Parallel Sort with MPI and OpenMP

High Performance Computing (CITS5507)

Daniel Tang (21140852)

Introduction

We investigate quick sort, enumerate sort, and merge sort algorithms written in C code, comparing the serial and parallel implementations of these algorithms using both MPI and OpenMP. As discovered in Project 1, there were a number of trade-offs between the number of threads used, the size of the arrays, and the OpenMP approach chosen. For this report, we expect the same trade-offs to be present in regards to the number of processes, the size of our data, communication overhead, and the overall MPI approaches utilised. Our aim is to investigate a number of methods that utilise MPI in order to discover optimisations for the sorting algorithms of interest.

Sorting Algorithms

Quick sort involves selecting a pivot value (e.g. the last element, the first, the median) and then traversing the array, swapping elements in-place until the array is partitioned into two sub-arrays with elements smaller than the pivot to the left and elements greater than the pivot to the right. This process of partitioning about the pivot and splitting into sub-arrays continues recursively until arrays are of length two, which results in the array being fully sorted.

Merge sort is a similar recursive sorting algorithm that splits the input array in half until the final sub-arrays have only two elements. For each sub-array, we copy elements into a temporary workspace array in the correctly sorted order, and then merge the corresponding output with the other array split at the same level of recursion. The data is copied back from the workspace array and this process repeats recursively until the full array size is sorted.

Enumeration sort works by iterating through each array element and counting how many other elements in the array it is greater than, storing the result in a temporary array of the same size. The resulting counts for each element specify the index positions to sort the array in ascending order.

Method

We recorded the mean runtime over 3 trials for each experiment with the following configurations:

- Array Sizes = [10,000, 100,000, 1,000,000, 10,000,000, 100,000,000].
- OpenMP Threads = [2, 4, 8] and a serial (1 thread) implementation.
- MPI Processes = [2, 4, 8, 16] and a serial (1 process) implementation.
- Sorting Algorithms (with OpenMP):
    - Quick Sort (serial and hybrid tasks with cutoff size 100).
    - Merge Sort (serial and hybrid tasks with cutoff size 100).
    - Enumeration Sort (serial and parallel for; with maximum array size of 100,000).

- Multiprocessing Approaches (with MPI):
    - MPI Merge (recursive sorting and aggregation across processes).
    - MPI Partition (recursive partitioning across processes, then independent sort).

In all cases where merge sort and quick sort were parallelised with multiple OpenMP threads, we utilised a hybrid cutoff algorithm that switches to a serial implementation when array sizes are below 100. This method was chosen due to our results in Project 1 that confirmed that this hybrid approach was significantly faster as it minimised overhead in instantiating new threads for trivially small array sizes. In both serial and parallel cases we ran the same verification function to check if the full array was ordered to ensure that the sorted results of the serial and parallel algorithms were the same.

MPI Parallelisation Approaches

Our approach to augment our sorting algorithms were to either partition or aggregate a collection of arrays spread across multiple processes with MPI. Both approaches assumed that the number of available processes was strictly a power of two and that the array chunks had already been loaded into memory for each process (i.e. with MPI parallel IO). Note that in both cases we have handled the case where processes have uneven array chunk sizes.

MPI Merge

This function was based on the merge function used in merge sort. In this approach we aggregated chunked arrays from a pair of processes onto one parent process, and performed a merge sort on the received array, recursively continuing until all array chunks were aggregated onto a final master process that was the same size as the total array size. Processes also communicate their array sizes prior to sending data in order to handle cases where arrays are not perfectly divisible by the world size. Note that in the initial stage where array chunks are loaded onto each process separately, we can use any sorting algorithm before the MPI Merge process begins, but the MPI-based merge sort is required as we step through the algorithm.

