Eugene Davis
CPE 435
Project
Report

Implementation

Random Number Generation

Random number generation is performed by the standalone executable, random_generator. It takes two arguments, the first is the number of ints to generate, the second is a file to save the ints to. It uses the C++ pseudo-random number generator.

Serial

The serial implementation is relatively straightforward. It reads in a random data file (specified as a command line argument), and recursively applies merge sort to it. Once done, it will write out the sorted file (the original name with "sorted_" prepended), assuming that the outputFileOptions preprocessor directive was defined at compile time (otherwise there is no output).

Parallel/Distributed

The basic architecture of the approach is a client-server approach. Though somewhat unusual, a push model was employed, in which the server connected to the clients, selecting them via a configuration file (ClientsList.conf). The push model was selected over the more typical pull model since it allowed easy configuration management when used in concert with a deployment script – as then only a single configuration file was needed both to deploy the clients, and to provide to the server to identify the clients. This had the downside of requiring all of the clients to be online before the server can run, but given the software is specialized this downside was outweighed by the ease of deployment.

Client

There were two major aspects to the client, the networking that allowed it to send and receive data to and from the server, and the merge sort that is parallelized via merge sort.

Networking

The client begins by getting the hostname, and setting up and binding the socket to a port (68015). Once bound, it listens on the socket, and waits until the server has made its connection. After the server has connected, the client will begin listening.

It first expects a single integer, which specifies the size of the data set that the client is to receive (in numbers of integers). With this data, it initializes an array to store the received data in, and begins receiving the integer data the server sends to it, until it reaches the amount that was specified by the server at the start of the communication.

With the data all received, it will then perform the merge sort (described later). Once the results of the merge sort have been received, it will send the data back. Since the server already knows the size of the data, it will simply send the data, rather than echoing back the size to the server. Once all the data is sent, it closes the socket.

Merge Sort

The merge sort was a typical, recursive merge sort modified to use pthreads. As is typical, it had a function to partition the data into ever smaller chunks, until the chunk size is one, then merge the chunks back together. The twist for the merge sort in the distributed client is that it used pthreads, which required logic to decide when to launch a pthread.

Since the requirements stated that the clients would run on machines with eight cores, the client only will have eight active threads at maximum, although fourteen threads are launched to get to that number.

To get eight active threads, the main merge sort function, mergeSplit, was modified to have an additional parameter to track its current level of recursion. The calling code started this at 0, and each level would increment it. Since, by the third level of recursion, it there would be a total of eight function calls, each level of recursion up-to-and-including the third level spawn each of the child calls in a new thread. Each of these threads will then join on their children, effectively pausing them until their children complete – meaning that by the time the third level of recursion hits there will be eight active threads. After the third level of recursion, the functions are no longer spawned as threads, and the merge sort returns to its normal behaviors.

Server

The server handles reading in the data file of random integers, and communicating with the clients to give them their portion of the data to sort. It roughly divides into three major areas, the file input and output, which is almost identical to the serial version and thus will not be discussed, the distribution and receiving of data from clients, and merging the data.

Client Communication

Having read in the data file, the server will next look at its configuration file, ClientsList.conf. It reads in up to twenty clients. It ignores any clients after twenty, and errors out if it has less than twenty.

Next it partitions the data into twenty segments, one for each client, and launches a separate pthread for each client, passing along a pointer to the data, and indicating the start and end point of the client's partition, as well as the client's hostname.

The thread for each client first attempts to connect to the client. Due to connectivity issues in the Linux lab, it will attempt to connect up to five times, waiting ten milliseconds between each attempt. Once it has successfully connected, it will start transmitting to the client, first sending the size of the data, then sending the data itself.

Once the data is all sent, the thread swaps into receiving mode, where it will read all of the data that the client sends back, and place it in it's portion of the data array. Once this is done, it will close the socket, and the thread exits.

Merging

The merging is straightforward. It starts with the first two partitions, and merges them, then merges the resulting array with the next partition, and so on until all the partitions have been merged. Once it is done, the resulting file is then written out (assuming the appropriate preprocessor define has been made).

Results

The performance of the serial and parallel/distributed merge sort programs were measured in the Linux lab. The data set sizes were selected by ranging from ten million to one hundred forty million in steps of ten. At each size, the programs were both run five times, each time with a freshly generated set of data. This section will be divided into the raw data, with a scatter plot graph, and a graph of the averages for each data size, with a line graph.

The time data was collected using GNU's time program (not the built-in BASH utility). This provided an advantage over employing a timer in the program, since it allowed collection of timing data for the entire process, including initialization and launching of the program, yielding better real world results. This does mean that delays caused by the network (such as waiting to connect to a client) adversely affect the times for the parallelized version.

