Homework 4
Parallel Sort

Geoffrey Ulman
CSI702

March 2010

1 Design

My serial sort is a simple quick sort implementation. My parallel sort is a bin sort very similar to the implementation analyzed in our previous homework. Each node receives an equal portion of the data set from node 0. Node 0 then calculates bin boundaries by selecting a small subsample of the entire data set, sorting it, and choosing data elements from it at regular intervals. This is done as an attempt to reduce communications by making the amount of data which ends in each bin as even as possible. Node 0 then informs the other nodes of the bin values it has chosen. The nodes then bin their local data and send the bins to the appropriate nodes. The nodes then perform local serial sorts on their subset of the data and send the results back to node 0.

2 Challenges

Writing a correct and efficient serial sort was a non-trivial challenge. I did not want to simply use the sort function built into the c standard library. I chose binary sort as my serial sort. My implementation works in place, but is not necessarily a stable sort. It also does not switch to a simpler sort for efficiency when the size of the recursive sub problems becomes small.

Debugging my parallel implementation was also quite difficult. My communication logic was relatively straightforward and no serious debugging was needed there. What proved difficult, was tracking down memory allocation problems which arose because of the somewhat complicated bookkeeping necessary to keep track of how much data had been stored in each bin. Originally, each processor allocated enough space in each bin for the entire data set (for simplicity). However, for large data sets and large numbers of nodes, I began running into memory errors. This took a significant amount of trial and error to find.

After fixing the bugs described above, my parallel implementation worked, but was quite slow. For simplicity, I was using far too much synchronous communication. My first implementation had the nodes sending their binned data to each other essentially one node at a time. The current implementation has all the nodes send their binned data to all other nodes asynchronously.

This problem was the first time that I had worked with variable sized messages in MPI, so I had to familiaraize myself for some of the functions for retrieving data from `MPI_Request` and `MPI_Status` objects as well as think through the logic necessary to combine the data from multiple variable sized messages into a single array.

Finally, I was not able to profile the code for arrays larger than 10^9 because the messages I was attempting to send exceeded the maximum message size. To experiment further with such large data sets would have required even more complicated algorithms to split up messages as they got too large.

3 MPI Commands

Like all MPI programs, the commands `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, and `MPI_Finalize` were used to initialize and get basic information about the MPI environment. Unsorted segments of data was distributed to nodes using `MPI_Isend` and `MPI_Waitall` was used by node 0 to wait for the sends to complete. Because binning was calculated on the host node, `MPI_Bcast` was used to distribute the bin edges to the other nodes. Data was sent using `MPI_Isend` and both `MPI_Irecv` and `MPI_Recv` were used for receving data. Synchronous receives were used when receving unknown quantities of data from multiple nodes. This greatly simplified the task of collecting the data into a single array for processing. `MPI_Probe` and `MPI_Get_count` were also very important for handling messages of unknown size.

4 Performance Analysis

5 Output Comparison

References

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall PTR, New Jersey, 2009.

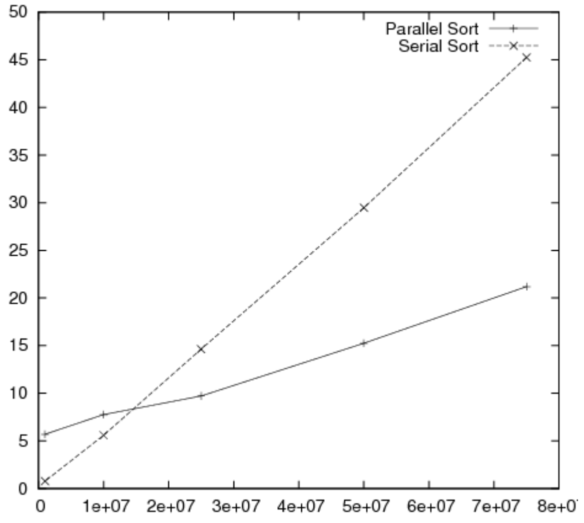

Figure 1: Parallel and Serial Timing Results for Variable Array Size