An Implementation of Parallelizing Dijkstra’s Algorithm

CSE633 Course Project
Zilong Ye
ID: 3715-8138
zilongye@buffalo.edu
Outline

- Problem statement
- Dijkstra’s algorithm
- Parallel Dijkstra’s algorithm
- Simulation results and analysis
- Reference
Problem statement

- Given a graph, Let $G = (V, E)$ be a directed graph, $|V| = n$, $|E| = m$, let $s$ be a distinguished vertex of the graph, and $w$ be the non-negative value to the weight of each edge, which represents the distance between the two vertexes.

- Single source shortest path: The single source shortest path (SSSP) problem is that of computing, for a given source vertex $s$ and a destination vertex $t$, the weight of a path that obtains the minimum weight among all the possible paths.
Dijkstra’s algorithm

- Dijkstra’s algorithm is a graph search algorithm that solves single-source shortest path for a graph with nonnegative weights.
- Widely used in network routing protocol, e.g., Open Shortest Path First (OSPF) protocol.

Fig. 1 24-node U.S. mesh network
Dijkstra’s algorithm

Fig. 2 8-node simple network

Table 1. The routing table for node A
Dijkstra’s algorithm-1\textsuperscript{st} round

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
cluster & B & C & D & E & F & G & H \\
A & 1, A & 4, A & \infty & \infty & \infty & \infty & \infty \\
AB & & & & & & & \\
\hline
\end{tabular}
\end{table}

Fig. 2  8-node simple network

Table 1.  The routing table for node A
Dijkstra’s algorithm-2nd round

Fig. 2  8-node simple network

Table 1.  The routing table for node A
Dijkstra’s algorithm-3rd round

Fig. 2  8-node simple network

Table 1. The routing table for node A
Dijkstra’s algorithm-$4^{th}$ round

Fig. 2 8-node simple network

Table 1. The routing table for node A
Dijkstra’s algorithm-5th round

Table 1. The routing table for node A

<table>
<thead>
<tr>
<th></th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>A</td>
<td>1, A</td>
<td>4, A</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>AB</td>
<td>3, B</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5, B</td>
<td>3, B</td>
</tr>
<tr>
<td>ABC</td>
<td>4, C</td>
<td>6, C</td>
<td>∞</td>
<td>5, B</td>
<td>3, B</td>
<td></td>
</tr>
<tr>
<td>ABCH</td>
<td>4, C</td>
<td>6, C</td>
<td>∞</td>
<td>5, B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5, D</td>
</tr>
<tr>
<td>ABCHDE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2 8-node simple network
Dijkstra’s algorithm-6th round

Fig. 2  8-node simple network

Table 1. The routing table for node A
Dijkstra’s algorithm-6th round

![Dijkstra's algorithm diagram](image)

Fig. 2 8-node simple network

Table 1. The routing table for node A

<table>
<thead>
<tr>
<th>Cluster</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d, n)</td>
<td></td>
<td>(d, n)</td>
<td>(d, n)</td>
<td>(d, n)</td>
<td>(d, n)</td>
<td>(d, n)</td>
<td>(d, n)</td>
<td>(d, n)</td>
</tr>
<tr>
<td>cluster</td>
<td></td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>A</td>
<td>1, A</td>
<td>4, A</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>AB</td>
<td>3, B</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5, B</td>
<td>3, B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABC</td>
<td>4, C</td>
<td>6, C</td>
<td>∞</td>
<td>5, B</td>
<td>3, B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCH</td>
<td>4, C</td>
<td>6, C</td>
<td>∞</td>
<td>5, B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHD</td>
<td>5, D</td>
<td>7, D</td>
<td>5, B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHDE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6, E</td>
<td>5, B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHDEG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6, E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHDEGF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Dijkstra’s algorithm

Create a cluster \( cl[V] \)
Given a source vertex \( s \)
While (there exist a vertex that is not in the cluster \( cl[V] \))
{
    FOR (all the vertices outside the cluster)
        Calculate the distance from non-member vertex to \( s \) through the cluster
    END
    ** \( O(V) \)**
    Select the vertex with the shortest path and add it to the cluster
    ** \( O(V) \)**
}
Dijkstra’s algorithm

- **Running time** \( O(V^2) \)
  - In order to obtain the routing table, we need \( O(V) \) rounds iterations (until all the vertices are included in the cluster). In each round, we will update the value for \( O(V) \) vertices and select the closest vertex, so the running time in each round is \( O(V) \). So, the total running time is \( O(V^2) \).

- **Disadvantages:**
  - If the scale of the network is too large, then it will cost a long time to obtain the result.
  - For some time-sensitive app or real-time services, we need to reduce the running time.
Parallel Dijkstra’s algorithm

- **Approach:**
  - Each core identifies its closest vertex to the source vertex;
  - Perform a parallel prefix to select the globally closest vertex;
  - Broadcast the result to all the cores;
  - Each core updates its cluster list.
Parallel Dijkstra’s algorithm

- **Step 1:** find the closest node in my subgroup.
- **Step 2:** use parallel prefix to find the global closest.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
<th>(d, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1, A</td>
<td>4, A</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>AB</td>
<td>3, B</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5, B</td>
</tr>
<tr>
<td>ABC</td>
<td>4, C</td>
<td>6, C</td>
<td>∞</td>
<td>5, B</td>
<td>3, B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCH</td>
<td>4, C</td>
<td>6, C</td>
<td>∞</td>
<td>5, B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHD</td>
<td>5, D</td>
<td>7, D</td>
<td>5, B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHDE</td>
<td></td>
<td></td>
<td></td>
<td>6, E</td>
<td>5, B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHDEG</td>
<td></td>
<td></td>
<td></td>
<td>6, E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCHDEGF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel Dijkstra’s algorithm