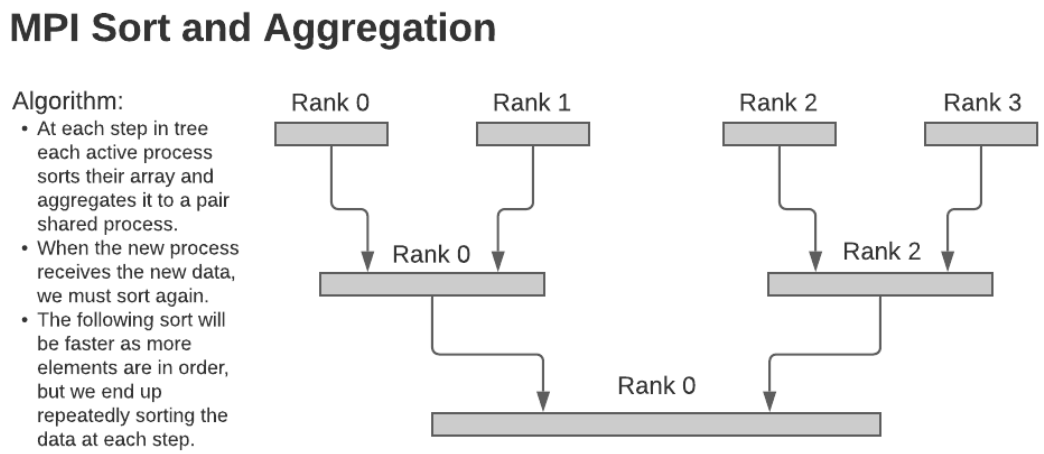

Figure 1. A sorting approach that leverages MPI based on merging sorted arrays and aggregating the results recursively.

This approach introduces obvious inefficiencies as we are re-sorting the array after each aggregation step. Even if the original source array is sorted on the source process, when it is aggregated with another sorted array we obviously cannot guarantee that the concatenation of these two arrays would be sorted. While we should expect that sorting the combination of two sorted subarrays should be faster than if they were entirely unsorted, we wanted to investigate whether or not an alternative approach could provide further optimizations.

MPI Partition

We wanted to address the inefficiencies of re-sorting the array after every step of aggregation (as in mpi_merge above) by implementing a new approach, based on the partition function used in quick sort. The goal was to achieve the following partition across processes: if we define R_i as the rank with id i, then if all values in R_{i-1} are less than the values in R_i and all values in R_{i+1} are greater than those in R_i, then we could independently sort each process with any sorting algorithm and concatenate the arrays in order of their rank, thus obtaining a fully sorted array shared across each process.

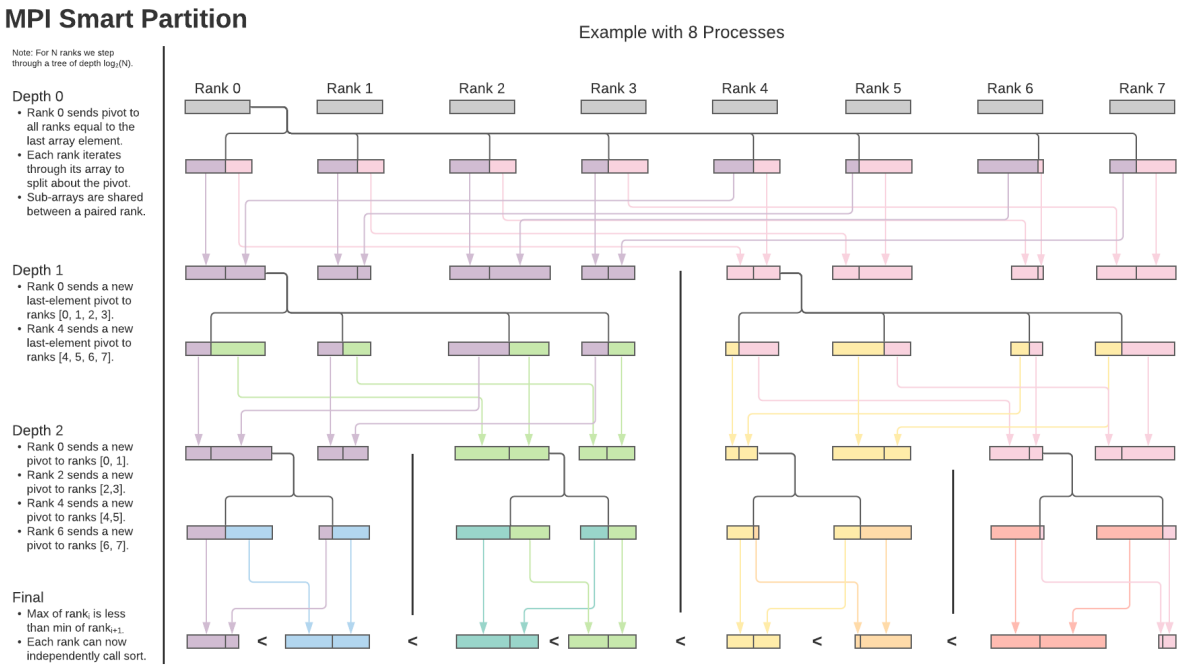

Figure 2. Partitioning process based on quick sort using MPI to organise arrays such that they would be sorted across ranks, based on Miller (2014). This avoids re-sorting data after each step although sizes differ between processes and depth.

