Parallelisation of graph algorithms in Java

Team 11

Bidirectional breadth first search

- What?
  - A search algorithm for Directed Acyclic Graph without negative edges
- Why?
  - Slow. Potentially can have a nice speed up, assuming it is suitable for parallelization
- Sequential implementation
- Parallel implementation
  - Known issues

What is a bidirectional BFS

BFS starts from both ends, for this project no heuristics are used hence => Bidirectional Dijkstra

Size complexity of blind bidirectional search :

- Assuming d is length of optimal solution path & that b is effective branching factor for both directions, then

  $$2 \times \frac{b^{d/2} - 1}{b - 1}$$

- Completeness: Yes
- Optimality:
  - True if the DAG is uniformly weighted, and the heuristic is admissible (or no heuristic)
  - False otherwise

Sequential implementation

Dijkstra starting from both source and sink. Given Q the queue for frontiers, s the source, d the sink, we have the following:

- After all Dijkstra iterations, for every node u not inside Q, L(u) is the length of the shortest s − u − path. (u, any intermediary nodes between s and d)
- At the same time we could execute another Dijkstra on the graph with reversed arcs. Now we have the length of the shortest v − d − path for each node v not in this second priority queue too.
- When a node gets outside both priority queues, we know the shortest path.
- A degree of freedom in this method is the choice whether a forward or backward iteration is executed.
- Simply alternate or choose the one with lower minimum d in the queue are examples of strategies.

Parallel implementation

Each side goes one level further before dispatching tasks.

Following is a possible approach (coarse grained):

- Assuming the number of processors is 8. Branching factor is greater than 4 for both sides.
- After one sequential iteration, both the source and the sink have fully discovered their children/parents. Now we have two sorted frontiers (priority queues) with each length equal to the number of vertices in the graph minus one (because the source has been removed from the source frontier, the same happens to the sink frontier).
- Then partition each frontier. Let’s call the nodes in the frontiers with assigned costs (meaning they are discovered) the heads and the rest of the frontier the tails. Divide the head list by 4 in this case. Combine them individually with the tails. Now they are partitioned frontiers.
- Assign partitioned frontiers to tasks, resulting in 4 source tasks and 4 sink tasks
- Then running them in parallel yields the desirable degree of concurrency (in this case, the degree of concurrency is 8)

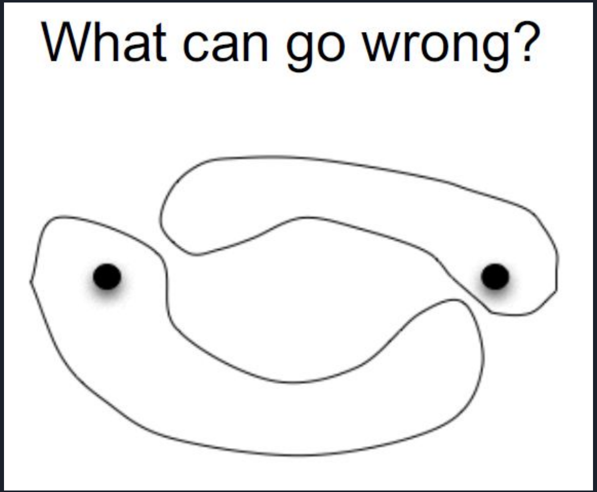
Issues

- Parallelizing inner loop (branching towards children), results in a massive amount of tiny tasks. Each of them requires almost negligible processing time comparing with the induced overheads. Hence, the benefit of parallelization is diminished. (Fine grained vs. Overhead)
- Outer loop has a strong inter loop dependence due to the frontier. The next iteration will pick a node to explore based on the frontier processed by the previous iteration. Refactoring the looping construct does help but it will cause other issues.
- Making everything private for each task and reducing the local costs when joining is an approach to decompose the outer loop. The tradeoff is extra instructions imposed by preparing the private ‘space’ for each worker, and those instructions are not parallelizable and take times.

