CPSC 524 Final Project Report:

Single-source Shortest Paths

kd538 - Keyang Dong

PART-0 | Directory Structure

Environment:

1) Log on to Omega clusters;
2) module load Langs/Intel/15.

Code:

1) serial.cpp: serial version of moore algorithm;
2) parallel1.cpp: loop-based parallelism schema;
3) parallel2.cpp: task-based parallelism schema (naive) ;
4) parallel3.cpp: task-based parallelism schema (advanced) .

To test:

Make executable targets using:

1) `make serial`; ./serial <path-to-data-file>
2) `make parallel1`; ./parallel1 <path-to-data-file>
3) `make parallel2`; ./parallel2 <path-to-data-file>
4) `make parallel3`. ./parallel3 <path-to-data-file>

Output example:

```
[kd538@compute-45-4 proj]$ ./serial Square-n.11.0.gr
d(1, 1) = 0
d(1, 1001) = 16502
d(1, 2001) = 19470
d(1, 2025) = 7955

Took 0.001 s to compute.
```

Data Format Notice:

Due to the complexity of text file formatting recognition of C/C++, I chose to modify the first several rows, delete extra comments at the start of USA-**.gr data sets, making the format look like:

```
c … // first row of comment
p sp <%d> <%d> // nNodes & nArcs
a <%d> <%d> <%d> // edges and weights
…
a <%d> <%d> <%d> // continue till the end of file
```

PART-1 | Serial Version

There are several popular algorithms and their variations for solving single-source shortest-path problems (within the domain of weighted undirected graphs), among which Moore Algorithm and Dijkstra Algorithm are two most famous.

The full version of Moore algorithm, also called Bellman–Ford algorithm, is expressed in the following pseudocode. It is generally slower than Dijkstra algorithm (O(VE) or O(V^2 * logV)), but more versatile and can dispose negative weights.

```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
       distance[v] := inf             // At the beginning , all
vertices have a weight of infinity
       predecessor[v] := null         // And a null predecessor

   distance[source] := 0              // Except for the Source, where
the Weight is zero
```

```
    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

As our problem assumes non-negative edge weights, Moore algorithm can be simplified and optimized using a FIFO queue:

```
// optimized moore algorithm that suits the very STP problem,
// which returns shortest distances from a given source vertex inplace;
// adjList is the adjacency list representation of graph,
// elements of which are vertices including referral to their neighbors.

void moore(int source, int nNodes, int dist[], vector<AdjListEntry>&
adjList) {
    for (int i = 0; i < nNodes; ++ i) dist[i] = INF;
    dist[source] = 0;

    queue<int> fifo_q;
    fifo_q.push(source);

    int vi, vj, new_dist;
    while (fifo_q.size() > 0) {
        vi = fifo_q.front();
        fifo_q.pop();

        for (int j = 0; j < adjList[vi].neighbors.size(); ++ j) {
            AdjListNode& adjNode = adjList[vi].neighbors[j];
            vj = adjNode.node;
            new_dist = adjNode.weight + dist[vi];

            if (new_dist < dist[vj] || dist[vj] == INF) {
                dist[vj] = new_dist;
                fifo_q.push(vj);
            }
        }
    }
}
```

A more well-known algorithm, Dijkstra algorithm for shortest paths, has time complexity of O((E+V)*logV) when enabling binary heap optimization. It performs better when encountering dense graphs.

