Parallelization and analysis of shortest path algorithms using OpenMP

Zachary Sims, April 29th, 2015

Introduction

The shortest path is defined as the path between two different vertices on a graph with the cost of moving from vertex A to vertex B being minimized(). There are two major algorithms that are used when designing a solution to the shortest path problem: Dijkstra's shortest path algorithm and Floyd Warshall's shortest path algorithm(). Both of these algorithms provide the shortest paths between all vertices.

The applications of theses solutions are varied. One major application is in route guidance and logistics(). Route guidance uses shortest path algorithms constantly to update and determine if the current route being assigned is in fact the shortest route possible. The algorithms that are explained below can be modified to check on various conditions to fit the application Route guidance has many varied weights for each route with weather, construction, speed limits, and traffic all affecting the shortest path. With these added weights, it can slow down the operation of the guidance system and negatively affect the route you end up taking.

Another major application is with network router paths. Network routers on the internet or in localized systems need to route information as quickly and as easily as possible. A

problem with route generation comes as the number of connections between different routers increases. For time sensitive applications, e.g. online gaming, the latency from taking a poor path will vary the overall quality of the application incredibly. By choosing the shortest path, latency can be minimized.

A third application is path finding in 3D environments. 3D Video games require an immense amount of constantly update path finding. The problem arises because the human players or the Artificial Intelligence players are constantly moving, requiring the path finding algorithm to constantly update. With a reduced running time shortest path algorithm, the video game could handle more AI/players with constantly updating paths.

Sequential Algorithms

The following two algorithms are the basis for finding the shortest paths between two different vertices. The both provide the same results, but go about finding the shortest paths differently. The inputs for each of the algorithms is an edge list for the entire graph. Both algorithms build the graph as an adjacency matrix and use that to determine all shortest paths. The output is the

Floyd-Warshall's algorithm

Floyd-Warshall's algorithm runs by first computing the shortestPath(*i*, *j*, *k*) for all (*i*, *j*) pairs for *k* = 1, then *k* = 2, etc. This process continues until *k* = *n*, and we have found the shortest path for all (*i*, *j*) pairs using any

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each vertex v
   dist[v][v] ← 0
for each edge (u,v)
   dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
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
```

intermediate vertices. Figure 1 shows the pseudocode for the algorithm. It first sets the diagonal of the distance matrix to 0, as the shortest path from vertex j to vertex j is zero. It then loads in the weight of each edge into the distance matrix. These are the values we will be checking using to determine the shortest path. The main point of the algorithm is to look through all possible routes that can be taken and find the smallest one.

The analysis of the algorithm is determined by looking at n vertices. To find all n^2 of shortestPath(i,j,k) from those of shortestPath(i,j,$k-1$) requires $2n^2$ operations. Since we begin with shortestPath(i,j,0) = edgeCost(i,j) and compute the sequence of n matrices the total number of operations used is $n \cdot 2n^2 = 2n^3$. Therefore, the complexity of the algorithm is $\Theta(n^3)$

Dijkstra's algorithm

Dijkstra's algorithm takes a different approach to finding the shortest path for each vertex. In the following algorithm the code searches for the vertex u in the vertex set Q that has the least dist[u] value. Then the length of the edge joining the two neighbor-nodes u and v is found. Next the length of the path from the root node to the neighbor node v if it were to go through u is calculated using dist[u] and the length of the edge between u and v. If this path is shorter than the current shortest path v, then the current path is replaced with this

```
dist[source] ← 0                        // Distance from source to source
prev[source] ← undefined                // Previous node in optimal path initialization

for each vertex v in Graph:  // Initialization
    if v ≠ source            // Where v has not yet been removed from Q (unvisited nodes)
        dist[v] ← infinity              // Unknown distance function from source to v
        prev[v] ← undefined             // Previous node in optimal path from source
    end if
    add v to Q                     // All nodes initially in Q (unvisited nodes)
end for

while Q is not empty:
    u ← vertex in Q with min dist[u]  // Source node in first case
    remove u from Q

    for each neighbor v of u:           // where v is still in Q.
        alt ← dist[u] + length(u, v)
        if alt < dist[v]:               // A shorter path to v has been found
            dist[v] ← alt
            prev[v] ← u
        end if
    end for
end while
```

new path. Figure 2 shows to pseudocode for this implementation using a Queue data type, but can be implemented using a simple matrix.

The running time of this algorithm depends on the number of vertices V and the number of edges E. Generally for a single vertex shortest path to all other vertices, the running time can be simplified to $\Theta(E*V)$. In order to determine the shortest path to each vertex from each vertex the running times expands to $\Theta(E*V^2)$.

OpenMP Algorithms

The OpenMP algorithms were developed by looking at where and how to algorithms could be parallelized. The two algorithms that were parallelized accessed the data differently and required different approaches in order to work.

Floyd-Warshall's OpenMP

Floyd-Warshall's algorithm is fairly straightforward in order to parallelize. There are no critical sections of the algorithm where multiple threads could try to access the same element. A simple approach was implemented where the middle loop was split into multiple chunks depending on the number of threads. The outer k loop is not able to be parallelized because the algorithm depends on the sequential determination of the k values. The middle loop can be split based on the number of threads and determine all the i,j combinations for the i values assigned to the thread. The implementation is shown in figure 3. The running time of this algorithm, with n vertices, is $\Theta((N^3/p))$.

