Performance comparison of OpenMP and OpenACC in Floyd Warshall Algorithm

Shengyuan Wang, Kaiyang Yao
COMP 445 - Parallel/Dist Processing

December 16, 2022

Abstract

Floyd-Warshall algorithm is a shortest path algorithm developed by a United States famous computer scientist R.W. Floyd in 1962. This paper presents the basic concept of this algorithm and its performance analysis. Then it compares three different parallel architectures – OpenMP, OpenACC-GPU, and OpenACC-CPU(Multicore) – for faster problem solving process.

1 Introduction

There are several famous shortest path algorithms, such as A*, Kruskal's, Dijkastra's, Bellman-Ford, Floyd-Warshall, etc. Floyd-Warshall is specialized in directed, weighted graphs. Although it is similar to the Dijkastra's algorithm that takes an assigned starting vertex and find the shortest distance from this vertex to the rest, Floyd-Warshall finds the shortest path between any two vertices, also known as the multi-source problem.

2 Algorithm

Floyd-Warshall algorithm utilizes the dynamic programming approach based on a concept of intermediate vertices. Specifically, it finds the distance between all pairs of vertices and checks if adding a new intermediate vertex could shorten the distance. For a given graph $G(V, E)$, we create a $|V| \times |V|$ grid. The element (i, j) in the grid indicates the shortest distance from vertex i to vertex j. The core formula, or state transition equation in terms of dynamic programming, is $dist = min(dist[i, j],\ dist[i, k] + dist[k, j])$. This formula means each time we find a shorter path between i and j via an intermediate vertex, we will update the distance. Otherwise, we will skip it and find other intermediate vertices[9]. After finding through all the possible vertices, the result of the grid is the shortest distance between any two pairs. The correctness of this algorithm can be proofed by mathematical induction, which is beyond the scope of this paper.

3 Example

For the graph below, we want to use Floyd-Warshall algorithm to find the shortest path among all pairs of vertices.

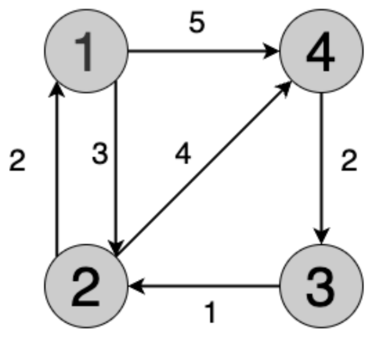
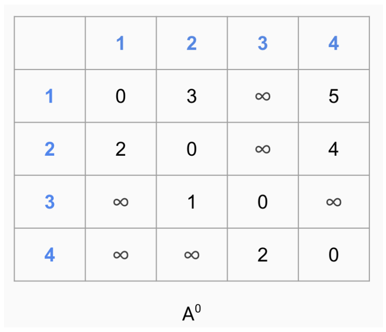


Figure 1: Graph Example

Initial Step

First, it is noted that the graph has 4 vertices, so we create a 4×4 matrix to store the initial distance. In the matrix, the cell (i, j) indicates the shortest distance from vertex i to vertex j at current step. So for the initial state, we have three different ways to fill the cell. First, if i and j are the same vertex, the distance should be 0. Thus, we set all cells in the main diagonal (top left to bottom right) to 0. Second, if i and j are different and there is a direct edge from vertex i to vertex j, then set cell (i, j) to the weight of that direct edge. Third, if there is no direct edge from i to j, we then set cell (i, j) to ∞, which means it is impossible to find a distance.



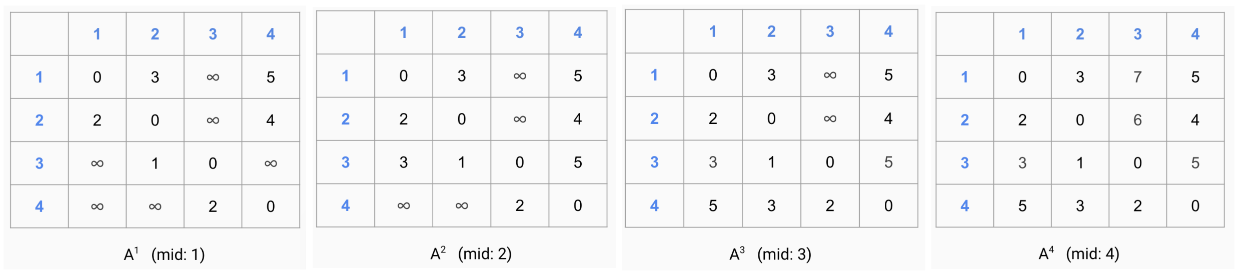
	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	∞	1	0	∞
4	∞	∞	2	0