Eugene Davis
CPE 435
Project
Report

Raw Data

Number of Ints	Time Parallel	Time Serial
50000000	14.702	14.243
50000000	19.25	19.683
50000000	15.361	14.215
50000000	19.227	19.724
50000000	19.311	19.68
60000000	21.749	23.174
60000000	22.642	19.642
60000000	22.511	23.755
60000000	20.301	18.909
60000000	20.146	16.553
70000000	26.395	20.323
70000000	21.706	22.752
70000000	24.44	21.355
70000000	21.445	26.047
70000000	23.512	27.84
80000000	28.54	25.23
80000000	28.23	31.77
80000000	27.21	26.44
80000000	29.82	29.92
80000000	25.07	22.4
90000000	34.43	27.86
90000000	34.54	36.05
90000000	34.34	35.94
90000000	34.33	35.67
90000000	34.24	36
100000000	31.08	28.22
100000000	29.08	28.48
100000000	37.28	35.11
100000000	31.51	40.01
100000000	38.29	36.84
110000000	65.41	40.69
110000000	37.08	44.14
110000000	41.35	31.24
110000000	34.21	44.14
110000000	36.58	36.62
120000000	37.36	44.48
120000000	44.66	37.65
120000000	36.71	48.26
120000000	45.82	48.29
120000000	45.6	48.18
130000000	38.67	52.38
130000000	37.9	40.6
130000000	48.45	41.07
130000000	48.73	38.46
130000000	48.8	38.05
140000000	47.68	41.88
140000000	50.39	40.2
140000000	52.15	56.54
140000000	53.36	56.63
140000000	53.3	39.63

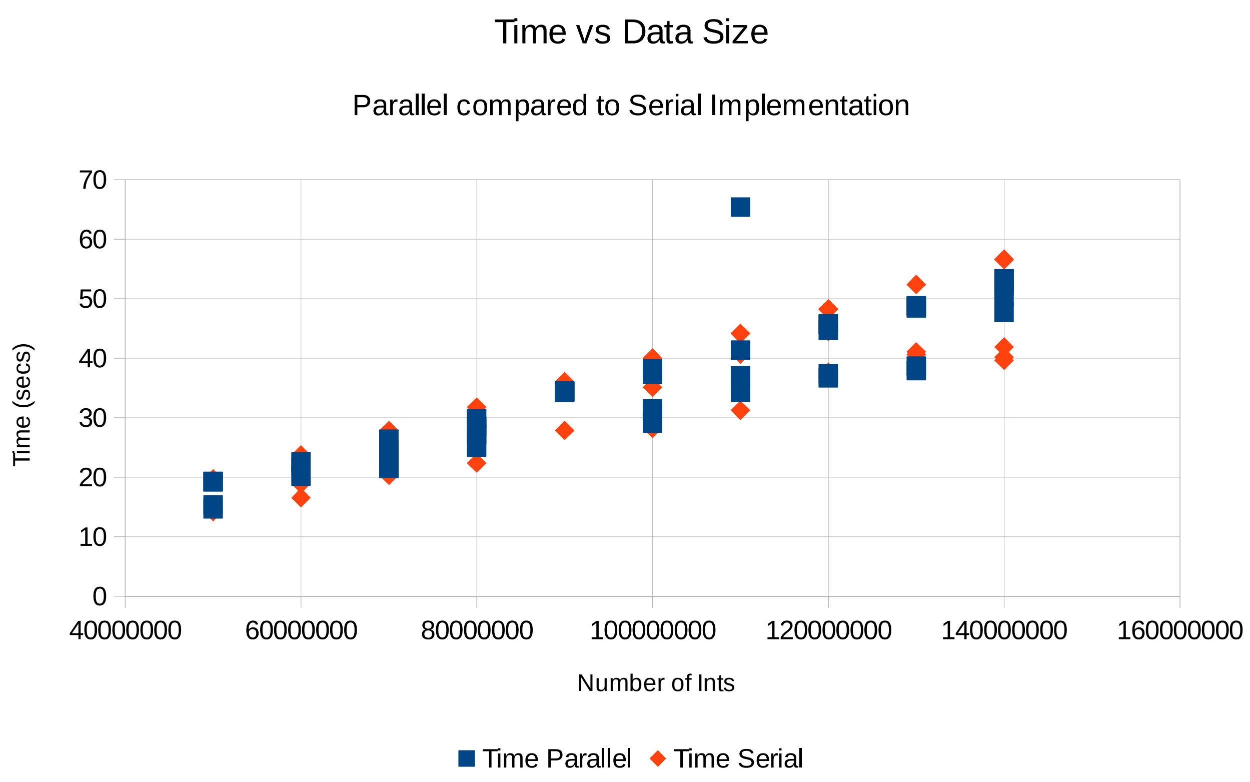

Figure 1: Scatter Plot of the Raw Data

Eugene Davis
CPE 435
Project
Report

Averaged Results

Number of Ints	Average Time Parallel	Average Time Serial
50000000	17.5702	17.509
60000000	21.4698	20.4066
70000000	23.4996	23.6634
80000000	27.774	27.152
90000000	25.2334	26.4484
100000000	33.448	33.732
110000000	42.926	39.366
120000000	42.03	45.372
130000000	44.51	42.112
140000000	51.376	46.976