There was a considerable amount of MPI communication used to implement this approach. Point-to-point communication was used in order to determine which group of processes would need to have a pivot shared from some source destination (i.e. rank 0 at depth 0, rank 0 and 4 at depth 1, and so on as in Figure 2). Additionally, a key difficulty in implementing this approach was dynamically allocating memory between each step of the tree as the choice of pivot will change how big the sub-arrays will be. Our solution was to use non-blocking communication to first send the size of the array, allocate the memory on the destination process, transfer the array data, and finally deallocate the old array memory. Barriers were required to prevent race conditions where large differences in

array sizes meant that one process might deallocate an array still in use by a partner array, and so we had to account for that problem in our code as well. To confirm the final array was sorted, we utilised MPI_Gatherv to aggregate variable size chunks onto a main process where we ran our sorted tests.

Parallel IO

In addition to sorting algorithms that leveraged multi-processing with MPI, we also used parallel IO. For all sorting algorithms that used MPI we read in array data in chunks to each process before running either mpi_merge or mpi_partition experiments. We also wrote tests that compared the performance of serial and parallel read and write for a variety of array sizes, shown in Figure 3 and 4.

Environment

All code was written in C with OpenMP and MPI and run with the following specifications:

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

Code was compiled (mpicc) and run (mpiexec) with n=4 processes with the commands:
Read and Write (IO) Experiments:

- mpicc -o experiments_io experiments_io.c random_array.c mpi_utils.c -lm
- mpiexec -n 4 experiments_io

Sorting Algorithm Experiments:

- mpicc -fopenmp -o main main.c experiments_sort.c random_array.c mpi_utils.c merge_sort.c quick_sort.c enumeration_sort.c -lm
- mpiexec -n 4 main

Results

Read and Write (IO) Experiments

io	read						write					
world_size	1	2	4	8	16	24	1	2	4	8	16	24
size												
1000	0.0000	0.0002	0.0004	0.0006	0.0013	0.1044	0.0001	0.0013	0.0018	0.0032	0.0032	0.0832
10000	0.0000	0.0002	0.0003	0.0005	0.0009	0.0505	0.0001	0.0002	0.0004	0.0006	0.0010	0.0587
100000	0.0003	0.0004	0.0004	0.0006	0.0010	0.0459	0.0005	0.0005	0.0009	0.0009	0.0018	0.0612
1000000	0.0036	0.0025	0.0015	0.0014	0.0016	0.0609	0.0075	0.0035	0.0051	0.0048	0.0053	0.0895
10000000	0.0378	0.0206	0.0127	0.0098	0.0080	0.0442	0.0860	0.0303	0.0293	0.0355	0.0374	0.1366
100000000	0.3518	0.2037	0.1122	0.0843	0.0694	0.1439	0.9280	0.2622	0.2761	0.2923	0.3130	0.4672
1000000000	3.0032	1.7220	0.9151	0.6454	0.5662	0.5831	7.2551	2.3322	2.5684	2.5032	2.8186	3.2280

Figure 3. Tabular data for IO performance for the generation and writing of a random double array, and the reading of the same array from file, across a number of processes.

