrind, it helps me find the errors quickly, and AI also help me to modify my code based on my results, I always get some useful feedback from AI.

References

[1] W3Schools. Dsa - quick sort algorithm, 2025. Accessed: July 11, 2025.

[2] GeeksforGeeks. Implementation of quick sort using mpi, omp and posix thread, 2025. Accessed: July 11, 2025.

[3] year = 2025 note = Accessed: 4 11, 2025 url = Quicksort.pdf Jarmo Rantakokko, title = Parallel Quick Sort.

[4] Reddit User Community. How does a double buffer allow things to be drawn smoothly?, 2022. Accessed: July 11, 2025.

[5] Eko Dwi Nugroho, Ilham Firman Ashari, Muhammad Nashrullah, Muhammad Habib Algifari, and Miranti Verdiana. Comparative analysis of openmp and mpi parallel computing implementations in team sort algorithm. *Journal of Applied Informatics and Computing*, 7(2):141–149, 2023.

[6] Peter S. Pacheco. *An introduction to parallel programming*. Elsevier/Morgan Kaufmann, Amsterdam ;, 2011.

[7] Kil Jae Kim, Seong Jin Cho, and Jae-Wook Jeon. Parallel quick sort algorithms analysis using openmp 3.0 in embedded system. In *2011 11th International Conference on Control, Automation and Systems*, pages 757–761. IEEE, 2011.

5 Appendix

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <string.h>


/*Reference: Quicksor.pdf, Lab5-lab10, Assignment2-4, chatgpt, deepseek,
Pacheco, P. S. (2011). An introduction to parallel programming. Elsevier/Morgan Kaufmann.
Nugroho, E. D., Ashari, I. F., Nashrullah, M., Algifari, M. H., & Verdiana, M. (2023). Comparative Analysis of OpenMP and MPI Parallel Computing Implementations in Team Sort Algorithm. Journal of Applied Informatics and Computing, 7(2), 141 149 . https://doi.org/10.30871/jaic.v7i2.6409
Kil Jae Kim, Seong Jin Cho, & Jae-Wook Jeon. (2011). Parallel quick sort algorithms analysis using OpenMP 3.0 in embedded system. 2011 11th International Conference on Control, Automation and Systems, 757 761 .
https://www.geeksforgeeks.org/dsa/implementation-of-quick-sort-using-mpi-omp-and-posix-thread/
```

```
*/

typedef struct {
    int* data;
    int size;
} ThreadData;

int compare(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

void sort_array(int* arr, int size) {
    qsort(arr, size, sizeof(int), compare);
}

int** global_medians = NULL;
int* global_pivots = NULL;
int* splitpoints = NULL;
int* exchange_sizes = NULL;

//Here I used double buffer system, I learn it from chatgpt
    and lab6(Temporal locality)
int** thread_temp_buffers = NULL;
int** thread_buffers_A = NULL;
int** thread_buffers_B = NULL;
// This buffer can check which buffer is active, and buffer
    A is 0, buffer B is 1
int* buffer_track = NULL;
// threadprivate, advoiding race condition
static int* findsplit_local_temp_buffer = NULL;
static int findsplit_local_temp_buffer_size = 0;
#ifdef _OPENMP
#pragma omp threadprivate(findsplit_local_temp_buffer,
    findsplit_local_temp_buffer_size)
#endif

int select_pivot(ThreadData* threadData, int num_threads,
    int group_size) {
    int myid = omp_get_thread_num();
    int locid = myid % group_size;
    int group = myid / group_size;

    int* data = threadData[myid].data;
    int size = threadData[myid].size;

    sort_array(data, size);

    int median;
    if (size == 0){
        median = 0;
```

```
    }else if (size == 1) {
        median = data[0];
    }else if (size % 2 == 0) {
        median = (data[size/2 - 1] + data[size/2]) / 2;
    }else{
        median = data[size/2];
    }

