CS 5220 Project 3 – Team 6 Final Report

Taejoon Song, Junteng Jia, Joshua Cohen
{ts693, jj585, jbc264}@cornell.edu

November 20, 2015

1 INTRODUCTION

The Floyd–Warshall algorithm, in computer science, is an alogorithm for finding the minimum patahs in a weighted graph. (i.e. All-pairs shortest path problem)
Our goal in this assignment includes:

1. Profiling: To find the bottleneck of the code.

2. Paralleization : To make it paralle by using MPI.

3. Optimization (Tuning) : Finally, tune it aggressively to get highest performance.

The rest of the report is organized as follows. In Section **??**, we introduce a baseline timing result from initial copy, and show our profiling result. Section 4 discusses our approach for parallelization-related work and Section 3 discusses vectorization. Our evaluation result will be shown in Section 5. Finally, Section **??** suggests what should be done more after peer reviews.

2 PROFILING

2.1 PROFILING

In order to find the bottlenecks of path, we first profiled our code using Intel's VTune Amplifier (It was broken in cluster, so we used it on local machie), as shown in Figure 1.

```
amplxe-cl -collect advanced-hotspots ./path
amplxe-cl -report hotspots -source-object function=<NAME>
```

Figure 1: VTune Amplifier Command

2.2 INITIAL PROFILING RESULT

Initial profile result can be found at Figure 2, and more detail in Figure 12.
We found that "square" function is the main bottleneck and do the most critical steps for this program, so this point is where we started to tune the code.
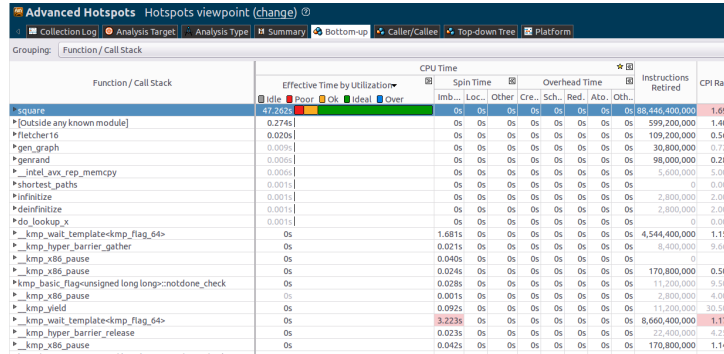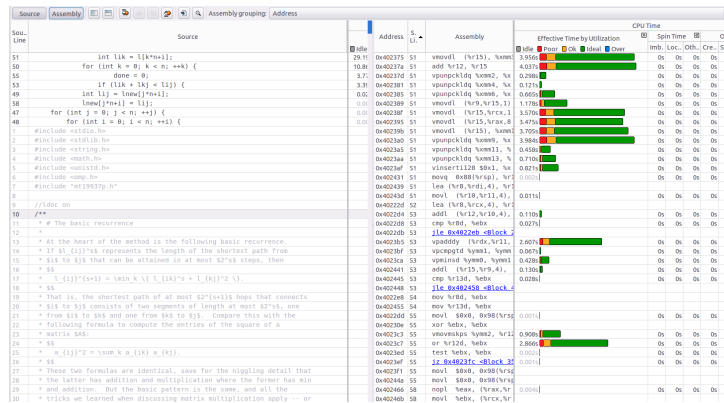

Figure 2: Initial Profile Analysis

Figure 3: Initial Assembly Result

2.3 INITIAL TIMING RESULT

Initial timing result is shown in Figure 4. As we can see the program is running with 8 threads using OpenMP and it takes 11.0818 seconds for 2000 vertices.

