Parallel Floyd's Algorithm for solving All-Pairs Shortest Path Problem Final Report

Zhao Chen zhaoc2@andrew.cmu.edu
Jianlin Du jianlind@andrew.cmu.edu

Summary:

This project parallelizes Floyd Algorithm to solve all-pairs shortest path problem with OpenMP & MPI and compare the performance.

Background:

Shortest Path Problem is an important part in graph theory. In contrast to single source shortest paths problem, for example Dijkstra's algorithm, all-pairs shortest path problems calculate the shortest path between all pairs of nodes in the graph. The Floyd algorithm solves the All-Pair-Shortest-Paths problem for directed graphs. A sample pseudocode is as follows.

```
1    func Floyd_All_Pairs_SP(A) {
2        D^(0) = A;
3        for k := 1 to n do
4            for i := 1 to n do
5                for j := 1 to n do
6                    d_{i,j}^(k) := min(d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1))
7     }
```

Figure 1: Pseudocode of Floyd Algorithm

As shown in the pseudocode, the sequential solution is a straightforward O(V^3) time complexity algorithm(V is the number of vertices), looping through all the vertices. This leaves some possibility for parallelization. We can explore the loops and try to divide the work into processors.

Challenges:
One challenge is associated with the nature of shortest path problem. When the program visits node after node to find the shortest path, the steps actually have strong sequential dependency, which means the parallelization is kind of limited. However, one of the reasons we are interested in the all pairs shortest path algorithm instead of single source shortest paths problem is that we hope to explore more parallelization in the process of calculating different pairs and to find potential data reuse and redundancy to accelerate the algorithm.
Another challenge is a common issue in parallelizing graph problems. If we divide the vertices into several processors, when different processors update the information of vertices, the information needs to be broadcast to other processors because they are sharing the same graph. This may require a lot of communications especially using message passing models.

Graph Representation

Adjacency Matrix is used to represent directed graph in this project. The direct path length from vertex i to vertex j is stored as a[i, j].
For a graph with N vertices, an N x N adjacency matrix keeps all edge info. If there is no direct path from vertex i to vertex j, the value can be infinity.
An example of adjacency matrix with 4 vertices are shown in the figure.

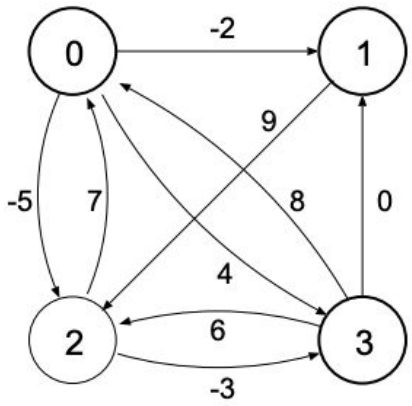

Figure 2 : Directed Graph

$$\begin{bmatrix} 0 & -2 & -5 & 4 \\ \infty & 0 & 9 & \infty \\ 7 & \infty & 0 & -3 \\ 8 & 0 & 6 & 0 \end{bmatrix}$$

Figure 3: Adjacency Matrix

MPI Approach:

- **Decomposition**

Divide matrix a into n*n elements
● Each a[i,j] is a task, representing the shortest distance between two vertices
● Finding the distance requires data from a[i,k] and a[k,j] for all k

Row-wise block decomposition: save broadcast within rows
Column-wise block decomposition: save broadcast within columns
Choose Row-wise because it could have better memory locality

