CSC 458 Assignment 4

Nick Graham

April 27, 2017

Introduction

Assignment Four had us look at two methods of parallelizing programs. One was using the map-reduce library Hadoop, and the other was using MPI. Hadoop was used to create a program that would take a set of input files and create a histogram of the word lengths found within them. For this case the only characters that were counted in the word length were characters, and not any other special character, punctuation, or number. MPI was used to parallelize a sequential program used to solve systems of equations, just as was done in Assignment 3.

Hadoop

The Hadoop portion of the assignment was written in java and was heavily based off of the word count example ($http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html$) that can be found in the documentation. To test the program I ran it on a set of files downloaded from project gutenberg of varying sizes. I ran the program on each file individually as well as all the files at once.

What I found was that for the limited number of files I ran it on the file size, number of files, and number of hadoop instances did not affect the results. Each test took between 14 and 16 seconds. To me this indicates that I did not have enough data in order to accuratly test the program. What I am able to gather from this however is that since performs about the same across files ranging from 4694 words to files as large as 1036117 as well as

when run on all the files at once, is that most likely scales well and would work for very large data sets.

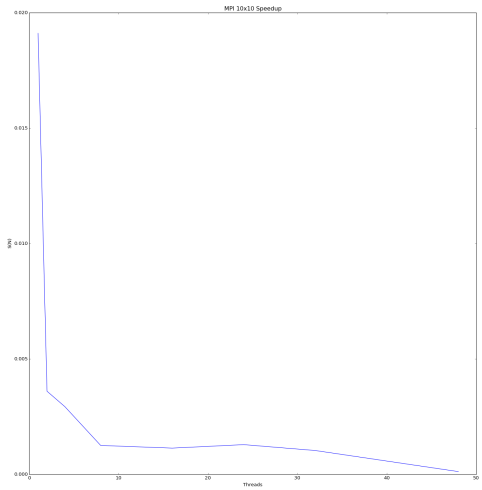
I was unfortunetly unable to test more due to time constraints both in using the cluster, as well as in gathering test data. If I were to test this further I would like to test it on datasets an order of magnitude larger in hopes of being able to better observe how it scales.

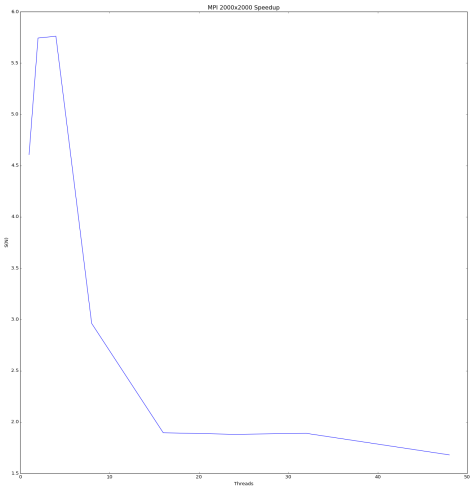
MPI

For the MPI portion of the assignment I parallelized the given serial code. My original attempt, which can be found in the gauss.c file, was to dynamically assign the number of columns to work on for each itteration, passing along the updates after each run through the array. This worked rather well for the 10×10 array I used for testing, however as soon as I moved to a larger data set, it slowed down dramatically.

In order to improve performace I desided to instead statically assign the portions of the array and only merge the results after each worker was finished. While this ended up being much faster (as in it actually finished), it did mean that each worker was not doing an equal amount of work. This method however dramatically decreased the number and size of the messages that needed to be sent, inproving performace.

When implementing this portion I ran into a few issues. The first was when the number of threads exceded the size of the dataset. To handle this, and to make my math simpler, I picked the closed number that the data set size was divisable by. The second issue I had was once I had done that, I had to create a new communication world so that only the threads that were working trying to talk to each other, and they did not wait for threads that had already exited. This issue was hard for me to track down, as it did not occure on my local machine, it only occured on the cluster.





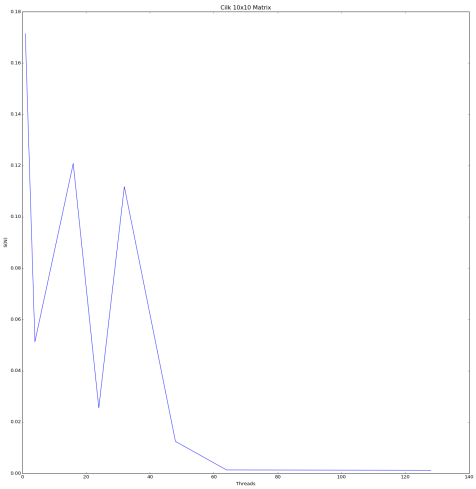
From these graphs you can see that my MPI solution did not fair very well. For the small data set the sequential program provided the best runtime, and the more threads only added overhead. For the larger data set, the parallelization did provide some improvements up until around 8 threads, and then the performance took a massive hit.

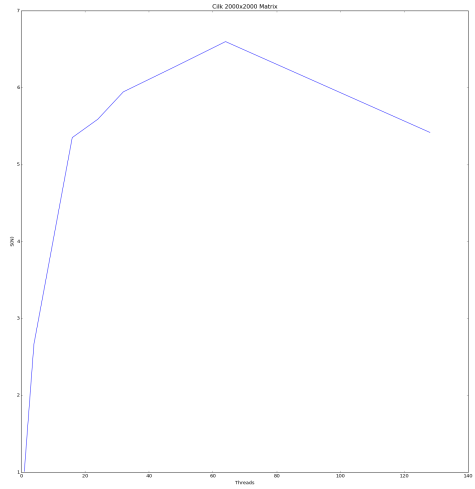
I believe these overhead costs are due to the ammount of message passing I still did in each iteration. In order to handle the smart pivoting I had all

the threads do a MPI_Bcast to share the location of the row to swap with.

I also had to share an array that contained the pivot values, as not all the threads would have the updated values to use for the computations.

As compared to the graphs below for Cilk, the most scalable implementation I had from project 3, you can see that the MPI Version is nowhere near as good.





In order to improve this in I would want to to either work on sending

smaller and less frequent messages, as well as trying to restructure the algorithm so that there were more independent portions.

Conclusions

Overall I found using both of these methods of parallelization to be rather interesting. Hadoop was interesting to work with, as I am not as familiar with java, however it provided amazing to results and ended up being easy to use.

The MPI portion of this project for me was fustrating. My first attempt proved to be horribly inefficient and I had to rethink my approach entirly. I then had a large number of issues that appeared due to the differences between my local install of MPI and the version on the cluster. While I liked the control that MPI provided for manually doing the parallelization, it certainly was much easier to use things such as Cilk or OpenMP to parallelize the code.