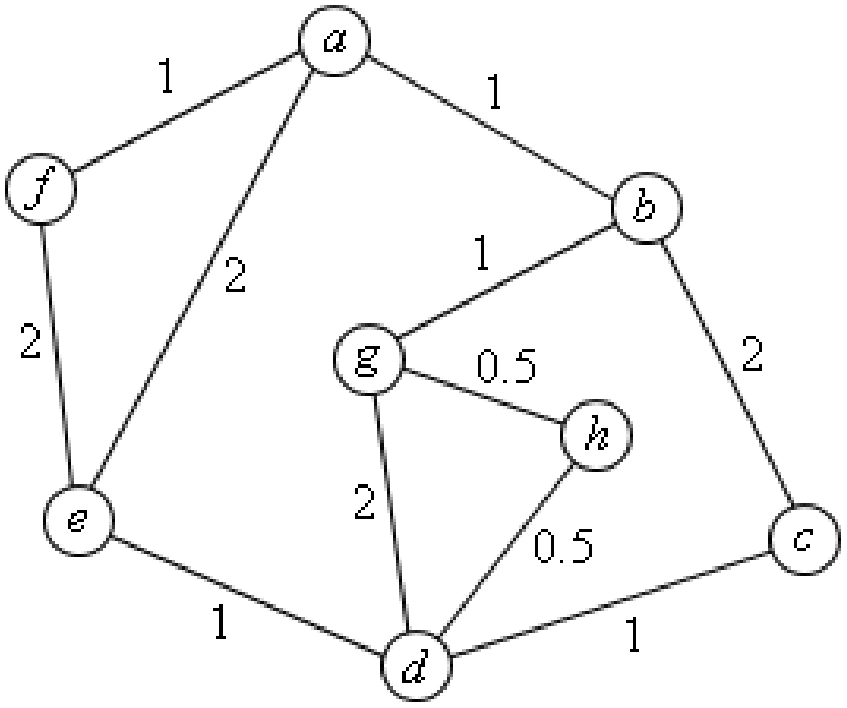
Team WilBauer:

Dijkstra's Algorithm in Parallel



Tom Wilson

Nicholas Hofbauer

Overview

The problem which we decide to parallelize was Dijkstra's Algorithm. It is a graph search algorithm that solves the single source shortest path problem for a graph with non-negative edge costs producing a shortest path tree. In short, it finds the shortest distance from point A to point B on a graph and the shows the route.

The pseudo-code for the algorithm is as follows:

```
 ----------------------------------
for each vertex v in Graph:
        dist[v] := infinity
        previous[v] := undefined
dist[source] := 0
Q := the set of all nodes in Graph
while Q is not empty:
        u := node in Q with smallest dist[]
        remove u from Q
        for each neighbor v of u:
                alt := dist[u] + dist_between(u, v)
                if alt < dist[v]
                        dist[v] := alt
                        previous[v] := u
return previous[]
-----------------------------------------
```

With the print out of the answer simply being:

```
-----------------------------------------

S := empty sequence
u := target
while defined previous[u]
        insert u at the beginning of S
        u := previous[u]
-----------------------------------------
```

This leads us to need the following data in each version of the software we create:

1. Distance from source - mapping a vertex to a distance
2. Unvisited Nodes

3. Previous node - mapping a vertex to the previous node
   This allows us to determine the path of all vertices from the source node to a given destination node

Software Design

The whole of this project involved 4 main classes, the two matrix based programs and the two collection based programs. We decided to try out the performance differences between the two, both in terms of sequential and parallel computation. Both of the collection classes are each supported by a few other classes to give them the needed functionality. The support classes include Edge.java, Vertex.java and Graph.java. Each of these classes were quite simply the objects they're named after. Their main purpose was to hold information and sort it.

The sequential matrix program was a simple graph searching program that used Dijkstra's Algorithm. The graph itself was an adjacency matrix of integer primitives, and the lists of unvisited nodes, previous nodes and the distances from the source were kept in arrays of integer primitives. Most of the loops involved are numerically iterating over one of these structures. The primary loop is the one which goes over the array of unvisited nodes.

The sequential collections program is a slightly less simple graph searching program that used Dijkstra's Algorithm. Instead of using a matrix, it uses a graph object that is made up of Vertex and Edge objects. The previously mentioned arrays are replaced with tree maps to provide quick searching for the correct element. The loops in this program are all archived through standard iteration under the hood.

The parallel Matrix program is structured almost identically to the sequential version. However, by using Parallel Java, the unvisited nodes array is broken up and computed by each of the available processors. This allows each processor to compute part of the distance to the destination node and then combine it to get the total answer at the end.

In a likewise fashion, the parallel collections version is also very similar to its sequential counterpart. However, not only is it different from the sequential version in that it runs in parallel, it's also different in function for the other parallel program in that it does it in a different way. Rather than spiting up the number of unvisited nodes, the parallel collections program split up the part out the algorithm where the distance to each neighbor is computed. The number of neighbors is for each node is split up amongst the processors when the distance calculation takes place.

Developer's Manual

First, it is important to note that all of the programs must have access to the Parallel Java library, including the sequential versions. This is needed to properly run on multiple processor machines. Secondly, all off the programs must be compiled using JDK v5. It is unknown why, but versions about 5v cause problems with running times involving the Parallel Java library.

