Parallel and Distributed Merge Sort

By: Bowen Liu

Introduction

With data sets becoming larger than ever with the advances of the internet and technology, handling large data sets are also becoming increasingly difficult. One common problem needed to handle large data sets is to sort a random collection of data into an orderly fashion.

Merge sort is often considered one of the fastest sorting algorithms (worst case time complexity of O(nlogn)), and has been widely implemented into many standard libraries. However, most of the standard implementations is serial does not exploit hardware parallelisms offered by the CPUs. As a result, sorting large data sets using serial sorting algorithms would take a significant amount of time. As an experiment, I will implement a parallelized version of mergesort that's capable of running on UMA and NUMA machines using OpenMPI. It would also be interesting to observe the performance trends of a sorting algorithm when parallelized across a single node machine, as well as a multi-node cluster.

Using the Parallel Mergesort Framework

The compiled executable for parallel merge sort requires at least 2 arguments to sort a file. The arguments take form in the following form:

psort input_file output_file <optional args>

The *input_file* and *output_file* arguments are self-explanatory. The user must specify the filenames (including paths if needed) to the input data file, and where the output data file would be. The input and output file can be the same. Currently, there are two optional args implemented.

The **–benchmark** argument will read the specified input data file, sort it, but will not save the sorted data into a file. This argument should only be used if the user wants to know how long the sorting process would take. Note that even though no files are outputted, the user is still required to specify a placeholder for the *output_file* field.

The **–noverify** argument disables verification after all sorting procedures have completed. Disabling verification could save a small amount of execution time if the user is confident that the user-defined functions are correct.

The number of nodes to use is not directly specified to the psort program. Please refer to **mpirun.sh** as an example of how to run the program on a fixed number of processes. The framework is designed to work with any number of processes. If only 1 process is specified, the framework will shortcut to a serial version instead. Furthermore, see **batchrun.sh** as an example of how to run the framework on a cluster with multiple nodes using a batch system (qsub).

Compile.sh is an example of how to compile the source codes together to form an executable needed by mpirun.sh.

Implementation details for my parallelized mergesort will be discussed below.

The Mergesort Algorithm

The mergesort algorithm is fairly simple. Given an array of unsorted elements, the array is recursively broken down into pairs of smaller sublists until there is only 1 element left in both pairs. The algorithm would then reverse the process and "merge" the pairs of sublists together by placing the smallest head elements in the front until both sublists are merged together. Eventually, the merging process would conclude with one big sorted list after numerous merges.

If we consider the data dependencies of mergesort, not a significant amount of dependencies is needed throughout the steps of the sorting. Mergesort works with blocks of data of virtually any size, therefore parallelizing the algorithm should be fairly easy. We can simply chop up the initial unsorted array of data into *p* blocks, where *p* is the number of processors/nodes available in the system. Each of the processes can then locally call mergesort and sort their sublists in parallel. At end of this step, there will be *p* sorted sublists distributed among the processes. The processes can then exchange their sorted sublists among each other for cross-process merging until one node is left with the entire sorted data array.

The code for simple serial mergesort is found in the file **mergesort.c** and **mergesort.h**. My implementation of mergesort is generalized to sort any data type specified by the user, as long as the user specifies the number of bytes needed by each data element, and a comparison function. Essentially, the mergesort function works very similarly to the qsort function in C's standard libraries.

Mandatory User-Definitions

The file **main_p.c** is the main structure and framework for parallelizing mergesort to run across many nodes and processors. This file is assumed to be correctly working and standalone, and should not be altered by the user. Just like the barebone implementation of our mergesort algorithm, the framework allows the user to sort any data type from any file without any restrictions.

In order to accomplish such generalization, the user must implement four functions and two definitions. The definitions of **DATA_TYPE** and **MPI_DATA_TYPE** must be defined in **userdef.h**. The other four functions specified in userdef.h must be implemented in **userdef.c**.

