Algorithms of All Pair Shortest Path Problem

Susmita
Dept. of Computer Science
MANIT, Bhopal
M.P, India

ABSTRACT

This paper is based on survey of various algorithms for all pair shortest path problem (APSP) on arbitrary real weighted directed graphs.This paper has summarized existing methods for solving shortest-path problems. In particular, we have addressed both sequential and parallel algorithms. We begin with a review of conventional sequential shortest-path algorithms and later, we have discussed blocked and vectorized implementation, thereby with the aim of reducing computational effort.

Keywords

APSP,Repeated Squaring Method,ADD Based Algorithm,Kleene's Algorithm,Blocked Implementation

1. INTRODUCTION

Today's social networks are getting larger and it needs to analyze datasets with millions of nodes and billions of edges.These networks require shortest path for transportation and communication.The All-Pairs Shortest Paths (APSP) problem seeks the shortest path distances between all pairs of vertices, and is one of the most fundamental graph problems.Given a weighted digraph G=(V,E) with weight function W :E -->R,(R is the set of real numbers)determine the length of the shortest path (i.e,distance) between all pair of vertices in G.We wish to find for every pair of vertices $u, v \in V$, a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. The all-pairs shortest path problem can be considered the fundamental of all routing problems.It has various applications areas such as Routing Protocol, Driving direction on Web mapping,transportation and traffic assignment problem, VLSI design ,wireless sensor network etc.

We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm $|V|$ times, once for each vertex as the source. If all edge weights are non negative, we can use Dijkstra's algorithm.With the use of linear-array implementation of the min-priority queue, the running time is $O(V^3 + V E) = O(V^3)$. The binary min-heap implementation of the min-priority queue yields a running time of $O(V E \log V)$, which is an improvement if the graph is sparse. Alternatively, we can implement the min-priority queue with a Fibonacci heap, yielding a running time of $O(V^2 \log V + V E)$.If negative weight edges are allowed, Dijkstra's algorithm can no longer be used. Instead, we must run the slower Bellman-Ford algorithm once from each vertex.The resulting running time is $O(V^2E)$, which on a dense graph is $O(V^4)$.We have proposed here various dynamic approaches for solving APSP problems.Rest of the paper is organized as follows:Section II gives different algorithms for finding all pair shortest paths and section III is devoted to conclusion and results of using these algorithms.

2. DIFFERENT ALGORITHMS FOR APSP PROBLEM REPEATED SQUARING METHOD [1]

It is a dynamic programming algorithm based on adjacency matrix representation. Input is an $n \times n$ matrix and W representing the edge weights of an n-vertex directed graph $G = (V, E)$. That is, $W = (w_{ij})$, where

$$W_{ij} = \begin{cases} 0 & \text{if } i=j, \\ \text{the weight of directed edge(i,j)} & \text{if } i\neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i\neq j \text{ and } (i,j) \in E. \end{cases}$$

Negative weight edges are allowed but we assume that the input graph contains no negative-weight cycles.The n*n output matrix D= (d_{ij}), where d_{ij} contains the weight of a shortest path from vertex i to vertex j. This algorithm works like repeated matrix multiplications.We start by developing a (V^4)time algorithm for the all-pairs shortest-paths problem and then improve its running time to $(V^3 \log V)$.To solve the all-pairs shortest-paths problem, we need to compute a ***predecessor matrix*** $\Pi = (\Pi_{ij})$, where Π_{ij} is NIL if either i = j or there is no path from i to j,and otherwise Π_{ij} is the predecessor of j on some shortest path from i. For each vertex $i \in V$, we define the ***predecessor subgraph*** of G for i as

$G\pi,i = (V_{\Pi,i}, E_{\Pi,i})$,where

$V_{\Pi,i} = \{j \in V : \Pi_{ij} \neq \text{NIL}\} \cup \{i\}$

and

$E_{\Pi,i} = \{(\Pi_{ij}, j) : j \in V_{\Pi,i} - \{i\}\}$.

