Name: Rohan Lagare

Student ID: 00001630346

Program 3 Report

Dijkstra's algorithm

You need to write two programs, dijkstra_mpi and dijkstra_omp/dijkstra_acc, that will take as input a single file (an adjacency matrix) and a number of random input source vertex IDs to test and output to a file with the weights of a shortest path from the source vertex 0 to every other vertex. As the names suggest, the first program will use MPI and the second OpenMP or OpenACC for parallelizing the algorithm. The input file will consist of n+1 lines, with the first line containing only the number n, and each other line containing n values (i.e., a dense matrix), with each value representing the weight of the edge. You should assume that the values in the input file are represented as quad-precision floats. An edge that does not exist in the graph is represented by the value inf. Your output file should also follow this convention.

For example, an adjacency matrix for a directed graph with five vertices and nine edges would be stored in a file as,

```
5
inf     0.2     inf     3.7     0.4
2.1     inf     3.9     1       inf
 2      0.01    inf     0.7     6
9       1.2     2.5     inf     inf
3       3       7       4       inf
```

The program takes the following parameters:

<PROGRAM> <graph> <num_sources> <nthreads> [<output_file>]

graph: input graph

num_sources: no of random source searches

output_file: B output file for result of program

num_threads : no. of threads for parallel block

Path finding

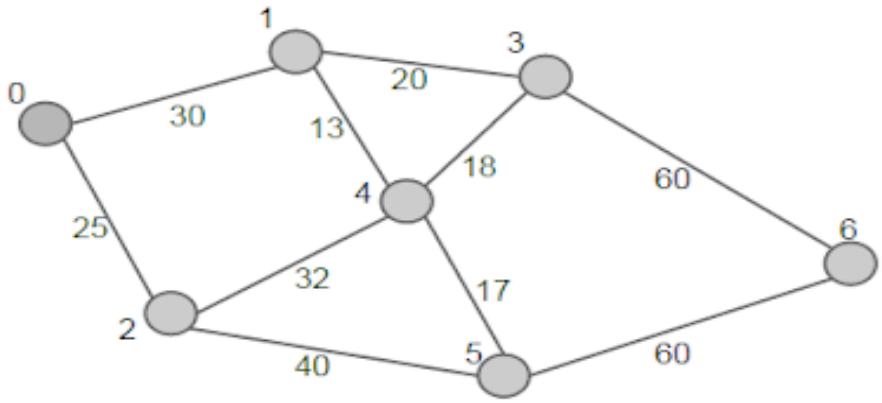
Given graph A, source, number of nodes.

We load graph A from provided file.

After loading the graph, we find the shortest paths using following algorithms.

Serial Algorithm:

1. Initialize min (float_int type) and l, m arrays to store the minimum length and, to store if the node is visited or not.
2. Populate l with length from current node to source for all nodes.
3. Set min.l=-1 and element of m for source as visited i.e. 1
4. loop for n (number of nodes) time finding local minimums and updating all the distance as below:
5. find the local minimum distance for current iteration by looping through all the nodes and checking if its distance is less than current minimum distance.
6. After getting minimum distance, mark minimum distance node as visited.
7. Loop through all the nodes and update distances for all by checking if current minimum plus distance from current node to minimum is less than that of l.
8. After looping, free all the unnecessary objects and return result array l.



Nodes	0	1	2	3	4	5	6
Distance	0	∞	∞	∞	∞	∞	∞

Fig: Graph and expected result format

Parallel Algorithm

In parallel approach, we are parallelizing the loop where we calculate local minimum, to divide the work between each thread as shown in fig 1 below. We are dividing rows of L between different threads, so each thread will get bunch of elements of L, and it will calculate local minimum for it. After calculating all the local minimums for each thread, we loop through all the thread minimums to get local minimum.

L	11.4	2.5	51.4	INF	INF	....	....	....	....	12.3
	0	1	2	3	4	....	....	....	....	N
	Process 1			Process 2				Process X		

Fig 2: logical distribution of data and work for parallel algo.

With just that we have parallelized our serial program, but if we use the one min variable as it is, there might be data race between different threads, and we might not get correct min values due to random execution of loop. And the result L will be incorrect. We can use the ordered OMP clause, but it will significantly impact our runtime as it will synchronize the threads, increasing overhead.

To tackle this, we are replacing the above min variable with array of float_int type, so that for each thread, we would have separate element, as shown in fig 3 below.

