Project #3 – MPI Reduce & Broadcast

Kyle Pontius

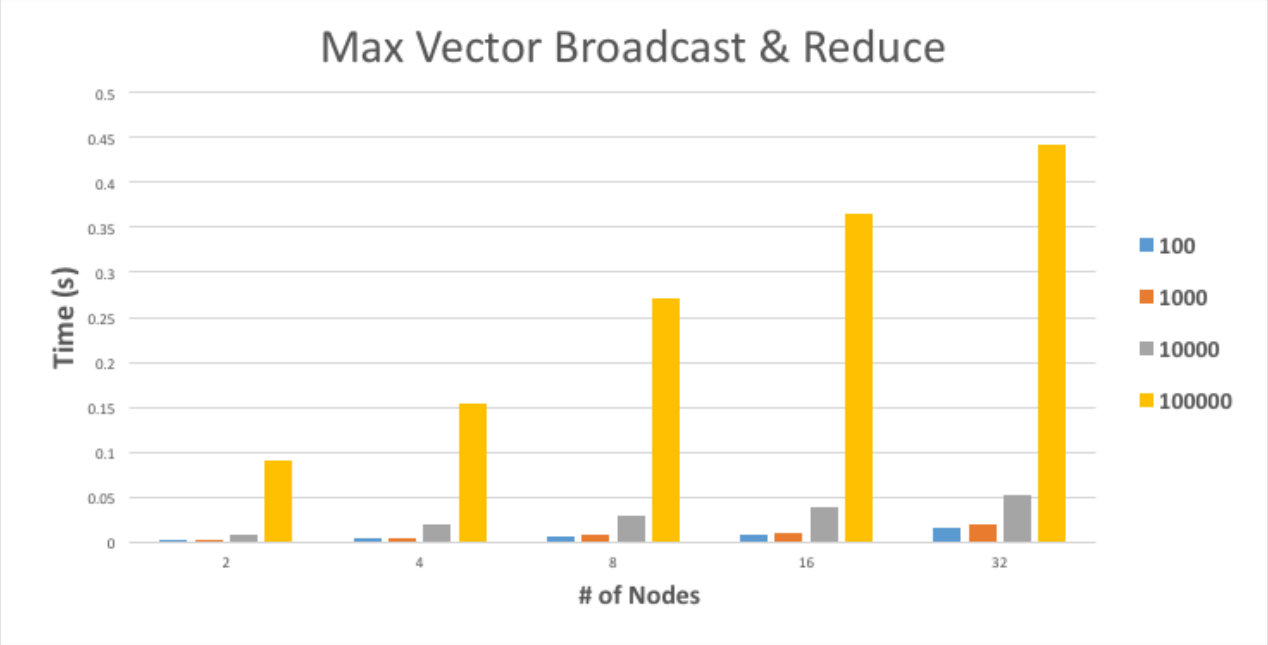
Introduction

The dawn of the transistor, and with it, the computer has brought on one of the greatest ages of innovation and technological growth this world has ever seen. Few lives have remained untouched by this relatively new and extraordinary development. One computer alone holds untold power, yet tethering these machines together helps us reach new heights unachievable with a single node. MPI is one of the most powerful, modern tools we have to create such a network of machines. The principle of this project is to flex MPI's communication muscle to create a meaningful example of distributed computing using vector max reduce & broadcast.

Methods

To facilitate efficient communication between our nodes, for vector max reduce & broadcast, we use the hypercube approach; which acts much like a binary tree and functions in logarithmic time. With the hypercube algorithm, in general, broadcasts begin at the root node then use MPI_Send() to message two child nodes (which use MPI_Recv() to receive each message). This pattern repeats for each segment of the hypercube, where a given child node then sends to two new children nodes of its own (hence the logarithmic growth). This is also the pattern of reduction, where two children reduce to a single parent, with two parent nodes reducing, in turn, with another parent until the root node is reached.

My approach was to create a program which does the following: First, each machine generates a new array of random numbers (size n) in its own memory. Second, each node then reduces (sends) this array to its child. Third, now that the child has two arrays it takes the two arrays that were sent to it and chooses the highest value at a given position, from either array, which value becomes the "max value" for that position in the resulting array. Fourth, the array reduces its maximized array to its designated parent node. If we're not at the parent node at this point, we go back to the 3rd step and repeat. Finally, after all the child nodes have reduced to their parent nodes, we end up at the root node with a maximized array of numbers. This array contains the maximum value of any nodes' array, for a given position.