As we know that all sub paths of a shortest path are shortest paths.Let $l^{(m)}_{ij}$ the minimum weight of any path from vertex i to vertex j that contains at most m edges.

$$l^{(0)}_{ij} = \begin{cases} 0 & \text{if } i=j, \\ \infty & \text{if } i\neq j. \end{cases}$$

Thus,we recursively define

$$l^{(m)}_{ij} = \min(l^{(m-1)}_{ij}, \min_{1\leq k\leq n} \{l^{(m-1)}_{ik} + w_{kj}\}) = \min_{1\leq k\leq n} \{ l^{(m-1)}_{ik} + w_{kj}\}.$$

If the graph contains no negative-weight cycles, then for every pair of vertices i and j for which $\delta(i, j) < \infty$, there is a shortest path from i to j that is simple and thus contains at most $n - 1$ edges. The actual shortest-path weights are therefore given by

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{n} = l_{ij}^{(n+1)} = \dots .$$

Conventionally we compute the shortest-path weights by extending shortest paths edge by edge.This is the following procedure, which, given matrices $l^{(m-1)}$ and W, returns the matrix $l^{(m)}$.

EXTEND-SHORTEST-PATHS(L,W)

```
1   n ← rows[L]
2   let L' = (l'_ij) be an n × n matrix
3   for i ← 1 to n
4      do for  j←1 to n
5            do l'_ij ←∞
6               for k← 1 to n
7                           do  l'_ij ← min(l'_ij, l_ik  +w_kj)
8   Return  L'.
```

We compute a series of matrices $L^{(1)}, L^{(2)},\dots, L^{(n-1)}$, where for $m = 1,2,\dots,n-1$,we have
$L^{(m)}= (l^{(m)}_{ij})$. The final matrix $L^{(n-1)}$ contains the actual shortest-path weights.

$$\begin{aligned} L^{(1)} &= L^{(0)}.W = W, \\ L^{(2)} &= L^{(1)}.W = W^2, \\ L^{(3)} &= L^{(2)}.W = W^3, \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)}.W = W^{n-1}. \end{aligned}$$

The matrix $L^{(n-1)} = W^{n-1}$ contains the shortest-path weights.This procedure computes the sequence in $\theta(n^4)$ time.But our goal is not to compute all the $L^{(m)}$ matrices. Our interest is only in matrix $L^{(n-1)}$.

Therefore, we can compute $L^{(n-1)}$ with only upper bound of $\log(n-1)$matrix products by computing the sequence.

$$\begin{aligned} L^{(1)} &= W, \\ L^{(2)} &= W^2 = W.W, \\ L^{(4)} &= W^4 = W^2.W^2, \\ L^{(8)} &= W^8 = W^4.W^4, \\ &\vdots \\ L^{(2\,\wedge\lg(n-1))} &= W^{(2\,\wedge\lg(n-1))} = W^{(2\,\wedge(\lg(n-1)-1))}.W^{(2\,\wedge(\lg(n-1)-1))}. \end{aligned}$$

Since, $2^{\lg(n-1)} \geq n-1$, the final product $L^{(2\,\wedge\lg(n-1))}$ is equal to $L^{(n-1)}$.

The running time of this repeated squaring technique is $\theta(n^3\log n)$. since each of the $\lg(n-1)$ matrix products takes $\theta(n^3)$ time.

The Floyd-Warshall algorithm[1]

This is also a graph analysis algorithm for finding shortest path in a weighted graphs with positive or negative edge weights but with no negative cycles and also for finding transitive closure of a relation R.We can say this algorithm as the quadratic version of Dijkstra's algorithm from each of the n vertices.Unlike matrix-multiplication-based algorithms,it considers the **intermediate** vertices of a shortest path.Consider a graph *G* with vertices *V* numbered 1 through *N*. Further consider a function shortestPath(i, j, k) that returns the shortest possible path from i to j using vertices only from the set {1,2,...,k} as intermediate points along the way. If w(i,j) is the weight of the edge between vertices i and j,we can define shortestPath(I,j,k+1) in terms of the following recursive formula:

Base Case: shortestPath(i,j,0) = w(i,j)

Recursive case:

shortestPath(i,j,k+1)=**min**(shortestPath(i,j,k),shortest Path(i,k+1,k)+ shortestPath(k+1,j,k))

The algorithm works by computing shortestPath(i, j, k) for all (i, j) pairs for k = 1 to n. Pseudocode for this is as follows:[2]

```
1  Let dist be a |V|*|V|array of minimum distances initialised to ∞ .
2  for each vertex v
3  dist[v][v] <- 0
4  for each edge (u,v)
5     dist[u][v] <- w(u,v)  // the weight of the edge (u,v)
6  for k from 1 to |V|
7     for i from 1 to |V|
8        for j from 1 to |V|
9           if dist[i][j] > dist[i][k] + dist[k][j]
10              dist[i][j] ← dist[i][k] + dist[k][j]
11          end if
```