    global_medians[group][locid] = median;
    #pragma omp barrier

    if (locid == 0) {
        sort_array(global_medians[group], group_size);
        int mid = group_size / 2;
        if (group_size % 2 == 0){
            global_pivots[group]=(global_medians[group][mid
                -1] + global_medians[group][mid]) / 2;
        }else{
            global_pivots[group]=global_medians[group][mid];
        }
    }
    #pragma omp barrier
    return global_pivots[group];
}

int findsplit(int* data, int size, int pivot) {
    int* temp = findsplit_local_temp_buffer;
    int left = 0;
    for (int i = 0; i < size; i++) {
        if (data[i] <= pivot) {
            temp[left++] = data[i];
        }
    }
    int split = left;
    for (int i = 0; i < size; i++) {
        if (data[i] > pivot) {
            temp[left++] = data[i];
        }
    }
    memcpy(data, temp, size * sizeof(int));
    return split;
}

void merge(int* dest, int* a, int size_a, int* b, int size_b
    ) {
    int i = 0, j = 0, k = 0;
    while (i < size_a && j < size_b)
        dest[k++] = (a[i] <= b[j]) ? a[i++] : b[j++];
    while (i < size_a) dest[k++] = a[i++];
    while (j < size_b) dest[k++] = b[j++];
```

```
}
/*Here, the data exchange algorithm is inspired MPI
1.Each thread has a partner thread for data exchange
2.Data is copied into temporary buffers (thread_temp_buffers), mimicking MPI's send/receive buffers.
3.shared memory for direct buffer access, avoiding the overhead of communication.
*/
void exchange_data(ThreadData* threadData, int group_size) {
    int myid = omp_get_thread_num();
    int locid = myid % group_size;

    int* current_data = threadData[myid].data;
    int current_size = threadData[myid].size;
    int my_split = splitpoints[myid];

    int *keep_src, *send_src;
    int keep_size, send_size;

    // For lower group, keeping the data less than pivot, and send the data larger than pivot
    if (locid < group_size / 2) {
        keep_src = current_data;
        keep_size = my_split;
        send_src = current_data + my_split;
        send_size = current_size - my_split;
    } else {
         // For upper group, keeping the data larger than pivot, and sednd the data less than pivot
        keep_src = current_data + my_split;
        keep_size = current_size - my_split;
        send_src = current_data;
        send_size = my_split;
    }

    // Copy data to be sent to temporary buffer
    memcpy(thread_temp_buffers[myid], send_src, send_size * sizeof(int));
    exchange_sizes[myid] = send_size;

    #pragma omp barrier

    // Partner thread for exchange
    int partner = (locid < group_size / 2) ? myid + group_size / 2 : myid - group_size / 2;
    int recv_size = exchange_sizes[partner];
    int* recv_data_from_partner = thread_temp_buffers[partner]; // Partner's temp buffer

    //target buffer(buffer A or buffer B)
```

```
    int target_buffer_idx = 1 - buffer_track[myid];
    int* target_buffer = (target_buffer_idx == 0) ? thread_buffers_A[myid] : thread_buffers_B[myid];


    merge(target_buffer, keep_src, keep_size, recv_data_from_partner, recv_size);


    threadData[myid].data = target_buffer;
    threadData[myid].size = keep_size + recv_size;
    buffer_track[myid] = target_buffer_idx;

    #pragma omp barrier
}


void global_sort(ThreadData* threadData, int current_group_size) {
    if (current_group_size <= 1) {
        int myid = omp_get_thread_num();
        sort_array(threadData[myid].data, threadData[myid].size);
        return;
    }

    int pivot = select_pivot(threadData, omp_get_num_threads(), current_group_size);

    int myid = omp_get_thread_num();
    splitpoints[myid] = findsplit(threadData[myid].data, threadData[myid].size, pivot);

    exchange_data(threadData, current_group_size);
    global_sort(threadData, current_group_size / 2);
}

int main(int argc, char** argv) {
    if (argc != 3) {
        printf("Usage: %s <array_size> <num_threads>\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    int num_threads = atoi(argv[2]);

    if ((num_threads & (num_threads - 1)) != 0 || num_threads == 0) {
        printf("Number of threads must be a power of 2 and greater than 0.\n");
```

