CS359- Parallel Computing Project Report

PARALLELIZING DIJKSTRA'S ALGORITHM

Submitted

In the partial fulfilment of the requirements of

CS 359 Parallel Computing

By

Shaik Suhana (220001073)

M.Hemadeepika(220001047)

Submitted to

Prof. Dr. Surya Prakash

Department Of Computer Science and Engineering

Indian Institute of Technology Indore

A.Y 2024-2025

Problem Statement:

Dijkstra's algorithm is a well-known algorithm used to find the shortest path between nodes in a graph, particularly in weighted graphs. However, the standard implementation has a time complexity of $O(V^2)$ using an adjacency matrix, where V is the number of vertices. This makes the algorithm inefficient for large graphs, particularly in real-time applications such as GPS systems and network routing. The objective of this proposal is to parallelize Dijkstra's algorithm using OpenMP to enhance its performance and scalability.

Objectives:

- Develop a parallelized implementation of Dijkstra's algorithm using shared-memory parallelism with OpenMP.
- Optimise the algorithm to efficiently divide computation across threads while minimising synchronisation overhead.
- Evaluate the performance of the parallelized algorithm on large synthetic graph datasets.
- Measure execution time, speedup, and scalability to assess improvements over the sequential version.

Mathematical Formulation:

The Sequential Dijkstra's algorithm is iterative. Each iteration adds a new vertex to the computed set. Since the value of $d[v]$ for a vertex v may change every time a new vertex u is added in S, It is hard to select more than one vertex. It is not easy to perform different iterations of the while loop in parallel. However, each iteration can be performed in parallel as follows.
Let p be the number of processes, and let n be the number of vertices in the graph. The set V is partitioned into p subsets using the 1-D block mapping. Each subset has n/p consecutive vertices, and the work associated with each subset is assigned to a different process. Let V_i be the subset of vertices assigned to process P_i for i = 0, 1, …, p - 1. Each process P_i stores the part of the array d that corresponds to V_i (Figure 1.a). Each process P_i computes $d_i[u] = min\{d_i[v] \in (V - S) \cap V_i\}$ during each iteration of the while loop. The global minimum is then obtained over all $d_i[u]$ by using the all-to-one reduction operation and is stored in process P_0. Process P_0 now holds the new vertex u, which will be inserted into S. Process P_0 broadcasts u to all processes by using one-to-all broadcasts. The process $P_i$responsible for vertex u marks u as belonging to set S. Finally, each process updates the values of $d[v]$ for its local vertices.

When a new vertex u is inserted into S, the values of $d[v]$ for v (V - S) must be updated. The process responsible for v must know the weight of the edge (u, v). Hence, each process P_i needs to store the columns of the weighted adjacency matrix corresponding to set S of the

vertices assigned to it. This corresponds to 1-D block mapping of the matrix. The space to store the required part of the adjacency matrix at each process is $\Theta\ (n^2/p)$. Figure 1.b illustrates the partitioning of the weighted adjacency matrix. The computation performed by a process to minimise and update the values of $d[v]$ during each iteration is $\Theta\ (n/p)$. The communication performed in each iteration is due to the all to-one reduction and the one-to-all broadcast. For a p-process message-passing parallel computer, a one-to-all broadcast to one word takes time $log\ p$. Finding the global minimum of one word at each iteration is $\Theta\ (log\ p)$. The parallel run time of this formulation is given by $T_p = \theta(n^2/p) + \theta(n\ log\ p)$.