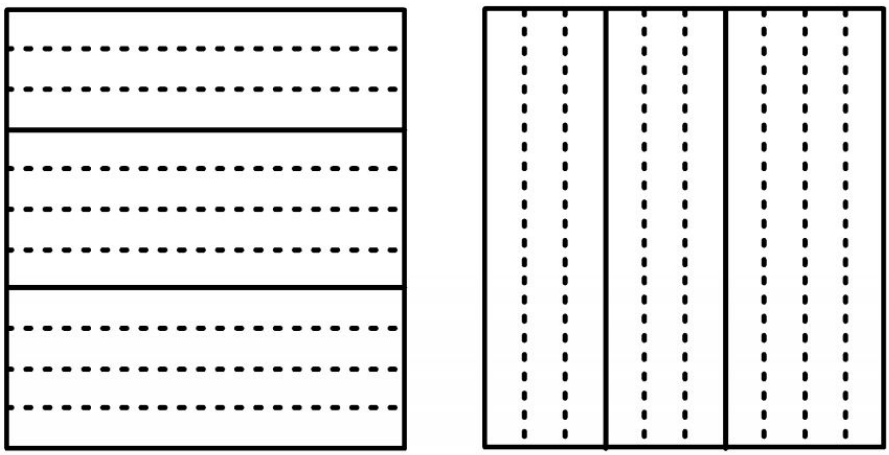

Figure 4: matrix decomposition

- **Source of Parallelism**

During the k'th iteration, the work is

```
for (i = 0; i <  n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = min(a[i][j], a[i][k] + a[k][j]
```

Note that a[i][k] = min(a[i][k], a[i][k] + a[k][j] will remain the value in a[i][k]
Note that a[k][j] = min(a[k][j], a[k][k]+ a[k][j] will remain the value in a[k][j]
k'th column and the k'th row remain the same during the k'th iteration

- **Communication**

During the k'th iteration, update a[i, j] needs values of a[i, k] and a[k, j]
- broadcast a[k, j] to a[0, j], a[1, j], …, a[n - 1, j]
- broadcast a[i, k] to a[i, 0], a[1, j], …, a[n - 1, j]

For example,
a[1, j] will be broadcast to a[0, j], a[2, j], a[3, j], a[4, j]
a[i, 1] will be broadcast to a[i, 0], a[i, 2], a[i, 3], a[i, 4]
Examples of communication are illustrated in the figures

The task of updating a[3, 4] needs a[3, 1] and a[1, 4] when k = 1

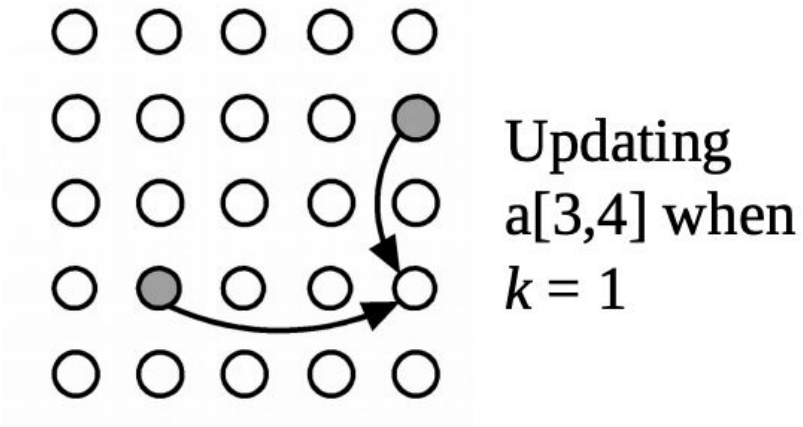

Figure 5: Communication Example1

Broadcast a[k, j] to a[0, j], a[1, j], …, a[n - 1, j]

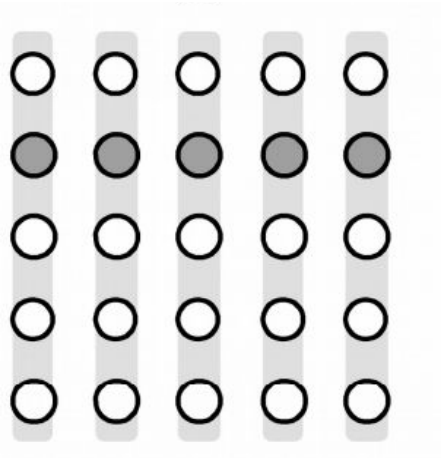

Figure 6: Communication Example2

Broadcast a[i, k] to a[i, 0], a[1, j], …, a[n - 1, j]

