A Multi-Source Label-Correcting Algorithm for the All-Pairs Shortest Paths Problem

Hiroki Yanagisawa

IBM Research - Tokyo
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Limited Distribution Notice

This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalties).
A Multi-Source Label-Correcting Algorithm for the All-Pairs Shortest Paths Problem

Hiroki Yanagisawa
IBM Research - Tokyo
yanagis@jp.ibm.com

Abstract

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. In this paper, a fast algorithm with a small working space for the APSP problem on sparse graphs is presented, which first divides the vertices into sets of vertices with each set having a constant number of vertices and then solves the multi-source shortest paths (MSSP) problem for each set in parallel. For solving the MSSP problems, we developed a multi-source label-correcting algorithm, as an extension of a label-correcting algorithm for the single-source shortest path problem. Our algorithm uses fewer operations on the priority queue than an implementation based on Dijkstra’s algorithm. Our experiments showed that an implementation of our algorithm with SIMD instructions achieves an order of magnitude speedup for real-world geometric graphs compared to an implementation based on Dijkstra’s algorithm.

1 Introduction

The All-Pairs Shortest Paths (APSP) problem [3] is, given a weighted directed graph \( G = (V, E, \ell) \), to determine the distance between every pair of vertices, where \(|V| = n\), \(|E| = m\), and \(\ell : E \rightarrow \mathbb{R}\) is the edge length (weight) function. The graph \( G \) may have negative length edges but we assume that \( G \) does not contain negative length cycles, because otherwise the APSP problem is not well-defined. The APSP problem is one of the most fundamental graph problems and has various applications such as routing problems on road networks.

When we solve the APSP problems on real-world graphs, not only the computation time of an algorithm but also the size of its working space is an important issue. Most real-world graphs such as road networks are large and sparse. Even for a moderate size graph with \( n = 100,000 \) vertices, we would need approximately 40 GB for storing the \( n^2 \) values in main-memory if each element is represented by 32 bits (4 bytes). This is too large even for current computer environments. However, many efficient algorithms for the APSP problem use as much as \( O(n^2) \) space, even though the input graph requires only \( O(m + n) \) space with an adjacency-list representation [3]. In this paper, we consider algorithms for the APSP problem on sparse graphs with a small \( O(m + n) \) working space.

One of the classical approaches for solving the APSP problem is to divide this problem into \( n \) single-source shortest path (SSSP) problems [3]. Such an approach runs an SSSP algorithm in parallel, once for each vertex as the source. If the input graph does not contain negative edge lengths, we can use Dijkstra’s algorithm [5] for solving the SSSP problems and can solve the APSP problem in \( O(mn \log n) \) time with a binary heap [22] and in \( O(mn + n^2 \log n) \) time with a Fibonacci heap [8]. To handle negative lengths, we can use Johnson’s algorithm [12] to reassign the edge lengths of the input graph so that the resulting edge lengths are nonnegative, requiring additional \( O(mn) \) time. Assuming that output can be written directly to secondary storage upon completion without using the main memory, these algorithms use only \( O(m + n) \) working space. While many algorithms have been proposed for computing SSSP, Dijkstra’s algorithm still appears to be a good practical choice in many real world settings [2]. We call such approach \( n \)-Dijkstra. A drawback of the \( n \)-Dijkstra algorithm is that, when computing an SSSP from a source vertex, it does not make use of any information on the shortest paths from other source vertices.

Many efficient algorithms use the information on the shortest paths from multiple source vertices. The Floyd-Warshall algorithm [7, 21] use it in a dynamic programming formulation. Demetrescu and Italiano [4] proposed an algorithm that makes use of the property “every subpath of a shortest path is a shortest path” [3] to reduce the number of operations on a priority queue, which is a bottleneck of many shortest path algorithms including Dijkstra’s algorithm. However these algorithms require as much as \( O(n^2) \) working space.

