# Parallel Sort on OpenMP

High Performance Computing (CITS5507)

Daniel Tang (21140852)

## Introduction

We investigate a number of different sorting algorithms written in C code, comparing the differences between sequential implementations of the algorithms against parallel versions implemented with OpenMP. The sorting algorithms in question include quick sort, enumerate sort, and merge sort. We expect there will be a number of trade-offs between the number of threads used for parallel computation, the size of the arrays that are to be sorted, as well as the OpenMP constructs used (i.e. sections, tasks, basic parallel for, and so on). Our objective is to find an optimised parallel implementation for each sorting algorithm and discuss efficiency parallel improvements.

## Sorting Algorithms

**Quick sort** is a divide-and-conquer sorting algorithm, which involves separating the input array into smaller sub-arrays and then sorting in a recursive fashion, finally combining the results from all smaller problems to get the final sorted output. Here we pick a particular element in an array to act as a pivot point - in our experiments we pivot on the last element in the array (however, we could also pick the first, the median value, or even select one at random).

Starting from the first index, we traverse the array in order finding elements smaller than the pivot element. If we find one, we swap it to the current index position in-place, and repeat until the array is partitioned into two arrays with elements smaller than the pivot to the left and elements greater than the pivot to the right. This recursive process of sorting about a pivot is repeated on smaller sub-arrays until they are of length two, swapping memory addresses in-place to sort each sub-array.

Quick sort is considered a memory efficient algorithm as it does not require an additional workspace array to store the results of intermediate computations for sorting in sub-arrays. However, a downside of this algorithm is that runtimes may have higher variance depending on the initial conditions of the random array as well as and the chosen pivot index.

**Merge sort** also leverages a divide-and-conquer sorting approach where the algorithm recursively splits the input array in half until final sub-arrays are of length two. For each sub-array, we copy elements into a temporary workspace array in correctly sorted order, and then merge the corresponding output with the other array split at the same level of recursion. The original array's values are then updated with the sorted values from the workspace array, and this process repeats recursively until the full array size is sorted.

One notable difference between merge sort and quick sort is the requirement for additional temporary arrays as it is not an in-place sort. However, merge sort is a more balanced algorithm compared to quick sort; it is not subject to variations in run-time due to different initial array values as it splits sub-arrays in half by index at each recursive step.

**Enumeration sort** works by considering each element in the array and counting how many other elements in the array it is greater than and storing the counts in a temporary array of the same size. These counts result in a unique enumeration for each array element that specifies the index position in sorted ascending order. Array duplicates can be handled simply by a convention that ranks duplicates by preserving whatever relative order they were in in the original array. Naturally, the for loop style comparison to each element, for each element, is a process that can be considered “embarrassingly parallel” and so we should expect to see easy run-time improvements from parallelisation.

## Method

Each sorting algorithm was developed with a serial and a parallel implementation and tested with a range of different input array sizes initialised with random double precision floating point numbers between 0.0 and 1.0. In particular, we tested a serial implementation (one thread without OpenMP) and a number of parallel implementations with multiple threads [2, 4, 6, 8, 16] with input arrays sizes of [1000, 5000, 10,000, 50,000, 100,000, 500,000, 1,000,000,000].

Larger input array sizes above 100,000 required dynamic memory allocation and in the end was not tested with enumeration sort due to its significant increase in run-time proportional to the size of the array. The mean over 5 trials for each experiment was used to minimise the variance in run-time due to random chance, such as unexpected external processes on the same operating system.

Additionally, overheads with instantiating new threads may result in circumstances where sequential operation is more efficient compared to parallel versions, especially for smaller array sizes. To address this expected difficulty, we design hybrid parallel algorithms that switch to serial implementations for small array sizes (our cutoff is an array size of 100).

We parallelise quick sort and merge sort algorithms with OpenMP using task and sections constructs and design a parallel algorithm for enumeration sort using parallel for and guided scheduling. The pseudo-code for our serial and parallel solutions are provided in the Appendix section.

