CS 5220
Project 3 - Floyd-Warshall Algorithm

Eric Gao (emg222)
Elliot Cartee (evc34)
Sheroze Sheriffdeen(mss385)

November 10, 2015

1 Introduction

The Floyd-Warshall algorithm computes the pair-wise shortest path lengths given a graph with a metric. The computational pattern of this algorithm is very much akin to matrix multiplication. If l_{ij}^s represents the the length of the shortest path from node i to j in at most 2^s steps, [1] then

$$l_{ij}^{s+1} = \min_k \{l_{ik}^s + l_{kj}^s\} \tag{1}$$

2 Design Decisions

2.1 Parallel Tuning

2.2 Message Passing Interface

In addition to the tuned parallel implementations, we explored an approach that uses the Message Passing Interface to achieve parallelism. In the MPI implementation, each process handles a certain region of the graph. To prevent a master process orchestrating the distance computation, we ideally want each process to only wait for information from the relevant part of the graph. To that end, we take the adjacency matrix on which the Floyd-Warshall algorithm is run and partition the graph by chunks of columns. Then each processors is responsible for update a contiguous sequence of columns in the matrix.

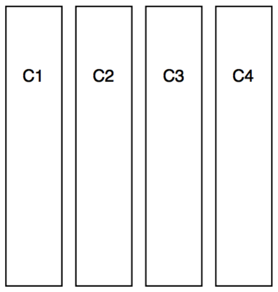

Figure 1: Initial partition of the graph where each C_i is a sequence of columns

Now each sequence of columns owned by a processor can be decomposed into square blocks. To compute the next iteration of the Floyd-Warshall algorithm for a single block, say block number i in processor 2, we need the i^{th} block from all other processors.

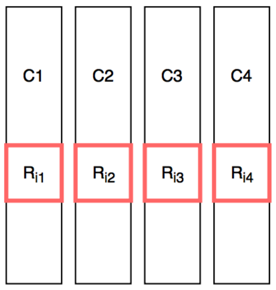

Figure 2: For block R_{i1} we need the i^{th} block from all other processors

Therefore, we do an `MPI_Allgather` operation which gathers the i^{th} block from all the processors and recreates the i^{th} row chunk in all processors. Now, we can update block R_{ij} for all j processors.

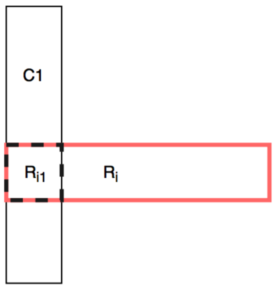

Figure 3: Updating the i^{th} square block by processor 1 using row chunk i

The `MPI_Allgather` is then repeated until all square blocks have completed a step in the Floyd-Warshall algorithm. Each processor individually checks whether an update was made to their sequence of columns. To complete the iteration, we perform a `MPI_Allreduce` operation to check whether any update was made across all processors. If an update was made, we continue the iteration. Otherwise, all processors terminate and the solution is reached.

2.2.1 Advantages

The MPI approach improves upon a serial implementation of the algorithm due to its ability to compute updates to multiple regions of the graph in parallel.

The MPI approach makes each processor be responsible of a contiguous sequence of columns in the graph. To update a a square block in this sequence of columns, the processor needs to recreate only the corresponding sequence of rows. Therefore, if the width of the sequence of columns is d and the side length of the graph is n, each processor only needs to hold $2nd$ information in memory instead of n^2. On larger graphs, this decrease in memory footprint will prevent thrashing since the space scales as $O(n)$ instead of $O(n^2)$ and will show improvements in performance over the OpenMP version.

2.2.2 Disadvantages

On smaller graphs, the communication and synchronization overhead of MPI may cause a decrease in performance.

2.2.3 Implementation

The MPI implementation can be found in `path-mpi.c`. We are still in the process of ironing out bugs in the implementation and hope to have a complete solution by the final report. Furthermore, if time permits, we aim to explore the Cannon's algorithm to improve upon the current MPI scheme. [2]