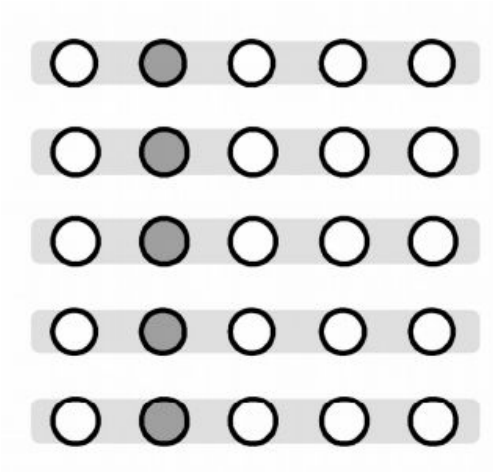

Figure 7: Communication Example2

Blocked OpenMP Approach:

- **Data Dependency**

For OpenMP approach, we partition the workload into blocks with size b x b, so that each block could be assigned to a thread and the cache locality is also taken advantage of. Here, we do not need to worry about communication, but data dependency exists between blocks, and the execution need to be divided into 4 phases. Below are the pseudo code:

```
int block_num = vertices_num / block_size;

for (int k = 0; k < block_num; k++) {
    // Phase 1: process block on the diagonal
    process(block_k_k)

    // Phase 2: process block on the kth row
    for (int j = 0; j < block_num; j++) {
        if (j == k) continue;
        process(block_k_j)
    }

    // Phase 3: process block on the kth column
    for (int i = 0; i < block_num; i++) {
        if (i == k) continue;
        process(block_i_k)
    }

    // Phase 4: process block on the ith row and jth column
    for (int i = 0; i < block_num; i++) {
        if (i == k) continue;
        for (int j = 0; j < block_num; j++) {
            if (j == k) continue;
            process(block_i_j)
        }
    }
}
```

As shown in Figure 3, the first block to be processed is the block on diagonal, since we need the value in the diagonal block to further process blocks on the kth row and kth column. Then, the second and the third phases is to process the blocks on the kth column and blocks on the kth row. The results in these blocks are needed for all the other blocks. As the kth row blocks and

the kth column does not depend on each other, so the order of second and third phase is interchangeable.