Both the sequential and parallel matrix programs can be complied alone, but the sequential and parallel collections programs need to be compiled along with Graph.java, Vertex.java, an Edge.java. Those class are required if the collections programs are to function.

If it is sure that you have the PJ library and are compiling with JDK v5, then you may use the following commands to compile each of program's code:

javac DijkstraMatSeq.java

javac DijkstraMatSmp.java

javac DijkstraColSeq.java Graph.java Edge.java Vertex.java

javac DijkstraColSmp.java Graph.java Edge.java Vertex.java

User's Manual

To run any of the programs, you must be using JDK v5. This is because the code was complied using JDK v5, and will not work otherwise. The rest that follows is assuming that the user has the for compiled class files of the four main programs, DijkstraMatSeq.java, DijkstraMatSmp.java, DijkstraColSeq.java, and DijkstraColSmp.java.

To run any of the 4 main programs you will need to have a text file that contains data representing a digraph with no negative edge values and build such that you can reach any node from any other node. The first line of the text file must be the integer number of nodes contained in the digraph represented by the text file. Every line thereafter will represent an edge of the digraph, in the format of:

Node1 Node2 EdgeDistance

The nodes must be integers from 0 to N-1, where N is the number of nodes in the graph. EdgeDistance may be any positive integer. There is a single tab character separating each integer. An example edge:

12 5 62

Upon the start of any of the programs, the command line arguments must contain the digraph text file, the node which the program is to start from, and the node which the program is trying to get to, in that order. The commands for running the sequential programs are as follows:

java DijkstraMatSeq <filename>.txt <sourceNode> <destinationNode>

java DijkstraColSeq <filename>.txt <sourceNode> <destinationNode>

To run the parallel programs, you must designate the number of threads (and therefore proccessors) to use at the start of the program. This is done using java -Dpj.nt=<K>, where K is the number of threads to be used. The full commands to use the parallel program are as follows:

java -Dpj.nt=<K> DijkstraMatSmp <filename>.txt <sourceNode> <destinationNode>

java -Dpj.nt=<K> DijkstraColSmp <filename>.txt <sourceNode> <destinationNode>

Performance Metrics

All times are recorded in milliseconds.

Sequential Matrix

T1	**T2**	**T3**	**Ts**
22413	19463	20913	19463

Parallel Matrix

K	**T1**	**T2**	**T3**	**Ts**	**Speedup**	**Eff.**	**EDSF**
1	17463	17268	17904	17268	1.127	1.127	----

2	11007	11253	10670	10670	1.824	0.912	0.096
3	7581	7579	80231	7579	2.568	0.856	0.084
4	6042	5912	6210	5912	3.292	0.823	0.072
8	3476	3613	3314	3314	5.872	0.734	0.052

Sequential Collections

T1	**T2**	**T3**	**Ts**
81241	81346	79951	79951

Parallel Collections

K	**T1**	**T2**	**T3**	**Ts**	**Speedup**	**Eff.**	**EDSF**
1	76245	76103	77012	76103	1.051	1.051	----
2	44926	45194	44368	44368	1.802	0.901	0.110
3	31464	31492	32423	31464	2.541	0.847	0.090
4	25268	25752	26319	25268	3.164	0.791	0.088
8	13634	13560	14218	13560	5.896	0.737	0.051

What Was Learned

All explanations and pseudo code that we found on Dijkstra's algorithm uses a Queue to store all the unvisited nodes. We then would iterate over the unvisited nodes until the queue is empty, removing the current node and performing the necessary computation. This worked fine in both our sequential program but when we performed this in our parallel version we ran into problems. Java would throw a ConcurrentModificationException since we are iterating over the Queue and remove from it simultaneously. We fixed this in the collections version by using a Map, mapping each Vertex to an Integer representation if the Vertex was visited. As for the matrix and array version we simply used an array that held integers that represented if a node was visited or not.

Future Work

As far as the future is concerned, our first priority is to fix the parallel matrix program. There is no reason that it should be getting such low efficiencies and speedups, so it must be a coding problem. We think that this is most likely due to how the unvisited nodes are distributed across the processors for computation. Another fix to make would be to the parallel collections program. While it might be less obvious, there should be a way to spit up the branching neighbors between the processors even for very low numbers of neighbors. As it stands, the program can only efficiently work for graphs where each node has a number of neighbors that is equal to or higher than the number of processors.

As for new projects, we would create a cluster version of both the matrix and collection versions of the program. We would need to make several modifications to the data structures because as it stands, neither version is capable of sizing up with the number of available machines. If we were successful with that, we would then proceed to make a hybrid SMP-Cluster program for each version. Hopefully we could optimize on both traits and make superior performance.

References

F. Benjamin Zhan, and Charle E. Noon. 1998. Shortest Path Algorithms: An Evaluation Using Real Road Networks. Transportation Science 32(1): 65-73.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 24.3: Dijkstra's algorithm, pp.595–601.

Dijkstra's algorithm. (2009, February 18). In Wikipedia, The Free Encyclopedia. Retrieved 20:14, February 19, 2009, from http://en.wikipedia.org

/w/index.php?title=Dijkstra%27s_algorithm&oldid=271627779

Meijster, A. and T.M. Roderdink . Computation of Watersheds Based on Parallel Graph Algorithms. University of Groningen, 2004.

Traff, Jesper. and Christos Zaroliagis. A Simple Parallel Algorithm for the Single-Source Shortest Path Problem on Planar Digraphs. Journal of Parallel and Distributed Computing, 2000.