Figure 4: Initial Timing Result

3 VECTORIZATION

3.1 TIMING AFTER VECTORIZATION

In order to vectorize properly vectorize our code, we looked at the output of `ipo_out.optrpt` after compiling with flags `-qopt-report=5 -qopt-report-phase=vec`. We first were able to vectorize our call to `square` within `shortest_paths` within `path.c` by explicitly precomputing the transpose of `l` during each call to `square`, and then replacing the assignment of `lik` directly from `l`, as

$$\texttt{int lik = l[k*n+i]}$$

to an assignment instead from the transpose, as

$$\texttt{int lik = l\_T[i*n+k]}$$

We also attempted to solve the issue of unaligned memory access from within `l` and `l_T` by replacing calls to `malloc` with `_mm_malloc` (and, correspondingly, calls to `free` with calls to `_mm_free`), using a byte alignment of 32 since we're compiling with AVX2 (using the flag `-xcore-avx2`). This solved some of the issues with unaligned access, according to the vectorization report, but there are still cases with unaligned access reported.

3.2 TIMING AFTER VECTORIZATION

Figure 5: Profile Analysis After Vectorization

As show in the picture, for the same problem size, the function `square` is more than 5 times faster than it was before, when we are running it with in Vtune, where one core is used to trace other threads. In fact, running it with command line and time it with build in function `omp_get_time()` shows the saving gives us a factor of more than 9 times faster.
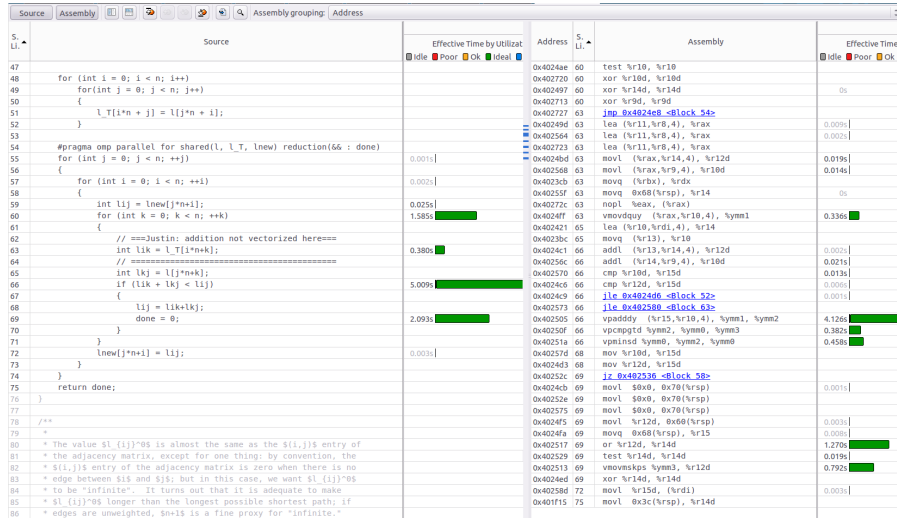

Figure 6: Timing Result After Vectorization

Figure 7: Assembly Analysis After Vectorization

The assembly analysis is more informative as it tells how well our functions have been vectorized as well as the timing. From the assembly analysis after vectorization, we can tell that the computations in the inner has been vectorized. Which is where the savings come from.

3.3 DATA-TYPE OPTIMIZATION

As we went through tests, we found that the maximum distance between two vertices is always below 20, even when we are calculating a 10000 by 10000 graph. Then it naturally follows that, we don't necessarily need a 4-byte int to store the distance information.

To fit more data into register so that we can carry out more operation per cycle, we used a prototype called ddt, which can be any data type such as long, short, and char. In fact, this gives us 30% saving when we are using char to store the distance between two vertices.

Notice in Figure 8, we printed out a variable called ddt_upper_range which tells us the largest distance we can have, in order to use one certain data-type. In the case with char, the largest distance is 127, which is proved to be more than enough by our tests.

