PARALLELIZING DIJKSTRA'S ALGORITHM

-Tejal Singh

Reg. Num-21011101133

AI&DS-B

1.What is Dijkstra's Algorithm?

Dijkstra's algorithm is an algorithm for finding the shortest path between nodes in a graph. The algorithm was published in 1959 by Dutch computer scientist Edsger W. Dijkstra, can be applied on a weighted graph. Dijkstra's original algorithm runtime is a quadratic function of the number of vertices.

The main feature of Dijkstra's algorithm is to extend the outer layer (the breadth-first search idea) around the source vertex until it reaches the end vertex. When calculating the shortest path in the Graph G, we specify the starting vertex s (that is, starting from the source vertex s). In addition, two sets S and U are introduced. The role of S is to record the vertices and the corresponding shortest path length for which the shortest path has been found. The set U is used to record the vertices and the distance from the vertices to the source vertex s which the shortest path has not been found. Initially, there is only the source vertex s in S; U contains vertices other than s, and the path of the vertex in U is the path from source vertex to this vertex. Then, find the shortest path for this vertex from U and add it to S, update the vertex and the corresponding path in U. Then, find the shortest vertex of the path from U and add it to S, update the vertex and the corresponding path in U … repeat the operation until all the vertices have been traversed.

(1) Initially, S only contains the starting vertex s; U contains other vertices except s, and the distances. The distance is the weight from the starting vertex s to the vertices in U. For example, the distance of the vertex v in U is ∞ if s and v are not adjacent.

(2) Select the shortest vertex u from U and add vertex u to S; meanwhile, remove vertex u from U

(3) Update the distance from each vertex in U to the source vertex. The reason why the distance of the vertices in U is updated is that in the previous step u is the vertex of the shortest path, so that the distance of other vertices can be updated by u; for example, the distance of (s, v) may be greater than the distance (s, u) + (u, v). (4) Repeat steps 2 and 3 until all the vertices have been traversed.

2.Why use parallel computing?

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. Here are some reasons why we need parallel computing:

- Save time and/or money.
- Solve larger/more complex problems.
- Provide concurrency.
- Take advantage of non-local resources.
- Make better use underlying parallel hardware.

From computational simulation in scientific and engineering applications to business applications in data mining and transaction processing, parallel computing has made a huge impact in various fields. The cost advantages of parallelism and the performance requirements of applications make compelling arguments for supporting parallel computing.

There are two main forms of data exchange between parallel tasks-accessing shared data space and exchanging messages.

3.Approach

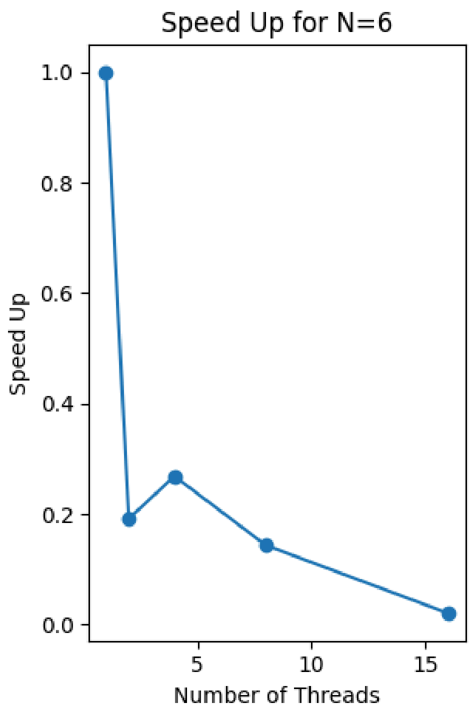
Given below is a pseudocode for the algorithm

