Parallel CPU-Based Shortest Path in Graph Algorithms

Abhiroop Kodandapursanjeeva and Muhammed Mohaimin Sadiq

Department of Electrical and Computer Engineering, The University of Texas at Austin

***Abstract*— Various shortest path finding algorithms are parallelized using multiple CPU hardware threads. Parallelized Dijkstra's (Java and C), Floyd-Warshall (C), Bellman-Ford (Java and C), and Delta-Stepping (Java) algorithms are implemented and compared to the sequential versions to examine changes in total run time. To further detail the effects of parallelization, tests are performed which vary both the number of hardware threads available to use, as well as the size of the graph. Significant speedups were observed for all algorithms implemented in C when parallelized using OpenMP, but only for Delta-Stepping in Java. Our source code is available at `https://github.com/mohaiminsadiq/EE382C_Term_Project`.**

I. INTRODUCTION

Finding the shortest path in a graph is useful in various applications. In computer networking, sending a packet from one node to another can be accomplished in the fastest possible time by finding the path of least cost. In this sense, the weights of an edge can represent the time taken to send information between nodes. Thus, path finding algorithms can be used to find the optimal path between nodes. Another application is finding the path of shortest distance or least time in transportation. For example, finding the optimal route between two locations on a geographic map is useful for any individual while driving [1]. The weight of an edge in this case can represent either the physical distance of a road, or the average time taken to traverse it (if traffic and speed limit are taken into account). Because either of these applications can feature large numbers of vertices and edges when networks are represented as graphs, optimizing the algorithms is essential to minimizing the time taken to perform common computational tasks. In particular, reducing the time taken to find shortest paths of a graph by utilizing multiple processors can be extremely advantageous. In this paper, we analyze four different path finding algorithms: Dijkstra's, Bellman-Ford, Delta-Stepping, and Floyd-Warshall. Pseudocode for the algorithms is shown in 1 [2], 2 [3], 3 [4], and 4 [5] respectively. These algorithms were selected because they cover a broad range of characteristics. The first three are single source shortest path (SSSP) algorithms, while Floyd-Warshall is an all-pairs shortest path (APSP) algorithm. Bellman-Ford and Floyd-Warshall can also handle negative edge weights.

Algorithm 1 Sequential Dijkstra's Algorithm

```
while visited.size < V:
    Find vertex u closest from source, unvisited
    visited.add(u)
    for Each neighbor n of u:
        if dist[u] + weight[u,n] < dist[n]:
            dist[n] = dist[u] + weight[u,n]
            prev[n] = u
//dist[v] init to INF, prev[v] init to UNDEFINED, dist[0]
//init to 0 =0
```

Algorithm 2 Sequential Bellman-Ford Algorithm

```
for $i < V - 1$:
    for all edges $e$:
        if dist[$e$.src] + $e$.weight < dist[$e$.dest]:
            dist[$e$.dest] = dist[$e$.src] + $e$.weight
            prev[$e$.dest] = $e$.src
//dist[v] init to INF, prev[v] init to UNDEFINED, dist[0]
//init to 0 =0
```

Algorithm 3 Sequential Delta-Stepping Algorithm

```
foreach $v \in V$ do tent($v$) := $\infty$
relax(s,0);
while $\neg$ isEmpty(B):
    $i$ := min{$j \geq$ 0: $B[j] \neq \emptyset$}
    $R := \emptyset$
while $B[i] \neq \emptyset$:
    Req := findRequests($B[i]$, light)
    $R := R \cup B[i]$
    $B[i] := \emptyset$
    relaxRequests(Req)
Req := findRequests($R$, heavy)
relaxRequests(Req)

function FINDREQUESTS($V'$, kind : {light, heavy}))
return $\{(w, tent(v)+c(v,w)) : v \in V' \wedge (v,w) \in E_{kind}\}$
end function

procedure RELAXREQUESTS(Req)
    foreach $(w,x) \in$ Req do RELAX$(w,x)$
end procedure

procedure RELAX(w,x)
    if x < tent(w):
        $B[\lfloor tent(w)/\Delta \rfloor] := B[\lfloor tent(w)/\Delta \rfloor] \setminus \{w\}$
        $B[\lfloor x/\Delta \rfloor] := B[\lfloor x/\Delta \rfloor] \cup \{w\}$
        tent($w$) := $x$
end procedure
=0
```

