Final Project Summary
CS 4230-001
Jeremy Bonnell
Adam Hartvigsen

This project was challenging to say the least. Our original plan was to parallelize the SVD algorithm using a combination of MPI and CUDA. After our initial efforts we found a couple of problems. The CHPC servers don't have the GPU's that we need to run CUDA. The CADE machines don't have large numbers of processors that take advantage MPI. After running into this complication and a discussion with Professor Hall we changed our approach. We wrote SVD in both CUDA and MPI and compared the sequential and parallel run times. We wrote and ran our MPI SVD program for the CHPC servers and our CUDA SVD for the CADE machines.

We worked and got our MPI SVD "working" and started to test it against the sequential SVD on the CHPC servers and got terrible results. We ran tests on matrices of sizes of 512, 1024, 2048, and 4096. The time increased by a factor of 10 for each matrix size increase. The parallel version was about twice as long as the sequential. When we presented our poster this was the state we were in. We had our MPI SVD sequential and parallel code "working". We also had our sequential CUDA code up but not our sequential. After talking with Professor Hall our goal was to get the CUDA code working and get a speed up out of one of the two sequential programs. Since then we have been able to do that. We have a working CUDA parallel version and we were able to get speed up over our sequential code in our parallel MPI SVD program.

Implementation

The MPI implementation had a speedup factor of 2.9 for a 512x512 matrix! This was run on CHPC and used an algorithm, that after discovered, could have been used to speed up the CUDA version as well. All previous iterations of this program worked on the U_t matrix and always performed reductions it in a row by row fashion. The attempts at parallelizing the rotations had always fell short. But after glaring at the problem for a few days, it was found that if the matrix was transposed, all data dependencies can be broken and the rotations parallelized. Since the A[ ] matrix was already transposed, it was used to transfer chunks of data into a subset of the matrix for each process. Each process received N/comm_sz number of rows of A[ ] and stored them contiguously into a 1D matrix tempU[ ] and tempV[ ]. They could now rotate one row at a time without communicating with any other process. The threads all combined their results using MPI_Allgather() back into A[ ]. However, since c[ ] and s[ ] were still being calculated from U_t[ ] in the rotations, each i-iteration had to transpose A[ ] back in to U_t[ ] for the next reduction cycle. That was when it was found that if the same chunks of data could also be used for reductions, no transposes would be needed. In fact, each process would work on the same section of the matrix the whole “while-loop” and NEVER see any of the other data. The only caveat was each process still needed results of alpha, beta, and gamma to calculate c[ ], s[ ], and converge. So each process calculated a portion of beta[ ], and gama[ ] arrays. If the j-iteration includes i, that’s alpha. After the reduction loop ends, the results are combined with MPI_Reduce(…MPI_SUM); Each process now has a 1D array of up to (N-1) elements of beta and gamma. Each process, then computes an array for c[ ] and s[ ] and the value of converge. All dependencies

were now broken. A[ ] did not even have to be reconstructed until AFTER the while-loop ended. The results were transposed back in to U_t one time, then the program finishes up.

This technique can undoubtedly be incorporated in to the CUDA version as well. The version we have now completes, but is very slow. It uses the same row operations on the reductions as was first used on the MPI version and does not even have the rotations parallelized. It also has to resend the entire U_t, beta[ ], and gamma[ ] datasets to the GPU on each i-iteration. If the same MPI algorithm was applied, the GPU would have the data sent one time, check converge internally, then return once convergence is reached. We are confident that we could also combine CUDA and MPI on the CADE machines and get a speedup. We would try using MPI for reductions and CUDA for rotations and vice versa to see which is faster. Or run all rotations and reductions in MPI and calculate S[ ] with CUDA. ☺ Unfortunately, our time has expired. Although the instructor may be saddened by not ever viewing this achievement, the students will continue to work until this problem is solved.