In summary, there are two extreme approaches: one uses a small \( (O(m + n)) \) working space but does not make use of the shortest paths from other source vertices and the other makes use of the shortest paths from multiple source vertices but uses a large \( (O(n^2)) \) working space.
Our contributions. For the APSP problem on sparse graphs, we give a fast algorithm that is between the two extremes. It first partitions the vertices into sets of vertices with each set having at most $B$ vertices, where $B$ is a parameter, and then solves the multi-source shortest paths (MSSP) problem for each set in parallel. To solve the MSSP problem, we developed a multi-source label-correcting (MSLC) algorithm as an extension of the labeling method [2] for the SSSP problem. Since our MSLC algorithm computes the shortest paths from $B$ source vertices simultaneously, it uses much more information than the $n$-Dijkstra approach and therefore the number of operations on the priority queue is reduced. In addition, our algorithm uses only $O(m + Bn)$ working space.

Moreover, our MSLC algorithm has data-level parallelism. This means we can accelerate it by using Single Instruction Multiple Data (SIMD) instructions (e.g. the SSE instruction set [11]). Our implementation with SIMD instructions achieves a speed up of 2.3–3.7x compared to a scalar version.

Notation. In this paper, the value $\ell(v, w)$ represents the length of an edge $(v, w) \in E$ and the value $d^*(s, v)$ represents the shortest path length from a vertex $s$ to a vertex $v$.

2 Related Work

One of the classical approaches for the APSP problem is the Floyd-Warshall algorithm [7, 21], which is based on dynamic programming and runs in $O(n^3)$ time and $O(n^2)$ space. Many implementations have been described. For example, Penner and Prasanna [18] gave a cache-efficient implementation, Han et al. [9] proposed an automated tuning approach to accelerate on CPU, Bondhugula et al. [1] gave an FPGA-based implementation, and Katz and Kider [14] gave a GPU based implementation. In addition, many algorithms based on matrix multiplication have been suggested (see survey [20]). Duin [6] proposed another algorithm that avoids redundant updates of the distance labels by using the dynamic programming principle of the optimality of subpaths. Although these algorithms are efficient on dense graphs, there are many algorithms that are faster on sparse graphs. For example, Demetrescu and Italiano proposed an algorithm [4] that is a simple variant of Dijkstra’s algorithm, which reduces the number of operations on the priority queue. Karger et al. [13] and independently McGeoch [16] gave an algorithm that run in $O(mn^2 + n^3 \log n)$ time, where $m^*$ is the number of edges in $E$ that participate in some shortest paths. These algorithms use as much as $O(n^2)$ space and seem impractical for large sparse graphs.

There are some algorithms with a small $O(m + n)$ working space for sparse graphs. The $n$-Dijkstra algorithm [3] is a classical approach that runs in $O(mn + n^2 \log n)$ time with a Fibonacci heap. Another SSSP-based algorithm was proposed by Harish and Narayanan [10], which runs efficiently on GPUs. Recently, Okuyama et al. [17] improved this GPU implementation, which leads to an efficient use of on-chip shared memory and hides the latencies by generating many threads. This algorithm is similar to our algorithm in that it divides the vertices into sets of vertices, but this is only for generating the many threads. These SSSP-based algorithms do not make use of the information on the shortest paths from other source vertices.

The many-to-many shortest path problem is a generalization of the APSP problem, which seeks all of the shortest path distances between a set of source vertices $S$ and a set of target vertices $T$. (Note that this problem is equivalent to the APSP problem if $S = T = V$.) The applications of this problem include the preprocessing step of the routing problems on road-networks such as the vehicle routing problem. When $|S|$ and $|T|$ are large, the algorithms for the APSP problem can be applied directly. Knopp et al. [15] proposed an algorithm based on the highway hierarchies technique [19] for this problem, which exploits the hierarchical structure of a given graph. Since this algorithm computes APSP in a sub-routine, our algorithm could be used to accelerate this algorithm.