```
 1  function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph: // Initialization
 6          dist[v] ← INFINITY   // Unknown distance from source to v
 7          prev[v] ← UNDEFINED // Previous node in optimal path from
source
 8          add v to Q // All nodes initially in Q (unvisited nodes)
 9
10      dist[source] ← 0 // Distance from source to source
11
12      while Q is not empty:
13          u ← vertex in Q with min dist[u]    // Node with the least
distance will be selected first
14          remove u from Q
15
16          for each neighbor v of u:       // where v is still in Q.
17              alt ← dist[u] + length(u, v)
18              if alt < dist[v]:      // A shorter path to v has been
found
19                  dist[v] ← alt
20                  prev[v] ← u
21
22      return dist[], prev[]
```

As tested, because the provided datasets are mainly sparse graphs with *edges / vertices < 4*, moore generally performs better than dijkstra.

PART-2 | Parallel Version

This section will briefly depict the parallelism models I selected and implemented (using OpenMP) for calculation speed-up.

2.1 Loop-based Parallelism

In each iteration of popping from the head of FIFO queue, updating its neighboring vertices' shortest distance from the source, and append those updated vertices to the tail of queue, there is a clear parallelism in distributing the updating of neighbor nodes.

In this model, the only shared data structure that could cause probable confliction is FIFO queue, so each appending of updated node to its tail should be guaranteed an atomic operation. No other modifications are required.

2.2 Naive Task-based Parallelism

We can think the process of spreading calculation within the graph as a kind of analogy to BFS (breadth-first search), so a naive thought is to distribute each vertex current in FIFO queue to different threads as separate tasks, each has a local queue, updates its input vertex's neighbors, append modified ones to local queue, and finally link their local queues together to the new global FIFO queue maintained by task generator thread (root). Then the algorithm substitute the concatenated new queue with old FIFO queue, enter the next iteration to generate more tasks.

In this model, the only critical section is linking local queue to global FIFO queue, at the exit point of each task. If we implement queues as linked lists, then this operation need no copying of elements, making a least coordination between threads. So there is a possibility it will be faster than loop-based parallelism.

2.3 Advanced Task-based Parallelism

One can easily tell from the experimental results of the two parallelism models above (stated in the next PART), that in sparse graphs, we are overestimating the cost of thread coordination and locks, while failed to focus more on increasing the number of launched parallel threads doing calculation. Due to the serial nature of BFS-kind problems (like a tree spanning), more radical parallelism models should be introduced.

In this model, there is no more local queues linking together to substitute the old global queue. There is always one global queue distributing a task whenever it pops a vertex from its head, and as long as locks are acquired, any thread is allowed to append an updated vertex to its back. That saying, locking and unlocking can happen multiple times during any task.

This schema could be more efficient than the previous two, but also introduces the difficulty of determining termination of iteration (task generation). At any time point, the queue could be empty, but more updated vertices are on the way of calculating by other threads, haven't ready to append to it to generate new tasks immediately. So we have to keep an additional shared counter recording number of active threads at this very moment, and see if FIFO queue is empty as well as no worker process is still working. The updating of this counter (enter a worker thread will plus one, will submitted all of its work and exit will minus one) need to be in another critical section themselves.

PART-3 | Timing Comparation

3.1

Data Set	nNodes	nArcs	nArcs / nNodes
Square-n.14.0.gr	16384	65024	3.97

Version	Serial	FOR Parallel	TASK Parallel	TASK+ Parallel
Time (s)	0.012	0.012	0.023	0.012

Naive task (parallel 2) worst, other methods' performances are very close.

3.2

Data Set	nNodes	nArcs	nArcs / nNodes
Square-n.18.0.gr	262144	1046528	3.99

Version	Serial	FOR Parallel	TASK Parallel	TASK+ Parallel
Time (s)	0.557	0.546	1.041	0.560

Naive task (parallel 2) still worst, loop-based (parallel 1) best with little improvement.

3.3

Data Set	nNodes	nArcs	nArcs / nNodes
USA-road-d.NY.gr	264346	733846	2.78

Version	Serial	FOR Parallel	TASK Parallel	TASK+ Parallel
Time (s)	2.242	2.205	2.559	1.345

Very sparse, naive task (parallel 2) still worst, but advanced task (parallel 3) has a 40% improvement.

3.4

Data Set	nNodes	nArcs	nArcs / nNodes
Long-n.18.0.gr	262144	1015776	3.87

Version	Serial	FOR Parallel	TASK Parallel	TASK+ Parallel
Time (s)	26.625	26.611	59.867	29.068

No significance improvement, naive task (parallel 2) worst, loop-based best.

3.5

Data Set	nNodes	nArcs	nArcs / nNodes
USA-road-d.NY.gr	1524453	3897637	2.56

Version	Serial	FOR Parallel	TASK Parallel	TASK+ Parallel
Time (s)	32.683	32.163	36.737	22.247

Naive task (parallel 2) still worst, advanced task-based (parallel 3) best with almost 32% improvement.

PART-4 | Conclusions

These BFS-like problems in sparse graphs are generally of low level of parallelism, thus when implementing the parallel version of either moore or dijkstra, we don’t need to worry too much about the sharing data structure updates overhead caused by locks, critical sections management, etc.; rather, we should distribute the tasks as wide as possible, minimize any unnecessary serial part.

Regarding the implementations of parallel Moore algorithm, Advanced Task-based model (2.3) performs best in average, can sometimes achieve a speedup of nearly 30-40% in comparing to serial version. Meanwhile Naive task-based model is almost useless and degrading the performance.

To measure load balance of these task-based parallelism schemas is hard, almost impossible and not quite meaningful. Lightweight threads are frequently created and destroyed. In task-based schema, whenever a thread did fewer work than average and exits, its successors will sooner be granted a new task to compute, achieving a well-balanced computational load.

Meanwhile in loop-based schema, imbalance could exist, but not significant due to the sparsity of neighboring vertices. If should be designed to frequently process dense graphs, threads could be made first communicating with each other and share their neighbor information, but this cost is not ignorable, also greatly complicates the program logic.

One crucial difficulty of measuring load balance among threads is their lack of self-identification; in other words, when a thread is settled up and starts running, the id it obtains from omp_get_thread_num() is usually not unique, if there exist two threads in different parallel region or even on different cores.