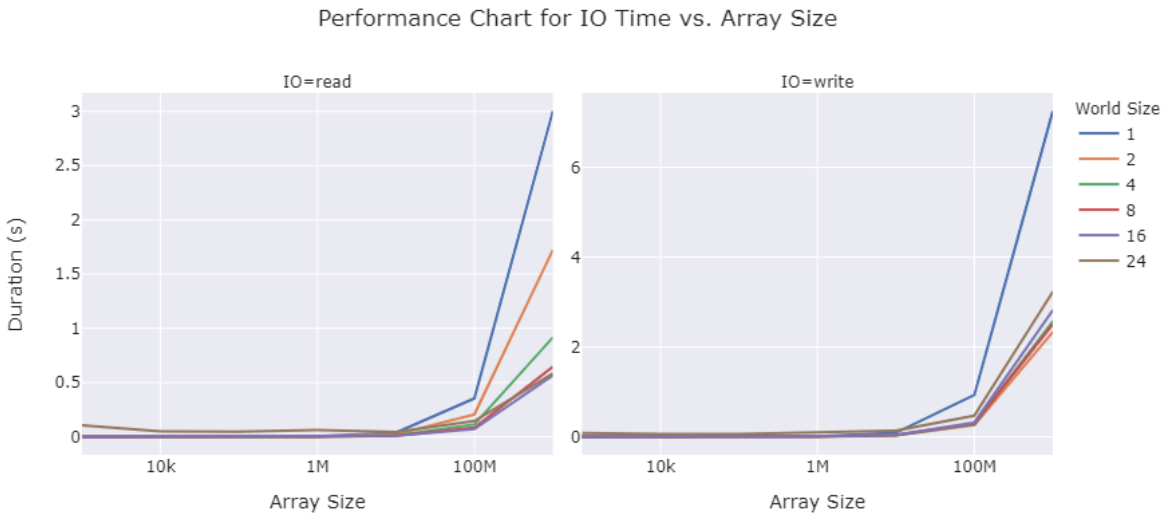

Figure 4. Read and write times for IO experiments on random double arrays comparing serial IO against MPI parallel IO.

Sorting Algorithm Experiments

	approach	serial		
	sort	enumeration_sort	merge_sort	quick_sort
world_size	threads			
1	1	24.007136	0.013737	0.015833
	2	24.378864	0.011116	0.018532
	4	12.296599	0.008322	0.018646
	8	6.217998	0.006515	0.018815

	approach	merge			partition		
	sort	enumeration_sort	merge_sort	quick_sort	enumeration_sort	merge_sort	quick_sort
world_size	threads						
2	1	6.038721	0.008327	0.008935	8.657768	0.009071	0.010455
	2	6.140236	0.010269	0.009826	8.810228	0.011311	0.011801
	4	3.110686	0.010378	0.009681	4.467161	0.011289	0.011717
	8	1.842067	0.010597	0.009894	2.402259	0.011381	0.011767
4	1	1.593489	0.005535	0.005746	5.442620	0.007574	0.008274
	2	1.563956	0.005999	0.005888	5.272327	0.008991	0.009124
	4	0.920553	0.006152	0.005737	2.712954	0.009223	0.009095
	8	0.743516	0.007053	0.006493	1.540687	0.009313	0.009342
8	1	0.436472	0.004981	0.004565	1.974359	0.005078	0.005223
	2	0.481349	0.004380	0.003903	1.846041	0.005984	0.005944
	4	0.383676	0.004912	0.004600	1.040454	0.006668	0.006798
	8	0.364904	0.004751	0.004575	0.743908	0.006332	0.006280
16	1	0.122133	0.004434	0.004267	4.101196	0.008032	0.007845
	2	0.218982	0.004546	0.004389	4.178977	0.010056	0.009034
	4	0.201651	0.006806	0.003960	2.191554	0.010396	0.009136
	8	0.188221	0.005598	0.003904	1.286648	0.009622	0.008802

Figure 5. Results tables for the single process (no MPI) and multi-processing implementations (MPI) for an array size of 100,000, including both mpi_merge and mpi_partition approaches for all sorting algorithms with and without OpenMP.