3 Algorithm

The pseudocode in Algorithm 3.1 shows the framework of our algorithm. Given an input graph $G$ and a parameter $B$, our algorithm first divides the set of vertices into sets of vertices $V_1, \ldots, V_p$, where the size of each set $V_i$ is at most $B$, and then solves the MSSP problem for each $V_i$ by using our multi-source label-correcting (MSLC) algorithm. Since the MSSP computations are independent of each other, our MSLC algorithm can run in parallel.

Algorithm 3.1 Our algorithm.

**Input:** A graph $G = (V, E, \ell)$ and a parameter $B$
1: Partition the set of vertices $V$ into $V_1, \ldots, V_p$ such that $|V_i| \leq B$ for each $V_i$
2: for each $V_i$ do
3: Run our MSLC algorithm for the set of source vertices $V_i$
4: end for

We discuss our MSLC algorithm in Sec. 3.1 and 3.2 and the graph partitioning in Sec. 3.3.
3.1 Multi-Source Label-Correcting Algorithm for MSSP

Many algorithms for the SSSP problem are classified as labeling methods [see 2] and Dijkstra’s algorithm is a typical representative. Before showing our multi-source label-correcting algorithm, we first review the labeling method for the SSSP problem. Then we extend this method for the MSSP problem.

The labeling method for SSSP. Algorithm 3.2 shows pseudo-code for the labeling method with a priority queue [3]. Here the priority queue is a data structure for maintaining a set of vertices, each with an associated value called a key (or priority). It supports the INSERT (adds a vertex), DECREASEKEY (decreases the value of vertex v’s key), and EXTRACTMIN (removes and returns a vertex with the minimum key) operations.

Given a source vertex s, the labeling method maintains three labels for each vertex v ∈ V: a distance label d(s, v), a vertex state s(v) ∈ {unreached, scanned}, and a vertex key k(v) that is used as the key in the priority queue Q. The distance label d(s, v) stores the shortest path length from s to v among the paths found so far, and hence it is an upper bound of the shortest path length d∗(s, v). The method starts by initializing d(s, v) = ∞ and s(v) = unreached for every vertex v. Then it sets d(s, s) = 0, s(s) = scanned, and k(v) = 0, and then adds s to Q. This Q holds some vertices with the scanned state and initially Q contains only s. At each step in the computation, the method removes a vertex v with the minimum k(v) from Q if any, then scans (or relaxes) each outgoing edge (v, w) ∈ E. The scan operation (line 6–10 in Algorithm 3.2) determines whether or not there is a better path from s to w via v, i.e. if d(s, w) > d(s, v) + ℓ(v, w). When a better path is found, the distance label d(s, w) is updated with the value of d(s, v) + ℓ(v, w), the vertex state s(v) is set to scanned, the key k(w) is updated, and the vertex w is added to Q if Q does not contain w. When Q becomes empty, the method outputs d(s, v) for all v ∈ V and terminates.

Algorithm 3.2 Labeling method.

Input: A graph G = (V, E) and a source vertex s
1: d(s, v) = ∞ and s(v) = unreached for every vertex v ∈ V
2: d(s, s) = 0, s(s) = scanned, k(s) = 0, add s to priority queue Q
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) ∈ E outgoing from v do
6: if d(s, w) > d(s, v) + ℓ(v, w) then
7: d(s, w) = d(s, v) + ℓ(v, w) and s(w) = scanned
8: Compute k(w) // e.g. Dijkstra’s algorithm sets k(w) = d(s, w).
9: Insert w to Q if w is not contained in Q
10: end if
11: end for
12: end while
13: Output d(s, v) for all v ∈ V

We can use any value as the key k(v) for each vertex v and this value determines which vertex is extracted from the priority queue in line 4 of Algorithm 3.2. It seems better to extract a vertex v that is close to the source vertex s and therefore Dijkstra’s algorithm uses the distance label d(s, v) as the key, i.e. it sets k(v) = d(s, v).