### Environment

All code was written in C with OpenMP and run on a machine with the following specifications:

- **Operating System:** Ubuntu 18.04.5 LTS
- **CPU:** AMD Ryzen Threadripper 2920X 12-Core Processor
- **RAM:** 62.7GB

Source code was compiled with the command:

```
gcc -fopenmp -o main main.c experiments.c quick_sort.c merge_sort.c enumeration_sort.c random_array.c
```

## Results

Our results for serial implementations compared to our parallel implementations with an array size of 100,000 are presented below. The best implementations of quick sort and merge sort utilised tasks rather than sections constructs and switched to serial code for array sizes below 100.

| [ARRAY SIZE = 100000 \| CUT OFF = 100] | **Enumeration Sort** | **Merge Sort (Tasks)** | **Quick Sort (Tasks)** |
|---|---|---|---|
| **Serial (1 Thread)** | 26.825874s | 0.014113s | 0.016350s |
| **Hybrid Parallel (2 Threads)** | 23.931718s | 0.011017s | 0.012862s |
| **Hybrid Parallel (4 Threads)** | 12.060942s | 0.012159s | 0.007813s |
| **Hybrid Parallel (8 Threads)** | 6.115675s | 0.007759s | 0.006974s |
| **Hybrid Parallel (16 Threads)** | 3.458822s | 0.007116s | 0.008051s |

Table 1. A summary of serial sorting algorithms against hybrid parallel sorting algorithms for an array size of 100,000.

Relative to their serial baselines, quick sort has a x2.34 speed-up (with 8 threads), merge sort has a x1.97 speed-up (with 16 threads), and we find a x7.76 speed-up for enumeration sort (with 16 threads) - although enumeration sort is order of magnitudes slower than the first two algorithms.

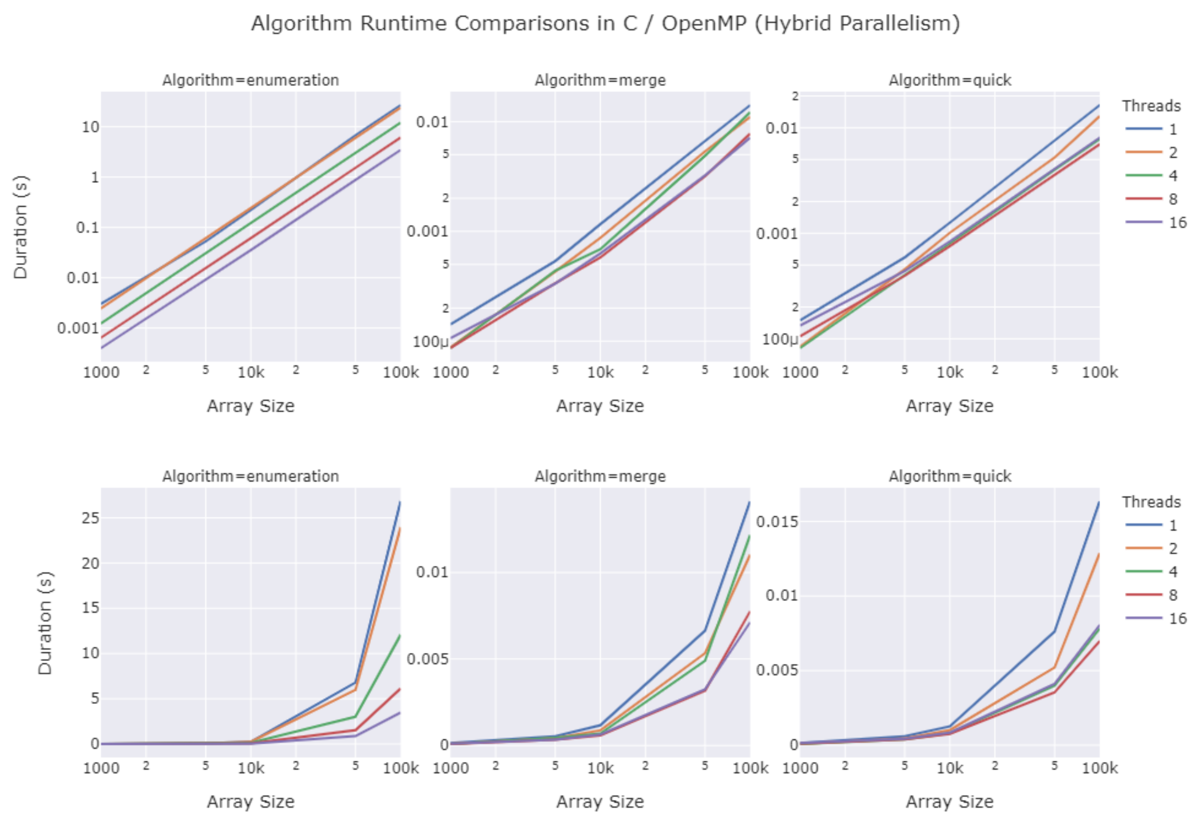

Figure 1. Results for the three sorting algorithms (tasking for quick sort and merge sort and parallel for enumeration sort) showing duration vs. array size. The first row results are displayed as a log-log plot and the second row shows the same data but as a log-linear plot (y-axis are not shared to show the extent of differences in run-time between all algorithms).