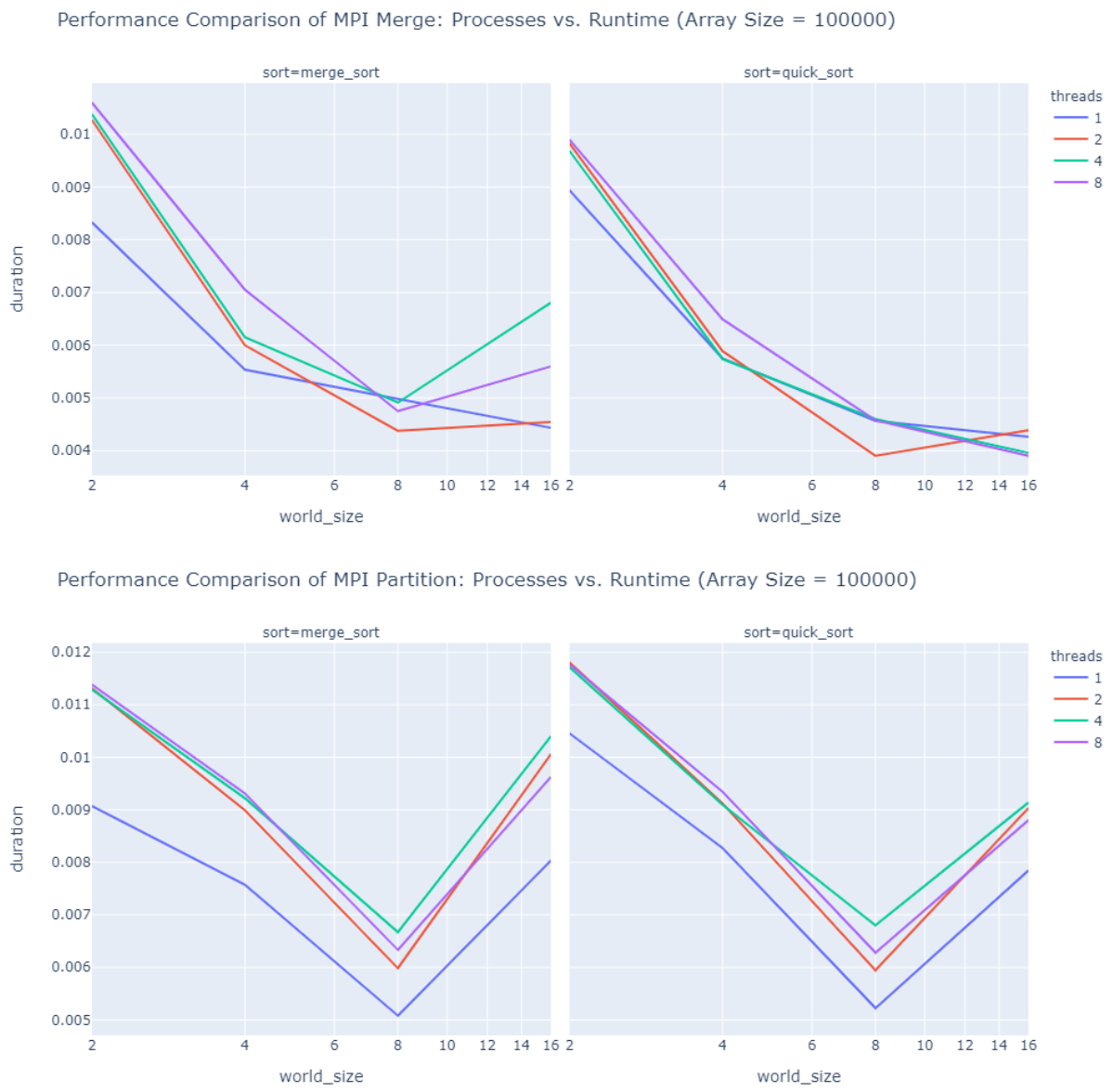

Figure 6. Performance results for number of processes vs. duration for both MPI Merge and MPI Partition approaches for a fixed array size of 100,000 (excluding enumeration sort). We exclude enumeration sort from these charts as it is orders of magnitude slower (see Figure 5). We can see there is an optimal number of 8 processes for the MPI Partition approach, with quick sort performing slightly better. MPI Merge does not decrease in performance for 16 processes, although there is slight uptick in performance for the merge sort implementation with 16 processes.

Discussion

As expected, we generally found that increasing the number of processes and threads yielded performance improvements across the board. However, we found that this speed tapered off if we increased this number too high (i.e. 16 processes with multiple threads). Given our machine had only 24 logical cores, an excessive number of threads or processes that were collectively greater than our hardware limitations did not yield significant improvements, as expected. Additionally, there are trade-offs in terms of communication cost when more data must be transferred between processes. Both quick sort and merge sort were optimal with 8 processes for both MPI Merge and MPI Partition.

Enumeration sort was fastest with the maximum number of processes for MPI Merge, but the MPI Partition approach also had the best results at 8 processes.

Notably, we found that the MPI partition approach did not outperform the MPI Merge approach. Even though MPI Merge is performing a full sort at each aggregation (merge) step of the tree, this sorting process is not as slow compared to sorting an unsorted array as sub-arrays are already partially sorted. Conversely, while the MPI Partition approach only performs a full sort once after arrays have been partitioned in order across arrays, it is still, in general, slower. We believe that one of the major contributions to the slower speed in the MPI Partition case is because sub-array sizes are highly variable. In extreme cases we sometimes observed that pivots could be selected such that one process has an array of size 5000, while others would have orders of magnitude less elements (i.e. less than 10). This variability in chunk size between processes means that we have a fairly large bottleneck in our algorithm which may result in idle processes.