The above algorithm only outputs the shortest distances. We can modify the solution to give the shortest paths also by storing the predecessor information in a separate 2D matrix.To find all n^2 of shortestPath(i,j,k) (for all i and j) from those of shortestPath(i,j,k−1) requires $2n^2$ operations.The total number of operations used is $n \cdot 2n^2 = 2n^3$.So this algorithm runs in $\theta(n^3)$ time and with $O(n^2)$ space.Floyd-Warshall algorithm can be easily modified to detect cycles.If we fill negative infinity value at the diagonal of the matrix and run the algorithm,then the matrix of predecessors will contain also all cycles in the graph(the diagonal will not contain only zeros,if there is a cycle in the graph).

Johnson's algorithm for sparse graph[1]

For sparse graphs, this algorithm is asymptotically better than either repeated squaring of matrices or the Floyd-Warshall algorithm. It also allows negative weight edges but not negative weight cycle.The algorithm outputs weights for all pairs of vertices or reports that the input graph contains a negative-weight cycle. Johnson's algorithm uses as subroutines both Dijkstra's algorithm and the Bellman-Ford algorithm.The idea of Johnson's algorithm uses the technique of ***reweighting*** all edges and make them all positive,then apply Dijkstra's algorithm for every vertex.Bellman Ford algorithm is used to transform the input graph for removing all negative weights.

For transformation of graph a new vertex is added to the graph and connected to all existing vertices. The shortest distance values from new vertex to all existing vertices are h[] values.Following is the complete algorithm:

JOHNSON*(G)*

```
1   compute G', where V[G'] = V[G] ∪ {s}, E[G'] =
    E[G] ∪ {(s, v) : v ∈ V[G]}, and w(s, v) = 0 for all v ∈
    V[G]
2   if BELLMAN-FORD(G',w, s) = FALSE
3    then print "the input graph contains a negative-
    weight cycle
4    else for each vertex v ∈ V[G']
5              do set h(v) to the value of δ(s, v) computed
    by the Bellman-Ford algorithm
6         for each edge (u, v) ∈ E[G']
7              do w'(u, v) ← w(u, v) + h(u) − h(v)
8   for each vertex u ∈ V[G]
9              do run DIJKSTRA(G, w', u) to compute
    δ'(u, v) for all v ∈ V[G]
10                     for each vertex v ∈ V[G]
11                          do duv ←δ'(u, v) + h(v) − h(u)
12             return D
```

The main steps in the algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is O(VE) and time complexity of Dijkstra is O(VlogV). So overall time complexity is $O(V^2\log V + VE)$ by a Fibonacci heap. The time complexity of Johnson's algorithm becomes same as Floyd-Warshall when the graph is complete($E=O(V^2)$).Without assumption made by Johnson's ,it is difficult to break the $O(n^3)$boundary. The binary min-heap implementation yields a running time of *O(V E* lg *V)*, which is still asymptotically faster than the Floyd-Warshall algorithm if the graph is sparse.

ADD Based Algorithm[3]

Algebric decision diagram,an symbolic algorithm is a new kind of BDD(Binary Decision Diagram) with a set of constant values different than the set {0,1}.ADD based procedures for weight calculation are very effective for very large graph(over 10^{27} vertices and 10^{36} edges).This symbolic computation technique is based on the ***triangulation rule***.

Formal Definition:

An ADD is a directed acyclic graph (V∪Φ∪T,E) representing a set of functions $f_i : \{0,1\}^n \rightarrow S$, where S is the finite carrier of the algebraic structure over which the ADD is defined. **V** is the set of the internal nodes. The out-degree of v ∈ V is 2**.**Both arc are labeled else and then, respectively. Every node of a set of internal nodes has a label *l*(v) ∈ {0,1,….n-1}.The label identifies a variable on which the ***fi*** depends. Φ is the set of function nodes with out-degree 1 and in-degree 0**.** The function nodes are in one-to-one correspondence with the *fi's*. **T** is the set of terminal nodes and it is labeled with an element of S,s(t).Its out-degree is 0. E **is** the set of edges connecting the nodes of the graph;(v_i,v_j) is the edge connecting node v_i to v_j. The variables of the ADD are ordered; if v_j is a descendant of v_i(i.e.,$(v_i,v_j) \in E$), then $l(v_i) < l(v_j)$.