Our Algorithm for MSSP. Here we extend the labeling method for the MSSP problem and obtain a multi-source label-correcting (MSLC) algorithm. Algorithm 3.3 shows pseudocode for this algorithm. The main difference is that our algorithm maintains a vertex potential d(s, v) for each s ∈ S and thus each vertex v is associated with |S| distance labels, while each vertex v is associated with a single distance label in Algorithm 3.2. The scan step is extended for updating multiple distance labels d(s, v) for every s ∈ S. When at least one of the |S| distance labels for a vertex w is updated, the key k(w) is updated and w is added to the priority queue Q if Q does not contain w.

As in the SSSP problem, we can use any value as the key k(v) for each vertex v and it determines the behavior of Algorithm 3.3. When we remove a vertex v from the priority queue (line 4 in Algorithm 3.3), in an analogous way for the SSSP problem, it is better to extract a vertex v that is “close” to a set of source vertices S for the MSSP problem. While there are various definitions for the closeness between a vertex v and a set of vertices S, we used the minimum distance between v and every s ∈ S as the closeness. That is, we set k(v) = min{d(s, v) | s ∈ S} for each v.

For other definitions, we compared them in Sec. 5.

By computing the MSSP simultaneously, our algorithm can update multiple distance labels in a single scan, while Algorithm 3.2 can update only one distance label in a single scan. Though our algorithm cannot always update multiple distance labels in a single scan, we can expect if the vertices in S are close to each other. An observation behind this expectation is that, if two source vertices s, s′ ∈ S are close to each other, the shortest path tree rooted from s and that rooted from s′ are similar. This means that, if two source vertices s and s′ are close to each other and
Algorithm 3.3 Our MSLC algorithm.

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

1: $d(s, v) = \infty$ and $s(v) = \text{unreached}$ for every vertex $v \in V$ and $s \in S$
2: $d(s, s) = 0$, $s(s) = \text{labeled}$, $k(s) = 0$, 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: \hspace{1em} updated = false
7: \hspace{1em} for all $s \in S$ do
8: \hspace{2em} if $d(s, w) > d(s, v) + \ell(v, w)$ then
9: \hspace{3em} $d(s, w) = d(s, v) + \ell(v, w)$, $s(w) = \text{scanned}$, and updated = true
10: \hspace{1em} end if
11: \hspace{1em} end for
12: \hspace{1em} if updated = true then
13: \hspace{2em} Compute $k(w)$ \hspace{1em} \hspace{1em} // e.g. set $k(w) = \min\{d(s, w) \mid s \in S\}$
14: \hspace{2em} Add $w$ to $Q$ if $Q$ does not contain $w$
15: \hspace{1em} end if
16: \hspace{1em} end for
17: end while
18: Output $d(s, v)$ for all $s \in S$ and $v \in V$

Figure 3.1: An example.

an edge $(v, w)$ is on the shortest path from $s$ to $w$, the edge $(v, w)$ is also likely to be on the shortest path from $s'$ to $w$. Based on this observation, our algorithm often encounter the following simple case: there are two source vertices $s$ and $s'$ that are close to each other, two other vertices $v$ and $w$, and an edge $(v, w) \in E$ that is on the shortest paths from $s$ and $s'$ to $w$ (see Fig. 3.1). In this case, (*) $d^*(s, w) = d^*(s, v) + \ell(v, w)$ and $d^*(s', w) = d^*(s', v) + \ell(v, w)$ hold. After our algorithm have reached a point that the distance labels $d(s, v)$ and $d(s', v)$ are optimal (that is $d(s, v) = d^*(s, v)$ and $d(s', v) = d^*(s', v)$), our algorithm scans the edge $(v, w)$ and this single scan operation updates two distance labels $d(s, w)$ and $d(s', w)$ with the optimal values $d^*(s, w)$ and $d^*(s', w)$ respectively, since (*) hold. Thus, our algorithm is likely to update multiple distance labels in a single scan, if the source vertices in $S$ are close to each other.

The advantage of our algorithm is that a single scan operation can update multiple distance labels, whereas the $n$-Dijkstra algorithm can update at most one distance label in a single scan. Hence, while the $n$-Dijkstra’s algorithm requires exactly $|S|m$ scans for solving the MSSP problem on a strongly connected graph, our algorithm requires only $m$ scans in the best scenario. Though we cannot expect the best scenario all of the time, our experiment in Sec. 5 shows that our algorithm uses much fewer scans than the $n$-Dijkstra algorithm.