```
for (k = 0; k < n; ++k)
    #pragma omp parallel for private(i,j)
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
            if ((dist[i][k] * dist[k][j] != 0) && (i != j))
                if ((dist[i][k] + dist[k][j] < dist[i][j]) || (dist[i][j] == 0))
                    dist[i][j] = dist[i][k] + dist[k][j];
```

Dijkstra's OpenMP

Dijkstra's algorithm also provided a simple approach to parallelize. The simplest is to split the workload for each vertex to a specific thread in order to perform all shortest paths. This approach avoids breaking down the code any further than it needs to and allows for simple but effective improvements to be made over the base code. The running time of this algorithm, with V vertices and E edges is Θ($(E * V^2)/p$.).

Evaluation

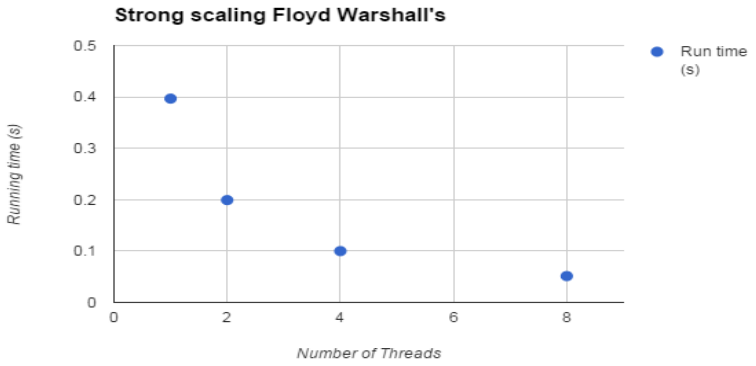
The effectiveness of the was tested three different ways. The first of these is strong scaling where the number of Vertices and Edges remained constant but the number of processors/threads started at 2 and increased to 8. The second is weak scaling where both the number of processors/threads and the number of vertices varied. The number of processors scaled from 2 to 8 while the number of vertices increased from 1000 to 4000 with it to match the increased number of processors. For both of these test groups, the density of the graph (number of edges) remained constant. These two test groups provide a qualitative explanation of how the OpenMP versions of these algorithms performed compared to their sequential versions.

Strong Scaling

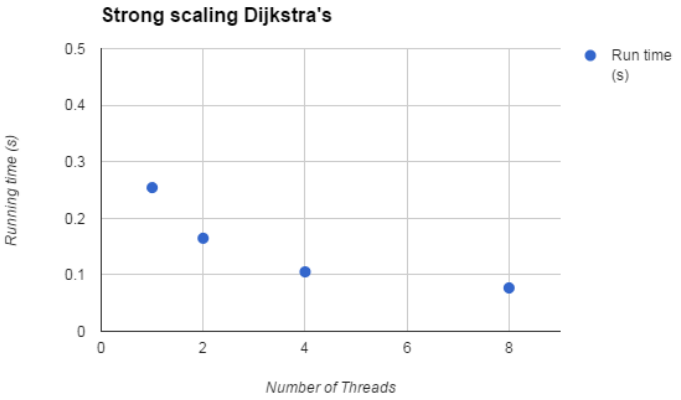
The results from scaling the number of processors while keeping the problem size constant show the speedup of the parallel algorithm. For this evaluation it's important to compare the running times of the parallel algorithms to a base running time. For this, the sequential algorithm with the same number of vertices is used, but with only one processor. The expected running time here should be reduced by the number of processors. For

example, if the base running time for X vertices with P processors is T, then with good strong scaling the running time for X vertices with 2P processors should be T/2.

First, Floyd-Warshall's OpenMP algorithm is compared to its sequential version. For these tests, the number of vertices was fixed at 500. The sequential version of Floyd Warshall's algorithm runs 500 vertices in 0.356s. By looking at the graph to the right, the overhead from thread generation actually slows down the performance of the algorithm, but only by 0.04 seconds. Furthermore, by increasing the number of threads, an improvement is noticeable. The reduction from 1 thread to 2 threads is a 50% reduction in the running time. Continued, the reduction from 2 to 4 threads is 50% as well and the reduction from 4 to 8 threads is 51%. This means the OpenMP algorithm shows good strong scaling as a double in the number of threads results in a halving of the run time. Overall, a maximum speedup of 6.9 is found when the number of threads is 8.



Next, Dijkstra's OpenMP algorithm is ran against its sequential version as well. The sequential version of Dijkstra's algorithm runs 500 vertices in 0.453 seconds. By looking at the graph above, running the openMP

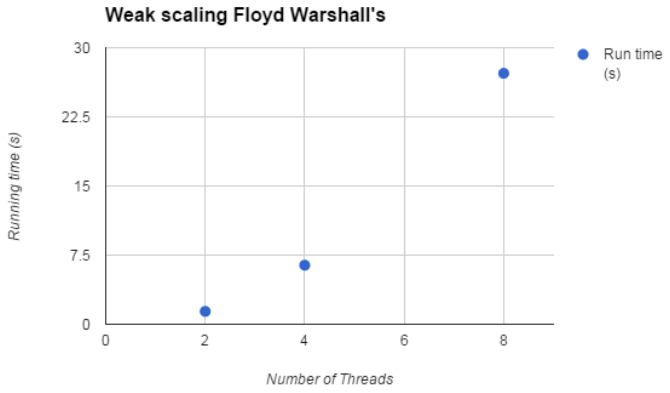