Results

General performance results from my algorithm are charted above. Please note, that the spec specified using 65K as the max vector size, I decided to go with 100K as the max to

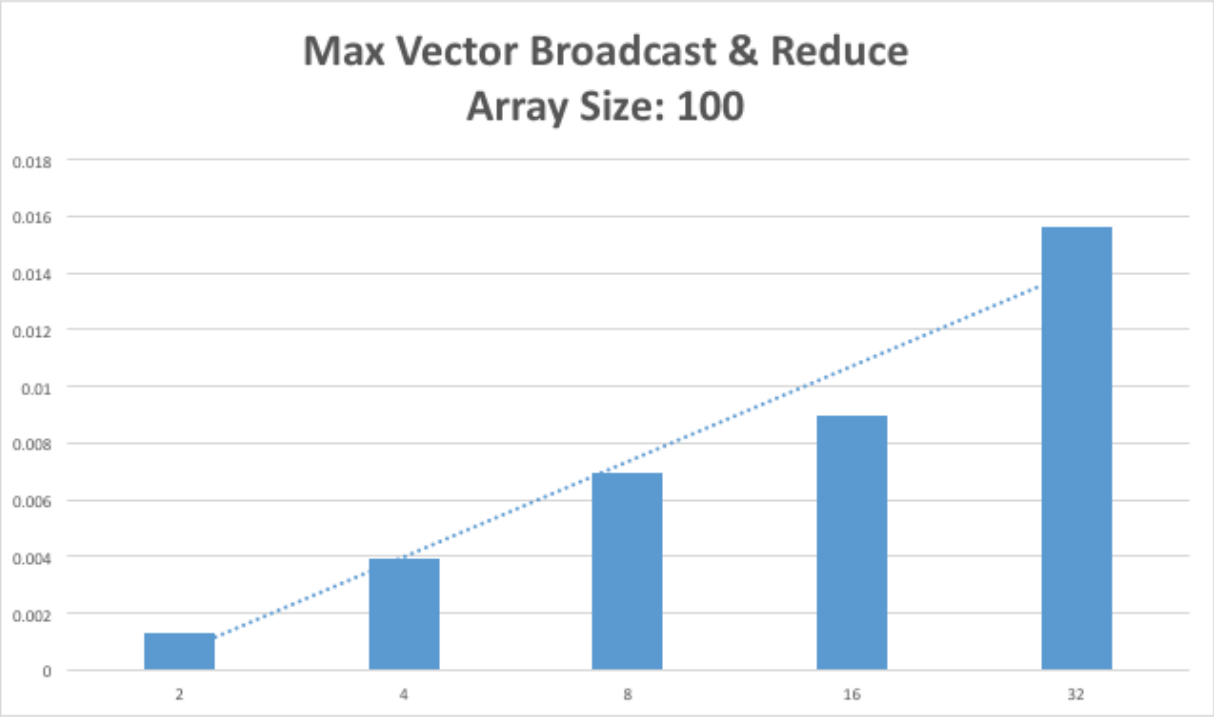
more effectively visualize the differences in performance. This graph above depicts the execution time of different problem sizes, ranging from 100- to 100,000- count arrays of randomly generated integers. These numbers show us some very interesting results, a few of which I'd like to touch on:

First, for a given number of nodes, say 32 nodes, the execution time takes almost 10x longer for each array-size respectively. In other words, the time for a 10,000-sized array to execute on 32 nodes takes about 10x less than 100,000 (.05 seconds vs .45 seconds). This is interesting because it shows us that the difference, in communication time, to transfer the data for the large array and a small array is relatively small.

Second, observe the difference in speed for each array size, for a given number of nodes. The coefficient of growth between for problem size 1000 on 2, 4, 8, 16, and 32 nodes is nearly identical to the scale of growth on problem size 100,000 for 2, 4, 8, 16, and 32 nodes. Again, I believe this attests to the fact that communication time between the nodes is not the primary factor in increasing execution time. However, this leads us to the final point I'd like to make.

While I did my best to make the communication time dominate the calculation time of the program, I just couldn't do it. In an attempt to scale down max-array calculation time to be smaller than communication time, I dropped the array size to just 100 integers. However, this didn't change the execution time growth coefficient at all (see graph below). As we can see from this graph, and the linear trend-line I added, the growth of the execution time was nearly linear even with a small array size. If I had to surmise the reasoning behind these results, and the failure of communication time to largely affect the outcome, I would guess that if we

drastically increased the number of nodes or shrunk the problem size even further, we'd finally observe a more significant impact from communication on total runtime. Additionally, the low communication time could also be simply due to the type of problem we're dealing with, that



is, we're just passing arrays back and forth. This is probably far different than transmitting large, unconventional, data objects between nodes, instead.

Bandwidth Calculation

As per the spec, I calculated the max bandwidth time between the host node and node #2 to be about: 265 Mbps. I used a for-loop with the blocking MPI_Send() and MPI_Recv() calls, sending data back and forth for 100 iterations, then measuring the total execution time. The formula I used was: ***dataSize * iterations / executionTime***

Conclusion

First, this particular problem is dominated by problem size far more than communication time. Second, execution time grew, for each problem size, with a nearly identical growth coefficient; which is to say that each time we increased from 2, to 4, to 8, and so on nodes, our execution time growth was proportionally very similar for both a 100-sized array and a 100,000-sized array. Finally, despite shrinking the problem size down to a relatively small number, we saw that communication time between nodes was still insignificant for the type and scale of this particular problem.