Algorithm 4 Sequential Floyd-Warshall Algorithm

```
let dist be a |V| x |V| array of minimum distances
initialized to $\infty$
foreach edge $u, v$ do dist[$u$][$v$] := w($u, v$)
foreach vertex $v$ do dist[$v$][$v$] := 0
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[$k$][$j$]
=0
```

II. Project Description

Graphs are constructed for testing by creating square matrices of specified sizes. The edge weights are randomized between 1 and 5, and an average of $\frac{V}{2}$ edges exist for each vertex. Only non-negative edge weights are used, although the Bellman-Ford and Floyd-Warshall algorithms can function for negative edge weights. For the scope of this paper, the presence of negative edge weights will not affect the time complexity of the algorithms, thus was not a point of study. Test conditions were defined as a combination of two variables: number of processors and graph size. Graph sizes of [100..1000] and number of processors between 2 and 256 were used in testing. Times were reported as an average of 5 samples for smoother results. All tests were performed on the Texas Advanced Computing Center's Stampede2 supercomputer,which consists of 68 Intel Xeon Phi 7250 cores. There are 4 hardware threads per core, resulting in a total of 272 hardware threads per node, each operating at 1.4 GHz clock rate. Dijkstra's and Bellman-Ford algorithms are implemented in both Java and C to compare times between the two languages. In addition to timing the algorithms, the results of the algorithms, primarily the minimum path distances from the source vertex to other vertices, were checked to ensure they are the correct values. This was done by first ensuring the sequential version of each algorithm found the same minimum distances as the parallel version (i.e., Dijkstra's sequential found the same minimum distances as Dijkstra's parallel). Cross checking was also done to ensure different algorithms found the same results as well. The actual paths found by the algorithms were checked, though this was not thought to be important to the focus of the study.

To parallelize Dijkstra's, the unvisited, neighboring vertices of the minimum-distance vertex are assigned to individual processors to run concurrently. This becomes an $O(\frac{V}{P})$ operation in time. The outer loop cannot be parallelized because the selected minimum distance vertices must be visited in proper order. Finding the minimum vertex is also parallelized, such that each processor finds the minimum of their subset of vertices, and the global minimum vertex is selected afterwards. This operation is an $O(\frac{V}{P}+P)$ operation in time. Overall, the parallelized Dijkstra's algorithm is expected to run in $O(\frac{V^2}{P}+VP)$ time, compared to the sequential complexity of $O(V^2)$ time.

Bellman-Ford is parallelized by assigning processors a subset of the edges to handle concurrently in the inner loop. Whereas the sequential algorithm will run with a time complexity of $O(VE)$, the parallelized version will run in $O(\frac{VE}{P})$. In our testing, the number of edges, E, is approximately $\frac{V^2}{2}$ for each graph. Therefore, the Bellman-Ford algorithm has a relatively high time complexity for an SSSP algorithm of $O(V^3)$, while the parallelized version has a time complexity of $O(\frac{V^3}{P})$. An improvement to the Bellman-Ford algorithm is also implemented in the C version by using an early termination condition, which is met by checking if no new distance updates are made after each iteration of the inner loop [3]. If the inner loop finishes without updates, no further iterations will lead to updates, thus allowing the algorithm to terminate early and possibly save significant amounts of time. In the case of dense graphs, in which each vertex has a significant number of edges, it is likely that the shortest path from the source to all other vertices is limited to only a few edges. Thus, there will likely not be a need to iterate through all edges $V-1$ times, as paths will likely be much shorter than $V-1$ vertices.

Floyd-Warshall was parallelized by parallelizing the first nested for-loop, thereby assigning each processor one row of the distance matrix to compute. In the sequential algorithm, the two nested loops run in $O(V^2)$ time. For P processors, this would be reduced to $O(\frac{V^2}{P})$ time so that the overall run time of the parallelized version is $O(\frac{V^3}{P})$. A recursive version of the Floyd-Warshall algorithm was also implemented and then parallelized. The details are discussed in III.