The user must first define **DATA_TYPE**, which is the C data type composes the target data set. All elements in the target dataset must be using this data type.

The user must also define the corresponding MPI data type to the same C data type as mentioned. This is defined as **MPI_DATA_TYPE.** If the user is using a complex data type such as a struct for DATA_TYPE, a custom MPI data type is needed.

The first function needed to be implemented by the user is **userInit()**. The function is optional, but must be present in userdef.c. The user can use this function to create the custom MPI data type needed, or any additional code for initialization needed. The userInit function is called right after MPI initializes. Therefore, the user must node if the initialization code is intended to run on all processes, or only the master.

The function **compFunc** takes the pointers of two elements from the target dataset (of type DATA_TYPE) and compares between them. The function must return -1 if a <b; return 0 if a=b; and return 1 if a>b.

The function **readDataFromFile** is intended to read the raw input file containing the unsorted data set into memory. The name and path of the input file is passed in as *filename*.

The function must first dynamically allocate an array (of type DATA_TYPE) large enough to hold each of the data elements in the input file. The user then needs to implement how to read the raw data from the input file into the structured data array that was allocated. Once all data has been read, structured, and placed into the array, the

function directly returns pointer on the allocated data array. The function is also required to use the output argument *elements_read* to return how many elements are in the data array.

The last function needed from the user is **writeDataToFile**. This function is essentially the opposite of readDataFromFile. The function passes in the pointer of a sorted array of data (of type DATA_TYPE) as *data*, and the number of elements in the data array is passed in as *n*. The user needs to implement how to save each of the structed data elements in the data array into the output file. The output file's filename and path is passed into the function as *filename*.

A sample implementation of userdef.c is included. In this sample dataset, the input file has a "double" written to each line of the file. You can generate a compatible sample data set using **generate.c**. Essentially, the application of the sample userdef.c and dataset sorts a large array of doubles.

Master/Slave Processes

Upon entering the program, MPI is initialized to assign all available processes an ID, or "rank". The process with a rank of 0 is treated as the master process, while all other processes are worker nodes for the master.

Most of the code dedicated to the master process is wrapped in the function masterProcess. The master process first needs to parse the arguments that were passed in from the command line using the function **parseArgs**. The function would then return a *ParsedArgs* struct, which contains a pointer to the data array that has been read in to the file, as well as information about other arguments that were found.

The master informs all slave processes to get ready by broadcasting the number of elements in the data array to all processes. All processes (including the master) will then call the function **calcCountDisp** to create an initial global snapshot of how many elements each node will be receiving, and index numbers of where to break the data array for distribution.

Once the master and slaves have calculated the same global snapshots, the master uses MPI_Scatterv to distribute the blocks of unsorted data to all nodes. All processes will then call the mergesort routine in parallel to locally sort their newly assigned block of unsorted data. At the end of this step, there will be *p* blocks of sorted sublists distributed among the *p* processes.

Phase 2 Merging

The purpose of phase 2 is to take the *p* distributed sorted sublists and merge them into one large sorted list for outputting. This phase is the most complex part of the entire algorithm as it requires non-trivial inter-process communications that cannot be handled by a single MPI function call.

At the start of every phase 2 merging iteration, the function **planMerge** is called. This function updates the global snapshot of which process would have data for the next iteration, as well as instructing the calling process on where to send or receive a sorted sublist to/from. The algorithm used in planMerge can complete the inter-process merging phase in at most $\mathbf{\log_2 P + 1}$ iterations. All communications and computing during each iteration occurs in parallel.