A^0

Figure 2: Initial Matrix

Iterative Step

Next, we move on to the core part of the algorithm. For each of the iteration, we choose a new intermediate point k. Then we copy a new matrix A^n from A^{n-1} and update A^n based on the distance by passing point k. Recall that the update rule is to compare $A^{n-1}_{i,j}$ with $(A^{n-1}_{i,k} + A^{n-1}_{k,j})$. If the later is smaller, then we update $A^n_{i,j}$ with the new smaller value.



	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	∞	1	0	∞
4	∞	∞	2	0

A^1 (mid: 1)

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	3	1	0	5
4	∞	∞	2	0

A^2 (mid: 2)

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	3	1	0	5
4	5	3	2	0

A^3 (mid: 3)

	1	2	3	4
1	0	3	7	5
2	2	0	6	4
3	3	1	0	5
4	5	3	2	0

A^4 (mid: 4)

Figure 3: Iterative Matrix

Code Example

```
for(int k = 0; k < N; k ++) {
    for(int i = 0; i < N; i ++) {
        for(int j = 0; j < N; j ++) {
            int i0 = i*N + j;
            int i1 = i*N + k;
            int i2 = k*N + j;
            if(mat[i1] != -1 && mat[i2] != -1){
                  int sum =  (mat[i1] + mat[i2]);
                  if (mat[i0] == -1 || sum < mat[i0])
                     mat[i0] = sum;
            }
        }
    }
}
```

4 Parallel Architectures

It is important to note that we have an $O(n^3)$ for loop in the sequential version, which is undesirable in performance when the data size is large. In the rest of the paper, we will consider improvements on the giant for loop by using parallel architectures. While these architectures utilize different techniques in implementations, they all parallelize the code so that multiple instructions are run synchronously.

4.1 OpenMP

OpenMP is a shared-memory multithread architecture, in which a primary thread forks a specified number of sub-threads and they work together to divide tasks. The updated code is shown below.

```
    void Floyd_Warshall(int* matrix, int size) {
    int *row_k = (int*)malloc(sizeof(int)*size);

    #pragma omp parallel default(none) shared(row_k)
    for (int k = 0; k < size; k++) {
        #pragma omp master
        memcpy(row_k, matrix + (k * size), sizeof(int)*size);
        #pragma omp for schedule(static)
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                if (matrix[i * size + k] != -1 && row_k[j] != -1) {
                    int new_path = matrix[i * size + k] + row_k[j];
                    if (new_path < matrix[i * size + j] || matrix[i * size + j] == -1)
                        matrix[i * size + j] = new_path;
                }
            }
        }
    }
}
```

First, we added `#pragma omp parallel` at the outermost for loop to create a parallel region on the entire nested for loop. This indicates that the code uses the multiple instruction / multiple data (MIMD) CPU architecture as well as the thread pool shared memory hardware. Then we used `#pragma omp master` for the `memcpy` instruction, which means it will be run only by the master thread. This reflects the use of collective synchronization parallel pattern. Moreover, we added `#pragma omp for`, which indicates the use of the parallel for loop program structure. Inside the loop, the program will decompose the data among each thread and join them together when the loop is done, a process reinforcing the fork-join structure. Note that `schedule(static)` specifies that the for loop has the static scheduling type. OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread. This is the data decomposition parallel strategy.