One could mitigate the variability in the array size across processes by picking a pivot element in a more intelligent manner. For example the median value (across all arrays) could be used as the pivot which might help to reduce variability and balance the load across all processes. A notable difference between our two MPI approaches is that our MPI Partition implementation completes with sorted arrays on each process. This means that we can directly leverage MPI parallel IO whereas the MPI Merge approach will either need to use serial IO to write the aggregated array to disk or scatter the data back to all processes before it can use parallel IO, introducing considerably more overhead. This is one major advantage we would expect the MPI Partition approach to have in a distributed setting although we did not necessarily see major performance improvements in our simple case. According to experiments, if we were to write 100,000,000 doubles then our MPI Partition approach with parallel IO would gain an extra five seconds over MPI Merge when writing to disk is required.

Summary

We implemented two main multiprocessing approaches using MPI, as well as MPI parallel IO, to achieve performance improvements in a number of (multi-threaded) sorting algorithms - namely quick sort, merge sort and enumeration sort. Our implementation of MPI Partition was an attempt to improve a more naive implementation for MPI Merge, but we found that due to array size variability and potentially communication cost this MPI Partition approach did not yield the performance improvement we would expect in practice. A better pivot and partitioning process may optimise the MPI Partition approach, and this approach may have benefits when directly to disk across nodes with MPI parallel IO, whereas MPI Merge completes sorting with all data on one single process.

We found that a balance of processes and threads is necessary to achieve optimal performance as one has to consider both hardware constraints (i.e. number of cores) and communication overhead. In general, combining multi-threading (OpenMP) across processes (MPI) should typically produce the performance increase we are looking for when running in distributed environments rather than only on a single machine. These two points highlight the importance considering hardware constraints and the necessity of tailoring optimisations to the circumstances of both the algorithm and the infrastructure at hand for any high performance computing problem.

References

Miller, R. 2014, Implementation of Parallel Quick Sort using MPI, University at Buffalo Department of Computer Science and Engineering, CSE 633: Parallel Algorithms, viewed 21 October 2021 <https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ramkumar-Spring-2014-CSE633.pdf>.

Appendix

Pseudocode

Pseudo-code is provided for relevant algorithms below.

Quick Sort

```
def partition(array, low, high):
  // select pivot value (i.e. last element of array)
  pivot = array[high]

  // let i be number of elements less than pivot
  // we will iterate through the array and count
  i = -1
  for (j = low; j < high; j++)
    if array[j] < pivot:
      i++  // increment i to add to count
      swap(array[i], array[j])  // swap position

  return i + 1 // the starting index of the upper sub-array

def quick_sort(array, low, high, cutoff=100):
  // with OpenMP tasking (hybrid - thread continues if array is below cutoff)
  partition_idx = partition(array, low, high)

  #pragma omp task shared(arr) firstprivate(low, pi) if (high - low > cutoff)
  quick_sort(array, low, partition_idx - 1)

  #pragma omp task shared(arr) firstprivate(low, pi) if (high - low > cutoff)
  quick_sort(array, partition_idx + 1, high)
```

MPI Partition

This algorithm requires that the number of processes to be a power of 2 (based on Miller (2014)).

1. Group all ranks and select a (last element) pivot from the first rank in the group
2. Send this pivot value to each rank in the group.
3. Swap values in the array of each rank until we have two sub-arrays split by the pivot.

4. Send lower half subarrays to arrays in the lower half of the group, and send upper half subarrays to arrays in the upper half of the group.
5. Split the group of processes into two (lower and upper).
6. Repeat steps 1-5 until the group size is equal 1, at which point all of the values in rank i should have values greater than rank i-1, and less than rank i+1.
7. Call a sorting algorithm on each rank independently.
8. Array chunks across processes can either be gathered (MPI_Gatherv as we cannot ensure that each rank now has equal sized arrays) or written directly to disk in parallel chunks.

MPI Merge and Merge Sort

The following merge algorithm is implemented with two versions:

- Version 1: two half arrays of size n, and a full array of size 2n;
- Version 2: a full array size of 2n and a temp array of size 2n.

While tasking and sections have been implemented for version 1, we show version 2 as this was used for the MPI merge approach and we previously showed version 1 in Project 1's pseudocode.