An ADD represents a set of Boolean function for each node, which defined as follows:[4]

1. The function of a terminal node, *t*, is the constant function *s*(*t*). The constant *s(t)* is interpreted as an element of a Boolean algebra larger than or equal in size to *S*.

2. The function of an internal node v ∈ V is based on recursive Shannon expansion and is given by $l(v). f_{then} + l(v)'. f_{else}$, where '.' and '+' denote Boolean conjunction and disjunction, and f_{then} and f_{else} are the functions of the *then* and *else* children.

3. The function of *Φ* ∈Φ is the function of its only child.

ADD algorithm determines all the paths p_i for a given shortest path weight δ(source,sink) such that $w(p_i)$ = δ(source,sink) .

The Algorithm

Using the triangulation rule,Bahar, Frohm,Gaona,Hachtel, Macii,Pardo and Somenzi wrote the symbolic backtracing algorithm in ADD form. The procedure takes as parameters the ADD representing the adjacency matrix of the graph A_G, the ADD of the matrix of the shortest path weights, S and the ADD's of the source and sink nodes. At each iteration, given the characteristic function of the sink nodes, the predecessors of the current sink are computed and added to the matrix of the shortest path weights S to obtain **K.** The ADD ***K*** is the characteristic function of vertices *x* which are connected to the vertices given by sink. The procedure returns an acyclic sub-graph of the original graph, which contains all the paths from source to sink of smallest weight. The valid predecessors represented by ***k"*** become the new sinks for the next iteration.

```
procedure ShortestPath-BackTsace (A_G, S, source, sink) {

  path = addConst(+∞);

  visited = addConst(0);

  while (sink != addConst(0)) {

     E = A_G + sink;

     K = S + E ;

     Kmatch = addmatch (K, S);

     k = ∃y Kmatch ;

     k' = addMask(k,sink);

    path = UpdatePath (path, k');

    visited = visited + k';

    k" = addMask(k',source);
    sink = k";
    sink = sink - visited;

  }

return path;
}
```

This algorithm finds the maximum possible number of these negative cycles.It prevents the back-tracing of negative **(or** zero) weight cycles, thus implicitly recording all the occurrences of these cycles.

A Multi Source Label Correcting Algorithm[5]

For the APSP problem on sparse graphs, Hiroki Yanagisawa proposed a fast algorithm that is between the two extremes. It first partitions(using any of the graph partitioning types like BFS,DFS,*K*NN) the vertices into sets of vertices $V_1,V_2,\ldots V_p$ with each set having at most *B* vertices such that the vertices in

each set are close to each other, where B is a parameter, and then solves the multi-source shortest paths (MSSP) problem for each set in parallel.This algorithm is an extension of **labeling method**[6]. This algorithm computes the shortest paths from B source vertices simultaneously and our algorithm uses only $O(m + Bn)$ working space where m is the number of edges and n is the number of nodes.It's implementation with SIMD instructions achieves a speed up of 2.3–3.7x compared to a scalar version.

Here the priority queue is a data structure for maintaining a set of vertices, each with an associated value called a key (or priority). Given a source vertex s, the labeling method maintains three labels for each vertex $v \in V$: a distance label $d(s,v)$, a vertex state $s(v) \in$ {unreached; scanned}, and a vertex key $k(v)$ that is used as the key in the priority queue Q. The main difference between this algorithm and labeling method is that MSLC algorithm maintains a vertex potential $d(s, v)$ for each $s \in S$ and thus each vertex v is associated with |S| distance labels, while each vertex v is associated with a single distance label in labeling method.

The algorithm

Input: A graph $G = (V;E)$ and a set of source vertices S

1: $d(s,v) = \infty$ and $s(v)$ = unreached for every vertex $v \in V$ and $s \in S$

2: $d(s,s) = 0$, $s(s)$ = labeled, $k(s) = 0$, add s to priority queue Q for every $s \in S$

3: **while** Q is not empty **do**

4: Remove a vertex v with the minimum key $k(v)$ from Q

5: **for** each edge $(v,w) \in E$ outgoing from v **do**

6: *updated* = false
7: **for** all $s \in S$ **do**
8: **if** $d(s,w) > d(s,v) + l(v,w)$ **then**
9: $d(s,w) = d(s,v) + l(v,w)$, $s(w)$ = scanned, and *updated* = true
10: **end if**
11: **end for**
12: **if** *updated* = true **then**
13: Compute $k(w)$ // e.g. set $k(w) = \min \{ d(s,w) \mid s \in S \}$

14: Add w to Q **if** Q does not contain w
15: **end if**
16: **end for**
17: **end while**
18: Output $d(s,v)$ for all $s \in S$ and $v \in V$

SIMD Implementation

Hiroki Yanagisawa used 4-way SIMD instruction set by using 128-bit vector registers where each register contains four 32-bit values.Assume that a vertex $v \in V$ is assigned an integer id from the interval [0, n- 1] and that a source vertex $s \in S$ is assigned an integer id from the interval [0,B- 1].The distance labels are stored as $d(s,v)$ for $s \in S$ and $v \in V$ in the form of an array d of length Bn, where each d[v*B+s] stores $d(s; v)$. The array d is equivalent to an array of vector v_d of length B_n=4, where each vector element vd[i] consists of the four values of d[i*4] to d[i*4+3].Here it is assumed that the first element of the array v_d is aligned on a 128-bit boundary.

They used the loop-unrolling technique, which achieves a slight speedup and single-precision (32-bit) floating point numbers to store edge lengths and path lengths.Then they used the binary heap for the priority queue implementation and a parameter value $B = 128$ and the BFS strategy for the graph partitioning. Note that, the scan ratio is at most B for a strongly connected graph, since the n-Dijkstra algorithm performs exactly mn scans and our algorithm performs at least mn/B scans.The graph partitioning is implemented as a single-thread since the execution time required for the graph partitioning is negligible. They used multi-thread to solve the MSSP problems in parallel, because each MSSP computation is independent.

Drawback of this algorithm is that if this algorithm runs in a massively parallel and distributed environment, the sequential graph partitioning algorithm may become a bottleneck for this algorithm.Since it uses simple heuristics for the key on the priority queue and the graph partitioning, improving heuristics would be a research focus.

In-place Parallel Recursive approach using kleene's algorithm[7]

Kleene's algorithm is used for finding transitive closure that computes the path existence between every possible pair of vertices(i, j). Kleene's algorithm divides the nodes of the graph into $n/\sqrt{s}$ zones. Adjacency matrix corresponding to the graph is divided into n^2/s sub matrices each having size $\sqrt{s} * \sqrt{s}$. Each entry $e_{ij} \in M_{ij}$ refers to the shortest path from every possible vertex from zone 'i' to zone 'j' going through no more zone greater than zone 'k' and is computed using $e_{ij} += \Sigma_{k=1}^{n} e_{ik} * e_{kj}$. It uses data locality to improve cache performance.

ALGORITHM : KLEENE'S_TANSITIVE_CLOSURE(A, N)

1 /* Divide the graph 'A' into $n/\sqrt{s}$ zones */
2 for k = 1 to $n/\sqrt{s}$ do
3 /*compute $M^*_{k,k}$, the transitive closure of $M_{k,k}$*/
4 $M_{k,k}= M^*_{k,k}$
5 for i = 1 to $n/\sqrt{s}$ do
6 for j = 1 to $n/\sqrt{s}$ do
7 $M_{i,j} += M_{i,k} * M_{k,k} * M_{k,j}$
8 end for
9 end for
10 end for

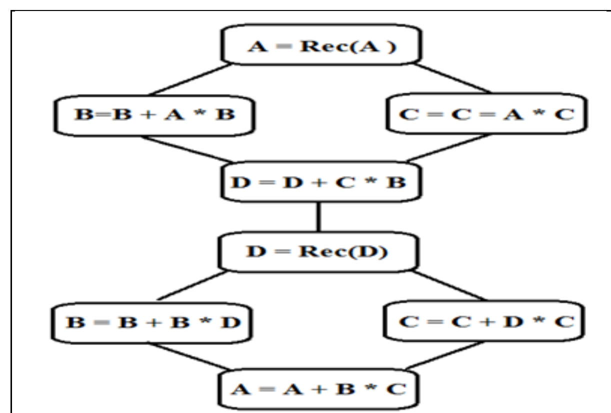

Fig. 1 : Precedence Graph for Kleene's Operation

Since,OpenCL does not support recursion so implementation of recursive function is done in host program which calls OpenCL kernel recursivelyThis Kleene's based parallel recursive algorithm shows a significant speedup over OpenCL parallel Floyd Warshall's algorithm over same GPU.

A Blocked Implementation of All Pairs Shortest –Paths Algorithm[8]

It is a blocked organization of Floyd Warshall's all pairs shortest paths algorithm to make better utilization of cache.Several models for computer with different organization of memory has been developed although L2 cache is architecture dependent.La Marca and Ladner develop a model for single level direct mapped cache.They used this model to analyze the performance of binary heaps and cache aligned d-heaps and optimized the cache performance for several sorting methods.Authors obtained a lower bound for the L1 and L2 cache miss rate by determining the minimum number of cache misses and making the reasonable assumption that cache optimization will not decrease the total memory references.

i.e, execution time >= (CPI* IC +IC* memory reference per instruction * L1 miss rate * L1 miss penalty) * clock cycle time

They declared all variables other than n^2 elements of integer array as register variables.Since line size of cache decides the unit of memory transfer so number of L1 and L2 cache miss rate depend on line size of cache.

Blocked version of Floyd's algorithm

Partition the cost adjacency matrix into sub matrices of size B*B,where B is a blocking factor.That means it will perform B iterations of the outer most loop of Floyd's algorithm.Each set of B iteration are divided into three phases.**In phase 1** of the first set of B iterations,top left block (1,1) is a self dependent block in the first B iterations.**In phase 2** of the first B iterations,a modified equation is used to compute the shortest path. 1<= k<= B for the remaining blocks (1,*) and (*,1) that are on the same row or column as the self dependent block.For the remaining (1,*) blocks

$D^k(i,j) = \min \{D^{k-1}(i,j), D^{k-1}(i,k) + D^B(k,j)\}, k >= 1$

Where $D^0(i,j) = A^0(i,j)$.

In phase 3,D^k is computed for the remaining blocks where 1<=k<=B.(i.e for the blocks that are not on the same row or column as the self dependent block) as

$D^k(i,j) = \min \{D^{k-1}(i,j), D^B(i,k) + D^B(k,j)\}, k >= 1$

.And this phase is followed by the next round of B iterations.When computing the D values in a block during any round of function,atmost three blocks are active.During the self dependent block computation only 1 block is active.

L1 cache misses are minimized by choosing the largest block size B.And second requirement is necessary as the smallest unit of data transferred to L1 cache should be contiguous bytes of memory.Blocked version obtains speedups close to the maximum possible for a cache optimized version of Floyd's algorithm.Experiments indicate that the blocked algorithm delivers a speedup (relative to the unblocked Floyd's algorithm) between 1.6 and 1.9 on a Sun Ultra Enterprise 4000/5000 for graphs that have between 480 and 3200 vertices.The measured speedup on an SGI O2 for graph with between 240 and 1200 vertices is between 1.6 and 2.

Optimizing All Pair shortest Path Algorithm Using Vector Instructions[9]

Sungchul Han and Sukchan Kang presented a vectorized version of Floyd-Warshall's algorithm to improve the performance.The vectorized implementation utilizes the SIMD instruction available in state-of-the-art architectures.Various other papers concentrated on the exploitation of data locality to improve the cache performance but they didn't work on the parallel execution of multiple instructions.

They analyzed the blocked version of the FW algorithms that include the straight-forward iterative implementation(FWI), the recursive version (FWR), and the tiled version (FWT).

The conventional Floyd Warshall's algorithm is an in-place algorithm that overwrites the result of each iteration to the input matrix i.e, If the reconstruction of the actual shortest path is desired, an additional output matrix V is also generated.This V matrix is the **via matrix**.

Blocked Floyd Warshall's algorithm is a generalized iterative approach without using **via matrix.**Iterative method is very similar to matrix matrix multiplication.It can be performed in a blocked manner with P* P matrices are invoked $(N=P)^2$ times, where P is the subblock size after blocking. Therefore, it is possible to perform iterative method recursively.A tiled version of FW, which is simply a recursion by only one level. When the via matrix is not included, the operations counts for all variants of the blocked versions are the same as that of the original FW, which is $2N^3$ integer additions, counting a comparison and a minimum operation as two operations.via matrix involves atleast one comparison. Furthermore, They used three logical operators (i.e., four integer operations in total) for the via matrix in efforts to reduce the branch instructions. For fair comparison between conventional algorithms and the vectorized algorithms to follow,they assumed an operation count of $6N^3$ integer operations for any FW algorithm with the via matrix. They modified blocked recursive algorithm to use vector instructions,specifically Intel single instruction multiple data extensions2 (SSE2), which provides eight parallel arithmetic or logical operations on 16-bit integer data.

