

CS 5220

Project 3 - All-Pairs Shortest Path

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

November 19, 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 length of the shortest path from node i to j in at most 2^s steps, then

$$l_{ij}^{s+1} = \min_k \{l_{ik}^s + l_{kj}^s\} \quad (1)$$

represents the shortest path from i to j of at most 2^{j+1} hops. [1]

2 Design Decisions

2.1 Parallel Tuning

Since the Floyd-Warshall algorithm is structured very similarly to matrix multiplication, we decided to try taking some of the tuning methods used in the first project on matrix multiplication, and applying them to the Floyd-Warshall algorithm.

The first of these methods uses a blocking scheme, so that updating the shortest path lengths is done through repeated calls to a small kernel. Since the size of this small kernel is known at compile-time, the compiler is able to optimize this small kernel extremely efficiently. Currently our implementation of blocking only works when the number of vertices is divisible by the block size, but we plan to correct this soon.

The second method changes the loop ordering, so that in the innermost loop, memory is being accessed with unit stride, which increases performance.

As the project goes on, we also hope to add other optimizations attempted in the matrix multiplication project, such as copying the blocks into a contiguous chunk of memory.

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.

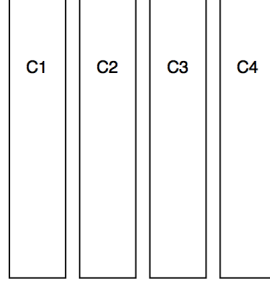


Figure 1: Initial partition of the graph where each C_i is a sequence of columns

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.

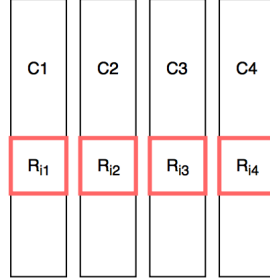


Figure 2: For block R_{i1} we need the i^{th} block from all other processors

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.

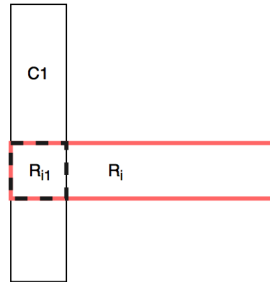


Figure 3: Updating the i^{th} square block by processor 1 using row chunk i

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.

The MPI implementation can be found in `path-mpi.c`.

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 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.

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.

Function	CPU Time			Spin Time		
	CPU Time	Idle	Poor	Ok	Ideal	Serial
square	42.888s	0s	7.252s	3.375s	32.261s	0
__kmp_barrier	14.312s	0.030s	13.113s	1.029s	0.140s	14.31
__kmpc_reduce_nowait	4.654s	0.030s	4.186s	0.398s	0.040s	4.583
__kmp_fork_barrier	2.877s	1.549s	1.187s	0.121s	0.020s	2.876
__intel_sse3_rep_memcpy	0.030s	0s	0.030s	0s	0s	0
gen_graph	0.030s	0.010s	0.020s	0s	0s	0
__kmp_itt_metadata_loop	0.028s	0s	0.026s	0.002s	0s	0
fletcher16	0.020s	0s	0.020s	0s	0s	0
__kmp_join_call	0.010s	0s	0.010s	0s	0s	0.01
__kmp_launch_thread	0.010s	0s	0.010s	0s	0s	0.01

3.1.2 Scaling Study

The speedup plots of the original solution shows linear improvement in performance but an exponential decrease in efficiency as shown in figure 5. This can be attributed to the increased overhead in synchronization and spin time.