4.2 OpenACC-CPU Multicore

Next, we tried the parallel version using the OpenACC standard. In the CPU multicore version, we added `#pragma acc parallel loop` to declare the loop below is parallel. Since OpenACC assumes all variables to be shared by default, we don't need to explicitly declare the shared variables `row_k` and `k`.

```
void Floyd_Warshall(int* matrix, int size) {
    int *row_k = (int*)malloc(sizeof(int)*size);

    for (int k = 0; k < size; k++) {
        memcpy(row_k, matrix + (k * size), sizeof(int)*size);
        #pragma acc parallel loop
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                if (matrix[i * size + k] != -1 && row_k[j] != -1) {
                    int new_path = matrix[i * size + k] + row_k[j];
                    if (new_path < matrix[i * size + j] || matrix[i * size + j] == -1)
                        matrix[i * size + j] = new_path;
                }
            }
        }
    }
}
```

In terms of the parallel pattern, CPUs with multicores are examples of MIMD architecture. The implementation shows the Parallel For Loop, Single Program Multiple Data(SPMD) program structure and Shared Data data structure. Moreover, Data Decomposition is used as the parallel algorithm strategy.

4.3 OpenACC-GPU

Finally, we run the GPU version under the OpenACC standard. We added `#pragma acc kernels` so that the for loop will be transformed into kernel functions that can be executed on many cores of the GPU. Then we added `#pragma acc loop independent` to declare that the loop below is independent among cores. Note that this have to be done one more time before the inside loop to make the loop truly independent.

```
void Floyd_Warshall(int* matrix, int size) {
    int *row_k = (int*)malloc(sizeof(int)*size);
    #pragma acc kernels
    #pragma acc loop independent
    for (int k = 0; k < size; k++) {
        memcpy(row_k, matrix + (k * size), sizeof(int)*size);
        #pragma acc loop independent
        for (int i = 0; i < size; ++i) {
            #pragma acc loop independent
            for (int j = 0; j < size; ++j) {
                if (matrix[i * size + k] != -1 && row_k[j] != -1) {
                    int new_path = matrix[i * size + k] + row_k[j];
                    if (new_path < matrix[i * size + j] || matrix[i * size + j] == -1)
                        matrix[i * size + j] = new_path;
                }
            }
        }
    }
}
```

In terms of the parallel pattern, GPU is an example of the SPMD architecture. The implementation reflects a Shared Data data structure and Data Decomposition parallel algorithm strategy.

5 Result

We did a holistic test on the performance of four different versions of this algorithm: sequential, OpenMP, OpenACC-CPU Multicore, and OpenACC-GPU. The problem size for these tests is the number of vertices, because it is the only variable that influences the algorithm complexity as well as the size of our matrix.

5.1 Scalability Test

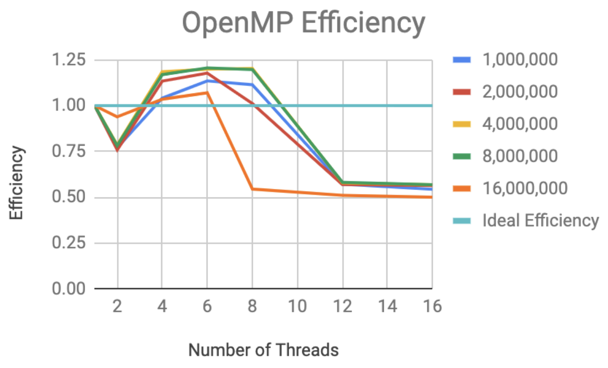
First, we did the scalability test on OpenMP and Multicore to check how good do they in handle increasing problem size.

OpenMP

In strong scalability test, we tried 1, 2, 4, 6, 8, 12, 16 different threads on problem sizes 1000, 1400, 2000, 2800, 4000. In weak scalability test, we created 5 scale lines starting from 250000, 500000, 100000, 200000, 400000 with 1 thread. Then for each line, we doubled the size and doubled the threads for 5 times.

To generate the data according to the strong scalability test method in OpenMP, we can go to `./OpenMP` and run the following instruction.

```
make
bash run_strong_tests.sh $repeat_time
bash run_weak_tests.sh $repeat_time $start_size $num_line
```



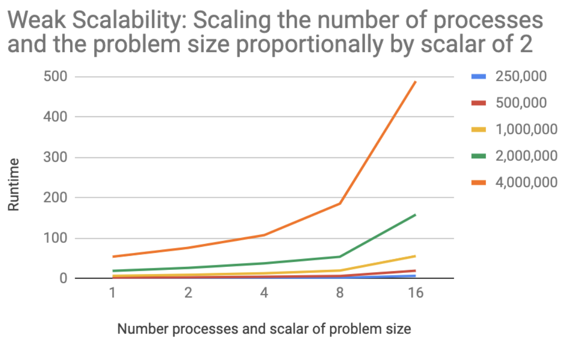


Figure 4: OpenMP Scalability Test (Left : Efficiency Right : Weak Scalability)

In strong scalability test, a efficiency greate than 75% is considered strong. Our OpenMP version has a good scalability from 2 to 8 threads, with the middle range even better than the ideal efficiency. When the thread number change to 10 and above, our program lose the strong scalability.