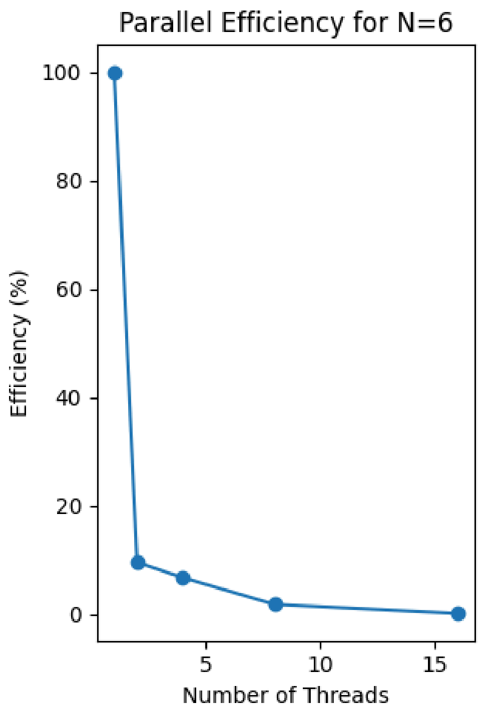
```
 1. procedure DIJKSTRA_SINGLE_SOURCE_SP_OPENMP (V, E, w, distances, s)
 2. begin
 3. #pragma omp parallel private
 4. shared ()
 5. omp_get_thread_num ( );
 6. omp_get_num_threads ( );
 7. Each thread finds the min distance u unconnected vertex inner
 8. # pragma omp critical // update overall min
 9. # pragma omp barrier
10. # pragma omp single // mark new vertex as done
11. for all v in each thread
12. distances[v] := min{distances[v], distances[u]+ w[u][v]}; 12. Endwhile
13. end DIJKSTRA_SINGLE_SOURCE_SP_OPENMP
```

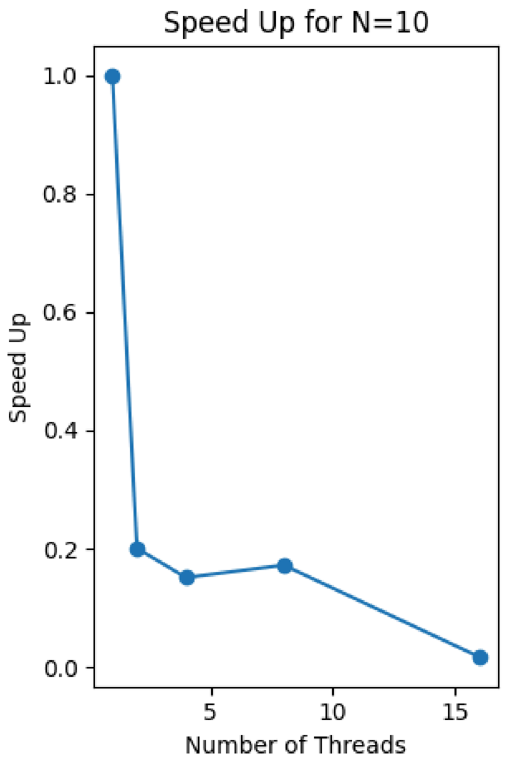
OpenMP has less lines of code compared to other methods of parallelization. The function *omp_set_num_threads* sets the default number of the threads that will be created on encountering the next parallel directive. We use this function in the main function.
The *omp_get_num_threads* function returns the number of threads participating in a team. The *omp_get_thread_num* returns a unique thread id for each thread in a team. This integer lies between 0 and *omp_get_num_threads()* – 1. The critical directive ensures that at any point in the execution of the program, only one thread is within a critical section specified by a certain name.
OpenMP provides a critical directive for implementation critical regions. There is a critical region that allows different threads to execute different code while being protected from each other.A barrier is one of the most frequently used synchronization primitives. OpenMP provides a barrier directive. On encountering this directive, all threads in a team wait until others have caught up, and then release. A single directive specifies a structured block that is executed by a single thread. On encountering the single block, the first thread enters the block. All the other threads proceed to the end of the block.