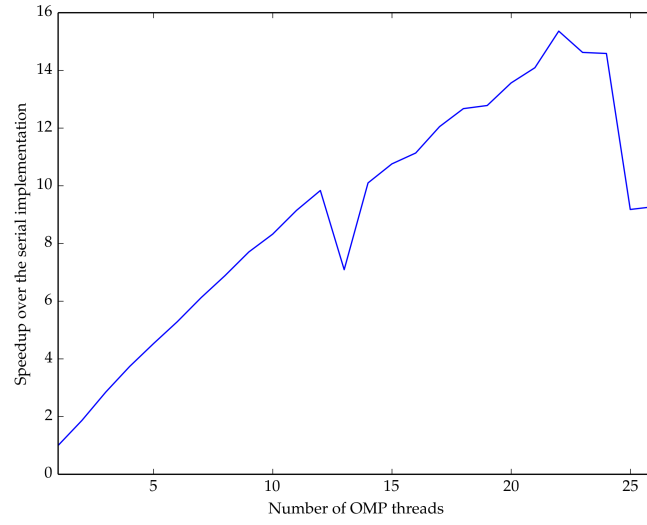


Figure 4: Strong scaling study of the original solution

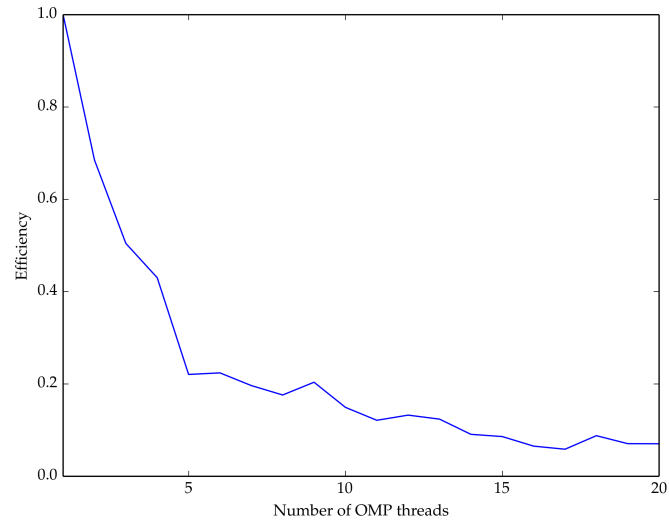


Figure 5: Weak scaling study of the original solution

The strong scaling study was performed on a graph with 2000 nodes. The weak scaling varies the number of threads but keeps the problem size per processor be 1000 nodes.

3.2 Tuned Parallel Implementation

3.2.1 Profiling

We had two version of the tuned parallel code in OpenMP. The output from the first version is shown here:

Function	CPU Time	CPU Time				Serial	Spin Time		Other
		Idle	Poor	Ok	Ideal		OpenMP		
square	963.365s	0s	249.551s	257.437s	456.377s	0	0s	0s	
square	145.393s	0s	13.859s	40.891s	90.642s	0	0s	0s	
__kmp_barrier	67.634s	1.649s	62.657s	3.178s	0.150s	67.6337	65.017s	2.617s	
__kmp_fork_barrier	60.432s	0.134s	56.620s	3.320s	0.358s	60.4215	57.691s	2.730s	
square_ref	39.427s	0s	6.070s	1.450s	31.907s	0	0s	0s	
__kmpc_reduce_nowait	19.949s	0.550s	15.825s	3.394s	0.180s	19.6886	18.860s	0.829s	
square	4.453s	0.020s	0.542s	0.482s	3.410s	0	0s	0s	
genrand	2.819s	0s	2.819s	0s	0s	0	0s	0s	
basic	2.779s	0s	0.829s	0.651s	1.299s	0	0s	0s	
gen_graph	2.521s	0.010s	2.511s	0s	0s	0	0s	0s	
__intel_ssse3_rep_memcpy	2.435s	0.010s	2.184s	0.201s	0.040s	0	0s	0s	
deinfinite	0.511s	0s	0.511s	0s	0s	0	0s	0s	
infinite	0.380s	0s	0.380s	0s	0s	0	0s	0s	
__kmp_join_call	0.270s	0.010s	0.260s	0s	0s	0.260004	0.240s	0.020s	
__int_free	0.113s	0s	0.113s	0s	0s	0	0s	0s	
genrand	0.110s	0s	0.110s	0s	0s	0	0s	0s	

The output from the second version is shown here. Note that this was run with 16 threads.

Function	CPU Time	CPU Time				Serial	Spin Time		
		Idle	Poor	Ok	Ideal		OpenMP	Other	
__kmp_fork_barrier	0.520s	0.520s	0s	0s	0s	0.519948	0.520s	0s	
__kmpc_barrier	0.420s	0.147s	0.273s	0s	0s	0.420102	0.420s	0s	
square	0.130s	0s	0.120s	0.010s	0s	0	0s	0s	
__intel_ssse3_rep_memcpy	0.020s	0s	0.020s	0s	0s	0	0s	0s	
__kmp_barrier	0.010s	0s	0s	0.010s	0s	0.0100003	0.010s	0s	
genrand	0.010s	0.010s	0s	0s	0s	0	0s	0s	

3.2.2 Scaling Study

Although the scaling studies for the tuned parallel implementation times show similar speedup and efficiency patterns as the original solution, the overall time taken to achieve the solution is lower than the original OpenMP solution.