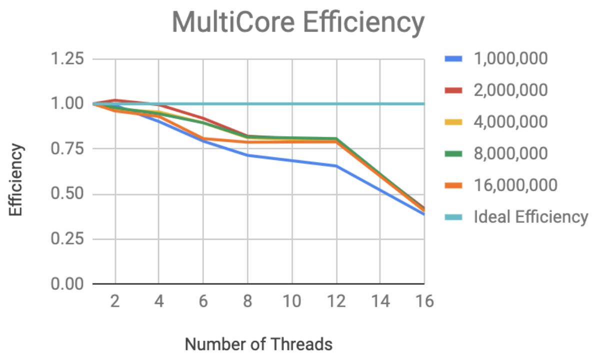
In weak scalability test, all lines excepts the top one (with the largest problem size) suggests a good weak scalability within 8 threads threshold. This indicates that for problem sizes less than 2000000 and threads number less than 8, our OpenMP solution performs good weakly scalablity. However, when the problem size is larger or the threads number turns to 16, the performance is not ideal. A reasonable speculation is that add more threads after 8 negatively effect the performance because forking/joining threads and communicating between threads create extra time that outweights the time saved.

Multicore

In the multicore version, we did the strong and weak scalability tests with the same data as we used in the OpenMP.
To generate the data according to the strong scalability test method in OpenMP, we can go to `./MultiCore` and run the following instruction.

```
make
bash run_strong_tests.sh $repeat_time
bash run_weak_tests.sh $repeat_time $start_size $num_line
```

We get a similar set of result. In the strong scalability test, we have a good efficiency from 2 to 8 threads. The efficiency drops when the number exceeds 8. For weak scalability test, the program is weakly scalabl with threads number less than 8 as well.



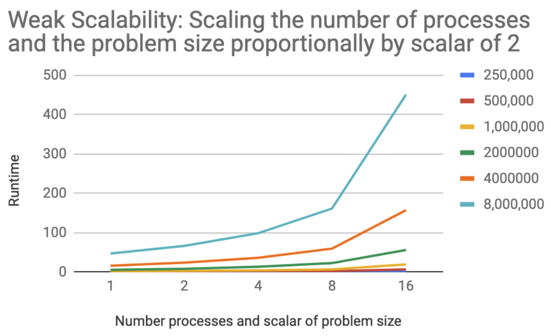


Figure 5: Multicore Scalability Test (Left : Efficiency   Right : Weak Scalability)

5.2 Run Time Comparison

Since we can not set thread number for the GPU device, we need to find an alternative way to test the performance. One way to do is to fix the thread number of OpenMP and CPU Multicore as 8 and test their run time together with the GPU and Sequential. We tried the problem size from 50×50 to 6400×6400, with each number scale by approximately $\sqrt{2}$ each time. Note that this will let the total problem size be doubled. We compare the running time under different parallel architecture in the following Table (1). Also, to be more clear and straightforward, we make Table (2) to calculate speed up rate with the running time of sequential version as benchmark.

size	50x50	71x71	100x100	141x141	200x200	282x282	400x400	564x564	800x800	1128x1128	1600x1600	2256x2256	3200x3200	4512x4512	6400x6400
Sequential	0.003582	0.008782	0.024708	0.053147	0.091291	0.163811	0.605895	1.539944	3.868914	10.706321	29.06987	85.293091	267.105098	772.953232	2109.134367
OpenMP	0.001082	0.001827	0.003834	0.008233	0.027712	0.050268	0.092047	0.204183	0.844486	2.49502	7.091797	19.717603	56.091517	155.908222	426.455864
MultiCore	0.00407	0.003815	0.004507	0.005951	0.010765	0.020807	0.04098	0.076662	0.172534	0.566916	1.478288	3.765112	11.364483	32.018555	89.444232
OpenACC	0.001697	0.002026	0.003281	0.00691	0.013154	0.02513	0.04919	0.09546	0.169452	0.315167	0.456212	0.819619	1.395318	2.547093	5.489022