Delta-Stepping was parallelized as follows. In the *FindRequests* function, nested Java parallel streams were used: the outer level to process each v in V' in parallel, and the inner level to find the outgoing edges of each of those v's whose weight matched the required criteria in parallel. In the *RelaxRequests* function, each of the *Relax* tasks was done in parallel using the Java *Runnable* interface. This is close to the description of the simple parallelized version mentioned in [4]. For a discussion of run time complexity see [4] Section 1.2.2.

III. Design Alternatives

The choice of programming language was the first design decision. Java is a good choice because of its well developed *Collections* library and parallel stream functionality. However, using OpenMP in C to parallelize the algorithms is much simpler to code, and as will be shown, results in much faster run times. This is thought to be due to the larger time overhead of the JVM handling threads. Dijkstra's and Bellman-Ford were implemented in both languages, while Floyd-Warshall only in C, and Delta-Stepping only in Java.

Regarding Dijkstra's algorithm, it can be improved by utilizing a priority queue to pick the minimum distance node. This is done by adding the neighboring nodes to the queue based on their distance. The time complexity then drops to O($V + Elog(V)$) in sequential.

Regarding Floyd-Warshall, a recursive version of the algorithm was considered, since the original algorithm is based on a recurrence relation. [6] proposes a recursive Floyd-Warshall implementation that splits up the distance matrix, $dist$, into 4 quadrants which are passed into 8 sequential recursive calls in a very specific sequence to ensure the correct result. Recursion proceeds until the dimensions of the quadrants are smaller than a threshold (i.e. the base case) after which sequential Floyd-Warshall is run on the quadrants. The purpose is to improve cache performance through data access patterns that reduce processor-memory traffic. [7] proposes a parallelization for this algorithm. Inspired by this algorithm, and the blocking Floyd-Warshall algorithm in [8] we used a different sequence of recursive calls as proposed in the original auto-blocking matrix multiplication algorithm paper [9] which used the sequence for a different purpose entirely: a quadtree based recursive matrix multiplication algorithm. The pseudocode is shown in 5, where the quadrants are defined as

$$A = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}$$

From the original sequential Floyd-Warshall algorithm, one can observe that the distance matrix is updated in place and each element that is updated depends on 2 other elements of the matrix that should not have been updated yet. For an explanation of why this is possible see [8]. Fortunately, the recursive sequence we tried succeeded in taking care of the interdependency of the elements of the matrix so that the matrix could surprisingly be modified in place with 8 recursive calls in parallel without creating any copies. Further, we implemented the recursive calls using row and column indexes of the quadrants instead of the quadrants themselves so that no data was copied at all. We used the *omp task* directive to parallelize each recursive call.

Algorithm 5 Recursive Floyd-Warshall Algorithm

```
0: function FLOYDWARSHALLSEQUENTIAL(A,B,C)
 for $k$ from 1 to $n$ do
   for $j$ from 1 to $n$ do
     for $i$ from 1 to $n$ do
       if C[$i$][$j$] $>$ A[$i$][$k$] + B[$k$][$j$] then
         C[$i$][$j$] := A[$i$][$k$] + B[$k$][$j$]
 end function

 function FLOYDWARSHALLRECURSIVE(A, B, C, n)
 if $n \leq$ base case threshold then
     FLOYDWARSHALLSEQUENTIAL(A, B, C, n)
 else
     FLOYDWARSHALLRECURSIVE(A11, B11, C11, n/2)
     FLOYDWARSHALLRECURSIVE(A21, B12, C22, n/2)
     FLOYDWARSHALLRECURSIVE(A12, B21, C11, n/2)
     FLOYDWARSHALLRECURSIVE(A11, B12, C12, n/2)
     FLOYDWARSHALLRECURSIVE(A22, B22, C22, n/2)
     FLOYDWARSHALLRECURSIVE(A21, B11, C21, n/2)
     FLOYDWARSHALLRECURSIVE(A12, B22, C12, n/2)
     FLOYDWARSHALLRECURSIVE(A22, B21, C21, n/2)
 end function
   =0
```