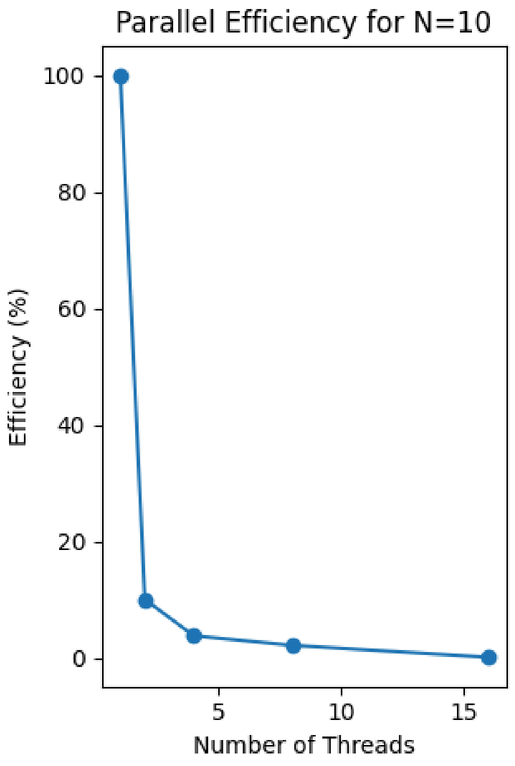
4.Performance Metrics

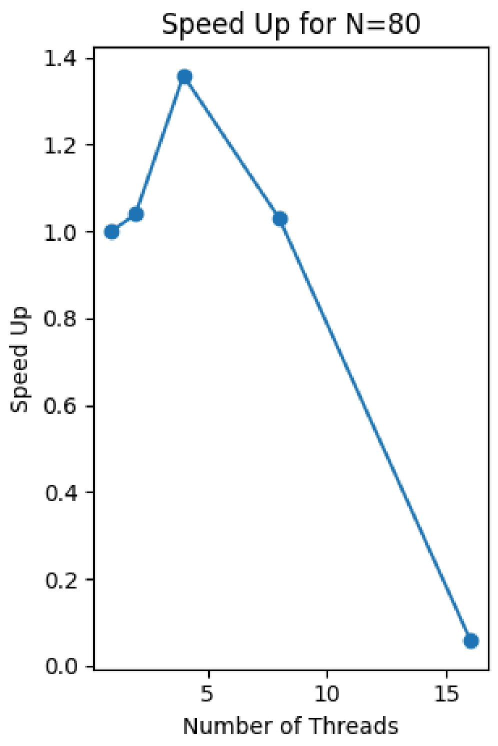
Initially, smaller problem sizes may exhibit higher speedup due to easier task division and lower communication over- head. However, as the problem size grows larger, the parallel efficiency may diminish due to factors such as increased communication overhead and resource contention among processing elements *p*. Therefore, the decrease in speedup for larger problem sizes,, suggests that the algorithm's performance becomes less optimal as the computational workload increases. Understanding this trend is crucial for optimizing the parallel algorithm's performance and resource allocation for different problem sizes *N*.