Issues

- Some data structure cannot be shared. For example, maps that record the least cost for each node from both the source and the sink. Another one is the map that records the 'path', i.e., the unique predecessor for each node in the final solution. Hence additional synchronization must be in place, which induces overheads.
- 'Mismatching peers'. Due to lost update, a worker can spend more iterations before discovering the meeting point from its peer. In the worst case, the search space of a particular worker can be way larger than its peer, hence, unbalancing loads.
- Overlapping search space, the search space of each worker is very likely to overlap with the others'. Therefore, lots of time wasted in repeated searches in which the least cost path is not updated at all.
- Stopping criterion in sequential search is not reusable in parallel search, because of lost update in the meeting points. Parallel stopping criterion takes an additional O(n) time to traverse the closed lists of each side.

A systematic illustration of mismatching peers



Floyd Warshall Algorithm

An algorithm for finding shortest paths

- Features
    - Dynamic programming
    - Any of all the nodes can be a source
    - Edge weight can be negative
    - Limitation
        - No negative cycles

Floyd Warshall Algorithm

- Algorithm
  - Sequential version
    - Compare all possible paths through the graph between each pair of vertices
    - Calculate shortestPath(i,j,k) In one iteration

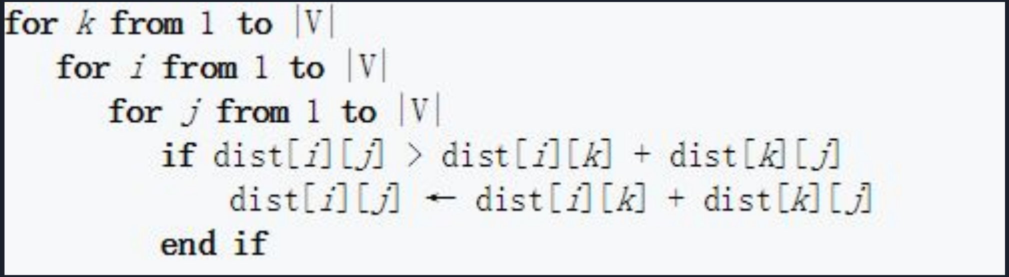
      If distance[i][j] > distance[i][k] + distance[k][j]

      Distance[i][j] = distance[i][k] + distance[k][j]

    - K is the intermediate point and could be any node in the graph

Floyd Warshall Algorithm

- Sequential version



```
for $k$ from 1 to |V|
    for $i$ from 1 to |V|
        for $j$ from 1 to |V|
            if dist[$i$][$j$] > dist[$i$][$k$] + dist[$k$][$j$]
                dist[$i$][$j$] ← dist[$i$][$k$] + dist[$k$][$j$]
            end if
```

- How to parallel
    - Implement parallelisation on for loops
        - Not every loop
        - Loop scheduler

Floyd Warshall Algorithm

- Known issues for parallelisation
  - Dependences
    - Inter loop dependences
  - Possible solution
    - Make TASK private
    - Reduce

Iterative deepening depth-first search

- Features
  - Combined DFS and BFS
  - DFS's space-efficiency
  - BFS's fast search (for closest node to root)

	Time complexity	Space complexity
DFS	O(b^d)	O(d)
BFS	O(b^d)	O(b^d)
IDDFS	O(b^d)	O(b*d)

b is number of children of each node
d is depth

Iterative deepening depth-first search

- Algorithm
    - IDDFS calls DFS for different depths starting from the source.
    - In every call, DFS is restricted from going beyond given depth.
    - IDDFS is kind of a BFS version of DFS.

Iterative deepening depth-first search

- How to parallel
  - Recursive implementation of DFS is hard to parallel
  - Using stack to implement a loop version of DFS
  - Then multiple threads can access the stack together to resolve nodes.
- Problem
  -

Thank You