IV. Results

Times of both C and Java versions of Dijkstra's algorithm, as well as the sequential and parallel versions of each, are shown in figure 1. It can be observed that the Java versions of the algorithm, run for 1000 vertices, is much slower than the C versions. Time for the Java versions were minimized by first creating the thread objects required for both neighbor-processing and minimum-vertex finding, and utilizing an ExecutorService object to run the threads. This is likely due to the time overhead of handling threads in the JVM, which introduces large amounts of time to properly execute and shut down threads. In addition, we can see that the sequential version in Java is in fact faster than the parallel version, for any number of threads utilized. Therefore, our implementation of Dijkstra's SSSP algorithm in Java does not benefit from any parallelization. The C version however, shows that parallelization is beneficial when using less than 32 threads. As more threads are used, the time taken for this algorithm increases quickly. This is expected, because of the minimum vertex-finding portion of the algorithm. The parallelized algorithm takes $O(\frac{V^2}{P}+VP)$ time, in which the minimum-finding portion accounts for the VP term. As number of processors increase, this term will increase as well, meaning the overall parallel algorithm's speed is limited by this term. At best, we observe that the parallel C algorithm for this graph size is 45% faster when utilizing 8 threads.

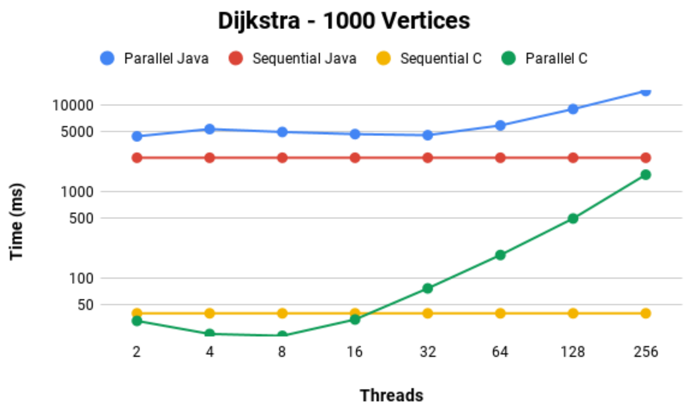

Fig. 1: Times for Dijkstra's Algorithm

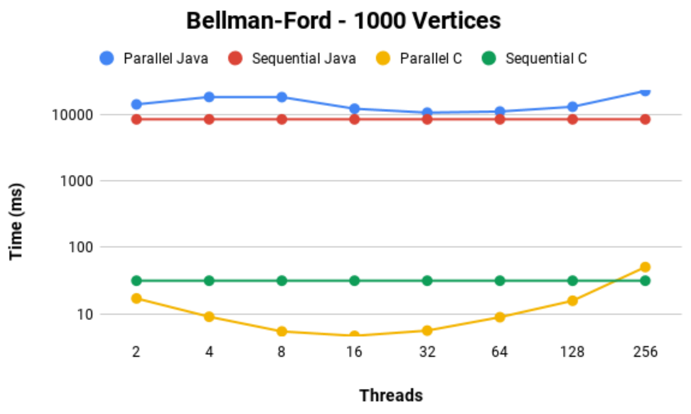

Fig. 2: Times for the Bellman-Ford Algorithm

Times of both C and Java versions of the Bellman-Ford algorithm, as well as the sequential and parallel versions of each, are shown in figure 2. Once again, we can observe that the parallel Java version is similar in time, albeit slightly slower, than the sequential Java version. Again, times were minimized by creating the thread objects before running the path finding algorithm, and allowing an ExecutorService object to handle the threads when needed. This time overhead in Java once again led to longer run times compared to the sequential version. Therefore, our parallel Java solution for Bellman-Ford's algorithm is not an improvement over the sequential Java solution. Again, we can note that the C versions performed much faster than the Java versions, and the parallel C version saw vast improvements in time performance over its sequential counterpart. In addition to lower thread overhead, the C versions utilized the early termination condition, which contributed to significant speedups.