The optimum parameters for optimizing the FW blocked algorithm are as follows:

- For recursive FW, blocking factor of 2 and base size of 256.
- For tiled version of FW, tile size of 256.

The data type of the distance matrix and the via matrix is defined as 16-bit integers. This makes it possible to vectorize eight integer additions with the SSE2 128-bit registers.

Performance of the Blocked Algorithms With the via matrix or without via matrix, the recursion-all-the-way strategy yields the poorest performance due to the excessive recursion overhead. With 32-bit integers,the blocked algorithms are about 20% better than iterative approach.

Effect of Unrolling The performance of any unrolled FW algorithm was only about 60% of their non-unrolled counterparts.

For higher cache performance, they divided the input matrix into small tiles of appropriate size and performed each tile with vectorized FW routines. Then, unrolling was applied again to reduce the loop overhead. The Intel SSE2 instruction set allows the packing of eight 16-bit integers into one 128-bit register. They designed three unrolled versions with the unrolling factor of 2, 4, and 8.

It is observed that between 95% and 130% of speed-up against tiled based FW has been obtained with the non-unrolled version and between 133% and 170% of increase with the unrolled version. Higher unrolling factor improves the

performance except that the horizontally unrolled version was only as good as the one unrolled by a factor of 2. Without the via matrix, for the unrolled version ,the increase was between 231% and 359%.with the via matrix, the best performance is observed from the most unrolled version which gave an increase between 369% and 417%.

By the unrolled versions,improvement can be achieved by the parallel execution by vectorization and the elimination of branch instructions. Vectorized FW implementation improved the performance by a factor of between 2.3 and 5.2 over the conventional blocked algorithms. Unrolling works effectively for vectorized versions.Vector instruction based algorithm improve the performance over the convensional blocked algorithms.

3. CONCLUSION

Repeated squaring method is used in ADD based data structure to store graph.The Floyd-Warshall algorithm is a simple and widely used algorithm to compute shortest paths between all pairs of vertices in an edge weighted directed graph. It can also be used to detect the presence of negative cycles.Johnson algorithm is better for sparse graph but without the assumption made in this algorithm, it is not possible to break the boundary of $O(n^3)$.ADD reduces the space required to store graph by eliminating the redundant node.MSLC algorithm can run on small working space. Implementation of MSLC algorithm with SIMD instructions achieves an order of magnitude speedup for real-world geometric graphs compared to an implementation based on Dijkstra's algorithm.Kleene's based parallel recursive algorithm gains significant speedup over OpenCL parallel Floyd Warshall's algorithm over same GPU.The blocked algorithm delivers a speedup (relative to the unblocked Floyd's algorithm) between 1.6 and 2 on a graph of large size.

4. REFERENCES:

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Second Edition. The MIT Press, Sep. 2001.

[2] http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

[3] R.Iris Bahar,Gary D.Hachtel,Enrico Macii,Abelardo Pardo,Massimo Poncino,Fabio Somenzi, "*An ADD Based Algorithm for Shortest Path Back-Tracing of Large Graphs*",1066-1395/94, 1994,pages-248-251,IEEE

[4] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel,E. Macii, A. Pardo, F. Somenzi, "*Algebraic Decision Diagrams and their Applications*", ICCAD-93: ACM/IEEE 1993 International Conference on Computer Aided Design, Santa Clara, CA, November 1993.

[5] Hiroki Yanagisawa*," A Multi-Source Label-Correcting Algorithm for the All-Pairs Shortest Paths Problem*", RT0882,sept 2009.

[6] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "*Shortest paths algorithms: theory and experimental evaluation*", Mathematical Programming, Vol. 73, Issue 2, pp. 129–174, 1996.

[7] Paolo D'Alberto, A. Nicolau, "R-Kleene: a high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks", Algorithmica 47 (2) (2007) pp. 203-213.

[8] Gayathri Venkataraman,Sartaj Sahni,and srabani Mukhopadhyaya,"*A Blocked All Pairs Shortest-Paths Algorithm*"

[9] Sungchul Han and Sukchan Kang,"*Optimizing All-Pairs Shortest-Path Algorithm Using Vector Instructions*"