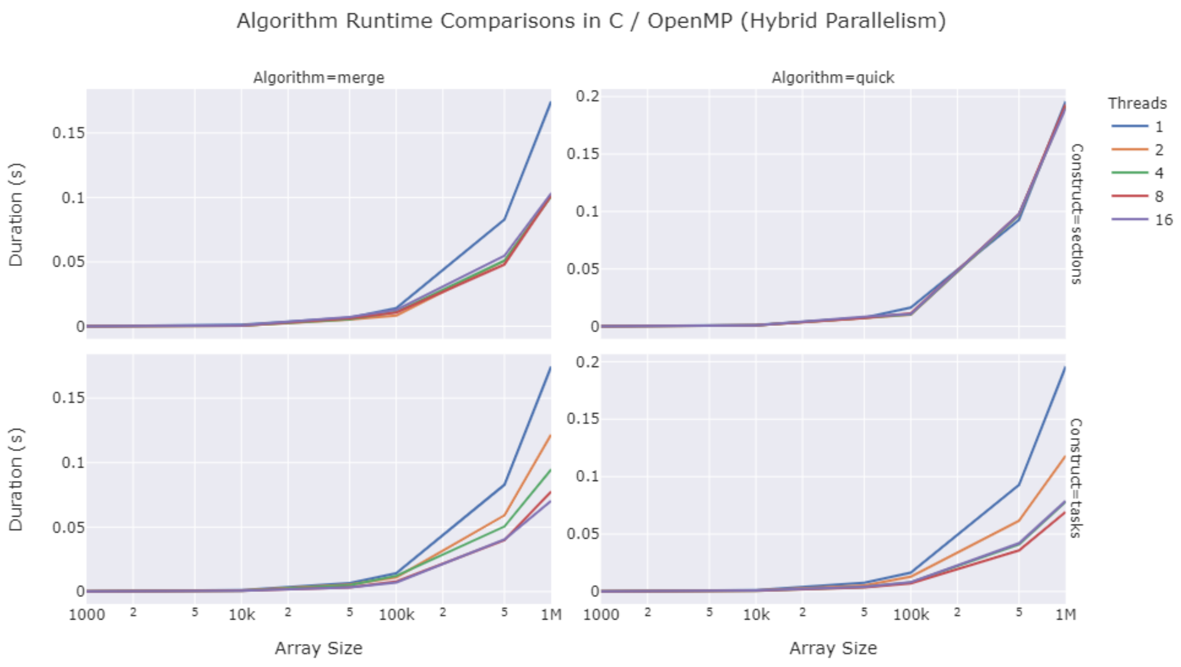

Figure 2. Mean sorting algorithms run-time over five trials vs input array size for both quick sort and merge sort algorithms each implemented with tasking and sections for a number of threads. For threads greater than one, we use hybrid parallelism where our algorithm switches to serial below an array size of 100. The plot shown with log-linear axes; and duration (y axis) is shared between each subplot to show variation in run-times between algorithms was minimal.