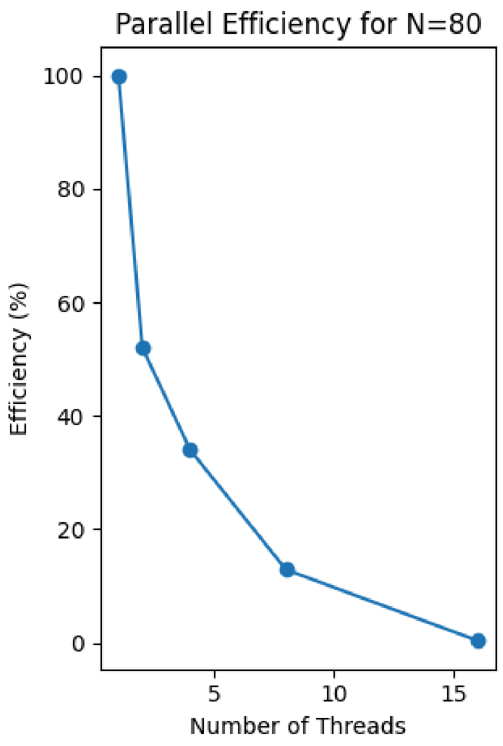

Speed Up for N=6
Speed Up
Number of Threads
0.0
0.2
0.4
0.6
0.8
1.0
5
10
15

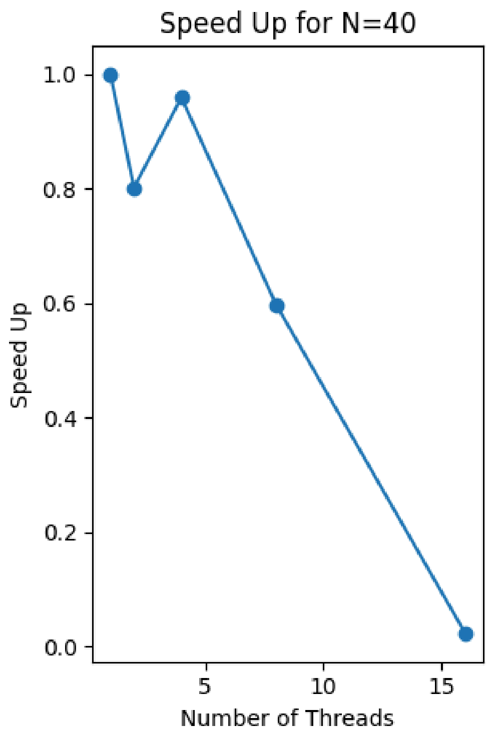

Parallel Efficiency for N=6
Efficiency (%)
Number of Threads
0
20
40
60
80
100
5
10
15

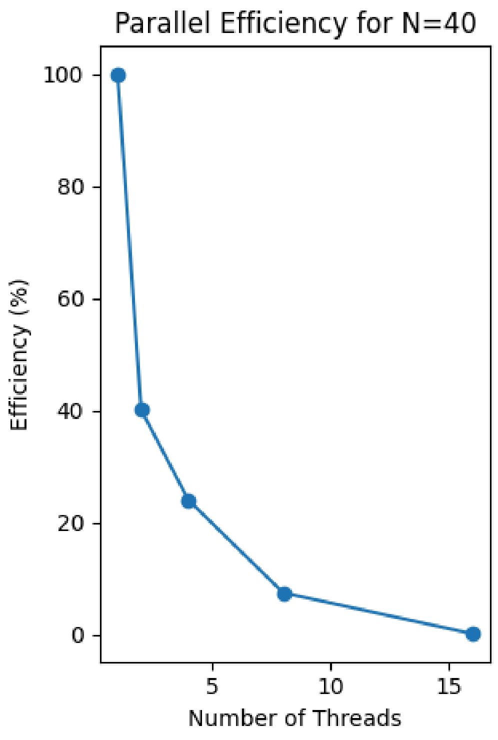

Speed Up for N=10
Speed Up
Number of Threads
0.0
0.2
0.4
0.6
0.8
1.0
5
10
15


Parallel Efficiency for N=10
Efficiency (%)
Number of Threads
0
20
40
60
80
100
5
10
15


Speed Up for N=80
Speed Up
Number of Threads
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
5
10
15


Parallel Efficiency for N=80
Efficiency (%)
Number of Threads
0
20
40
60
80
100
5
10
15


Speed Up for N=40
Speed Up
Number of Threads
0.0
0.2
0.4
0.6
0.8
1.0
5
10
15


Parallel Efficiency for N=40
Efficiency (%)
Number of Threads
0
20
40
60
80
100
5
10
15

5.References:

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3 rd ed. MIT Press, 2009.
- A. Grama, A. Gupta, G. Karypis, and V. Kumar, Introduction to Parallel Computing, 2 nd ed. Addison Wesley, 2003.
- C.Wong.“Parallel-Dijkstra’s-Algorithm,” November 2015, https://github.com/ courtniwong/Parallel-Dijkstras-Algorithm .
- “Dijkstra's algorithm,” https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm .