Parallel CPU Based Graph Algorithms - Shortest Path

EE 382C - Multicore Computing

Aravind Srinivasan, Brendan Ngo

Abstract

This paper will explore the algorithms used to solve the single-source shortest path and the all-pairs shortest path problems on graphs with non-negative edge weights. We first look at attempts to parallelize the sequential algorithms that solve these problems such as Dijkstra's and Floyd-Warshall. Then we will look at algorithms based on matrix operations such as Seidel and Shoshan-Zwick. Comparing the two paradigms we see that parallelizing matrix operations provides a larger speed up in theory, but is not practical on CPU based implementations.

Introduction

The single source shortest path (SSSP) problem and the all pairs shortest path (APSP) problem on graphs with non-negative edge weights are common problems in graph theory. The SSSP problem is as follows: Given a digraph *G* with non-negative weights on its edges and a source node *S*, and a destination node *T*, determine a directed path from *S* to *T* with the minimum total weight. The APSP problem is similar to SSSP. Given a digraph G with non-negative weights on its edges, determine a directed path from every node to every other node that has the minimum total weight.

Single-Source Shortest Path

A well-known solution to this problem is Dijkstra's algorithm[1] that finds the shortest path from *S* to all the nodes the graph. Given *V* nodes and *E* edges, it runs in *O(E + VlgV)*. Dijkstra's

greedily picks the node with the lightest weight and “relaxes” the edges. It does this using a min-heap which inherently does not lend itself to parallelization since every iteration of the while loop in line 12[1] depends on the previous iteration. The previous iteration updates the weights of the nodes in the heap which affects the next element that will be extracted.

Our implementation of parallelizing Dijkstra's attempts to reduce the runtime cost by reducing the time it takes to find the minimum element in the set of unvisited nodes. Finding the minimum in parallel is a widely studied problem and depending on the algorithm used we can see a reduction from $O(n)$ up to $O(1)$, taking the runtime of our algorithm from $O(E + VlgV)$ to $O(E + V)$. The initial matrix is set up the same as in Dijkstra's (line 5 - 8)[1]. Instead of using a min-heap, we maintain a set of nodes that have yet to be visited. At each iteration, we find the minimum of the set in parallel.

Our results comparing Dijkstra’s algorithm sequentially and in parallel are shown in Figure 1. In theory we should be seeing a large reduction in runtime. But in practice the cost of setting up the threads for finding the minimum in parallel in Java is rather high, so some of the gains in run-time is offset by the cost of setting up the parallelization. Using more than 4 threads also leads to a decrease in performance. Since we were using a machine with 4 cores, using more than 4 threads degrades performance due to thread scheduling and context switching. As we use more threads, the cost of context switching and thread scheduling starts goes higher and our runtime goes up.

All-Pairs Shortest Path

The common solution to APSP is the Floyd-Warshall algorithm[2]. It returns a V x V matrix where each row represents the cost of the shortest path from the V-th node to every other node. Sequential Floyd-Warshall runs in $O(V^3)$ time. We notice that Floyds lends itself to parallelization a lot better than Dijkstra's algorithm. The 2 for loops on line 7 and 8[2] can easily parallelized using since their operations are mutually independent. Given P processors, we can bring the runtime down to $O(V)*O(V^2/P)$. Given a large enough P, i.e. V^2 processors, the runtime can be reduced to $O(V)$.

APSP can also be solved by running Dijkstra's algorithm[1] on all V nodes. The sequential runtime of this algorithm is $O(V * (E + VlgV))$. Parallelizing this algorithm can be done by running the V instances of Dijktra's algorithm in parallel, which brings down the runtime to $O(E + VlgV)$ (same as the runtime for sequential Dijkstra's algorithm). The results comparing the two algorithm is shown in Figure 2.

As expected, Parallel Floyd-Warshall runs faster than Parallel Dijkstra's. As seen in the Figure 1, the runtime for both algorithms start going down as we use more than 4 threads. Another point of consideration is the number of edges. Though Floyd-Warshall has a better runtime in the optimal case, in practice we are unlikely to have V^2 cores available to us, especially when dealing with large graphs. Given a large graph that is sparse, and a limited number of cores P, parallel Floyd-Warshall's can actually be slower than parallel Dijkstra's which has a runtime of $O(V/P) * O(E + VlgV)) \cong O(V/P)*O(VlgV) \leq O(V)*O(V^2/P)$.

All-Pair Shortest Path - Matrix Multiplication Based Implementations

Besides the classical methods discussed previously, APSP can also be calculated using matrix calculations. First, we look at the most basic APSP matrix solution, Seidel's Algorithm[3]. Seidel's Algorithm solves the APSP problem for undirected, unweighted graphs in $O(D(V)*log(V))$ runtime, where $D(V)$ is the time to compute the distance matrix. $D(V)$ is the cost to perform matrix multiplication on two VxV matrices in this case. Seidel's Algorithm finds log(V) distance matrices through repeated squaring. Given W, a VxV matrix containing the edge weights of the graph, W^k gives the distances realized by paths that use at most k edges. By recursively computing the adjacency matrices of the squared graphs, we check for all paths in log(V) time. Seidel then calculates the actual path lengths deciding, using one matrix multiplication, for every two vertices u, v, whether their distance is twice the distance in the square, or twice minus 1.

Sequentially, matrix multiplication can be calculated in $O(V^3)$ through brute force, in $O(V^{2.8074})$ using Strassen's Algorithm[4], and, at best, in $O(V^{2.375477})$ using the Coppersmith-Winograd Algorithm[5]. In the brute force methods, each matrix location takes $O(V)$ time to calculate and can be calculated independently. Thus, matrix multiplication can be performed in $O(V)*O(V^2/P)$, where P is the number of available parallel processors. With V^2 processors, matrix multiplication can be performed in $O(V)$. This can be further optimized by performing the tree-based algorithm[6] for each matrix location. Thus, with V^3 processors, matrix multiplication can ideally be performed in $O(log(V))$. Thus, theoretically, Seidel's Algorithm can be run in $O(log^2(V))$, which is a significant speedup.

The Shoshan-Zwick Algorithm[7][8][9] is a matrix calculation based APSP algorithm that works on weighted, undirected graphs where existing edge weights are integers in the range of {1, 2, …, *N*}. Like Seidel, this algorithm also computes a logarithmic number of distance products in order to determine the shortest paths. The runtime of Shoshan-Zwick is *O(D(n)log(N*V))*, where *D(n)* is the cost to find the distance product of two *V*x*V* matrices. The distance product for Shoshan-Zwick[8], computed differently than Seidel's, can be computed in *O(N*M(V))*, where *M(V)* is matrix multiplication cost. This can be reduced to *O(N*log(V))*. The Shoshan-Zwick must also do more additional work in order to recover the shortest paths. The algorithm is based on not allowing the range of elements in the matrices is uses to increase, hence it uses additional matrix calculations such as chop() and clip()[7]. These calculations can theoretically be done all in parallel and thus do no add to the overall runtime of the algorithm. The algorithm then uses each distance matrix calculated to reconstruct the shortest path distances.

Although these matrix-multiplication based implementations seemed optimal in theory, logarithmic vs linear runtimes, we found that in practice, when implemented on CPU, they are not as practical. For one, matrix calculations are better suited for GPU. Although matrix multiplication can theoretically be performed in log(*V*) time, we do not have access to that many processors on CPU. Our implementation consisted of giving threads different parts of the matrix and having each thread run in parallel, so our runtimes are severely hindered by the number of processors available. A GPU-based implementation would perform better here.

Looking at our results in Figure 3, Seidel ran the fastest on large inputs, as expected. However, the speedup was not as significant as expected due to the problem of performing matrix

multiplication on CPU as discussed earlier. Seidel also suffers from efficiency because it is a recursive function. So for large matrices, Seidel uses the stack substantially. Alternatively, we found that Shoshan-Zwick's Algorithm performed the worse on large inputs. This is because the algorithm uses multiple matrix reductions that could not be fully parallelized on CPU using threads. Although operations such as chop() and clip() can be done in *O(1)* in theory, this runtime is not feasible on CPU. Thus performing these functions in practice adds significant overhead to the runtime. Shoshan-Zwick also suffers in terms of space because it stores *O(log(V))* versions of the distance matrix, taking up a lot of space for large graphs.

Conclusion

We looked at various solutions to SSSP and APSP. For SSSP, we looked at Dijkstra's algorithm and attempted to parallelize it by finding the minimum node through parallel reduction. For APSP, we looked at the Floyd-Warshall algorithm and parallel solutions such as parallelized Floyd-Warshall, Dijkstra's, Seidel's and Shoshan-Zwick.

In general, using more than 4 threads in any of the algorithms lead to a degradation in runtime since our machines are bounded by 4 cores. Given more cores, we expect to see faster runtimes with increasing number of threads. Looking at Figure 4, we see that the speedup for using parallelization is significant only for large inputs. Given a small input size, the benefits of parallelization are negligible and the costs of parallelization (thread creation and scheduling) is too high.

Figures

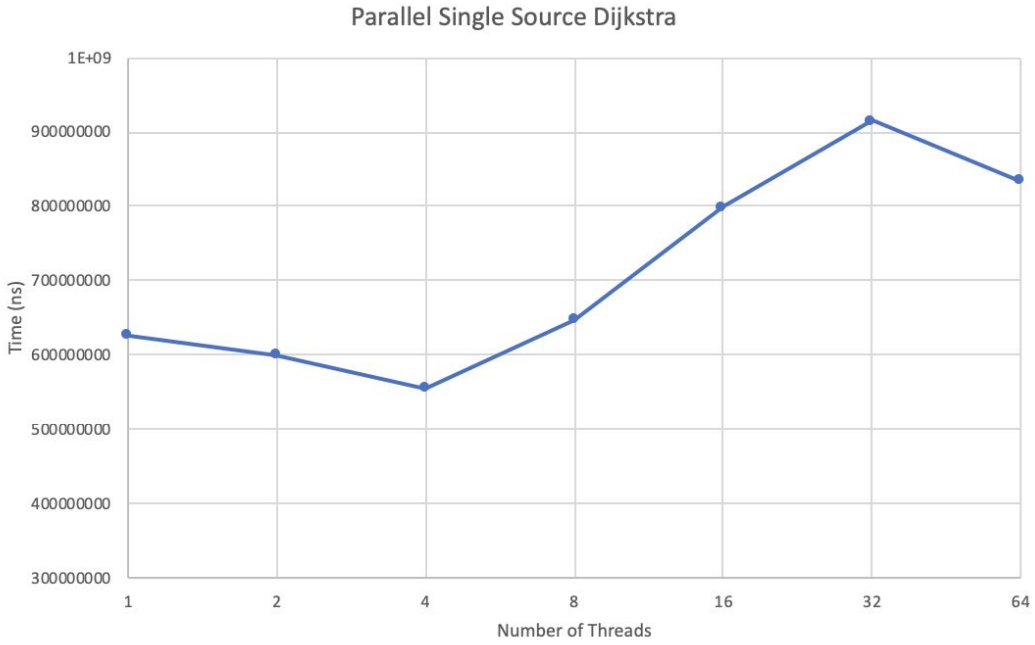


Figure 1. Graph of Dijkstra's Algorithm with varying number of threads

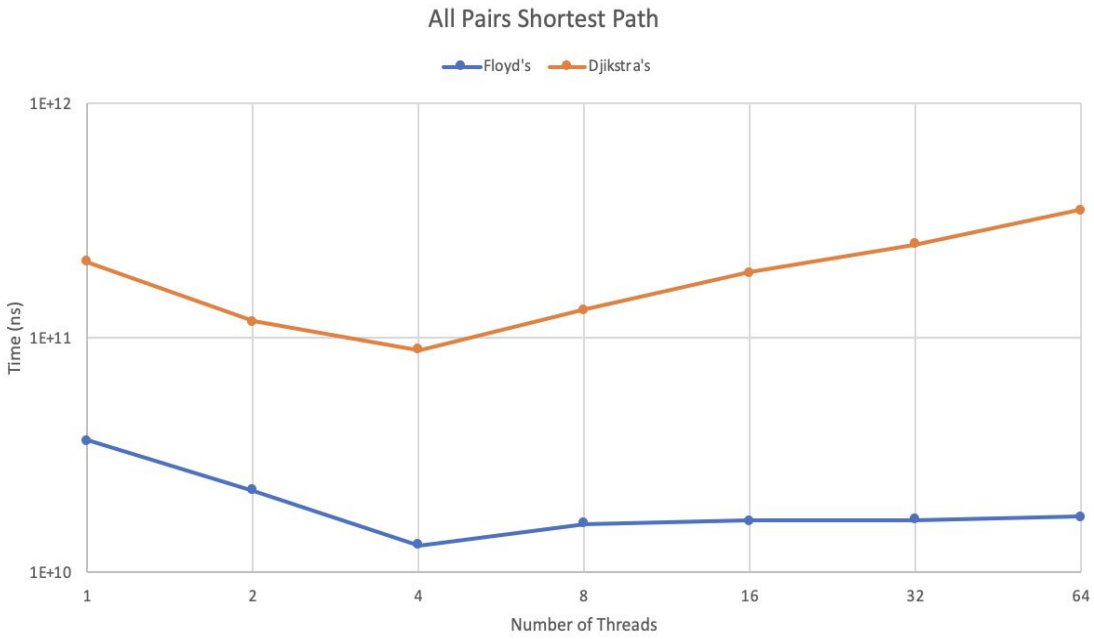


Figure 2. Graph of Parallel Floyd-Warshall and Parallel Dijkstra's

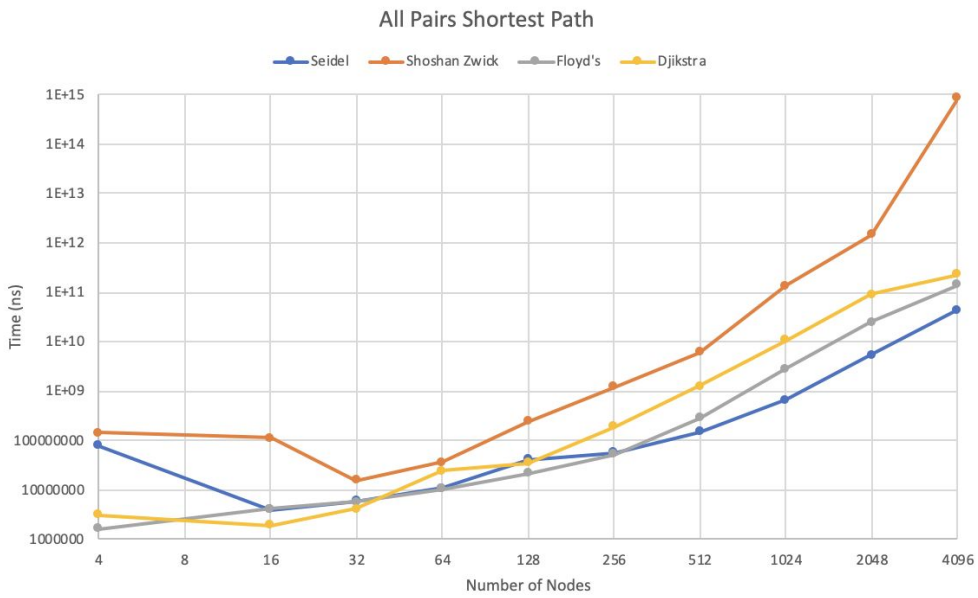


Figure 3. Runtimes for various APSP algorithms as the number of nodes increases

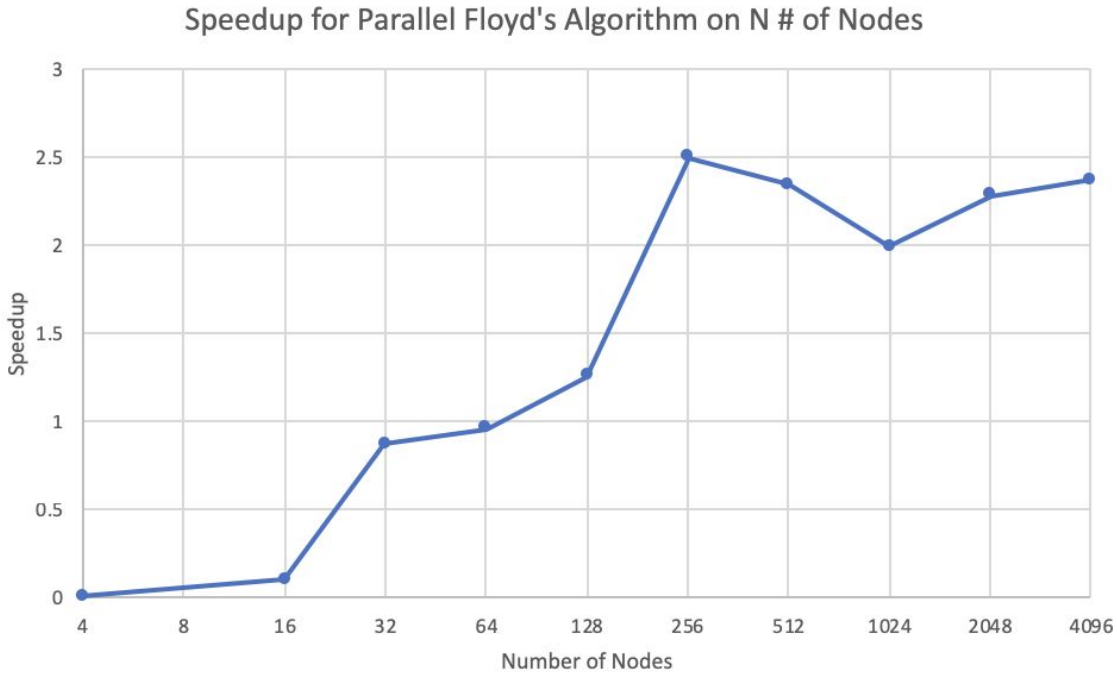


Figure 4. Graph of Speedup for varying number of nodes for parallel Floyd-Warshall using 4 threads

Source Code: https://github.com/aravindsrinivasan/parallel-shortest-path

References

[1] Djikstra's Algorithm, Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs"

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:             // Initialization
        dist[v] ← INFINITY                  // Unknown distance from source to v
        prev[v] ← UNDEFINED                 // Previous node in optimal path from source
        add v to Q                          // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                        // Distance from source to source

    while Q is not empty:
        u ← vertex in Q with min dist[u]    // Node with the least distance
                                            // will be selected first
        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

    return dist[], prev[]
```

[2] Floyd-Warshall Algorithm, Floyd, Robert W. (June 1962). "Algorithm 97: Shortest Path".

Communications of the ACM. 5 (6): 345.

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

[3] Seidel's Algorithm, Seidel, R. (1995). "On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs". *Journal of Computer and System Sciences*. **51** (3): 400–403.

Seidel(A)
Let $Z = A \cdot A$
Let B be an $n \times n$ 0-1 matrix,
where $b_{ij} = \begin{cases} 1 & \text{if } i \neq j \text{ and } (a_{ij} = 1 \text{ or } z_{ij} > 0) \\ 0 & \text{otherwise} \end{cases}$
IF $\forall i, j,\ i \neq j\ b_{ij} = 1$ then
Return D = 2B-A
Let T = Seidel(B)
Let $X = T \cdot A$
Return $n \times n$ matrix D,
where $d_{ij} = \begin{cases} 2t_{ij} & \text{if } x_{ij} \geq t_{ij} \cdot degree(j) \\ 2t_{ij} - 1 & \text{if } x_{ij} < t_{ij} \cdot degree(j) \end{cases}$

[4] Strassen Matrix Multiplication Algorithm, Strassen, Volker, *Gaussian Elimination is not Optimal*, Numer. Math. 13, p. 354-356, 1969

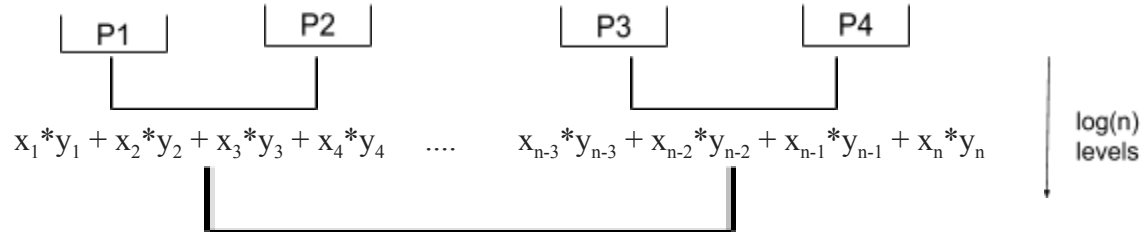
[5] Coppersmith-Winograd Matrix Multiplication Algorithm, Coppersmith, Don; Winograd, Shmuel (1990), "Matrix multiplication via arithmetic progressions" (PDF), *Journal of Symbolic Computation*, **9** (3): 251

[6] Parallel Matrix Multiplication

$$[x_1\ x_2\ x_3\ x_4 \ldots x_{n-3}\ x_{n-2}\ x_{n-1}\ x_n] * [y_1\ y_2\ y_3\ y_4\ \ldots y_{n-3}\ y_{n-2}\ y_{n-1}\ y_n]$$

$$= x_1*y_1 + x_2*y_2 + x_3*y_3 + x_4*y_4 + \ldots + x_{n-3}*y_{n-3} + x_{n-2}*y_{n-2} + x_{n-1}*y_{n-1} + x_n*y_n$$

$$= x_1*y_1 + x_2*y_2 \quad x_3*y_3 + x_4*y_4 \quad \ldots \quad x_{n-3}*y_{n-3} + x_{n-2}*y_{n-2} \quad x_{n-1}*y_{n-1} + x_n*y_n$$



$$x_1*y_1 + x_2*y_2 + x_3*y_3 + x_4*y_4 \quad .... \quad x_{n-3}*y_{n-3} + x_{n-2}*y_{n-2} + x_{n-1}*y_{n-1} + x_n*y_n$$

$$. \quad x_1*y_1 + x_2*y_2 + x_3*y_3 + x_4*y_4 + \ldots + x_{n-3}*y_{n-3} + x_{n-2}*y_{n-2} + x_{n-1}*y_{n-1} + x_n*y_n$$

[7] Shoshan-Zwick Algorithm, A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In FOCS, pages 605–614, 1999.

Function SHOSHAN-ZWICK-APSP($\mathbf{D}$)

$l = \lceil \log_2 n \rceil$.
$m = \log_2 M$.
for ($k = 1$ to $m + 1$) **do**
  $\mathbf{D} = clip(\mathbf{D} \star \mathbf{D}, 0, 2 \cdot M)$.
end for
$\mathbf{A}_0 = \mathbf{D} - M$.
for ($k = 1$ to l) **do**
  $\mathbf{A}_k = clip(\mathbf{A}_{k-1} \star \mathbf{A}_{k-1}, -M, M)$.
end for