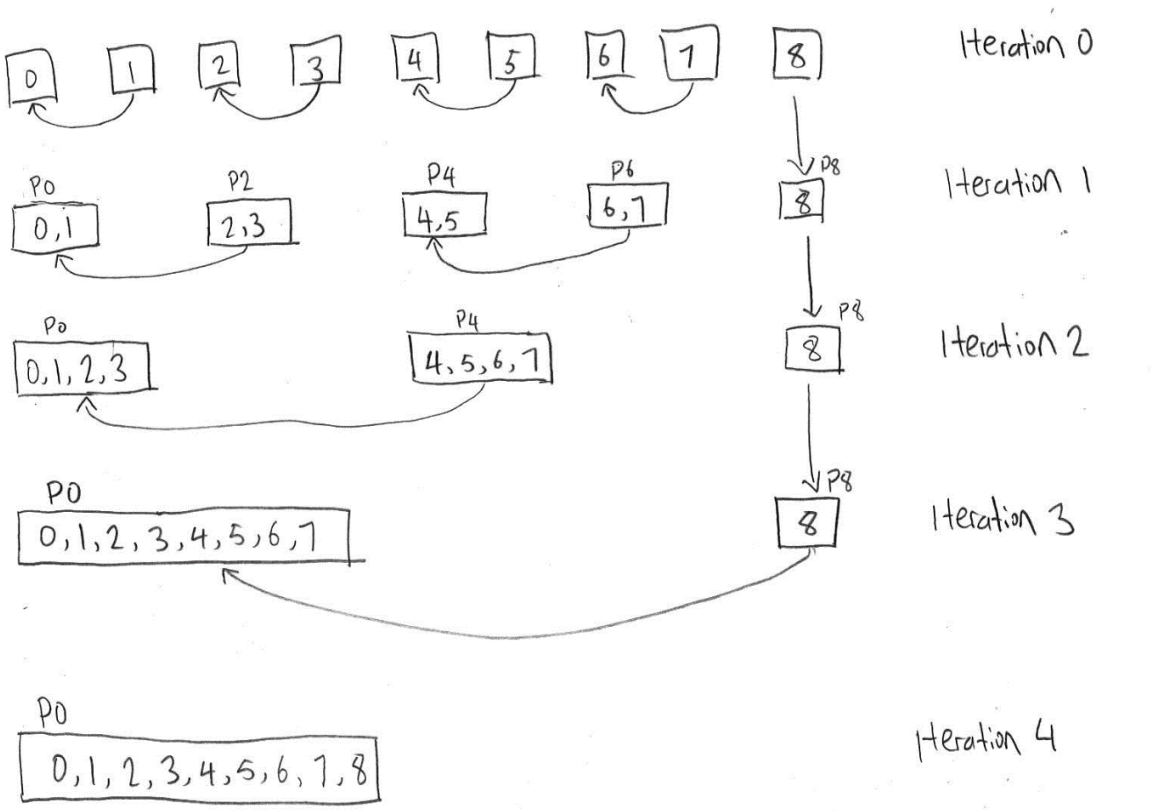
If a process is instructed to receive a sorted sublist, the process will call the recvAndMerge function. The function uses the updated snapshot to know how much data to receive, and using the merge instruction to know who to receive it from. Upon receiving the sorted sublist, the process will directly call the **Merge** function to join its current sorted sublist with the newly received one together.

The master process will only be receiving at any given iterations. All slave processes will be required to send its current sorted sublist to another process at some iteration. When a node has been instructed to send its sorted sublist, it will also use the updated snapshot and merging instruction returned by planMerge to know who to send

the sublist to. Once a slave process has sent out its sorted sublist, it will remain dormant for the rest of the iterations. Planmerge will return “NONE” instructions for any subsequent iterations for a dormant slave process.

The master process will receive a “NONE” instruction once planMerge has declared completion. That is, only the master process has data remaining. At this point, the master process has a complete sorted data list, and it can proceed to verification and output, if enabled.

To understand how the planMerge algorithm works, consider the example below with p=9.



The 9 sorted sublists are distributed among the 9 processes at the starting iteration. The planMerge function recognizes a starting iteration when all processes still have data. During iteration zero, all odd numbered processes (p1, p3, p5, p7) sends their sorted sublist to their neighbor, process i-1.