```
        return 1;
    }

    omp_set_num_threads(num_threads);


    int* data = malloc(n * sizeof(int));

    // Fixed seed for reproducibility, learn from chatgpt
    srand(42);
    for (int i = 0; i < n; i++) {
        data[i] = rand() % 10000;
    }


    ThreadData* threadData = malloc(num_threads * sizeof(ThreadData));

    global_pivots = malloc((num_threads / 2) * sizeof(int));

    global_medians = malloc((num_threads / 2) * sizeof(int*));

    for (int i = 0; i < num_threads / 2; ++i) {
        global_medians[i] = malloc(num_threads * sizeof(int));

    }

    splitpoints = malloc(num_threads * sizeof(int));
    exchange_sizes = malloc(num_threads * sizeof(int));
    thread_temp_buffers = malloc(num_threads * sizeof(int*));
    thread_buffers_A = malloc(num_threads * sizeof(int*));
    thread_buffers_B = malloc(num_threads * sizeof(int*));
    buffer_track = malloc(num_threads * sizeof(int));


    double start_time, end_time; // Declare timing variables here

    // Each thread initializes its own buffers and calls global_sort
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        thread_buffers_A[tid] = malloc(n * sizeof(int));
        thread_buffers_B[tid] = malloc(n * sizeof(int));
        thread_temp_buffers[tid] = malloc(n * sizeof(int)); // Max possible size for temp buffer
```

```c
        findsplit_local_temp_buffer = malloc(n * sizeof(int));
        findsplit_local_temp_buffer_size = n;
        // Start with A as active
        buffer_track[tid] = 0;

        int chunk_size = n / num_threads;
        int offset = tid * chunk_size;
        int current_thread_actual_size = (tid == num_threads - 1) ? (n - offset) : chunk_size;
        // Copy data to this thread's buffer A
        memcpy(thread_buffers_A[tid], data + offset, current_thread_actual_size * sizeof(int));
        threadData[tid].data = thread_buffers_A[tid];
        threadData[tid].size = current_thread_actual_size;

        #pragma omp barrier
        // only master thread record runtime
        #pragma omp master
        {
            start_time = omp_get_wtime();
        }

        global_sort(threadData, num_threads);

        #pragma omp master
        {
            end_time = omp_get_wtime();
        }

        // Each thread frees its own threadprivate buffer, so we need free it after sort function
        if (findsplit_local_temp_buffer != NULL) {
            free(findsplit_local_temp_buffer);
            findsplit_local_temp_buffer = NULL;
        }
    }

    // Collect results back into the original data
    int current_offset = 0;
    for (int i = 0; i < num_threads; i++) {
        memcpy(data + current_offset, threadData[i].data, threadData[i].size * sizeof(int));
        current_offset += threadData[i].size;
    }

    int is_sorted = 1;
    for (int i = 1; i < n; i++) {
        if (data[i-1] > data[i]) {
            is_sorted = 0;
```

```c
                break;
            }
        }

        if (is_sorted) {
            printf("OK! Array is sorted.\n");
            printf("Time: %.4fs\n", end_time - start_time);
        } else {
            printf("Error! Array is not sorted.\n");
            return 1;
        }


        for (int i = 0; i < num_threads; i++) {
            free(thread_buffers_A[i]);
            free(thread_buffers_B[i]);
            free(thread_temp_buffers[i]);
        }
        free(thread_buffers_A);
        free(thread_buffers_B);
        free(thread_temp_buffers);

        for (int i = 0; i < num_threads / 2; ++i) {
            free(global_medians[i]);
        }
        free(global_medians);
        free(buffer_track);
        free(splitpoints);
        free(exchange_sizes);
        free(global_pivots);
        free(threadData);
        free(data);

        return 0;
}
```