CS 547
Fall 2013

Conclusions

Parallelized Dijkstra's Algorithm
Project 2

Elijah Houle
Gregory Norton
Bo Wang

December 6, 2013

Introduction

The team selected the “Parallel Twist” implementation of Dijkstra's Algorithm as presented in the instructors lecture notes as one of the technical showpieces for Project Two. As a real time simulation, our game depended heavily on units navigating across “ground” and “open sky” maps independently once directed to attack/move, and the concern with using a traditional single-threaded implementation of Dijkstra's Algorithm was that performance would be poor and frame rate would suffer.

Initially, only one JUnit test was created to test the single-threaded version of Dijkstra's Algorithm against the parallelized implementation using a grid map similar to those that the game would employ. However, once we started testing, the surprising result of the test proved that for the limited edge count of the game map, the single-threaded implementation would always execute the path mappings faster, and the team developed a second test with higher edge counts in an attempt to create a situation where a parallel implementation of Dijkstra's Algorithm would prove to be more appropriate than a single-threaded version.

Test Method

The team tested the two implementations of Dijkstra's Algorithm created for the project using a pair of JUnit classes, included in the source code listings under Appendix A and in the project repository on GitHub at https://github.com/gsnorton/CS-547-Project-Two. The classes differed in the topology of the graph, edge counts, and potential path lengths.

GridTest.java	This test simulated the kind of graph that the team utilized for movement maps in Project Two. Each vertex node has between 3 (corners) and 8 (inside nodes) edges leading to adjacencies.
WumpusTest.java	The graph is similar to a maze created for the classic computer game “Hunt the Wumpus”. Borrowing the maze generation concepts from Barski's “Land of Lisp”, the intent of this test was to emphasize the strength of the parallelized implementation with shorter paths than the GridTest due to large edge counts connecting each node.

Dijkstra's Algorithm Implementations

Single-Threaded	The traditional implementation of Dijkstra's Algorithm as presented in “Introduction to Algorithms” and other computer science texts. The development team utilized the code Greg modified for his Project One effort which he obtained from the website referenced in the comments of DijkstraAlgorithm.java.
“Parallel Twist”	Using the single-threaded version as a starting point, the team implemented their interpretation of the “Parallel Twist” algorithm as presented in the instructor's lecture notes. To make better use of CPU

resources, the team opted to use the Fork/Join parallel programming technique introduced in JSR 166 which became a permanent Java feature as of JDK 1.7.

Test Hardware/OS/JRE

The test system consisted of an Intel Q6600 CPU, 2.4 GHz quad core on an ASUS P5Q Pro Turbo motherboard equipped with 16 GB of RAM running Windows 7. The JRE used for testing was 1.7.0_45.

Testing Results

1. The development team ran the single-threaded implementation of Dijkstra's Algorithm through the GridTest class to establish a base line for comparison against the parallel version. The loop calling the dijkstra.execute() method was called just once to get an approximate set-up time for the adjacency processing by the constructor.

Test	Time
1	0.135 Seconds
2	0.134
3	0.136
4	0.134
5	0.135

Giving an approximate setup time require of 0.135 Seconds.

2. The single threaded implementation was run again with the GridTest class, calling the dijkstra.execute() method 100 times by reconfiguring the 'for' loop appropriately. Subtracting the average time above as the setup time, the result divided by 100 would be the approximate running time of the implementation with the map size of the test.

Test	Time
1	0.254 Seconds
2	0.257
3	0.256
4	0.257
5	0.256

Averaging and subtracting the setup time gave an approximate running time of 1.21 mS for each call to dijkstra.execute().

3. Repeating (1) for the parallelized implementation with a 4-way split among adjacency processing resulted in the following times:

Test	Time
1	0.179 Seconds
2	0.178
3	0.178
4	0.182
5	0.183

Average: .180 Seconds

4. Repeating (2) for the parallelized implementation resulted in the following times:

Test	Time
1	1.478 Seconds
2	1.51
3	1.491
4	1.493
5	1.497

Averaging and subtracting setup time from (3) gave an approximate running time of 13.1 mS per call to dijkstra.execute().

Lowering the parallel split from 4-way to 2-way resulted in the following times:

Test	Time
1	0.503 Seconds
2	0.771
3	0.458
4	0.773
5	0.776

Average: .656 S or an approximate running time of 4.8 mS per call once the setup time from (3) was subtracted. Decreasing the parallel split of adjacency processing improved the running time of the test, but the running time (2) was still significantly better.