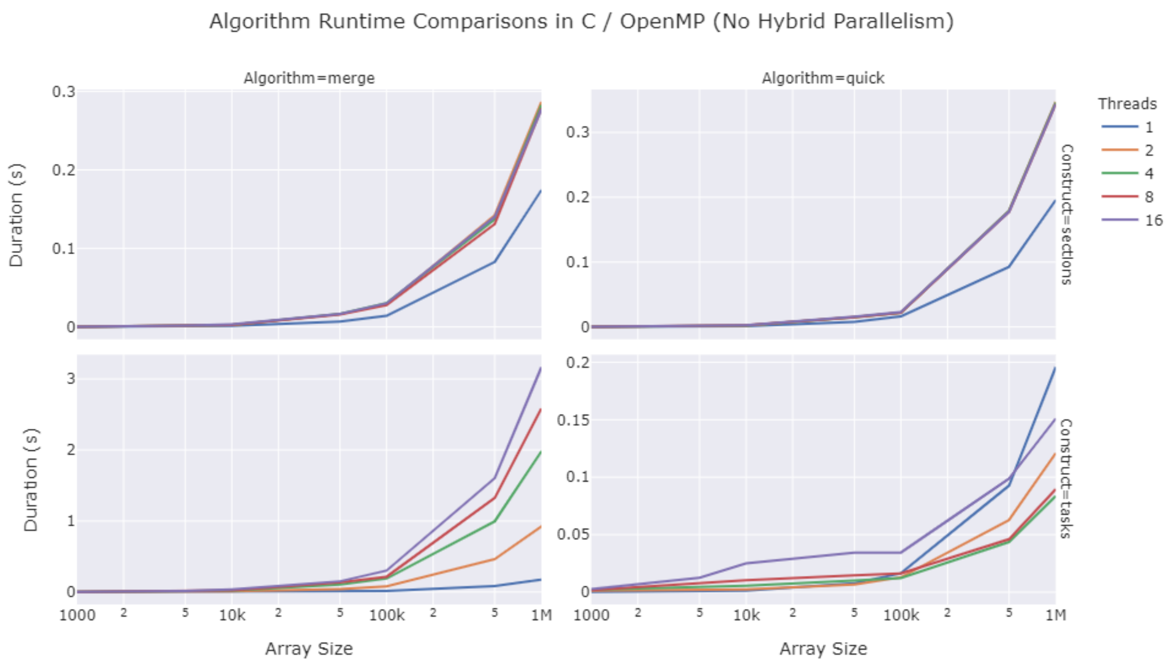

Figure 3. Mean sorting algorithms run-time over five trials vs input array size for both quick sort and merge sort algorithms each implemented with tasking and sections for a number of threads without hybrid parallelism. The plot shown with log-linear axes; and duration (y axis) is *not* shared between each subplot to be able to show the variation in run-times for large array sizes.

## Discussion

Firstly, we found that enumeration sort is considerably slower than the other two sorting algorithms, especially with larger array sizes. However, as this algorithm functions by comparing every element with every other element in the array, we should expect it to scale poorly for larger arrays with complexity $O(n^2)$. Conversely, quick sort and merge sort are expected to be faster than enumeration sort and of complexity $O(nlog(n))$in the best case, which appears consistent with our results.

We also tested both quick sort and merge sort with OpenMP sections constructs, but found that tasks performed better, especially when it was able to access more threads. Conversely, sections appeared to be capped in performance after 2 threads. While tasking can more efficiently distribute computing resources compared to sections, we assume the run-time difference may be due to the fact that our recursive implementation with parallel sections is executing depth first in pairs before another code block section starts and not taking advantage of more than two threads at a time.

Notably for merge sort, our implementation showed a large blow up in run-time for multi-threading with tasks without hybrid parallelism, which was not present for the quick sort case. This is likely a multi-threading issue as the speed decrease was proportional to the number of threads in this case (see Figure 3). We expect this is related to copying memory from the workspace array back to the original array, which significantly slows run-times when running multiple threads with small arrays. The fact that our sections implementation did not suffer from this slow down (likely as it seemed to only max out utilisation with 2 threads as discussed above). Our hybrid parallelism solution solves this but we expect there are more optimal merge sort implementations that solve this issue as well.

For quick sort, we find that the tasking implementation for the hybrid parallel version of the algorithm is considerably faster than using the sections construct. Sections specify the worksharing structure between threads at compile time, which means it is not as effective for handling code execution blocks with dynamic run-times. Dynamic run-times are particularly relevant for quick sort, as when the chosen pivot happens to be the middle value when sorted, sub-arrays end up being more balanced in size and less recursion is required, making the run-time of the algorithm tends towards its best case of $O(nlog(n))$. In the worst case when the pivot is always either the greatest or smallest value, run-time tends to $O(n^2)$to be equivalent to enumeration sort. As quick sort is more variable in its run-time (i.e. dependent on the random values in the array and the chosen pivot), we confirmed that tasking better handles sorting in a more efficient manner for this algorithm, especially when multiple threads are available.

## Summary

We find quick sort and merge sort to have extremely competitive performance for parallel implementations. Conversely, enumeration sort does not scale with larger array sizes, even with parallel implementations. Tasking appears to be the most efficient parallel construct for our case compared to sections, but only if a hybrid implementation is used (for merge sort) as managing multiple threads with smaller array sizes introduces unnecessary overheads. Our experiments confirm our expectations of quick sort and merge sort outperforming enumeration sort and highlight the importance of proper multithreading management and algorithm design for efficient parallelism.

## References

GeeksforGeeks, 2017. QuickSort - GeeksforGeeks. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/quick-sort/> [Accessed 17 September 2021].

GeeksforGeeks, 2017. MergeSort - GeeksforGeeks. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/merge-sort/> [Accessed 17 September 2021].

## Appendix

Pseudo-code is provided for each of the sorting algorithms below.