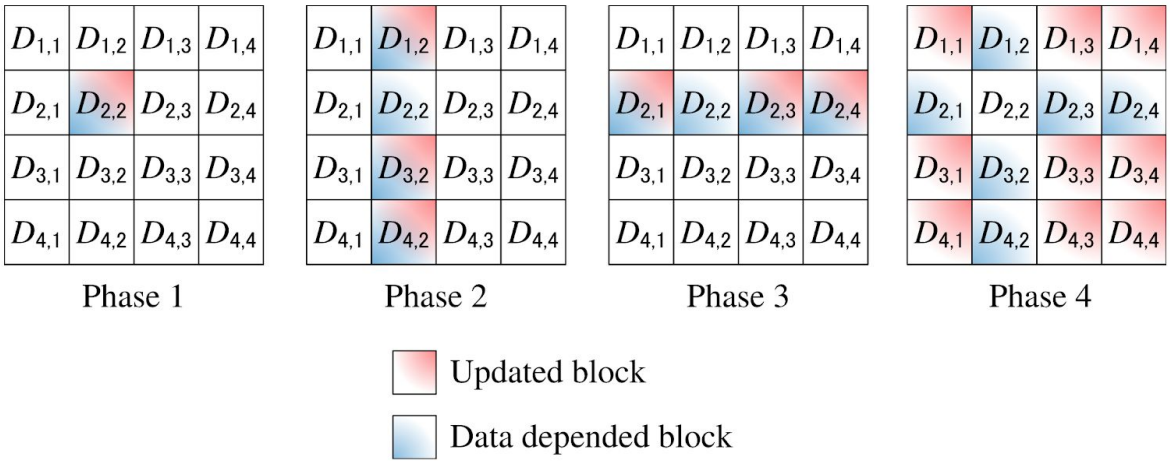

Figure 8: 4 phases of blocked OpenMP algorithm

Result

MPI result

A code snippet of MPI Implementation of Floyd Algorithm is shown below.
The basic idea is described above. Every processor owns certain rows. The shared data are broadcast by MPI function MPI_Bcast.

```
for (k = 0; k < n; k++) {
    root = get_myid(k, p, n);
    if (myid == root) get_rowk(matrix, n, p, row_k, k);
    MPI_Bcast(row_k, n, MPI_INT, root, comm);
    for (i = 0; i < n / p; i++)
        for (j = 0; j < n; j++) {
            dist = matrix[i * n + k] + row_k[j];
            if (dist < matrix[i * n + j]) matrix[i * n + j] = dist;
        }
}
```

Figure 9: MPI implementation

We tested performance with difference processors, 1, 2, 4, 8, 16 on latedays machine. Some screenshots of running time and execution commands.

```
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 16  --hostfile ./hosts ./mpi_floyd < matrix_4000x4000.txt

time elapsed 18.078935
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 8  --hostfile ./hosts ./mpi_floyd < matrix_4000x4000.txt

time elapsed 31.021252
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 4  --hostfile ./hosts ./mpi_floyd < matrix_4000x4000.txt

time elapsed 54.891464
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 2  --hostfile ./hosts ./mpi_floyd < matrix_4000x4000.txt

time elapsed 107.440743
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 1  --hostfile ./hosts ./mpi_floyd < matrix_4000x4000.txt

time elapsed 211.630531
```

Figure 10: MPI commands 4000 vertices

```
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 1  --hostfile ./hosts ./mpi_floyd < matrix_1000x1000.txt

time elapsed 4.108924
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 1  --hostfile ./hosts ./mpi_floyd < matrix_1000x1000.txt

time elapsed 3.990646
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 2  --hostfile ./hosts ./mpi_floyd < matrix_1000x1000.txt

time elapsed 2.363612
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 2  --hostfile ./hosts ./mpi_floyd < matrix_1000x1000.txt

time elapsed 2.341805
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 4  --hostfile ./hosts ./mpi_floyd < matrix_1000x1000.txt

time elapsed 1.483530
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 4  --hostfile ./hosts ./mpi_floyd < matrix_1000x1000.txt

time elapsed 1.486753
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 8  --hostfile ./hosts ./mpi_floyd < matrix_1000x1000.txt

time elapsed 0.851637
[zhaoc2@latedays floyd_mpi]$ mpirun --mca btl vader,self,tcp -np 8  --hostfile ./hosts ./mpi_floyd < matrix_1000x1000.txt

time elapsed 0.961445
```

Figure 11: MPI Performance 1000 vertices

Two performance figures corresponds to graphs with 1000 vertices and 4000 vertices are shown below. (For 1000 vertices, the maximum processors allowed is 8 as the number of vertices has to be divisible by the number of processors)
From the figures, the speedup is nearly linear to the number of processors. As the number of threads doubles, the time is cut into half. Besides, with the same number of processors, the time spent on 4000 vertices should be 4^3 = 64 times the time spent on 1000 vertices. The result meets the theoretical expectation.