During any non-starting iterations, every second process encountered with data will be instructed to send its current sorted sublist to its closest process on its “left” that has data. Conversely, corresponding receiving process will also be instructed to receive the sorted sublist from its closest process on its “right” that has data.

If the process *p-1* cannot be paired up during <u>any</u> iteration, it will skip the current iteration. This will proceed until process *p-1* can form a pair at an iteration where an even number of nodes still have data. In the example with p=9, only the last iteration can form complete pairs. However, process *p-1* can often be paired up in earlier iterations with different number of available processes.

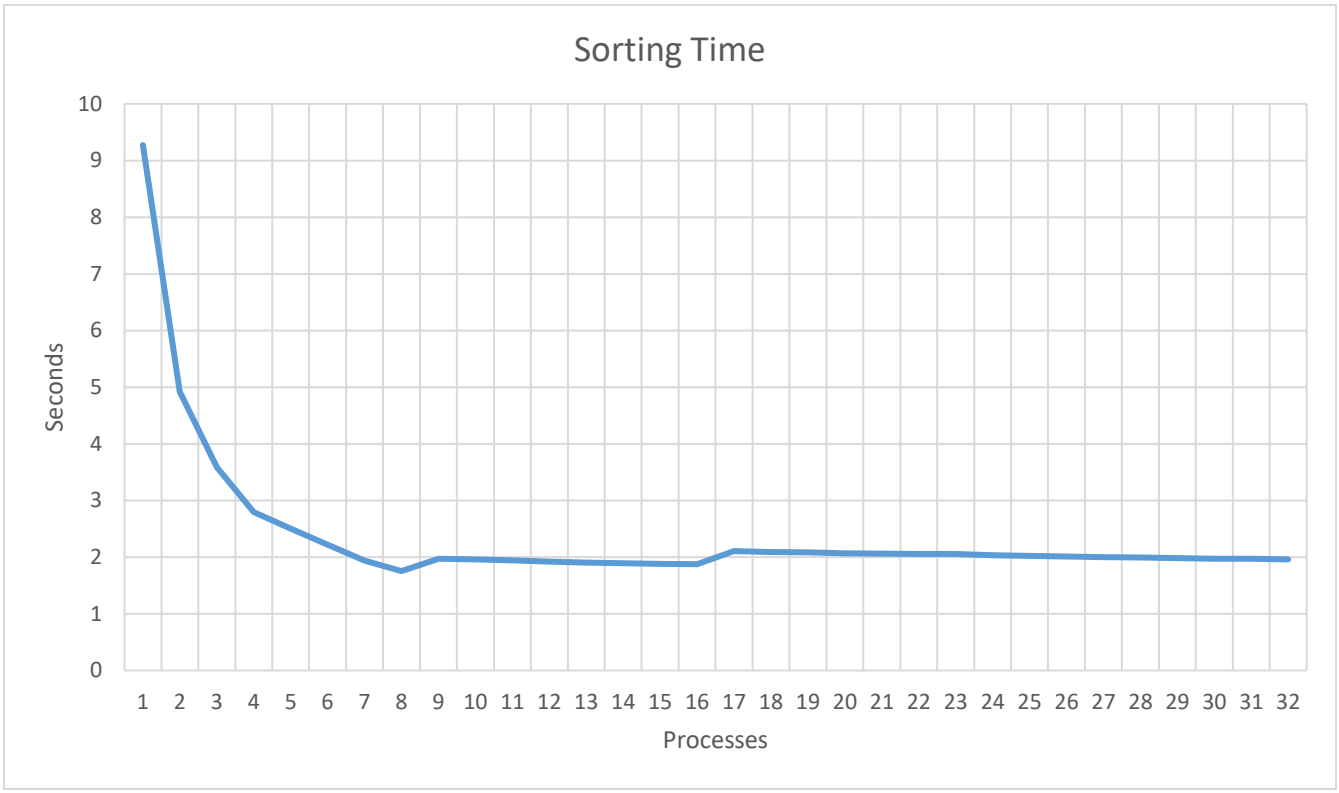
Parallel Performance

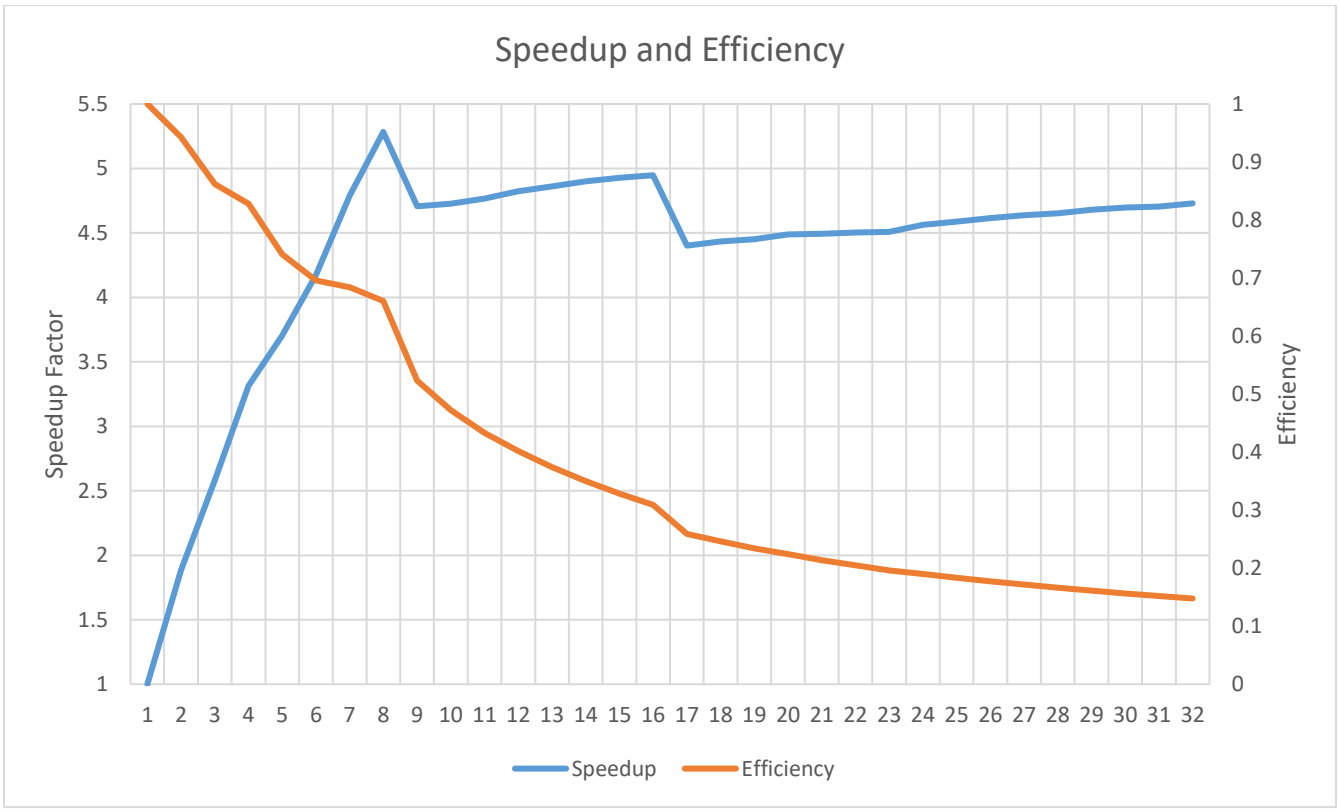
To benchmark the performance of my implementation of the parallelized mergesort, I ran the code on a cluster with 4 nodes. Each of the nodes has two Intel Xeon E5520 running at 2.27GHz, and each CPU has 4 cores. Each node also has 8GB of memory available. In total, the maximum amount of processes I can use is 32. The sample dataset was a file of 10,000,000 doubles generated using generate.c. The provided sample userdef.c and userdef.h are used with the main parallel framework.