The MPI Merge approach sends half2 from rank2 to rank1 with half1, and then merges the two results using merge sort on rank1. This process is recursive until the final rank has the full array.

```
def merge(half1, half2, array, n)
  // iterate through both half1 and half2
  // adding the smaller element to array

  i, j, k = 0

  // both subarrays are not empty
  while (j < n && i < n)
    if half1[i] < half2[j]
      array[k] = half1[i]
      i++
    else
      array[k] = half2[j]
      j++
    k++

  // one subarray has been emptied

  // if both conditions above not satisfied
  // then one subarray has been emptied
  // loop through remaining (sorted) elements
  while (i < n)
    array[k] = half1[i];
    i++
    k++

  while (j < n)
```

```
    array[n] = half2[j];
    j++
    n++

  return array
```

Enumeration Sort

```
def enumeration_sort(array, temp, size)
  // initialize enumerated array of ranks with zeros
  ranks[size] = 0;

  #pragma omp parallel for shared(array, temp)
  for (int i = 0; i < size; i++)
    for (int j = i+1; j < size; j++)
      if (array[i] > array[j])
        ranks[i]++
      else
        ranks[j]++

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

  return temp  // or copy temp to array
```

Auxiliary Data Tables

	approach	serial		
	sort	enumeration_sort	merge_sort	quick_sort
threads	size			
1	10000	0.2368	0.0013	0.0015
	100000	24.0071	0.0137	0.0158
	1000000	nan	0.1674	0.2033
	10000000	nan	1.9961	2.2887
	100000000	nan	22.7555	26.6629
2	10000	0.2453	0.0009	0.0015
	100000	24.3789	0.0111	0.0185
	1000000	nan	0.1216	0.2215
	10000000	nan	1.2654	2.4834
	100000000	nan	13.8979	28.5926
4	10000	0.1249	0.0009	0.0016
	100000	12.2966	0.0083	0.0186
	1000000	nan	0.0899	0.2200
	10000000	nan	0.8333	2.4976
	100000000	nan	9.4784	28.6804
8	10000	0.0693	0.0006	0.0015
	100000	6.2180	0.0065	0.0188
	1000000	nan	0.0757	0.2207
	10000000	nan	0.6009	2.5013
	100000000	nan	5.9809	28.8251

Figure 7. Serial performance results averaged over three trials for all sorting algorithms.

		approach	merge			partition		
		sort	enumeration_sort	merge_sort	quick_sort	enumeration_sort	merge_sort	quick_sort
world_size	threads	size						
2	1	10000	0.0600	0.0009	0.0009	0.0948	0.0011	0.0011
		100000	6.0387	0.0083	0.0089	8.6578	0.0091	0.0105
		1000000	nan	0.0930	0.1114	nan	0.1681	0.1981
		10000000	nan	1.0982	1.2184	nan	1.3993	1.5689
		100000000	nan	12.4001	14.1110	nan	12.7311	14.3739
	2	10000	0.0664	0.0012	0.0010	0.1051	0.0013	0.0013
		100000	6.1402	0.0103	0.0098	8.8102	0.0113	0.0118
		1000000	nan	0.1090	0.1201	nan	0.2033	0.2152
		10000000	nan	1.2773	1.3176	nan	1.6306	1.7104
		100000000	nan	14.2537	15.1680	nan	14.7826	15.4453
	4	10000	0.0352	0.0010	0.0010	0.0526	0.0012	0.0012
		100000	3.1107	0.0104	0.0097	4.4672	0.0113	0.0117
		1000000	nan	0.1097	0.1192	nan	0.2094	0.2182
		10000000	nan	1.2890	1.3228	nan	1.6545	1.7232
		100000000	nan	14.1806	15.0895	nan	14.6569	15.4295
	8	10000	0.0213	0.0011	0.0010	0.0293	0.0012	0.0012
		100000	1.8421	0.0106	0.0099	2.4023	0.0114	0.0118
		1000000	nan	0.1109	0.1188	nan	0.2007	0.2159
		10000000	nan	1.2996	1.3281	nan	1.6394	1.7462
		100000000	nan	14.2542	15.1598	nan	14.7205	15.5002
4	1	10000	0.0220	0.0008	0.0007	0.0412	0.0009	0.0008
		100000	1.5935	0.0055	0.0057	5.4426	0.0076	0.0083
		1000000	nan	0.0592	0.0650	nan	0.1766	0.1994
		10000000	nan	0.6496	0.7002	nan	1.0583	1.2028
		100000000	nan	7.2303	8.0031	nan	11.9595	13.3322
	2	10000	0.0188	0.0008	0.0007	0.0412	0.0010	0.0009
		100000	1.5640	0.0060	0.0059	5.2723	0.0090	0.0091
		1000000	nan	0.0649	0.0681	nan	0.2011	0.2207
		10000000	nan	0.7526	0.7648	nan	1.2403	1.2988
		100000000	nan	8.0879	8.4846	nan	13.7075	14.2998
	4	10000	0.0140	0.0008	0.0007	0.0229	0.0010	0.0009
		100000	0.9206	0.0062	0.0057	2.7130	0.0092	0.0091
		1000000	nan	0.0647	0.0680	nan	0.2065	0.2206
		10000000	nan	0.7530	0.7571	nan	1.2713	1.2999
		100000000	nan	8.0751	8.4566	nan	13.6246	14.2758
	8	10000	0.0122	0.0010	0.0008	0.0142	0.0010	0.0010
		100000	0.7435	0.0071	0.0065	1.5407	0.0093	0.0093
		1000000	nan	0.0651	0.0701	nan	0.2064	0.2278
		10000000	nan	0.7483	0.7560	nan	1.2536	1.2969
		100000000	nan	8.0347	8.4405	nan	13.6131	14.2784