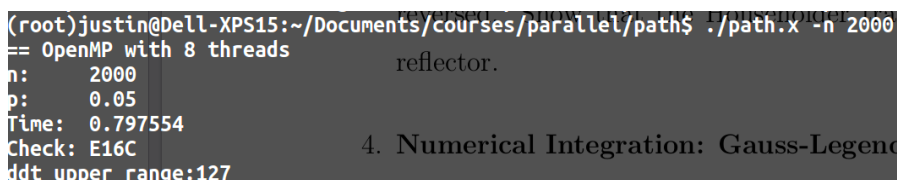## 3.4 TUNED OPENMP TIMING RESULT

Figure 8: Timing Result After OpenMP tuning

Timing result before introducing MPI is shown in Figure 8. As we can see the program is running with 8 threads using OpenMP and it takes 0.7976 seconds for 2000 vertices.
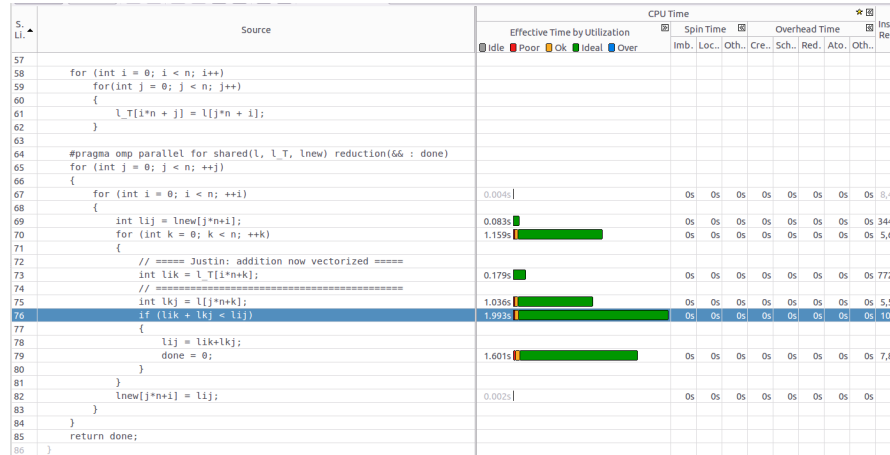
3.5 TUNED OPENMP PROFILING RESULT

Figure 9: Initial Assembly Result

The assembly analysis after using data-type char is down below. Comparing with Figure 7, we can tell that the major saving comes from the addition between lij and ljk, which consist with what we would expect, since we can now fit more additions into vector register to be calculated in one cycle.

4 PARALLELIZATION

4.1 OPENMP

Parallelization with OpenMP is already implemented in the original code. Within a single node, the jobs are statically distributed among different threads. Depending on the way of array accessing, we won't have problem with cache coherence, because we are reading from and writing to different memory locations.

4.2 BOARDCAST ALGORITHM

To make our code scales better, we implement an algorithm similar to broadcast algorithm for matrix multiplication. For details, see:

http://www.cs.berkeley.edu/ yelick/cs267-sp04/lectures/13/lect13-pmatmul-6x.pdf

Assuming we are working with number of ranks being a perfect square. Then, the procedure of our algorithm can be outlined as follows:

1. Generate the random graph from rank0.

2. On rank0, partition the whole matrix into a 2-d array of blocks. Each rank will be in charge of one block.

3. Rank0 send the information about the whole matrix to different processors.

4. Calculate the shortest path:

   a) MPI ranks within the same column share their blocks.

   b) MPI ranks within the same row share their blocks.

   c) Calculate the shortest path by the "square" method along the cols and rows. The "done" variable is determined by the "logical_and" operation.

   d) Block until all the MPI ranks return, then use MPI_reduce to decide whether the iteration has finished.

5. Once all the job has been finished, gather the updated matrix back to rank0.

4.3 MPI IMPLEMENTATION

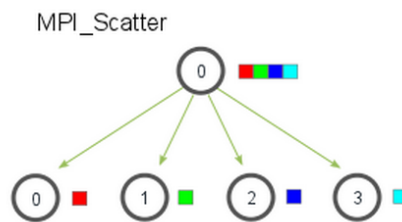1. To distribute blocks to all MPI ranks, MPI_Scatter method is used.