algorithm with only one thread improves the run time by almost half already. The reasoning behind this is unknown at the moment. This implementation also shows a noticeable improvement for strong scaling. The reduction from 1 thread to 2 threads is a 40% reduction in the running time. Continued, the reduction from 2 to 4 threads is 37% and the reduction from 4 to 8 threads is 28%. This means the OpenMP algorithm does not show good strong scaling as increasing the number of threads, does not have an equal effect on the run time. Overall, a maximum speedup of 5.9 is found when the number of threads is 8.

Weak scaling

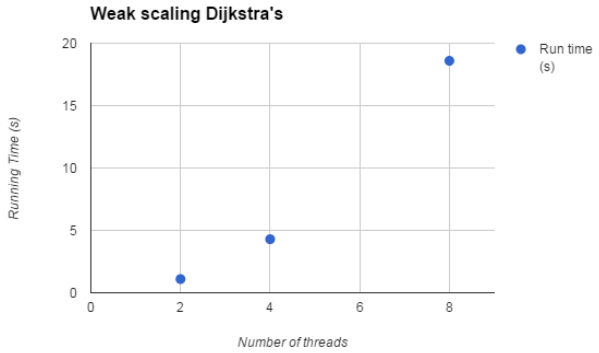
The results from weak the number of processors and the problem size show if the algorithm running time is linear based on the number of processors. For example, if the base running time for X vertices with P processors is T, then with good weak scaling the running time for 2X vertices with 2P processors should be 2T.

To begin, Floyd-Warshall's OpenMP algorithm is ran with a varying number of processors and problem size. The number of vertices goes from 1000 to 2000 and finally 4000, while the number of threads goes from 2 to 4 and finally 8. The running time of the Floyd Warshall algorithm for 1000 vertices is 8.35s and for 2000 vertices it is 114.66s. Running the algorithm for 4000 vertices takes an incredible amount of time, so was excluded in the data. By looking at the graph above, the results of weak scaling can be seen. Going from 1 thread and 500 vertices to 2 threads and 1000 vertices is a running time increase by a factor of 3.8.



Continued, the next step increase is 4.58 times longer to run and lastly the increase for the final step is 4.24. Overall, the increase appears to be linear in run time, as the constant factor increase is ~4 for each next step in weak scaling.

Finally, Dijkstra's OpenMP algorithm is ran with a varying number of processors and problem size. The number of vertices goes from 1000 to 2000 and finally 4000, while the number of threads goes from 2 to 4 and finally 8. The running time of the Dijkstra's algorithm for 1000 vertices is 3.58s and for 2000 vertices it is 32.0057s. By looking at the graph below, the results of weak scaling can be seen. Going from 1 thread and 500 vertices to 2 threads and 1000 vertices is a running time increase by a factor of 4.312. Continued, the next step increase is 3.88 times longer to run and lastly the increase for the final step is 4.33. Overall, the increase appears to be linear in run time, as the constant factor increase is ~4 for each next step in weak scaling.



Future Work

More work on this topic would involve further optimizations of the OpenMP code for both solutions. Another expansion into this field would be to implement these algorithms in other parallel programming languages, such as MPI and a PGAS language. Other algorithms can also be parallelized in the future to see if a specific algorithm would lead to faster running times for weak scaling and strong scaling. Future work also involves testing the algorithms

with various graph densities. While it should show no difference with Floyd-Warshall's, it would make an impact on Dijkstra's algorithm.

Conclusion

The parallelization of both Dijkstra's and Floyd Warshall's algorithm using OpenMP was successful at the very least. The greatest speedup was 6.9 for the Floyd Warshall algorithm. Dijkstra's algorithm provided a speedup of 5.9, but had poor strong scaling compared to Floyd Warshall's algorithm. Dijkstra's could be improved by using a queue, instead of an adjacency matrix, as this would reduce the time spent inside the algorithm for each thread. Overall, the implementation is simple but provides great speed ups for both algorithms.