8	1	10000	0.0069	0.0009	0.0008	0.0914	0.0014	0.0014
		100000	0.4365	0.0050	0.0046	1.9744	0.0051	0.0052
		1000000	nan	0.0433	0.0414	nan	0.1053	0.1167
		10000000	nan	0.4510	0.4484	nan	0.7308	0.7885
		100000000	nan	4.6336	4.8485	nan	9.8831	10.8789
	2	10000	0.0073	0.0009	0.0008	0.1063	0.0016	0.0015
		100000	0.4813	0.0044	0.0039	1.8460	0.0060	0.0059
		1000000	nan	0.0442	0.0442	nan	0.1227	0.1275
		10000000	nan	0.4725	0.4809	nan	0.8448	0.8776
		100000000	nan	5.0233	5.1471	nan	11.2764	11.8208
	4	10000	0.0063	0.0009	0.0008	0.0544	0.0015	0.0014
		100000	0.3837	0.0049	0.0046	1.0405	0.0067	0.0068
		1000000	nan	0.0427	0.0441	nan	0.1256	0.1274
		10000000	nan	0.4730	0.4767	nan	0.8342	0.8671
		100000000	nan	5.0062	5.1089	nan	11.2637	11.8275
	8	10000	0.0081	0.0010	0.0008	0.0318	0.0016	0.0015
		100000	0.3649	0.0048	0.0046	0.7439	0.0063	0.0063
		1000000	nan	0.0464	0.0437	nan	0.1222	0.1271
		10000000	nan	0.4638	0.4694	nan	0.8347	0.8674
		100000000	nan	4.9834	5.1253	nan	11.2704	11.8205
16	1	10000	0.0063	0.0012	0.0011	0.0263	0.0014	0.0013
		100000	0.1221	0.0044	0.0043	4.1012	0.0080	0.0078
		1000000	nan	0.0414	0.0370	nan	0.0574	0.0597
		10000000	nan	0.3818	0.3716	nan	0.8214	0.8400
		100000000	nan	3.8109	3.8224	nan	7.0912	7.4150
	2	10000	0.0080	0.0012	0.0011	0.0373	0.0016	0.0015
		100000	0.2190	0.0045	0.0044	4.1790	0.0101	0.0090
		1000000	nan	0.0376	0.0359	nan	0.0608	0.0651
		10000000	nan	0.3912	0.3741	nan	0.9512	0.9276
		100000000	nan	4.1116	4.0235	nan	8.6509	8.2721
	4	10000	0.0074	0.0020	0.0012	0.0219	0.0020	0.0033
		100000	0.2017	0.0068	0.0040	2.1916	0.0104	0.0091
		1000000	nan	0.0350	0.0346	nan	0.0705	0.0670
		10000000	nan	0.3910	0.3703	nan	0.9949	0.9480
		100000000	nan	4.1790	4.0270	nan	8.3350	8.3583
	8	10000	0.0149	0.0023	0.0010	0.0186	0.0024	0.0014
		100000	0.1882	0.0056	0.0039	1.2866	0.0096	0.0088
		1000000	nan	0.0367	0.0352	nan	0.0705	0.0672
		10000000	nan	0.3900	0.3724	nan	1.0173	0.9294
		100000000	nan	4.3853	4.4497	nan	8.1789	8.5381

Figure 8. Full summary of MPI parallel performance results for all sorting algorithms averaged over three trials.