Below is a table showing the time needed to sort the 10,000,000 doubles using different number of processes. The table also provides the parallel speedup factor (Speedup = $T_{serial}/T_{parallel}$), as well as the corresponding multicore efficiency (Efficiency = Speedup/p).

P	Sorting Time	Speedup	Efficiency
1	9.272932	1	1
2	4.917419	1.88573	0.94287
3	3.587708	2.58464	0.86155
4	2.798644	3.31337	0.82834
5	2.503733	3.70364	0.74073
6	2.221237	4.17467	0.69578
7	1.936385	4.78879	0.68411
8	1.755098	5.28343	0.66043
9	1.97025	4.70647	0.52294
10	1.961813	4.72672	0.47267
11	1.945961	4.76522	0.4332
12	1.922333	4.82379	0.40198
13	1.907294	4.86183	0.37399
14	1.891859	4.90149	0.35011
15	1.881377	4.9288	0.32859
16	1.873824	4.94867	0.30929
17	2.106373	4.40232	0.25896
18	2.091444	4.43375	0.24632
19	2.083559	4.45053	0.23424
20	2.065742	4.48891	0.22445
21	2.06397	4.49276	0.21394
22	2.058807	4.50403	0.20473
23	2.05719	4.50757	0.19598
24	2.032521	4.56228	0.1901
25	2.021786	4.58651	0.18346
26	2.009281	4.61505	0.1775
27	1.999459	4.63772	0.17177
28	1.993059	4.65261	0.16616
29	1.981827	4.67898	0.16134
30	1.973899	4.69777	0.15659
31	1.971066	4.70453	0.15176
32	1.961198	4.7282	0.14776

To help visualizing the collected data, I have plotted the data above onto graphs shown below.





We can observe a steep downwards trend in sorting time and a steep inverse increase in speedup for the first 8 processes used. However, as soon as more processes are introduced, the growing trend of speedup is interrupted with a sudden decrease, and continues to increase with a much slower rate of change. Similarly, the sorting time increased substantially. Although decreasing steadily with more processes, it fails to reach a lower sorting time than achieved by 8 processes.

The reason for this change in trend is due to introduction of inter-node communications. Because a single node has up to 8 cores available, the first 8 processes do not need to communicate with anyone outside the physical machine. By using 9-16 processes, the program now has to communicate between two nodes over the cluster's external interconnect to perform phase 2 merging. Similar trends can be observed when processes 17-24 are added, as the program now communicates between 3 nodes. However, when the 4th node is used for processes 25-32, we did not see a sudden drop in speedup. Speedup is slowly increasing at a steady rate during this interval and sorting time continues to drop slowly.

We can also observe that efficiency for the overall parallelized mergesort does not scale too well either. Within the first 8 processes used, efficiency is between 98% down to 70%. The efficiency plummets drastically when inter-node communication is needed, since the decreasing trend of the sorting time diminishes and converges to around 2 seconds.

Conclusion

By analyzing the graphs in the Parallel Performance section, we can conclude mergesort does not scale well across a network of clusters. This is because phase 2 merging is very heavily on inter-process communications, even though this step only requires roughly $\log_2 P$ iterations to complete. A steep increase in speedup and a steep decrease in sorting time can be observed when mergesort is parallelized across a single multicore machine. This trend diminishes immediately as soon as more nodes are added, and both speedup and sorting time starts to converge quickly. If a large amount of big data sorting is needed at a facility, a powerful single node UMA machine (with a large sum of cores available) should be considered for the task instead.