Remark. The labeling methods are classified as either the label-setting method or the label-correcting method. The label-setting method has the property that, once a vertex $v$ is extracted from the priority queue $Q$, the distance label does not change after this time. Since this property holds, each vertex is added to $Q$ at most once during the execution of this method and hence the outermost loop of this method is executed at most $n$ times. Dijkstra’s algorithm, which uses the value of $d(s, v)$ as the key $k(v)$, has this property and therefore is classified as the label-setting method, while our algorithm does not have this property and therefore is classified as the label-correcting method.

3.2 Correctness of Our Algorithm

Here we show that our algorithm terminates after a finite number of steps with an optimal solution for any set of source vertices $S$, independent of the value of key associated with each vertex. First we present the following theorem.

Theorem 3.1. Algorithm 3.3 terminates after a finite number of steps.
Proof. During the execution of this algorithm, the distance label \( d(s,v) \) stores the shortest path length from a source vertex \( s \) to a vertex \( v \) among the paths that were found so far. Since the shortest path is a simple path (a path without cycle) and the number of simple paths is finite, the number of updates of the distance label \( d(s,v) \) is also finite, which completes the proof.

Next we show that the solution obtained by our algorithm is correct. Before showing this, we see that the following claim holds:

**Claim 3.2.** When Algorithm 3.3 terminates, the followings hold: (i) \( d(s,s) = 0 \) for every \( s \in S \) and (ii) \( d(s,w) \leq d(s,v) + \ell(v,w) \) for every \( (v,w) \in E \) and \( s \in S \).

**Proof.** (i): Since the distance label \( d(s,s) \) for each source vertex \( s \in S \) is initialized with zero and since the input graph does not contain negative length cycles, the distance label \( d(s,s) \) cannot be updated after the initialization.

(ii): Whenever the distance label \( d(s,v) \) is updated, vertex \( v \) is added to the priority queue \( Q \). Thus vertex \( v \) is later extracted from \( Q \) and the scan step is applied for every edge \((v,w) \in E \) outgoing from \( v \). Therefore, \( d(s,w) \leq d(s,v) + \ell(v,w) \) hold for every \((v,w) \in E \).

By using this claim, we can show that our algorithm outputs a correct answer.

**Theorem 3.3.** Algorithm 3.3 outputs a correct answer.

**Proof.** Let \( p_{s,v} \) be a path from vertex \( s \) to vertex \( v \) and let \( \ell(p_{s,v}) \) be the length of the path \( p_{s,v} \). We first show that \( d(s,v) \leq \ell(p_{s,v}) \) for every \( s \in S \) and \( v \in V \).

We show this by induction. Let \( P_k \) be the set of paths with at most \( k \) edges. Suppose that \( d(s,v) \leq \ell(p_{s,v}) \) holds for every path \( p_{s,v} \in P_k \). Let \( p_{s,w} \) be a path of length \( k+1 \) whose last edge is \((v,w) \in E \). By Claim 3.2 (ii) and the assumption, \( d(s,v) + \ell(v,w) \leq \ell(p_{s,v}) + \ell(v,w) = \ell(p_{s,w}) \). Therefore \( d(s,w) \leq \ell(p_{s,w}) \) holds for every path \( p_{s,w} \in P_{k+1} \). Since the assumption holds for \( k = 0 \) by Claim 3.2 (i), we showed that \( d(s,v) \leq \ell(p_{s,v}) \) for every path \( p_{s,v} \).

It follows that \( d(s,v) \leq d^*(s,v) \) for every \( s \in S \) and \( v \in V \). By the definition of \( d(s,v) \), we have \( d(s,v) \geq d^*(s,v) \) for every \( s \in S \) and \( v \in V \). Hence we have \( d(s,v) = d^*(s,v) \) for every \( s \in S \) and \( v \in V \), which completes the proof.