Figure 10: Initial Assembly Result

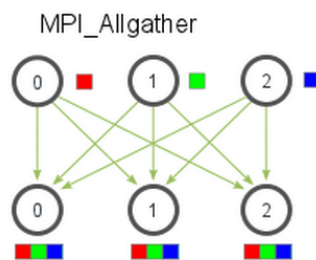2. During the calculation of the whole graph, Ranks in the same Column/Row are sharing blocks through MPI_Allgather.

Figure 11: Initial Assembly Result

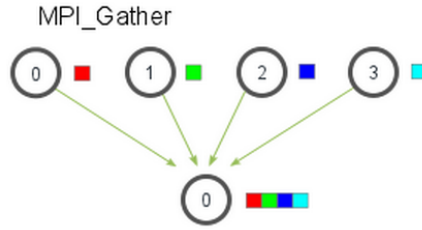3. To gather the final result to rank 0, MPI_Scatter method is used.

Figure 12: Initial Assembly Result
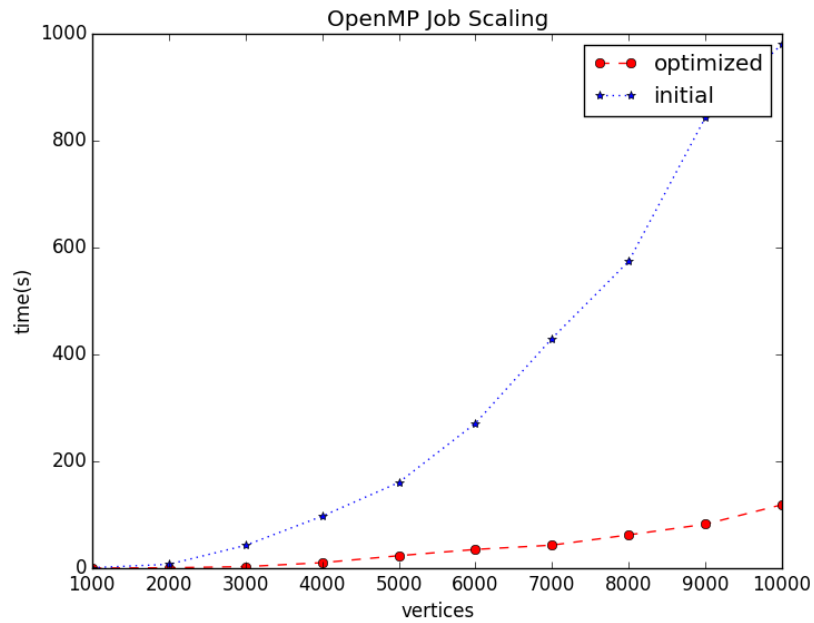
5 EVALUATION

5.1 OPENMP SCALING

Figure 13: OpenMP Job Scaling with 1 Node - 24 Threads

The picture shows the scaling paradigm of the original code and optimized OpenMP code. From this picture, we can see there is a factor of 10 improvement after our optimization. This picture also gives information about the scaling of the calculation. Non-linear regression fit of this curve gives a result of $O(N^{3.22})$.

In theory, this algorithm should have a $O(N^3)$ scaling with respect to one single iteration. The result different from the idea case because for a smaller graph, it can be fit into cache.
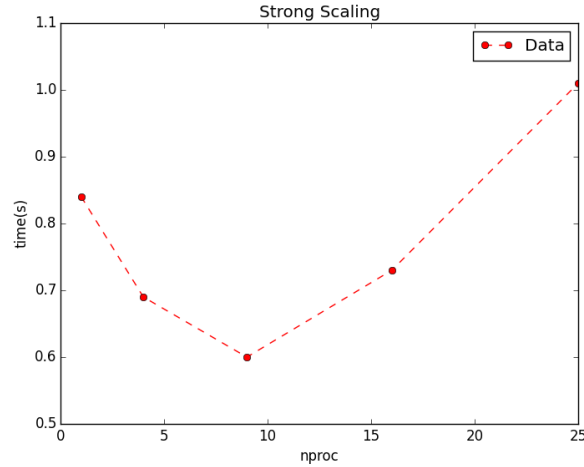
5.2 MPI SCALING

Figure 14: MPI Strong Scaling Naive

This is our first implementation of MPI code, tested with 2000 by 2000 graph. This strong scaling diagram shows the communication overhead is very big, and it's performance is no better than serial code at very large number of ranks.