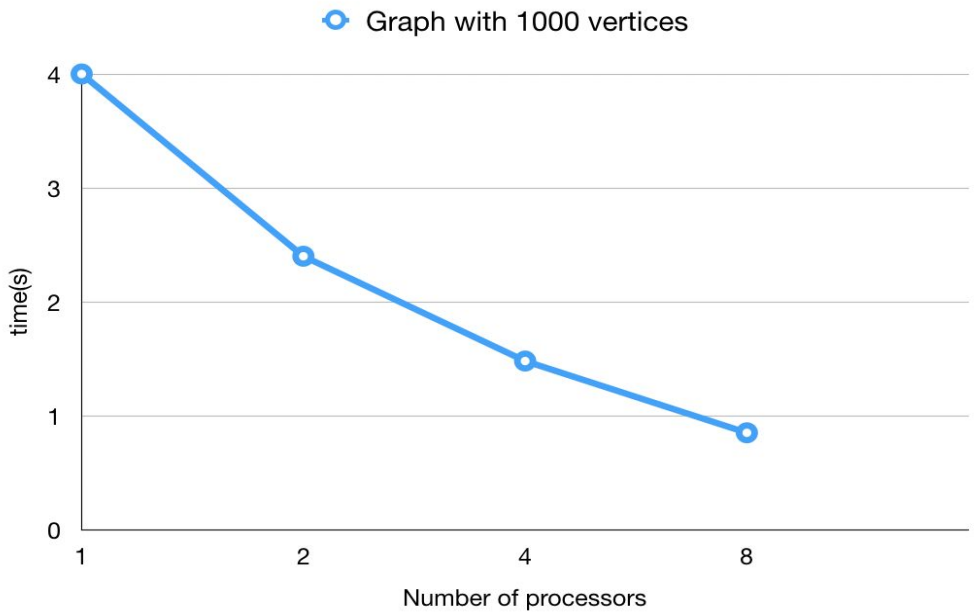

Figure 12: MPI Performance 1000 vertices

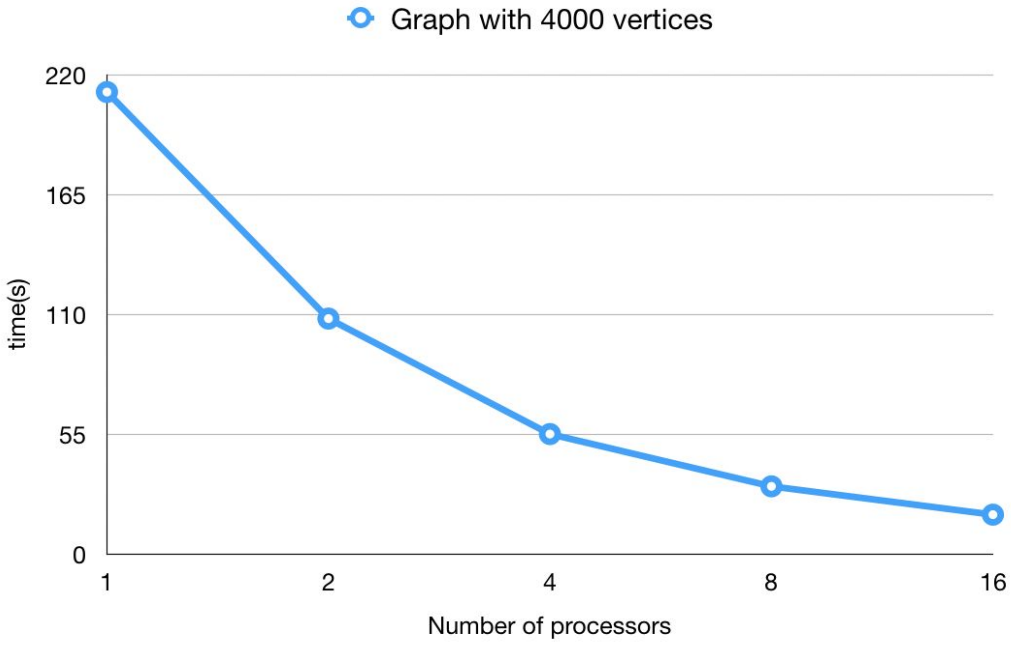

Figure 13: MPI Performance 4000 vertices

Blocked OpenMP result

For blocked OpenMP algorithm, we timed it with different thread numbers and block sizes. As we can see from figure 14 below, the best performance comes with 16 threads and a block_size of 8. This is expected, since 16 threads plus a small block_size is usually a good setting for CPU parallel code.

