Problem Statement

We will use MPI, OpenMP, OpenACC, and the quick sort algorithm with the parallel sorting by regular sampling method to sort a list of numbers across multiple machines, cores, and GPUs. The algorithm completes the sort in O(nlogn) time. Quick sort is ideal in the sense that it can sort in O(nlogn) time, and there does not exist an algorithm with better performance. It is non-ideal in the parallel sense because the isoefficiency relation is Clogp, and therefore will not scale perfectly to larger systems.

Approach to Solution

The source code is written in c and I used a few other software tools to accumulate the data. I used a Makefile for compiling the code with different compilers and on different machines. To aggregate the data I used a bash script to execute the code under different conditions, and collect the data in comma separated text files. I then used Excel to calculate relevant statistics on the data and produce plots to visualize the data. To save and version my work, I used a git repository on Github. This also proved useful in transferring my data and code to and from the Bridges machine to my local machine.

The first step to solving the problem is to set the conditions with the git repository and origination of all my code. I debugged the code in the simple serial environment, and steadily increased the size of the array I sorted as I worked out the bugs. Once the serial code was complete I added the OMP and ACC implementations. Once it came time for the MPI implementation, I executed each of the communication steps individually to ensure that all of the correct information was being transmitted.

Solution Description

To calculate the performance of the calculations we take the number of elements in the list and divide it by the time required to sort the list. In the MPI implementations we take the overall size of the list across all the machines. Formula (a) give the equation used to determine the performance.

(a) $$\text{Performance} = (\text{n})/\text{Execution Time}$$

The performance of the serial and OMP implementations degraded as the size of the array increases, which is to be expected. The algorithm runs in O(nlogn) time, so as the size of the array increases linearly, the execution time will increase by a logn factor. The OpenACC implementation yielded similar results. The complex data dependencies cause the data movement overhead to eclipse any gains from parallelism.

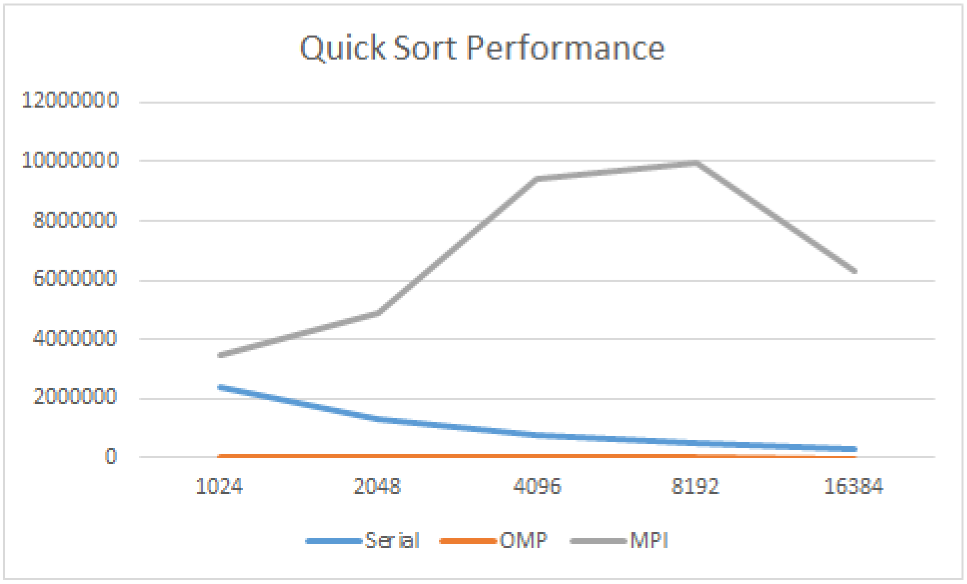

Figure 1: Quick sort performance for Serial, OMP, and MPI.

We achieved significant speedup with the MPI implementation, to a point. As the size of the array increases, the parallel gains overcome the overhead gains. Like the serial execution, as the size of the array grows the algorithm overhead eclipses the gains from the parallelism. Further work could be done on more machines to increase the parallelism and improve the performance. This works because the size of the initial sort is decreased, and it is easier to merge elements into a sorted array compared to sorting a larger array. Unfortunately, the isoefficiency matrix for this algorithm is not perfectly scalable, we will see a plateau in the performance.

Size	Serial	OMP	MPI
1024	2358974	32442	3447080
2048	1325616	33390	4910572
4096	762197	33769	9427030
8192	474133	31433	9932305
16384	283031	27615	6319738

Table 1: Performance across implementations and size.
Data is average performance across 10 iterations.