This is easily shown in figure 3, where we can observe that the Bellman-Ford algorithm is capable of finishing much faster than even Dijkstra's, though its time complexity is higher. This may be surprising, but the early termination condition implemented for Bellman-Ford leads to significant speedups, as the shortest paths are found relatively quickly in a dense graph. At best, the parallel Bellman-Ford algorithm in C was 6.75× faster than sequential, when run with 1000 vertices and 16 threads. As more threads are added however, the overhead of using OpenMP threads negates the speedup, thus leading to slower performance versus the sequential solution.

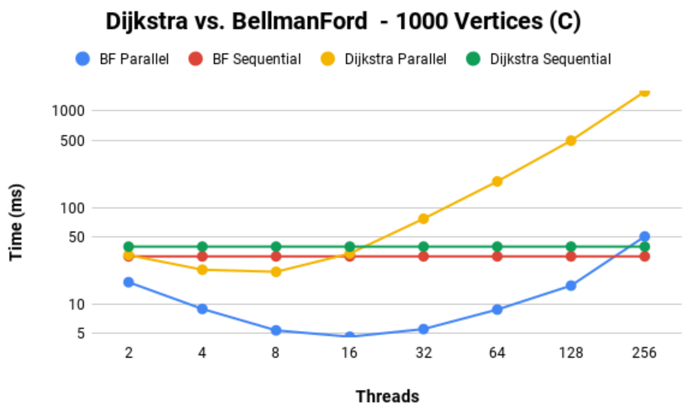

Fig. 3: Times for Bellman-Ford and Dijkstra algorithms, in C

Times for the C implementation of both sequential and parallel Floyd-Warshall versus the number of threads for a graph with 1000 vertices are shown in figure 4. The parallel algorithm is immediately better than the sequential version achieving approximately 2x speedup for just 2 threads. The rate of increase in speedup gradually decreases as we increase the number of threads, but the run times continue to decrease until 256 threads, reaching 42x speedup. We did not test for more threads as a single KNL node on Stampede2 only offers 272 threads.

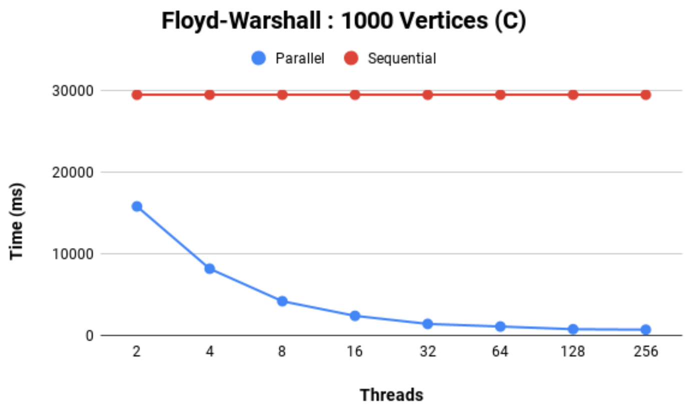

Fig. 4: Times for the Floyd-Warshall Algorithm

Times for the Java implementation of both sequential and parallel Delta-Stepping versus the number of threads for a graph with 1000 vertices are shown in figure 5. A delta value of 1 was used. The parallel algorithm is again immediately better than the sequential version, achieving approximately 2x speedup for just 2 threads. However, run times do not improve after 2 threads, remaining approximately constant up to 8 threads and then becoming gradually longer and exceeding the sequential run time at 64 threads.