Create a cluster $cl[V]$
Given a source vertex $s$
Each core handles a subgroup of $V/P$ vertices
While (there exist a vertex that is not in the cluster $cl[V]$)
{
    FOR (vertices in my subgroup but outside the cluster)
        Calculate the distance from non-member vertex to $s$
        through the cluster;
        Select the vertex with the shortest path as the local
        closest vertex;
    END
    ** Each processor work in parallel $O(V/P)$ **
    Use the parallel prefix to find the global closest vertex
    among all the local closest vertices from each core.
    ** Parallel prefix $log(P)$ **
}
Parallel Dijkstra’s algorithm

- Running time $O \left( \frac{V^2}{P} + V \cdot \log(P) \right)$
  - $P$ is the number of cores used. In order to obtain the routing table, we need $O(V)$ rounds iteration (until all the vertices are included in the cluster). In each round, we will update the value for $O(V)$ vertices using $P$ cores running independently, and use the parallel prefix to select the global closest vertex, so the running time in each round is $O(V/P) + O(\log(P))$. So, the total running time is $O \left( \frac{V^2}{P} + V \cdot \log(P) \right)$. 
Simulation results and analysis

- **Experiment 1:**
  - Run on one 32-core node, with different size of mesh network model (50*50, 100*100, 150*150).
  - Analyze the performance in terms of different size of network

- **Experiment 2:**
  - The mesh network size is fixed-150*150. The task is run on one 32-core node, three 12-core nodes, sixteen 2-core nodes, respectively.
  - Analyze the performance in terms of different distribute way.

- Implement using OpenMP and all the statistics are the average values for 10 rounds of running.
Experiment 1

- The running time
  - It is obvious that, for the large size network (150*150), the running time is decreasing as the number of cores increases until it reaches the smallest value, then the running time will increase because of the communication latency.
  - For middle size network (100*100), the phenomenon of a reducing running time is not that obvious.
  - For a small size network (50*50), the running time is even increasing as the number of cores increases, because the communication latency outperforms the benefit from using more cores.

Fig. 3   The running time v.s. the number of cores
Experiment 1

- The speed up
  - The speed up is increasing as the number of cores increases until it reaches the maximum value, then the speed up is decreasing.
  - The speed up is increasing because of using more cores.
  - The speed up is decreasing because the communication latency outperforms the benefit from using more cores.
  - As the network size increases, the number of cores used to get the maximum speed up increases. (As shown in the figure, 50*50-4 cores, 100*100-8 cores, 150*150-16 cores)
Experiment 1

- The cost
  - The cost is increasing because the speed up (or the benefit of a reduced running time) cannot outperforms the cost of using more cores.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>50*50</td>
<td>0.06587</td>
<td>0.08351</td>
<td>0.13073</td>
<td>0.33906</td>
<td>1.16115</td>
<td>7.37132</td>
</tr>
<tr>
<td>100*100</td>
<td>1.04358</td>
<td>1.11022</td>
<td>1.22707</td>
<td>1.89479</td>
<td>4.29782</td>
<td>14.0981</td>
</tr>
<tr>
<td>150*150</td>
<td>5.23908</td>
<td>5.38029</td>
<td>5.75562</td>
<td>6.64937</td>
<td>12.4086</td>
<td>36.0456</td>
</tr>
</tbody>
</table>

Fig. 5 The cost v.s. the number of cores
Experiment 2

- The running time
  - The running time is decreasing as the number of cores increases when all the cores are in the same node.
  - When cores from different nodes are used, the running time is increasing dramatically as shown for 16*2-core and 3*12-core.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>16*2-core</td>
<td>4.37263</td>
<td>2.36723</td>
<td>3.97442</td>
<td>5.38834</td>
<td>7.91071</td>
<td>12.9382</td>
</tr>
<tr>
<td>3*12-core</td>
<td>4.65692</td>
<td>2.40176</td>
<td>1.24577</td>
<td>0.69465</td>
<td>2.58422</td>
<td>5.41149</td>
</tr>
<tr>
<td>1*32-core</td>
<td>5.23908</td>
<td>2.69014</td>
<td>1.43890</td>
<td>0.83117</td>
<td>0.77554</td>
<td>1.12642</td>
</tr>
</tbody>
</table>

Fig. 6 The running time v.s. the number of cores
Experiment 2

- The speed up
  - The speed up is increasing as the number of cores increases if the cores are from the same node.
  - When cores from different nodes are used, the speed up is decreasing significantly as shown for 16*2-core and 3*12-core.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>16*2-core</td>
<td>1.84715</td>
<td>1.10019</td>
<td>0.81149</td>
<td>0.55274</td>
<td>0.33796</td>
<td></td>
</tr>
<tr>
<td>3*12-core</td>
<td>1.93895</td>
<td>3.73818</td>
<td>6.70394</td>
<td>1.80205</td>
<td>0.86056</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7 The speed up v.s. the number of cores
Experiment 2

- The cost
  - The cost is increasing as the number of cores increases.
  - The cost of a 16*2-core is much higher than the cost of 3*12-core and 1*32-core.
Reference


Questions?
Thank you!