However, we did not expect the performance when block_size == 512 to be so bad, the reason may be that cache is full thus a lot of cache misses are introduced. As a result, data accessing become the bottleneck and thread number is no longer affecting the performance when the block_size is too big.

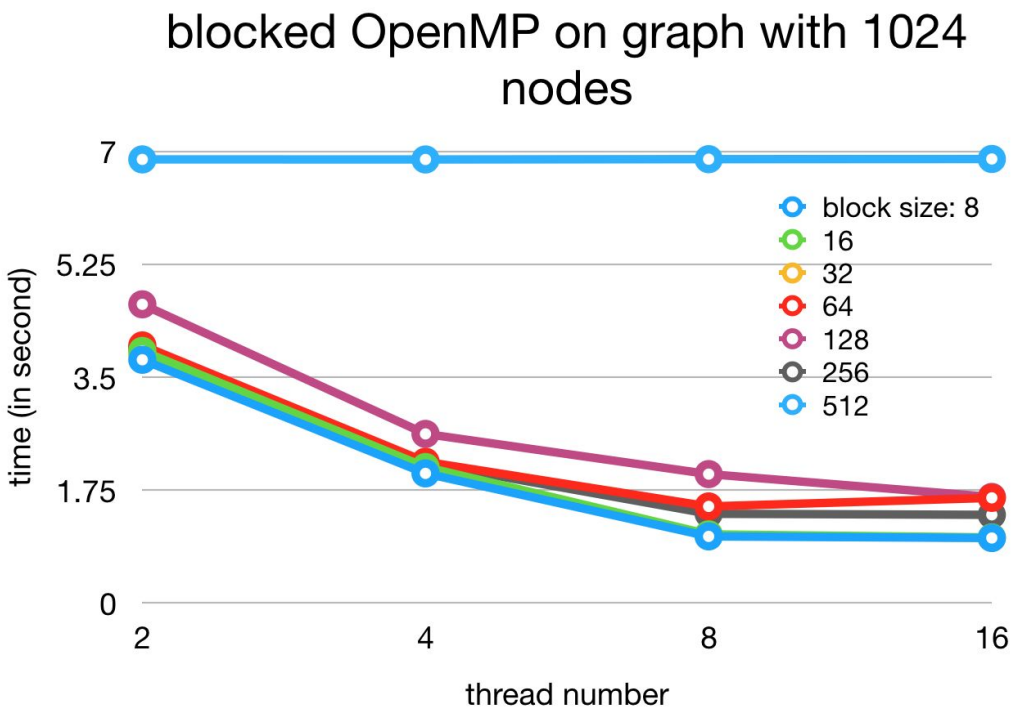

Figure 14: Blocked OpenMP performance on graph with 1024 vertices

Reference

[1] J.S. Park, M. Penner, and V. K. Prasanna, Optimizing Graph Algorithms for Improved Cache Performance IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 9, SEPTEMBER 2004.

[2] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Blocked united algorithm for the all-pairs shortest paths problem on hybrid CPU-GPU systems," IEICE TRANSACTIONS on Information and Systems, vol. 95, no. 12, pp. 2759-2768, 2012.

[3] J.S. Park, M. Penner, and V. K. Prasanna, Optimizing Graph Algorithms for Improved Cache Performance IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 9, SEPTEMBER 2004.

[4] http://acc6.its.brooklyn.cuny.edu/~cisc7340/examples/mpifloyds16.pdf

[5] https://en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm

Distribution of Credit:

50% - 50%