### 3.3 Graph Partitioning

As mentioned before, in Algorithm 3.1, we should partition the vertices into sets of vertices such that the vertices in each set are close to each other. We used a simple approach for the graph partitioning. Initially all vertices in the input graph are marked as unused. First we arbitrarily select an unused vertex \( s \) as a start vertex and then select other \( B - 1 \) vertices that are close to \( s \). The start vertex \( s \) and the selected \( B - 1 \) vertices are marked as used and we create the set that consists of these \( B \) vertices. We repeat this procedure until all vertices are marked as used. Since our MSLC algorithm works even if a given source vertices \( S \) are not connected each other, we can partition the vertices into exactly \( [n/B] \) sets.

To select \( B - 1 \) vertices that are close to a start vertex \( s \), we used three simple strategies: Breadth First Search (BFS), Depth First Search (DFS), and k-Nearest Neighbor (kNN). The BFS and DFS strategies traverse the graph starting from \( s \) and select the first \( B - 1 \) vertices in the traversal. See [3] for the details of the BFS and DFS traversals. The kNN strategy first constructs a shortest path tree from \( s \) with an algorithm for the SSSP problem such as Dijkstra’s algorithm and then selects first \( B - 1 \) unused vertices in order of increasing shortest path length.

Fig. 3.2 illustrates an example of a result of each graph partitioning strategy. The BFS strategy first selects vertex \( A \) as a start vertex and then selects vertices \( B \) and \( C \) with the BFS traversal. Next, it selects vertex \( D \) as a start vertex and then selects vertices \( G \) and \( I \). Finally, it selects the rest of vertices \( E, F \) and \( H \). Similarly, the DFS strategy constructs three sets \( \{A,B,E\}, \{C,G,H\}, \) and \( \{D,F,I\} \) and the kNN strategy constructs three sets \( \{A,B,D\}, \{C,G,H\}, \) and \( \{E,F,I\} \).

We compared these partitioning strategies and gave its experimental results in Sec. 5.

### 4 SIMD Implementation

Modern processors provide a set of Single Instruction Multiple Data (SIMD) instructions and they can operate on multiple data values in parallel to accelerate computationally intensive programs for a broad range of applications.

A computationally intensive part of our algorithm is the scan operation on edges (line 6-15 in Algorithm 3.3). We exploit the data-level parallelism in the scan operation and implemented this operation with the SIMD instructions. We used a 4-way SIMD instruction set by using 128-bit vector registers where each register contains four 32-bit values. For simplicity, we assume that \( B = |S| \) and \( B \) is divisible by 4. We also 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 \([0, B - 1] \).
Figure 3.2: Three types of the graph partitionings: BFS, DFS, and $kNN$.

First we explain our array layout for the distance labels. We store the distance labels $d(s,v)$ 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 $vd$ of length $Bn/4$, where each vector element $vd[1]$ consists of the four values of $d[i*4]$ to $d[i*4+3]$. We assume the first element of the array $vd$ is aligned on a 128-bit boundary.

Algorithm 4.1 shows pseudo-code of the SIMD implementation of the scan operation in our algorithm. The $updated$ and $compResult$ are Boolean variables and the $edgelength$ and $tempKey$ are vector variables. The $simd_set$ is a vector instruction for loading a vector value consisting of four identical scalar values from the input scalar register. The $simd_add$ and $simd_min$ are vector instructions for add and min instructions respectively. These instructions read from two input vector registers and write to one output vector register. The $simd_add$ instruction adds each value in the first register and the corresponding value in the second input register and returns the sum of them in the output register. The $simd_min$ instruction compares each value in the first register and the corresponding value in the second input register and returns the minimum of them in the output register. The vector compare instruction ($simd_lt$) reads from two input vector registers and compares each value in the first register to the corresponding value in the second input register. If all four values in the first register are smaller than those in the second register, it returns true; otherwise it returns false.

Algorithm 4.1 A SIMD implementation of the scan operation.