3 Analysis

3.1 Original Implementation

3.1.1 Profiling

Profiling the original solution shows that the most CPU time goes into the square function and significant portion of that time is considered by VTune to be ideal. The next most expensive functions in terms of CPU time is the barrier and the reduction in OpenMP due to the high spin times.

```
-------------------------------------------------------------------------------------------------------
|                           |          |                 CPU Time                   |      Spin Time       |
| Function                  | CPU Time |  Idle      Poor        Ok    Ideal  | Serial   OpenMP  Other |
|-------------------------- | -------- |-------  --------   ------  ------- | -------  ------- ------ |
| square                    |  42.888s |      0s    7.252s   3.375s  32.261s |       0       0s     0s |
| __kmp_barrier             |  14.312s | 0.030s   13.113s   1.029s   0.140s |   14.31  13.712s 0.600s |
| __kmpc_reduce_nowait      |   4.654s | 0.030s    4.186s   0.398s   0.040s |   4.583   4.324s 0.260s |
| __kmp_fork_barrier        |   2.877s | 1.549s    1.187s   0.121s   0.020s |   2.876   2.746s 0.131s |
| __intel_ssse3_rep_memcpy  |   0.030s |      0s    0.030s       0s       0s |       0       0s     0s |
| gen_graph                 |   0.030s | 0.010s    0.020s       0s       0s |       0       0s     0s |
| __kmp_itt_metadata_loop   |   0.028s |      0s    0.026s   0.002s       0s |       0       0s     0s |
| fletcher16                |   0.020s |      0s    0.020s       0s       0s |       0       0s     0s |
| __kmp_join_call           |   0.010s |      0s    0.010s       0s       0s |    0.01   0.010s     0s |
| __kmp_launch_thread       |   0.010s |      0s    0.010s       0s       0s |    0.01   0.010s     0s |
-------------------------------------------------------------------------------------------------------
```

3.1.2 Scaling Study

The speedup plots of the original solution shows linear improvement in performance

3.1.2.1 Strong Scaling Study

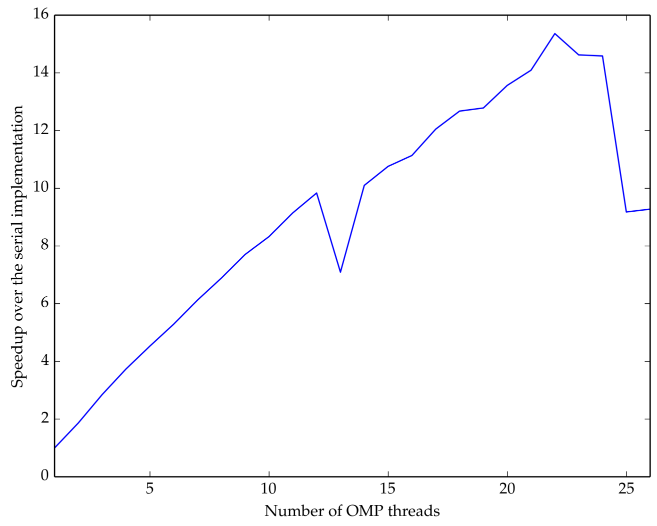

Figure 4: Strong scaling study of the original solution

3.1.2.2 Weak Scaling Study

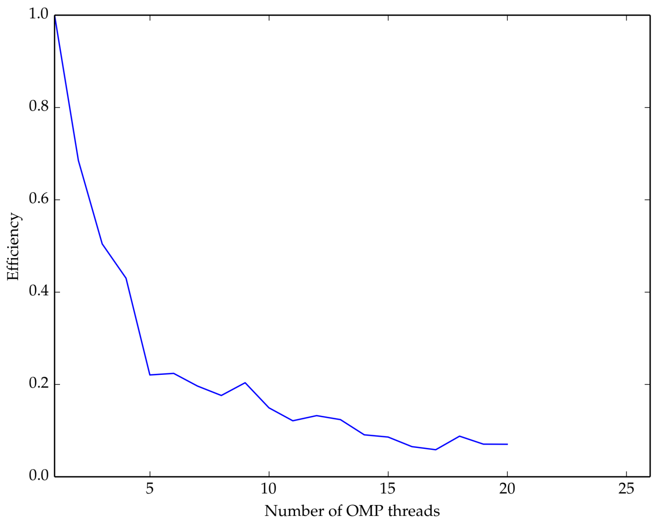

Figure 5: Weak scaling study of the original solution

3.2 Tuned Parallel Implementation

3.2.1 Profiling

3.2.2 Scaling Study

3.2.2.1 Strong Scaling Study

3.2.2.2 Weak Scaling Study

References

[1] Bindel, D. *All-Pairs Shortest Paths.* Retrieved November 10, 2015, from `https://github.com/sheroze1123/path/blob/master/main.pdf`

[2] Hyuk-Jae Lee, James P. Robertson, and Jos A. B. Fortes. 1997. *Generalized Cannon's algorithm for parallel matrix multiplication.* In Proceedings of the 11th international conference on Supercomputing (ICS '97). ACM, New York, NY, USA, 44-51. DOI=http://dx.doi.org/10.1145/263580.263591