$\mathbf{C}_l = -M$.
$\mathbf{P}_l = clip(\mathbf{D}, 0, M)$.
$\mathbf{Q}_l = +\infty$.
for ($k = l - 1$ down to 0) **do**
  $\mathbf{C}_k = [clip(\mathbf{P}_{k+1} \star \mathbf{A}_k, -M, M) \bigwedge \mathbf{C}_{k+1}] \bigvee [clip(\mathbf{Q}_{k+1} \star \mathbf{A}_k, -M, M) \bar{\bigwedge} \mathbf{C}_{k+1}]$.
  $\mathbf{P}_k = \mathbf{P}_{k+1} \bigvee \mathbf{Q}_{k+1}$.
  $\mathbf{Q}_k = chop(\mathbf{C}_k, 1 - M, M)$.
end for
for ($k = 1$ to l) **do**
  $\mathbf{B}_k = (\mathbf{C}_k \geq 0)$.
end for
$\mathbf{B}_0 = (0 \leq \mathbf{P}_0 < M)$.
$\mathbf{R} = \mathbf{P}_0 \mod M$.
$\Delta = M \cdot \sum_{k=0}^{l} 2^k \cdot \mathbf{B}_k + \mathbf{R}$.
return Δ.

Functions defined in the Shoshan-Zwick Algorithm

$$(clip(\mathbf{A}, a, b))_{ij} = \begin{cases} a & \text{if } a_{ij} < a \\ a_{ij} & \text{if } a \leq a_{ij} \leq b \\ +\infty & \text{if } a_{ij} > b \end{cases}$$

$$(chop(\mathbf{A}, a, b))_{ij} = \begin{cases} a_{ij} & \text{if } a \leq a_{ij} \leq b \\ +\infty & \text{otherwise} \end{cases}$$

$$\left(\mathbf{A} \bigwedge \mathbf{B}\right)_{ij} = \begin{cases} a_{ij} & \text{if } b_{ij} < 0 \\ +\infty & \text{otherwise} \end{cases}$$

$$\left(\mathbf{A} \bar{\bigwedge} \mathbf{B}\right)_{ij} = \begin{cases} a_{ij} & \text{if } b_{ij} \geq 0 \\ +\infty & \text{otherwise} \end{cases}$$

$$\left(\mathbf{A} \bigvee \mathbf{B}\right)_{ij} = \begin{cases} a_{ij} & \text{if } a_{ij} \neq +\infty \\ b_{ij} & \text{if } a_{ij} = +\infty, b_{ij} \neq +\infty \\ +\infty & \text{if } a_{ij} = b_{ij} = +\infty \end{cases}$$

$$(\mathbf{C} \geq 0)_{ij} = \begin{cases} 1 & \text{if } c_{ij} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$(0 \leq \mathbf{P} \leq M)_{ij} = \begin{cases} 1 & \text{if } 0 \leq p_{ij} \leq M \\ 0 & \text{otherwise} \end{cases}$$

[8] Distance Product for Shoshan-Zwick Algorithm

$$(\mathbf{A} \star \mathbf{B})_{ij} = \min_{k=1}^{n} \{a_{ik} + b_{kj}\}, 1 \leq i, j \leq n.$$

[9] Correction of the Shoshan-Zwick Algorithm, Pavlos Eirinakis, Matthew Williamson, and K. Subramani. On the Shoshan-Zwick algorithm for the all-pairs shortest path problem. J. Graph Algorithms Appl., 21(2):177–181, 2017.

21: $\hat{\mathbf{B}}_0 = (-M < \mathbf{P}_0 < 0)$.
22: $\hat{\mathbf{R}} = \mathbf{P}_0$.
23: $\Delta = M \cdot \sum_{k=1}^{l} 2^k \cdot \mathbf{B}_k + 2 \cdot M \cdot \hat{\mathbf{B}}_0 + \hat{\mathbf{R}}$.
24: **return** Δ.