5. Moving on to the WumpusTest.java JUnit test, the first run using the single-threaded Dijkstra's Algorithm implementation with a single call to dijkstra.execute() produced the following times:

Test	Time
1	5.403 Seconds
2	5.379
3	5.38
4	5.377
5	5.381

Average: 5.384 Seconds

6. Running the single-threaded implementation with a loop of 100 calls to dijkstra.execute() produced the following times:

Test	Time
1	6.218 Seconds
2	6.177
3	6.185
4	6.193
5	6.245

Average: 6.20 Seconds or a running time of 8.20 mS per call once the setup time found in (5) was subtracted.

7. Repeating (5) with the parallelized implementation with a 4-way split of adjacencies resulted in the following times:

Test	Time
1	5.506 Seconds
2	5.486
3	5.434
4	5.46
5	5.474

Average: 5.472 Seconds.

8. Repeating (6) with the parallelized implemention and 4-way adjacency split:

Test	Time
1	11.638 Seconds
2	11.682
3	11.61
4	11.525
5	11.564

Average: 11.6 Seconds giving a running time of 61.3 mS per call to dijkstra.execute() once the setup time established in (7) was subtracted.

Reducing from 4-way to 2-way adjacency count produced the following run times:

Test	Time
1	6.736 Seconds
2	8.022
3	6.71
4	8.007
5	7.964

Average: 7.49 Seconds giving a running time of 20.1 mS per call once the setup time was subtracted.

Conclusion

Based on our initial test data above, the conclusion was that the parallel implementation of Dijkstra's algorithm was insufficient for our project's requirements. More interesting was that in the WumpusTest, a test designed to take advantage of the strengths of the parallel version in handling large numbers of adjacencies departing each node, performance was still significantly better.

When implementing Dijkstra's Algorithm according to the Parallel Twist method outlined in the instructor's notes, an expensive coordination effort is required to select a “winner” from the adjacent nodes processed by the threads. In addition to the cost of stopping/starting the adjacency processing threads, certain cost calculations must be shared from each thread to the others in our implementation. It could be that this sharing is unnecessary, but, given our time frame and the ambition of our project, further exploration was not possible.

Further expansion of this test method should involve additional test machines featuring both newer and older CPUs and, if possible, different architectures such as ARM and PowerPC.

Appendix A

GridTest.java

```
/*
 * BASED ON Dijkstra Java code from
 *
 * http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html
 *
 * Version 1.1 - Copyright 2009, 2010, 2011, 2011 Lars Vogel
 *
 * MODIFIED BY Gregory Norton to address performance concerns.
 *
 * Eclipse Public License
 */

package dijkstra.test;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import static org.junit.Assert.*;

import org.junit.Test;

import dijkstra.engine.ParallelDijkstraAlgorithm;
import dijkstra.engine.DijkstraAlgorithm;

import dijkstra.model.Edge;
import dijkstra.model.Graph;
import dijkstra.model.Vertex;

public class GridTest {

      private List<Vertex> nodes;
      private List<Edge> edges;

      private Random rand = new Random();

      @Test
      public void test() {
            nodes = new ArrayList<Vertex>();
            edges = new ArrayList<Edge>();

            final int x = 20;
            final int y = 20;

            for (int j = 0; j < y; j++) {
                  for (int i = 0; i < x; i++) {
                        int n = j*x + i;

                        Vertex location = new Vertex("Node_" + n);
                        nodes.add(location);

                        if(0 < i) addLane(n, n-1, 100);
                        if(0 < j) addLane(n, n-x, 100);
                        if((0 < j) && (0 < i)) addLane(n, n-x-1, 141);
```

```
                        if ((0 < j) && ((x - 1) > i)) addLane(n, n-x+1, 141);
                  }
            }

            List<Vertex> path = null;

            Graph graph = new Graph(nodes, edges);

            /* ------------------------------------------------------- */

            /* TESTING: Select implementation and execution pass count */

            DijkstraAlgorithm dijkstra =
                        new DijkstraAlgorithm(graph);

//          ParallelDijkstraAlgorithm dijkstra =
//                new ParallelDijkstraAlgorithm(graph);

#define PASS_COUNT 100

            int source = 0, start = x*y;

            for (int n = 0; n < PASS_COUNT; n++) {
                  source = rand.nextInt(x*y);
                  dijkstra.execute(source);

                  start = source;
                  while (start == source)
                        start = rand.nextInt(x*y);

                  path = dijkstra.getPath(start);
            }

            /* ------------------------------------------------------- */

            System.out.println(start + " to " + source);

            assertNotNull(path);
            assertTrue(path.size() > 0);

            for (Vertex vertex : path) {
                  System.out.println(vertex);
            }
      }

      private void addLane(int sourceLocNo, int destLocNo, int duration) {
            String laneId = String.format("lane_%d_%d", sourceLocNo, destLocNo);
            Edge lane = new Edge(laneId, nodes.get(sourceLocNo),
                        nodes.get(destLocNo), duration);
            edges.add(lane);

            laneId = String.format("lane_%d_%d", destLocNo, sourceLocNo);
            lane = new Edge(laneId, nodes.get(destLocNo), nodes.get(sourceLocNo),
                        duration);
            edges.add(lane);
      }
}
```

