HW 3: All-Pairs Shortest Path (CPU) Report

106033233 資工21 周聖諺

1. Implementation
   (1) Design of The Testcase:
      (a) Dense & Huge Graph: It is a dense graph in which the 95% of the pairs of the elements of the matrix have edges.
      (b) Shortest Path in the Last: The last node of the graph has the shortest path to all the other nodes. As a result, the program should scan the whole graph.
   (2) Design of The Algorithm
      (a) Algorithm: Block Floyd Warshall, it divides the matrix into n * n submatrice and each of the thread handles one matrix. It increases the locality of the memory access pattern and reuse the data in the cache as many times as possible.
      (b) Vectorization: Use SSE2 to parallel 4 floats in a time.
      (c) Store graph in 1D array: It only needs to allocate the whole memory once.
      (d) Read / Write buffer: Use *fread()* function to read the whole file into a buffer. Use *fwrite()* function to write and flush the whole buffer into the file.
      (e) Build graph in parallel: Translate the data of the buffer into a graph in a dense matrix with multi-threads.
      (f) Dependency Graph(Fail): Actually if we check the dependency of each block of the graph, you can see each block only depends on several specific blocks. Once dependent blocks are finished, the blocks can be executed. Thus it should be faster. However, it is hard to design the dependent mechanism. I gave up.
2. Other Implementation
   (1) Sliding Window Floyd Warshall:

   I've tried to parallel the Floyd Warshall with the sliding window. That is, for sequential Floyd Warshall, it iterates k, i, j in sequence and the sliding window parallel iterates the j index in parallel. It assigns a segment for each thread along the j index in sequence. It can also keep the locality of the memory access and it is also more friendly for cache which always accesses contiguous memory. As a result, it shouldn't be slow. But in my implementation, it is still slower than Block Floyd Warshall, even the sequential one.

A possible reason for this situation is that the Block Floyd Warshall access memory and reuse it much more times than the Sliding Window version.