5.3 NON-BLOCKING COMMUNICATION

Notice that we don't need the whole column and row of the matrix to begin the shortest path calculation, we just need the first column block and first row block, then we can do the calculation. So we implemented a new version of MPI based on non-blocking communication MPI_Ibcast. The procedure is as follows:

1. The rank owning the first column block or row block do a blocking board-cast.

2. The rank owning the second column block or row block do a non-blocking board-cast.

3. Wait for the first column block and row block.

4. Update the graph based on the first column block and row block.

5. The rank owning the third column block or row block do a non-blocking board-cast.

6. Wait for the second column block and row block.

7. Go to the 4th step.
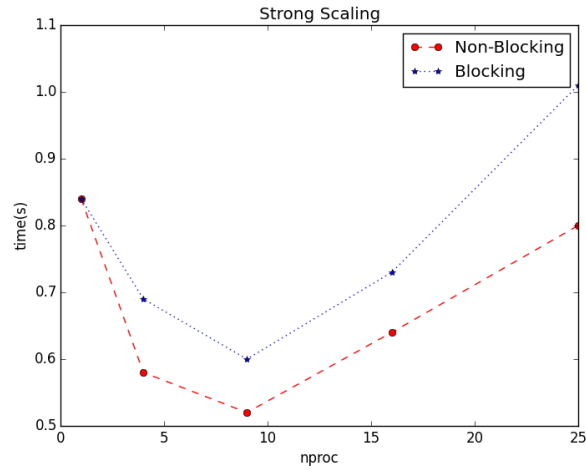
After the optimization, we can see improvement.

Figure 15: MPI Strong Scaling Non-Blocking

5.4 MPI WEAK SCALING

Since this algorithm is not linear scaling, it is hard to define weak scaling, because the problem size grows $O(N^3)$. The data tested is listed below:

	1	4	9	16	25
2000	0.84				
4000	5.74	2.79			
6000	13.30		16.40		
8000	23.59			68.91	
10000	33.18				162.419

From this graph, we can calculate the speed up factor represented in the following graph.
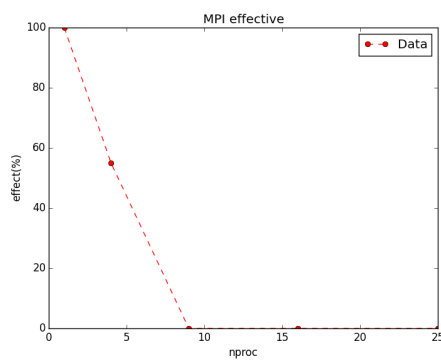

Figure 16: MPI Effective

9

From the result above, MPI is not used effectively. We think one reason is that the calculation of one single job is too small, because this algorithm uses additions to update the graph. So the fraction of Communication vs Calculation will be really high.

6 SUMMARY

In this project, we did the following:

1. Profiled to find the hotspots, look into the vectorization report.

2. Optimized our code based on our finding, get a factor of more than 10.

3. Successfully parallel our code through MPI, board-cast algorithm, get the right answer.

4. Use non-blocking communication to speed up the MPI code.

5. Build up a mode to analysis the performance of our MPI code in terms of strong scaling and MPI efficiency.

From our result, we learned:

1. Non-blocking communication can help speed up the program, but it might not be that ideal.

2. When the size of a job is not large enough, the efficiency of MPI calculation maybe influenced.