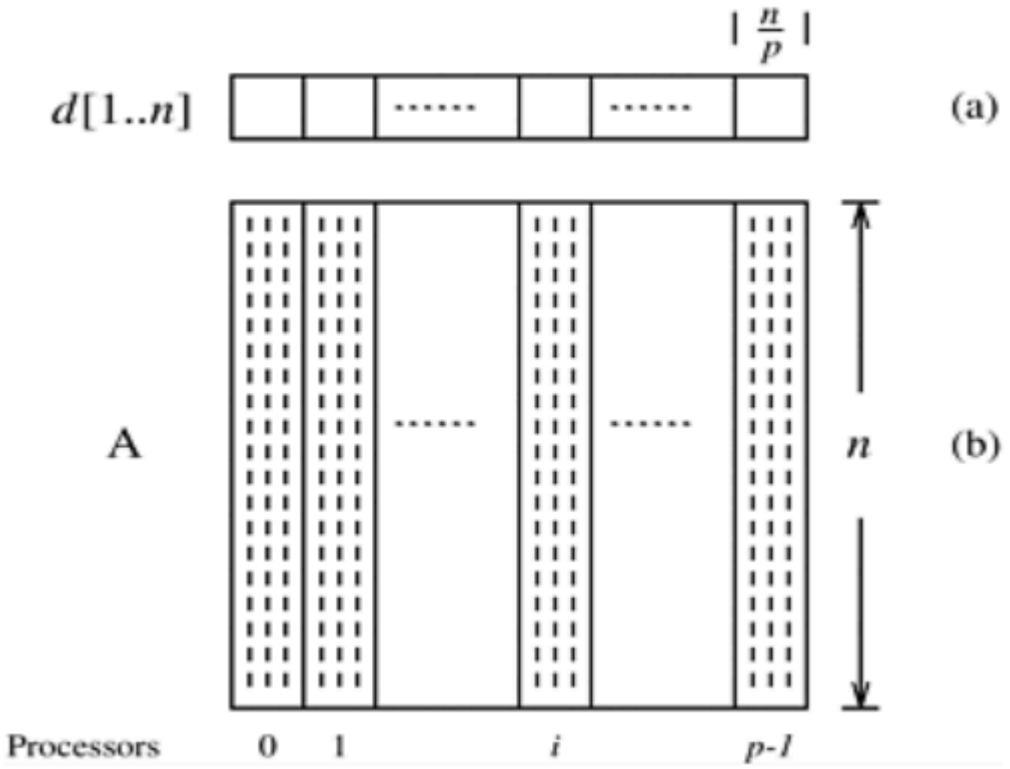

Figure 1. The partitioning of the distance array d and the adjacency matrix A among p processes.

Implementation:

View the Code on GitHub

```
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
```

As the pseudocode shows, OpenMP Dijkstra's algorithm implementation is very similar to the sequential one. 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` function 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 during the execution of the program, only one thread can be within a specified critical section. OpenMP provides the `critical` directive to implement critical regions. This allows different threads to execute different code while being protected from one another.

A barrier is one of the most frequently used synchronisation primitives. OpenMP provides a `barrier` directive. Upon encountering this directive, all threads in a team wait until all others have caught up, after which they are released.

The `single` directive specifies a structured block that is executed by a single thread. When encountering the `single` block, the first thread to reach it enters the block, while all other threads skip the block and proceed to the end of it.

Results and Analysis:

For OpenMP parallel computation, 2, 4, 8, 16 processors (the numbers of vertices should be larger than processors) are used to run the code.

Table 1. Execution time in seconds for both the implementations.

	8 vertices	64 vertices	256 vertices	512 vertices	1024 vertices
Seq	0.0035	0.0029	0.0099	0.1162	0.3568
OpenMP2	0.0003	0.0015	0.0093	0.0651	0.2493
OpenMP4	0.0003	0.0016	0.0109	0.0635	0.2488
OpenMP8	0.0007	0.8831	0.0140	0.0854	0.2807
OpenMP16		0.0574	0.0233	0.2736	0.6794

Table 2. The best execution time in seconds for both the implementations (the numbers in brackets indicate how many threads/processors).

	8 vertices	64 vertices	256 vertices	512 vertices	1024 vertices
Seq	0.0035	0.0029	0.0099	0.1162	0.3568
OpenMP	0.0003(2)	0.0015(2)	0.0093(2)	0.0635(4)	0.2488(4)

We can see the performance is better when using OpenMP to run the algorithm. For a small number of vertices, more time could be spent on parallelization and synchronisation than it is spent on execution of code as sequential. So, when the number of vertices is less than 512, it is not obvious that parallelization is superior to sequential. We can predict the cost time of OpenMP to be significantly better than sequential for a higher number of vertices.

Table 3. The comparison for theoretical speedup and experiment speedup for OpenMP implementation with 1024 graph vertices and different number of processors.

number of processors	Theoretical speedup 1024 vertices	Experiment speedup 1024 vertices
2	1.9961	2.7112
4	3.9690	2.7853
8	7.8168	2.6947
16	15.0588	1.7363

Speedup is defined as the ratio of the time taken to solve a problem using the best sequential algorithm to the time required to solve the same problem on a parallel computer with *p* identical processing elements. If speedup can maintain linear growth with the number of processors, multiple machines can significantly shorten the required time. However, this speedup is very difficult to achieve, because when the number of machines increases, there is a problem of communication loss, as well as the problem of each computer node itself (the skew of the slaves). For example, the total time spent by the algorithm is usually determined by the slowest machine. If the time required by each computer is different, there is the problem of the skew of the slaves.

In the experiment, the speedup decreases maybe because the communication latency outperforms the benefit from using more processors. We should consider all the information needed to evaluate the performance of the parallel algorithm on a specific architecture with specific technology dependent constants, like CPU speed, communication speed.

Conclusion:

The results indicate that OpenMP outperforms the sequential implementation for larger input data scales. Interestingly, a smaller number of processors/threads yields the fastest results for the OpenMP implementation. However, the findings reveal that the average speedup achieved through parallelization is not satisfactory. Thus, the parallel implementation of Dijkstra’s algorithm may not always be the optimal choice.