

Shu Zhou 22552162

Shikai Wang 21938451

CITS3402

ASSIGNMENT2 REPORT

SHORTEST PATHS

**1. Execution**

* makefile

mpicc -g Floyd\_MPI.c Floyd\_MPI.h file.c

* Compilation of parallel code : mpiexec/mpicc -n [number of processor] ./output 4096.in
* Floyd-Warshall algorithms represented to solve the all-pairs-shortest path problem

**2. Introduction**

In this assignment, the goal is to design and implement parallel algorithm to solve the all pairs shortest path problem for a number of large graphs. We have decided to use Floyd-Warshall as Dijkstra’s algorithm finds the shortest path between a single pair of vertices. However, Floyd-Warshall finds the shortest paths between all pairs of vertices, so we decided to use Floyd-Warshall to parallel the program.

**3. Floyd-Warshall algorithm:**

We are going to use Floyd-Warshall algorithm which is using to computing the shortest paths between all pairs of nodes in a weight graphs with positive or negative edge.

The first integer represents how many vertices in this matrix, so number of elements are number of Vertices times number of Vertices which is (n\*n) of lengths.

In this matrix, the weight of all paths between nodes that do not contain any intermediate node is called distance matrix if there has any weight between vertices i and j that means this edge is weighted. It is noticed that the diagonal of the matrix contains only zeros.

The above algorithm represents the Floyd-Warshall algorithm. This process keeps looping until k = n, at the end will find the shortest path for all (i, j) pairs through any intermediate vertices.

As the pseudocode shown below: 

In order to return shortest paths among all pairs of nodes, two-dimensional matrix is generated. Firstly, initialize the matrix and then check if (i==j) which means all the diagonal of the matrix contains only zeros. If the value of edge between i and j is 0, then there is no path between these nodes. Conversely, the node between two vertices exist path but maybe not optimal. The most important step is comparing the previous path with the current path to check which is the shortest, as above pseudocode demonstrated that if there exist a path from ,then comparing to directed between i and j. If is smaller than previous path, then replace it otherwise do nothing.

As these matrices is going to be a single instance of two-dimensional array only so it will be updating the values for each vertex, k will keep changing for each term and result is to generate one matrix, so it needs another for loop as intermediate vertex. For instance, if k=1 then the first Matrix is generated, if k=2 then the second Matrix is generated and dynamically optimal the matrix. We repeat this procedure until the end to find the all pairs shortest path. The time complexity is O(n\*n\*n).

**4. Parallel Algorithm**

We are implemented MPI which is a communication protocol for parallel programming. MPI is specifically used to allow applications to run in parallel across a number of separate computers also called cluster. MPI generally run the same code we have written on multiple processors or cluster, which then communicate with one to another through library calls. First step we initialize argc and argv, and then terminate communications.

In order to allocate the number of values from matrix evenly to every thread, we defined an int type array called **offset\_of\_slice[size+1**], this array will record the start offset and end offset of matrix row for every thread. The next step is to allocate value for different threads.

The above picture shows how to allocate the start offset and end offset. Use one for loop to make sure each thread has allocated value. If vertex number mod the number of process is equal to or small to i that means, there are extra value need to be allocated into this thread.

Otherwise, find the start offset of this matrix row and put it into specific thread.

for (int i = 1; i <= size; i++)

{

if (i <= PGrph->vexnum % size)

offset\_of\_slice[i]=vexnum / size+offset\_of\_slice[i-1]+1;

else

offset\_of\_slice[i]=vexnum / size+offset\_of\_slice[i-1];

}

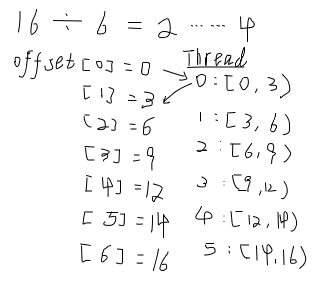
}

For instance, if we have got 16\*16 matrix and total processes is 6. If we use 16mod6=4 which means 4 is the remainder, so we are going to breakdown this and put each one into different thread sequentially. If current rank is smaller or equal to reminder then allocate one more into current rank, otherwise is rounding quotient then normally add the scope of values into current rank.

First one always start at 0 which is offset\_of\_slice[0] = 0. The second one offset\_of\_slice[1] = 16 / 6 + 0 + 1 = 3, because rank 1 is smaller than remainder etc.

Same as until we get offset\_of\_slice[5] = 12 + 2 which is 14.

The advantage of this method is to assign rank scope evenly instead of adding all of the remainder to the end of the rank.



Noticed that

Offset array records the scope of every thread. Therefore, the size of offset array should be equal that the size of threads adds one.

After allocate start offset and end offset to each thread, we need to use MPI\_Bcast function to broadcast in order to split the matrix for every thread and send to every thread as well as the number of vertices.

We have created two matrices, one called distance which is going to store the matrix of distance and another one called distance\_new to output the new matrix.

After created a distance matrix we need to use MPI\_Bcast function to broadcast to every thread.

The most important thing is how to parallel Floyd-Warshall algorithm.

Find the shortest distance is crucial what if you find a distance from

, but the issue is what if and they both from different thread , so the best way to solve the problem is after the first loop define one variable called k\_root to find which thread contain k row and then use MPI\_Bcast function to broadcast from k\_root to every thread. For example, if distance is (means this belongs to thread one) and find there have shorter distance + . These two vertices are from different thread.

We have defined our own function called minmum\_distance, because the disconnection is regarded as 0. Therefore, we didn’t use the build-in operations. What Minimum\_distance does is comparing current distance with previous one if distance is smaller than previous one then replaces it. Finally, we gather all information and print output through distance\_new function.

It is noticed that If the size is equal to one, then we use normal way to implement Floyd-Warshall algorithm.

5. Experiment with scalability

If the $\lfloor \frac{vertex\ number}{the\ number\ of\ process}\rfloor$ is equal to or small to $i$ (indicate the rank) that means, allocate one more into current rank, otherwise is rounding quotient then normally add the scope of values into current rank.

In general, the number of rows that each thread’s allocated is $\lfloor \frac{vertex\ number}{the\ number\ of\ process}\rfloor$. However, if the rank of a thread is smaller or equal to the reminder of above formula, then allocate one more into this rank.

Obviously, the reminder will equal to $16mod6=4$

Corresponding

After allocate start offset and end offset to each thread, MPI\_Bcast function was used to broadcast the corresponding rows for every thread, as well as the number of vertices.

Separate

In the design of parallel algorithm, the decomposition of the problem usually has two forms: one is to decompose the space of problems into several smaller problem areas and then solved separately; The other is to decompose a big problem into several sub-problems according to the function, and then each sub-problem is solved in parallel.[2] For the shortest path problem between all point pairs, it is more convenient to choose the former after we adopting Floyd algorithm. In this case, the strategy we used is that split its second layer ‘for’ loop. The basic idea is to separate a huge matrix to several smaller matrices by their rows and allocating these smaller matrices for every thread (or node). Like follow:

We will demonstrate the correctness of this method later.