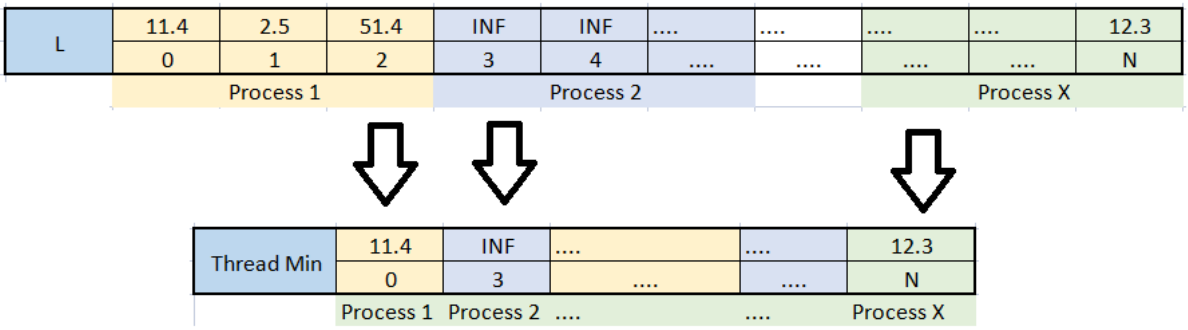
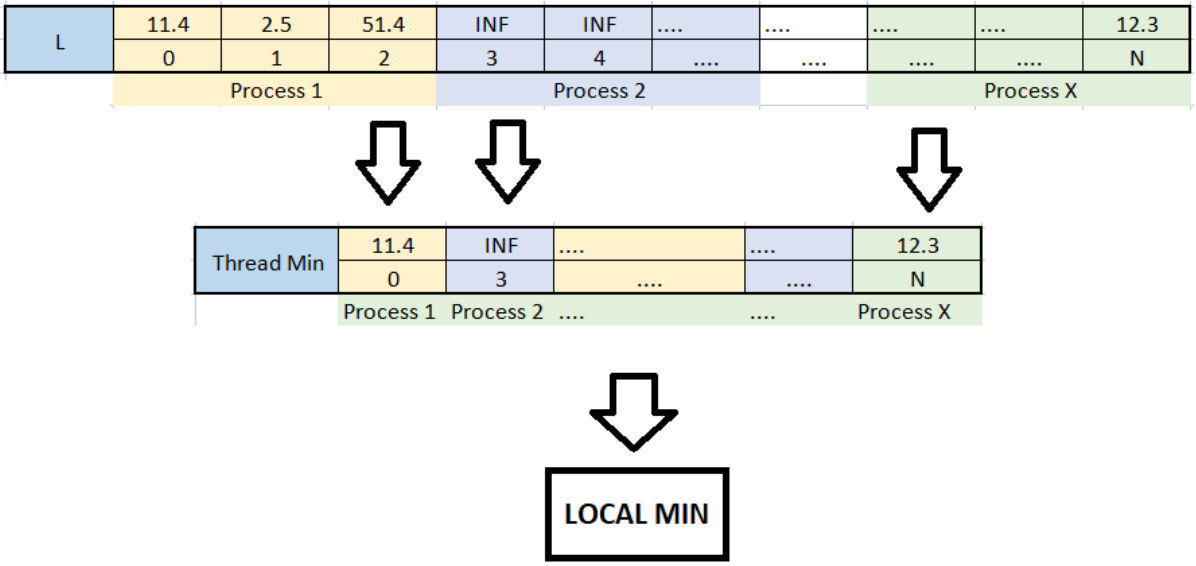


Fig 3: Finding thread minimums and storing in threadMin array.

So, at the end of 1st for parallel block, we will calculate local minimum from different thread minimums.

We will create threadMin array, as shown in figure below.



After this Loop through (2nd parallel for) all the nodes and update distances for all by checking if current minimum plus distance from current node to minimum is less than that of l.

After looping, free all the unnecessary objects and return result array l.

Execution Time and speedup

We have executed serial version of program for few data sets. We are comparing above serial time parallel version calculating the speedup.

We can see 2-7X speedup compared to serial program depending on problem size.