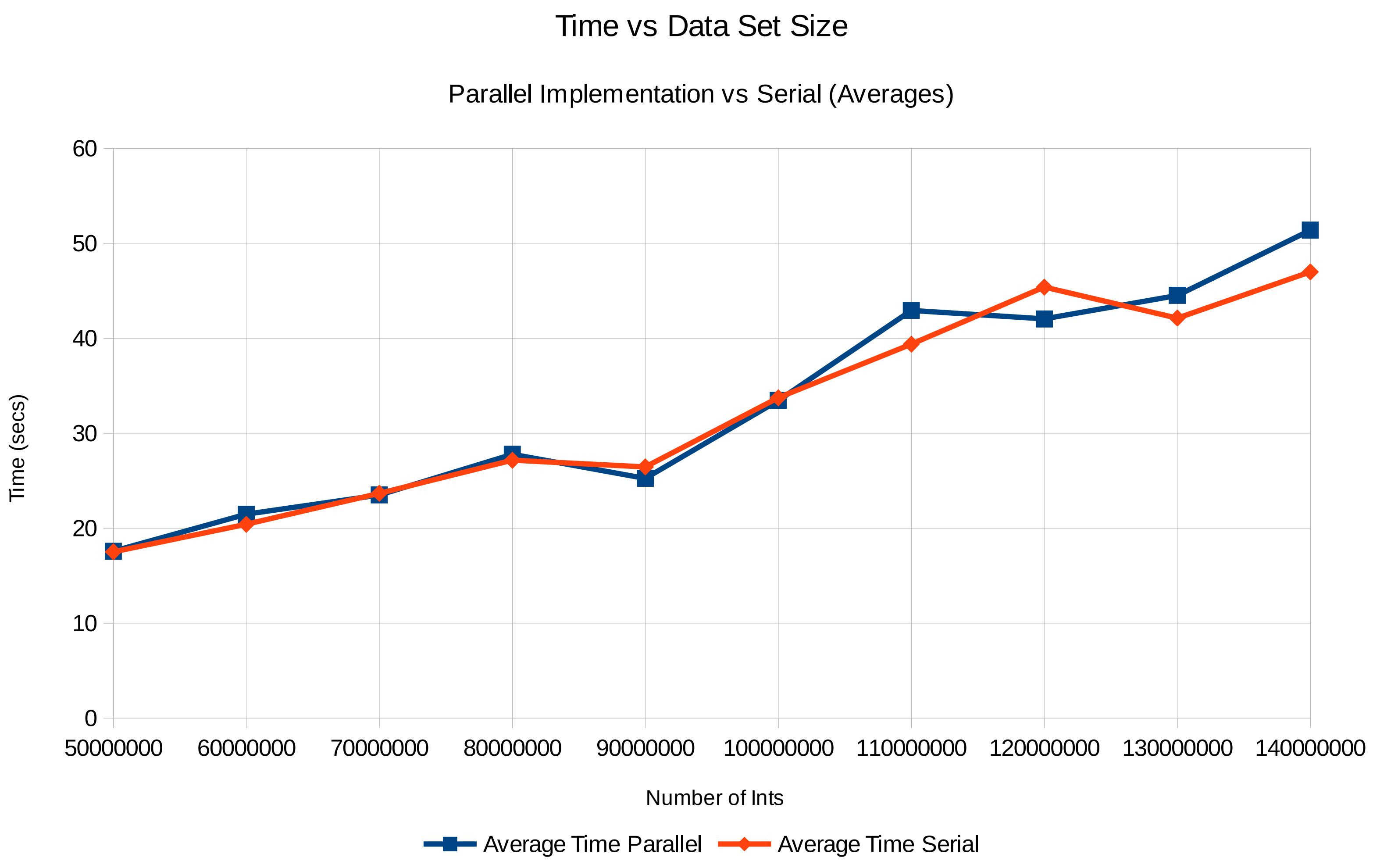

Figure 2: Line Plot of Averaged Data

Comments

The parallel/distributed version did not generally perform much, if at all, better than the serial version when measured just at pure speed. This is likely due to the reality that both the network and the individual machines were under load, potentially explaining the somewhat erratic performance even in the averaged times.

What is worth noting, however, is that with the exception of a single outlier at the eleven million mark (an outlier likely due to a spike in the load on a single machine), the parallel version tends to be significantly more consistent in the time it takes to complete the sort. Although for the data sets being sorted here, this consistency is of relatively little value, if the data sizes were in the billions, or even trillions, the improvement the parallel version would have over the worst cases for the serial version might be significant to warrant employing it.

There are certain optimizations that also could have been performed to improve the performance. The most obvious would be to run on dedicated machinery. However, given the limitation of shared computing equipment, an approach that uses employs load balancing rather than static partitioning of data could improve results. For example, employing a worker-pool approach, in which small chunks of the data set are provided at a given time, until a worker requests a new chunk, would have allowed the server to give the faster (i.e. less loaded) clients more work to do than the slower ones, potentially improving the over all time.

The times for the parallelized version appeared to be improving as the data sets got larger, suggesting that for larger data sets, the penalty paid for network transmissions was less of a factor in the total time usage.