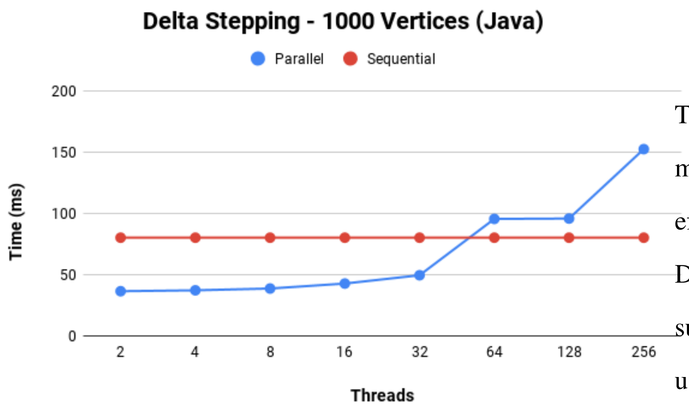

Fig. 5: Times for the Delta-Stepping Algorithm

The parallel recursive implementation of Floyd-Warshall did not improve upon the run time for the iterative parallel version. However, some interesting trends can be seen from the run time data for a graph with 1024 vertices and various dimensions for the quadrants in the base case (the legend indicates these dimensions) in figure 6 . The pink 1024 line is essentially sequential Floyd-Warshall with no recursion, as the base case is the same as the dimension of the input graph. It can be observed that as the dimensions for the base case decreases, run times become more stable until they are roughly constant for dimensions between 4 and 128. However, if the base case dimension is too small (i.e. 2), the recursion overhead causes an upward shift of the run times.

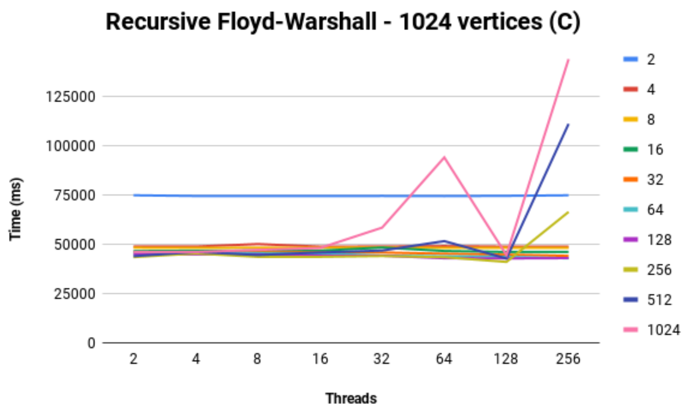

Fig. 6: Times for the Recursive Floyd-Warshall Algorithm

V. Conclusion

This study finds that using C and OpenMP to parallelize minimum path-finding algorithms is much more time efficient than using Java. In addition, we can observe that Dijkstra's, Bellman-Ford, and Delta-Stepping algorithms all suffer from time overhead increases as number of threads used is increased. This is observable in both C and Java, and indicates that thread management can utilize a significant amount resources which can be detrimental to the algorithm's time performance. The standard Floyd-Warshall algorithm however, did not show such a trend and saw only faster performances up to 256 threads.

References

[1] F. Zhan and C. Noon, "Shortest path algorithms: An evaluation using real road networks," *Transportation Science*, vol. 32, pp. 65–73, 02 1998.

[2] A. Javaid, "Understanding dijkstra's algorithm," *SSRN Electronic Journal*, 01 2013.

[3] M. J. Bannister and D. Eppstein, "Randomized speedup of the bellman-ford algorithm," in *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*, ANALCO '12, (USA), p. 41–47, Society for Industrial and Applied Mathematics, 2012.

[4] U. Meyer and P. Sanders, "Delta-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, pp. 114–152, 10 2003.

[5] D. Hochbaum, "Section 8.9: Floyd-warshall algorithm for all pairs shortest paths," tech. rep., University of California, Berkeley, 2014.

[6] J. . Park, M. Penner, and V. K. Prasanna, "Optimizing graph algorithms for improved cache performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 9, pp. 769–782, 2004.

[7] "Parallelizing the floyd-warshall algorithm on modern multicore platforms: Lessons learned," *online:* `https://gkaracha.github.io/papers/floyd-warshall.pdf`.

[8] P. Tang, "Rapid development of parallel blocked all-pairs shortest paths code for multi-core computers," in *IEEE SOUTHEASTCON 2014*, pp. 1–7, 2014.

[9] J. D. Frens and D. S. Wise, "Auto-blocking matrix-multiplication or tracking blas3 performance from source code," *SIGPLAN Not.*, vol. 32, p. 206–216, June 1997.