WumpusTest.java

```
/*
 * Test Dijkstra's Algorithm modules using a maze like the one generated for
 * "Hunt the Wumpus".
 */

/*
 * H/T: Generation of Wumpus maze is based on ideas presented in the "Grand
 * Theft Wumpus" chapter of Conrad Barski's "Land of Lisp". Copyright (C) 2011.
 *
 */

package dijkstra.test;

import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import org.junit.Test;

import dijkstra.engine.DijkstraAlgorithm;
import dijkstra.engine.ParallelDijkstraAlgorithm;
import dijkstra.model.Edge;
import dijkstra.model.Graph;
import dijkstra.model.Vertex;

public class WumpusTest {

	private static final int NODE_COUNT = 1000;
	private static final int LANE_COUNT = 10000;

	private List<Vertex> nodes;
	private List<Edge> edges;

	private static Random rand = new Random(System.currentTimeMillis());

	@Test
	public void test() {
		generateMaze();

		List<Vertex> path = null;

		Graph graph = new Graph(nodes, edges);

		/* ---------------------------------------------------------- */

		/* TESTING: Select implementation and execution pass count */

		DijkstraAlgorithm dijkstra =
				new DijkstraAlgorithm(graph);

//		ParallelDijkstraAlgorithm dijkstra =
//			new ParallelDijkstraAlgorithm(graph);

		int source = 0, start = 0;
```

```
#define PASS_COUNT 100

            for (int n = 0; n < PASS_COUNT; n++) {
                  source = rand.nextInt(NODE_COUNT);
                  dijkstra.execute(source);

                  start = source;
                  while (start == source)
                        start = rand.nextInt(NODE_COUNT);

                  path = dijkstra.getPath(start);
            }

            /* ------------------------------------------------------- */

            System.out.println(start + " to " + source);

            assertNotNull(path);
            assertTrue(path.size() > 0);

            for (Vertex vertex : path) {
                  System.out.println(vertex);
            }
      }

      private void generateMaze() {
            nodes = new ArrayList<Vertex>(NODE_COUNT);
            edges = new ArrayList<Edge>(LANE_COUNT*2);

            for(int n = 0; n < NODE_COUNT; n++) {
                  Vertex node = new Vertex(String.format("%d", n));
                  nodes.add(node);
            }

            int lanes_populated = 0;

            while (lanes_populated < LANE_COUNT) {
                  int source_num = 0;
                  int dest_num = 0;

                  while (source_num == dest_num) {
                        source_num = rand.nextInt(NODE_COUNT);
                        dest_num = rand.nextInt(NODE_COUNT);
                  }

                  String edgeId = String.format("%d_%d", source_num, dest_num);
                  Edge edge1 =
                        new Edge(edgeId,
                              nodes.get(source_num), nodes.get(dest_num), 100);

                  if (edges.contains(edge1)) {
                        //System.out.println("REJECTED " + edge1);
                        continue;
                  }

                  edgeId = String.format("%d_%d", source_num, dest_num);
                  Edge edge2 = new Edge(edgeId,
                              nodes.get(source_num), nodes.get(dest_num), 100);
```

```
                if (edges.contains(edge2)) {
                        //System.out.println("REJECTED " + edge2);
                        continue;
                }

                //System.out.println("ADDED " + edge1);

                edges.add(edge1); edges.add(edge2);
                lanes_populated += 1;
        }

        /* Look for "islands" */

        int node_num = 0;

        for (Vertex node : nodes) {
                boolean is_island = true;

                int edge_count = 0;

                for (Edge edge : edges) {
                        if (edge.getSource() == node ||
                                edge.getDestination() == node) {
                                is_island = false;
                                edge_count += 1;
                        }
                }

                if(is_island) {
                        System.out.println("ISLAND " + node);

                        Vertex dest = node;

                        int dest_num = -1;

                        while(dest == node) {
                                dest_num = rand.nextInt(NODE_COUNT);
                                dest = nodes.get(dest_num);
                        }

                        String edgeId = String.format("%d_%d", node_num, dest_num);
                        Edge edge = new Edge(edgeId, node, dest, 100);
                        edges.add(edge);

                        edgeId = String.format("%d_%d", dest_num, node_num);
                        edge = new Edge(edgeId, dest, node, 100);
                        edges.add(edge);
                }

                node_num += 1;
        }
    }
}
```