### Quick Sort

Below we have the partition function that splits arrays based specifically on the last element of the list, as well as a serial quick sort and a tasking implementation with OpenMP directives.

```
int partition (double *arr, int low, int high)
{
  double pivot = arr[high];
  int i = low - 1;

  // loop through partitioned array indices
  for (int j = low; j < high; j++)
  {
    // if current element is smaller than the pivot
    if (arr[j] < pivot)
    {
      i++; // increment index of smaller element
      swap(&arr[i], &arr[j]);  // pass mem address of array
    }
  }
  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}

void quick_sort(double *arr, int low, int high)
{
  {
    if (low >= 0 && high >= 0)
    {
      if (low < high)
      {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
      }
      }
   }
}
```

**void quick_sort_tasks(double *arr, int low, int high, int cutoff)**

```
{
   {
      if (low >= 0 && high >= 0)
      {
         if (low < high)
         {
            int pi = partition(arr, low, high);

            // separately sort elements before and after pivot
            // worker takes task itself if array bigger than cutoff
            #pragma omp task shared(arr) firstprivate(low, pi) if (high - low > cutoff)
            {
               quick_sort_tasks(arr, low, pi - 1, cutoff);
            }
            #pragma omp task shared(arr) firstprivate(pi, high) if (high - low > cutoff)
            {
               quick_sort_tasks(arr, pi + 1, high, cutoff);
            }
         }
      }
   }
}
```

## Merge Sort

We show code snippets for the main merge function, as well as a recursive serial implementation of merge sort and a recursive parallel implementation with sections.

**void merge(double *arr, double *temp, int size)**

```
{
   // begin loop to copy from temp
   // to main array in sorted order
   int i = 0;
   int j = size/2;
   int n = 0;

   // loop if both subarrays have elements
   while (i<(size/2) && j<size)
   {
      // add smaller element to temp array
      if (arr[j] > arr[i])
      {
            temp[n] = arr[i];
            i++;
        }
        else
        {
            temp[n] = arr[j];
            j++;
        }
        n++;
    }

    // if both conditions above not satisfied
    // then one subarray has been emptied
    // loop through remaining (sorted) elements
    while (i < (size/2))
    {
        temp[n] = arr[i];
        i++;
        n++;
    }

    while (j < size)
    {
        temp[n] = arr[j];
        j++;
        n++;
    }

    // copy temp array into main array
    memcpy(arr, temp, size*sizeof(double));
}

void merge_sort(double *arr, double *temp, int size)
{
    if (size < 2)
        return;

    // recursively sort subarrays
    // select sub-arrays by incrementing pointer positions
    merge_sort(arr, temp, size/2);
    merge_sort(arr + (size/2), temp + (size/2), size - (size/2));
    merge(arr, temp, size);
}

void merge_sort_sections(double *arr, double *temp, int size, int cutoff)
{
    if (size < 2)
        return;
    if (size > cutoff)
    {
        #pragma omp parallel shared(arr, temp)
        #pragma omp sections
        {
            #pragma omp section
            {
                // merge sort lower half
                merge_sort_sections(arr, temp, size/2, cutoff);
            }
            #pragma omp section
            {
                // merge sort upper half
                merge_sort_sections(arr + (size/2), temp + (size/2), size - (size/2), cutoff);
            }
        }
    }
    else
    {
        merge_sort(arr, temp, size/2);
        merge_sort(arr + (size/2), temp + (size/2), size - (size/2));
    }
    merge(arr, temp, size);
}
```

## Enumeration Sort

We show both a serial implementation and a parallel implementation for enumeration sort.

```
void enumeration_sort(double arr[], double temp[], int size)
{
    // initialize enumerated array of ranks with 0s
    int ranks[size];
    for (int i = 0; i < size; i++)
        ranks[i] = 0;

    for (int i = 0; i < size; i++)
    {
        for (int j = i+1; j < size; j++)
        {
            if (arr[i] > arr[j])
                ranks[i]++;
            else
                ranks[j]++;
        }

        temp[ranks[i]] = arr[i];
    }

    // copy temp array into main array
    memcpy(arr, temp, size*sizeof(double));
}
```

**void enumeration_sort_parallel(double arr[], double temp[], int size)**

```
{
    // compare each element against other elements in parallel
    // rank is how many other elements it is greater than
    #pragma omp parallel for shared(arr, temp) schedule(guided)
    for (int i = 0; i < size; i++)
    {
        int rank = 0;
        for (int j = 0; j < size; j++)
        {
            if ((arr[i] > arr[j]) || (i > j && arr[i] == arr[j]))
                rank++;
        }
        temp[rank] = arr[i];
    }

    // copy temp array into main array
    memcpy(arr, temp, size*sizeof(double));
}
```