# Problem Statement

We will use MPI, OpenMP, OpenACC, and Cannon’s algorithm for matrix multiplication across multiple machines, cores, and GPUs. The algorithm solves matrix multiplication in O(n3) time with O(n2) memory requirement, where n is the number of row in the matrix. Cannon’s algorithm is ideal for parallel computing environments because the memory requirements per processor is constant, making it extremely favorable for parallelization.

# 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 R 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 of the problem is to design the test matrices and figure out a way to test the output in a reasonable amount of time. I decided on multiplying a matrix with incrementing values multiplied by an identity matrix. This makes the creating and testing simple, while giving them enough variation to expose any issues in the code. It also keeps the maximum value of the elements low, allowing us to use floating point variables and reduce the communication overhead.

The problem is extremely easy to solve in the serial and single machine solutions. There is no loop carried dependencies across all three levels and we can simply run the iterations of the outside loop in parallel. Executing the problem across multiple machines becomes more difficult. If we precompute the neighbors for each machine, the communication involves two sends and two receive operations per iteration. I first executing the problem with small matrices and the minimal number of machines to test the code. Once the code was verified, I requested more machines and increased the size of the matrices.

# Solution Description

To calculate the number of floating point operations per second (FLOPS), we use the same formula as the previous assignment, given in formula (a). Each iteration requires one floating point add and one floating point multiply. Further graphics will represent the performance in Giga FLOPS.

(a) FLOPS = (2\*n3)/Execution Time

The first parallel execution that I attempted was the OMP implementation on a single machine. We were able to see significant speedup using the 28 core on the single machine. Calculation the serial execution portion with the Gustafson-Barsis equation and the speedup potential with the Karp-Flatt metric we can see that the speedup potential is good on the single machine. We can see in Figure 1 that the Karp-Flatt metric increases at first, but levels out when we start using greater than 10 cores.

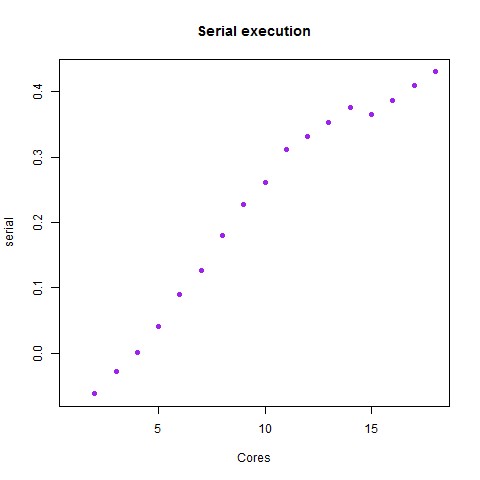
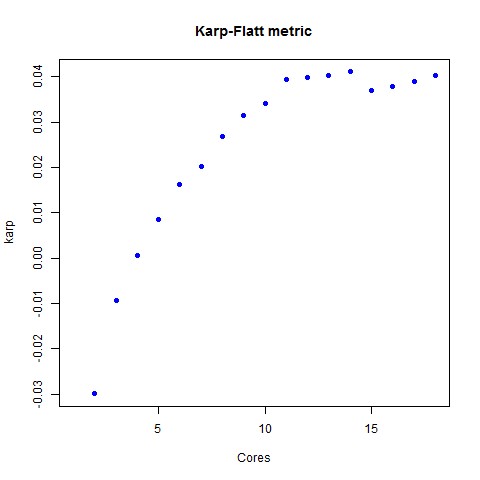


Figure 1: Karp-Flatt metric and serial execution ratio

The speedup levels off at around 12 Giga FLOPS, and the serial portion of the execution time is steadily increasing. With these two pieces of information in mind we know that the reduction in speedup is because of the data dependencies between the loops and not the parallelization overhead.

The next implementation that I tested was the OpenACC on one GPU. I will have to make future improvements to properly utilize all of the available GPUs on each machine. I didn’t see as much speedup with this implementation. I suspect that there was a flaw in my implementation that caused extra copy operations between the host and the device. Because the outside loop is not parallelizable, I suspect that the outer loop was being run on the host and the system was having to copy in and out the value of k.

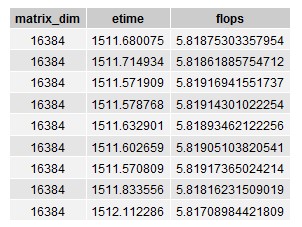


Figure 2: OpenACC implementation on single GPU

The next implementation I worked on was the pure MPI and OpenMP with MPI. Setting the OMP\_NUM\_THREADS to 1 I was able to test the performance with just the MPI functions. Using 8 machines with one core on each machine I was able to achieve almost the same performance as using 8 cores on 1 machine. I suspect that this would occur because the machines would have a high cache hit rate because the size of the data that they are operating on would be much smaller than the one machine with multiple cores.

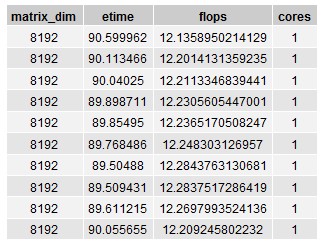


Figure 3: MPI performance across 8 machines

Using more machines and cores we were able to achieve pretty good speedup, but the communication overhead between the machines cased the speed to level off around 28 Giga FLOPS. Interestingly the performance dropped between 4 and 8 machines before rising again with 16 and 32 machines.

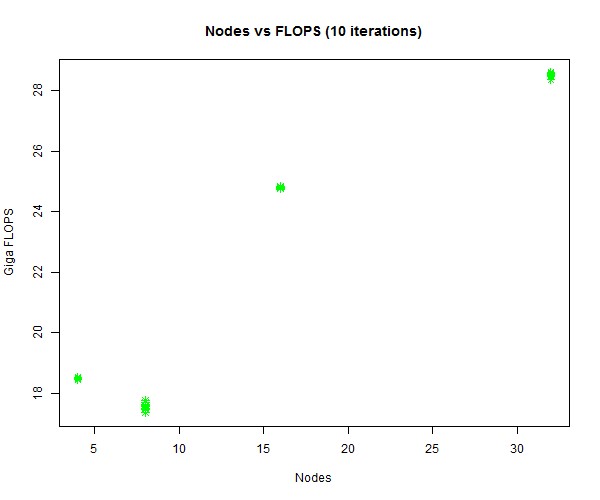


Figure 4: OpenMP and MPI execution.

The best speedup that I observed was executing the algorithm on the P100 GPUs on multiple machines with MPI. I believe that the speedup could be significantly increased by utilizing both of the P100s on each machine. This was the most complicated implementation because it required the GPU to copy the memory back to the host and then between the hosts and back into the GPUs. The highest observed performance was 43.67 Giga FLOPS.



Figure 5. OpenACC and MPI execution on 4 P100 GPUs