Table 1: Running Time

size	50x50	71x71	100x100	141x141	200x200	282x282	400x400	564x564	800x800	1128x1128	1600x1600	2256x2256	3200x3200	4512x4512	6400x6400
Sequential	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
OpenMP	3.310536044	4.806787083	6.444444444	6.455362565	3.294276848	3.258753083	6.582452443	7.541979499	4.581383232	4.291076224	4.099083772	4.325733255	4.761951758	4.957745153	4.945727202
MultiCore	0.8800982801	2.301965924	5.482138895	8.930767938	8.480352996	7.872879319	14.78513909	20.0874488	22.42406714	18.88519816	19.66455116	22.65353355	23.50349752	24.14079061	23.58044023
OpenACC	2.110783736	4.334649556	7.530630905	7.691316932	6.94017029	6.518543573	12.31744257	16.13182485	22.831917	33.97031098	63.72009066	104.0643165	191.4295508	303.4648645	384.2459307

Table 2: Speed-up Rate

From both Table(1) and Table(2), we can find that when problem size is smaller than $20,000$, all three parallel architecture performs about the same in speeding up the algorithm. As problem size increases, GPU and Multicore versions gradually perform their fast processing speed compared with OpenMP version. As for tasks with problem size more than $640,000$, GPU version perform far better than Multicore version.

6 Future Work

First, we are interested in making a fancy animation on the run time of the four approaches we mentioned above with different data sizes. The animation will make the comparison more visually appealing and will highlight the threshold moment when one approach overruns the other in speed. Moreover, we intend to explore more shortest path algorithms, like Dijkstra's or A*, to parallelize them and compare the performance with that in Floyd Warshall Algorithm.

7 Conclusion

In this paper, we introduced the famous Floyd-Warshall shortest path algorithm with a detailed example. Then we explored four different ways to implement the algorithm, one sequential approach and three parallel approaches. In the parallel implementations, we used the OpenMP standard as well as the OpenACC standard on both CPU and GPU. Next, we presented what modifications we made in those versions based on the sequential code and what parallel patterns our code follows. Finally, a holistic comparison is made among all approaches. The comparison includes the scalability of OpenMP and OpenACC-CPU and the execution time of all four approaches.

References

[1] Rajashri Awari. Parallelization of shortest path algorithm using openmp and mpi. pages 304–309, 2017.

[2] Hristo Djidjev, Sunil Thulasidasan, Guillaume Chapuis, Rumen Andonov, and Dominique Lavenier. Efficient multi-gpu computation of all-pairs shortest paths. pages 360–369, 2014.

[3] Ben Lund and Justin W Smith. A multi-stage cuda kernel for floyd-warshall. *arXiv preprint arXiv:1001.4108*, 2010.

[4] Nudžejma Pozder, Dalila Corovic, Esma Herenda, and Belmin Divjan. Towards performance improvement of a parallel floyd-warshall algorithm using openmp and intel tbb.

[5] Piyush Sao, Hao Lu, Ramakrishnan Kannan, Vijay Thakkar, Richard Vuduc, and Thomas Potok. Scalable all-pairs shortest paths for huge graphs on multi-gpu clusters. pages 121–131, 2021.

[6] Daisuke Takafuji, Koji Nakano, and Yasuaki Ito. Efficient parallel implementations to compute the diameter of a graph. *Concurrency and Computation: Practice and Experience*, page e5963, 2020.

[7] Edvin Teskeredžić, Kenan Karahodžić, and Novica Nosović. Comparison of the non-blocked and blocked floyd-warshall algorithm with regard to speedup and energy saving on an embedded gpu. pages 1–5, 2020.

[8] Quoc-Nam Tran. Designing efficient many-core parallel algorithms for all-pairs shortest-paths using cuda. pages 7–12, 2010.

[9] Eric W Weisstein. Floyd-warshall algorithm. *https://mathworld. wolfram. com/*, 2008.

[10] Li-yan Zhang, Ma Jian, and Ke-ping Li. A parallel floyd-warshall algorithm based on tbb. pages 429–433, 2010.