CS 5220
Project 3 - All Pairs Shortest Path

Marc Aurele Gilles (mtg79)
Edward Tremel (ejt64)
Yu NAME NETID

November 10, 2015

1 Introduction

In this project the goal is to profile, parallelize and tune an algorithm which computes the shortest path between all the pairs in a graph.

2 Profiling of Original Code

The profiling was done using amplxe. By looking at the hotspots report, we observe that most of the time (45 out of 67 seconds) is spent in the `square` function, which is to be expected as all of the computation is done in this function. A notable point, however, is that virtually all the rest of the time is spent waiting, which seems to indicate a significant load imbalance between the threads. Looking at the time spent inside of the `square` function, we observe that a majority of the time (27 out of 45 seconds) is spent fetching memory and writing it to a float, which could indicate that there is poor cache utilization.

This profiling leads us to believe that we should try to address two problems for the OMP implementation: have better load balancing, and increase cache reuse.

3 MPI Implementation

We assign each processor to a block of the the matrix instead of an entire column as the original code does. Note that to update one entry of the matrix for one step of the outer loop we need to read the whole column and row of that entry. Therefore, if we assign each processor to a column, then each processor needs to access each other column, and thus needs to communicate with all of the other processors. By assigning each processor to a block of the matrix instead, we allow each processor to have to communicate with only the blocks that share its row and its column.

To take advantage of this blocking strategy, we create $2n_{block}$ groups and communicators, where $n_{block} = n/blocksize$. Here n is the dimension of the matrix and $blocksize$ is the dimension of the blocks. That is, we have one communicator for each row of blocks, and one communicator for each column of blocks. Each thread, which owns its own block, is then added to the communicator of its row and to the communicator of its column. Therefore each thread is in two different communicators (in addition to the world communicator), but no two threads are in the same two communicators. To update its block, each processor makes one `MPI_Allgather` call in its column communicator and one `MPI_Gatherall` call in its row communicator, which reads all of the entries in its block column and block row, performs the update on its own block, and calls `MPI_Allgather` again to share its result with the other processors.

The number of entries to be shared in this manner is $2 \cdot n \cdot blocksize$ per thread and per time step, compared to n^2 if each thread owned a column instead.

The MPI implementation runs in 3.2 seconds for n=2000, which is about 10% more than the original OMP implementation, which runs in 2.9 seconds.

4 To do next

Next we will to tune our MPI implementation as well as the OMP implementation provided to us. We will also incorporate the blocking strategy into our parallel code to increase the cache reuse.