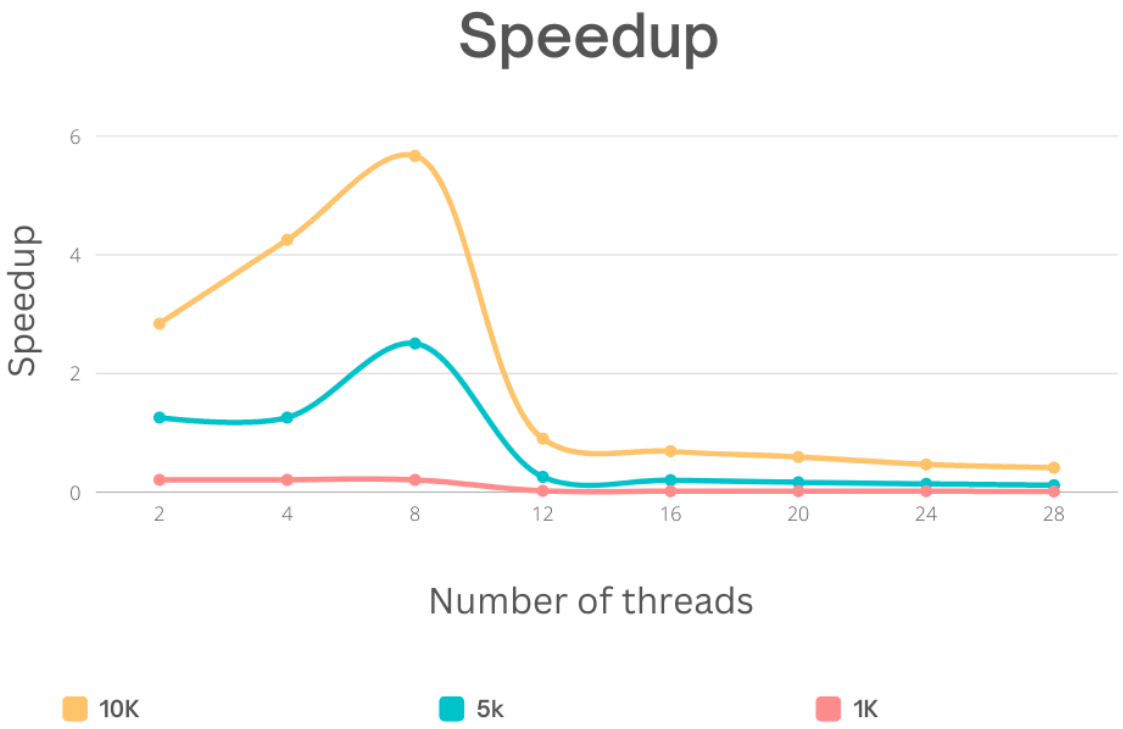
Please refer below graphs for more information.

Stats for 10k graph :

Graph used	Threads	ser time	Time_sec	speedup
10000.graph	2	17	6	2.833333333
10000.graph	4	17	4	4.25
10000.graph	8	17	3	5.666666667
10000.graph	12	17	19	0.894736842
10000.graph	16	17	25	0.68
10000.graph	20	17	29	0.586206897
10000.graph	24	17	37	0.459459459
10000.graph	28	17	42	0.404761905

Stats for 5k graph :

Graph used	Threads	ser time	Time_sec	speedup
5000.graph	2	2.5	2	1.25
5000.graph	4	2.5	2	1.25
5000.graph	8	2.5	1	2.5
5000.graph	12	2.5	10	0.25
5000.graph	16	2.5	13	0.192307692
5000.graph	20	2.5	16	0.15625
5000.graph	24	2.5	19	0.131578947
5000.graph	28	2.5	23	0.108695652



Graph : speed-up vs. No. of threads for 10k, 5k and 1k graphs

Data size and runtime

We can observe that, for the serial program, as we increase the data size the runtime/execution time is also increasing. This is simply due to the increase in the number of calculations.

However, for the parallel program tread is different, for small data sizes parallel program takes longer to execute due to overhead of threads, as we increase the data size the execution time decreases.

And for large enough datasets, we can see significant improvements in execution time as compared to serial program.

increase the data size the runtime/execution time is increasing. This is simply due to the increase in the number of calculations.

Performance scaling: no. Of threads and runtime

We can see from the above graphs and statistics that performance is increasing as we increase the no. Of threads, I.e., execution time is decreasing as we increase no. Of threads for same data size.

As we divide the work between different threads, it will be completed faster than previous ones.

However, we can see after certain no. Of threads performance is not increasing and is the same or is decreasing.

As we keep on increasing the no. of threads, the amount work that each thread must do is decreasing.

But to divide the work among the threads we must do some extra work (Overhead).

So, after a certain number of threads, the overhead and communication time between threads will add up to any performance gain that we might get from extra threads, as shown in graphs (For more info please refer stat sheets attached).

Performance characteristics

The performance of the program will depend on the properties of the matrices.

As discussed above, we are parallelizing the program by dividing the loop (which calculates local minimum) among different threads, so If matrix is converted to sparse matrix form, then we would get expected throughput as non-zeros will get divided equally.