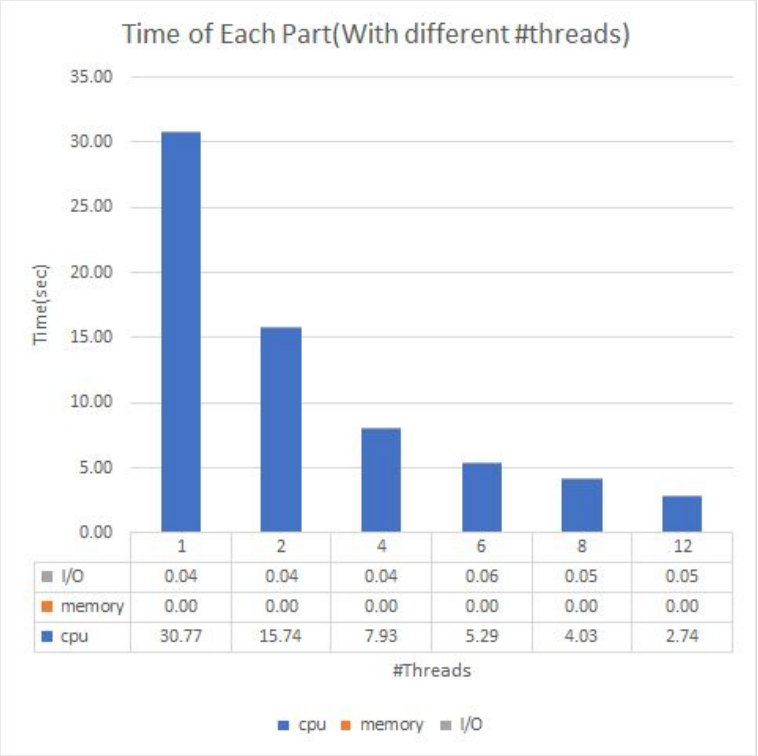
3. Experiment & Analysis

(1) System Spec

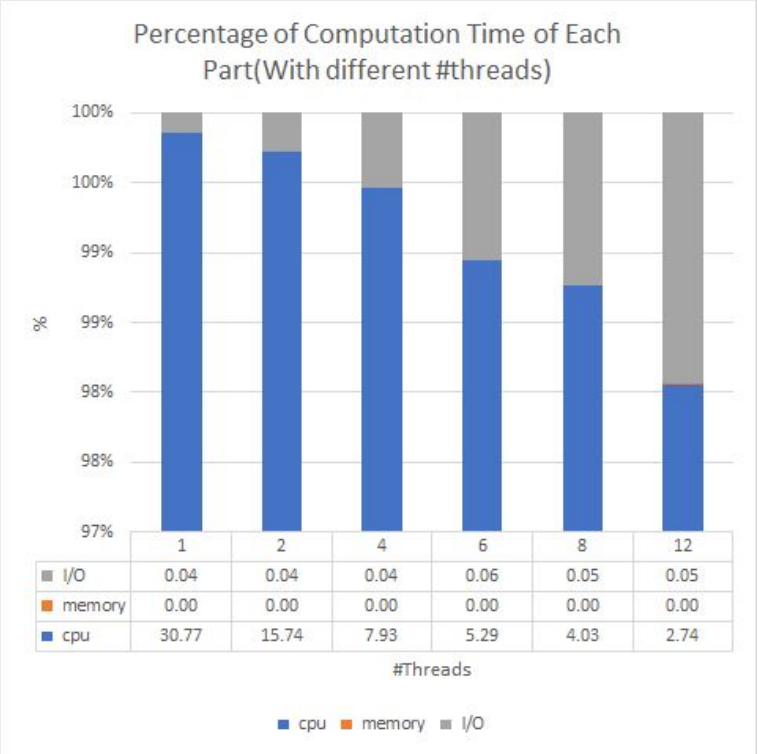
Apollo

(2) Strong scalability Test

(a) Execution Time of each part

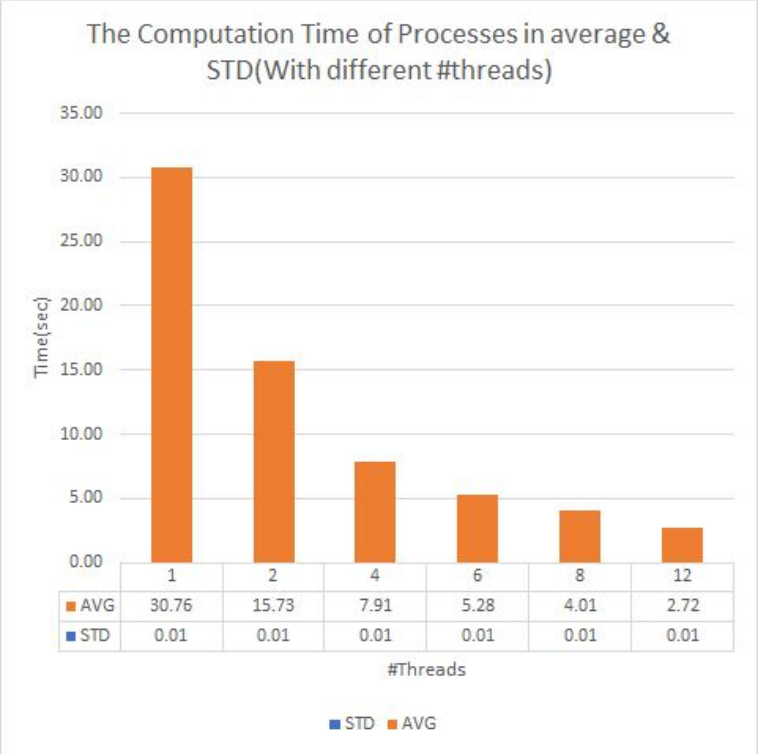


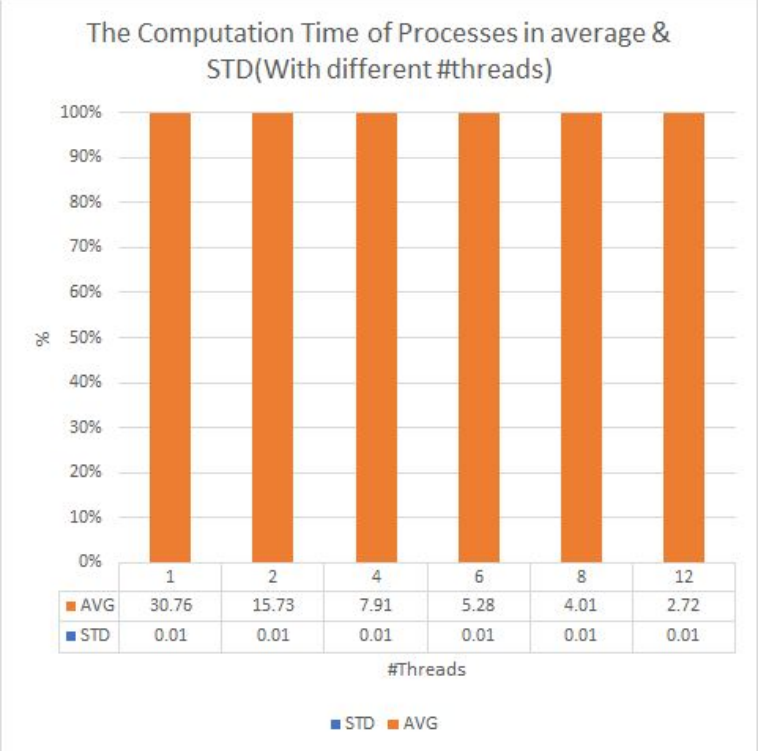
	1	2	4	6	8	12
I/O	0.04	0.04	0.04	0.06	0.05	0.05
memory	0.00	0.00	0.00	0.00	0.00	0.00
cpu	30.77	15.74	7.93	5.29	4.03	2.74



This assignment is a CPU bound task as the figures shown above. The CPU always takes the most time. The time of CPU execution occupies over 98% of the total execution time.

(b) Loading Balance

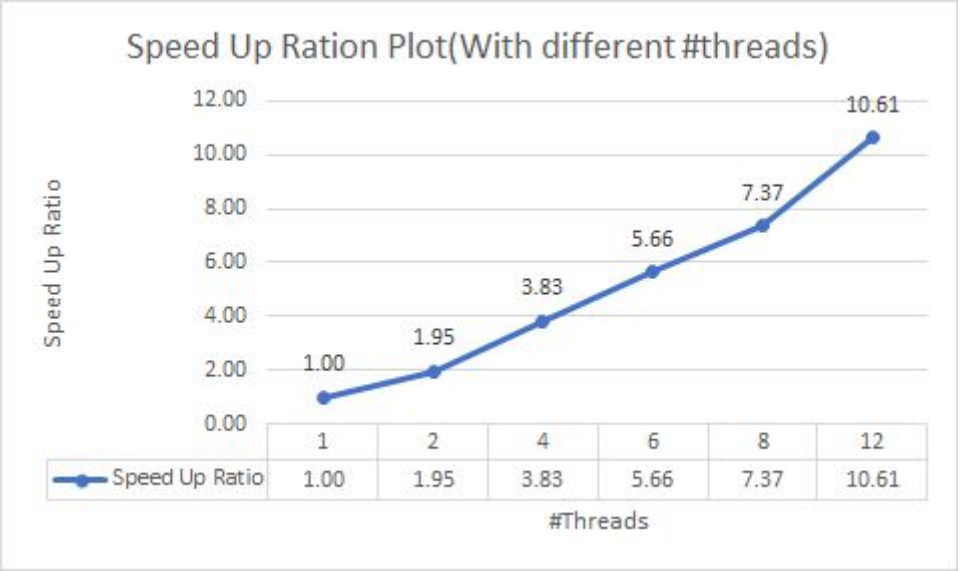




Here I calculate the Average and the Standard Deviation of the execution time of each thread. As the figures above, the standard

deviation is so small that we can almost ignore it. As a result, the load balancing is quite well.

(3) Speed Up Ratio Plot



(4) Compiler Tests

Here is the table of the total time that I ran hw3-judge with different compilers.

	1	2	3
g++ (in seconds)	35.72	35.45	35.76
clang++ (in seconds)	24.20	24.49	24.24

Clang++ is much better than g++

(5) Vectorization Speed Up Tests

Here is the table of the total time that I ran hw3-judge with SSE2 vectorized and non-vectorized versions. Both are compiled by g++ with -O3 optimization.

	1	2	3
Vectorization (in seconds)	35.72	35.45	35.76
Non-Vectorization (in seconds)	66.65	67.11	66.25

It is trivial that the code with vectorization is nearly 2 times faster than non-vectorization.

4. Experience & conclusion

This assignment is quite tricky that you should search for the paper of Block Floyd Warshall. Otherwise, you wouldn't know that optimize cache is the key of this assignment. It is also the first time that I try to compile my code with clang++. It is much and much faster than the g++. It saves almost 10 sec without any effort. I am quite happy with that.