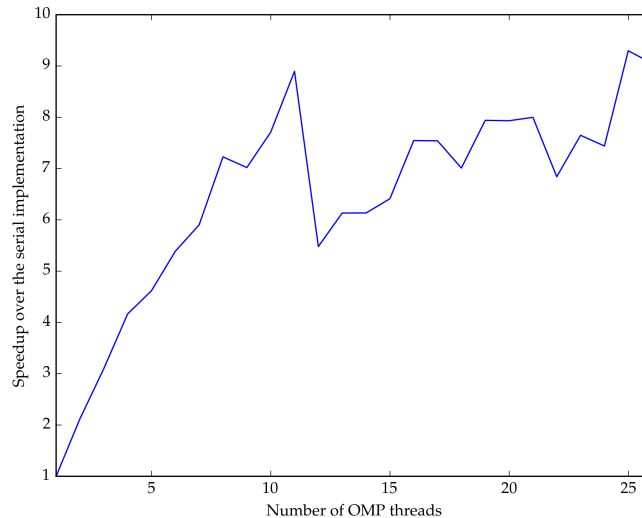


Figure 6: Strong scaling study of the tuned parallel solution

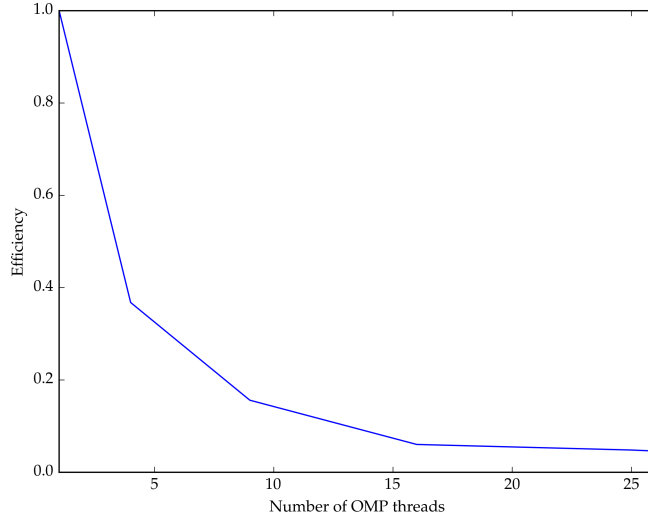


Figure 7: Weak scaling study of the tuned parallel solution

The strong scaling study was performed on a graph with 2048 nodes. The weak scaling varies the number of threads but keeps the problem size per processor be 512 nodes.

We then attempted to explore a different means of parallelization in which threads were assigned to much larger blocks based on their thread id, which could be obtained using the `omp_get_thread_num()` function. We obtained a significant speedup! For a boardsize of 2048 x 2048, our tuned implementation finished in 1.03249 seconds using 16 threads, whereas the naive finished in 11.0959, leading to a speedup factor of 11.

Looking at the vectorization report, the large majority of the computationally heavy calculations were being vectorized, resulting in a large speedup. However, we have not yet generalized this code and this can only run for some fixed problem sizes and fixed thread sizes. We plan to do this in the future. Therefore, we ran the strong scaling studying on a 2048 x 2048 graph. The weak scaling study varies the problem size but keeps the problem size per processor constant at a 512 x 512 board. We could only run it using a small number of threads: 1, 4, and 16 on the compute node.

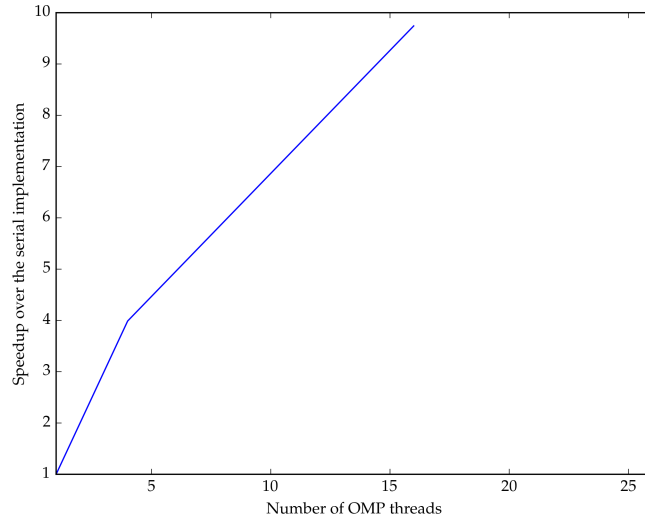


Figure 8: Strong scaling study of the tuned parallel solution

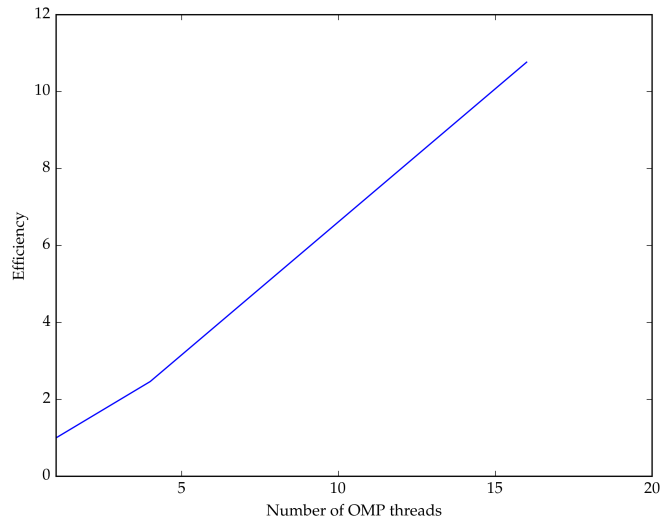


Figure 9: Weak scaling study of the tuned parallel solution

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>