**Input:** An edge $(v,w) \in E$

```
1: updated = false
2: edgelength = simd_set( \ell(v,w) )
3: tempKey = simd_set( \infty )
4: for $i = 1, \ldots , B/4$ do
5:    tentativeDistance = simd_add(vd[w*(B/4)+i],edgelength)
6:    compResult = simd_lt(vd[w*(B/4)+i],tentativeDistance)
7:    vd[w*(B/4)+i] = simd_min(vd[w*(B/4)+i],tentativeDistance)
8:    tempKey = simd_min(tempKey,vd[w*(B/4)+i])
9:    updated = true if compResult = false
10: end for
11: if updated = true then
12:    Set $k(w)$ to the minimum scalar value in tempKey
13:    Add $w$ to $L$ if $L$ does not contain $w$
14: end if
```

We implemented the instructions $simd_set$, $simd_add$, $simd_min$, and $simd_lt$ with the SSE instructions provided by Intel [11]. In our implementation, we used the loop-unrolling technique, which achieves a slight speedup (by a few percent).

We also used the MOVNTPS instruction in the SSE instruction set in the initialization phase of our algorithm to store the data without polluting the caches, which accelerated the code by 10–20%.

5 Experiments

In this section, we show experimental results of our algorithm. All tests were executed on a PC workstation, an Intel Quad Core Xeon (X5460) running at 3.16 GHz, with 12 GB of RAM and 2 x 6 MB L2 caches using Microsoft Windows Server 2003 R2 x64 Edition. All of the programs were written in C++ and compiled with Visual C++ 2008. We used the SSE intrinsics for the SIMD instructions. The input graphs were represented by the adjacency-
The execution times show the execution times of the scalar and SIMD versions of our algorithm respectively, and the fourth column shows the results. The first column shows the instance names, the second and third columns show the times (in seconds and do not include the times for reading the input graphs or the times for writing the outputs.

We first compared our algorithm to the Sequential program. Then we tested the parallel implementations. Unless otherwise noted, we used a parameter $B = 128$ and the BFS strategy for the graph partitioning. All of the execution times are expressed in seconds and do not include the times for reading the input graphs or the times for writing the outputs.

We performed our experiments with real-world road networks and synthetic graphs. The graphs we used in this evaluation are:

**Geometric Graphs.** We obtained TIGER/Line files from the 9th DIMACS Implementation Challenge site\(^1\), which consist of road networks in the United States. Each vertex corresponds to an intersection and each edge corresponds to a road connecting two intersections. Since the edges in the graphs are undirected, we converted each undirected edge into two directed edges.

**Grid Graphs.** A grid graph is a two-dimensional mesh with grid dimensions $X$ and $Y$. The vertices of grid graphs correspond to points on the plane with integer coordinates $[x, y]$, where $1 \leq x \leq X$, $1 \leq y \leq Y$. These points are connected by edges of the form $([x, y], [x + 1, y])$, $([x + 1, y], [x, y])$, $([x, y], [x, y + 1])$, and $([x, y + 1], [x, y])$ for $1 \leq x < X$ and $1 \leq y < Y$. Hence the number of vertices is $XY$ and the number of edges is $4XY - 2X - 2Y$. The length of each edge is an integer chosen uniformly and independently from the interval $[0, 1000]$. Note that the edges are not symmetric, that is the lengths of edges $(v, w)$ and $(w, v)$ are different with high probability. We generated two families of grid graphs: Long-$n$ ($X = n/16$, $Y = 16$) and Square-$n$ ($X = Y = \sqrt{n}$).

**Summary.** Table 5.1 gives a summary of the graph instances we used in our experiments. Note that the numbers of edges of the geometric graphs are twice the values shown on the DIMACS site, since we converted each undirected edge into two directed edges.

<table>
<thead>
<tr>
<th>Instance</th>
<th># Vertices</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>9559</td>
<td>29318</td>
</tr>
<tr>
<td>RC</td>
<td>49109</td>
<td>121024</td>
</tr>
<tr>
<td>RI</td>
<td>53658</td>
<td>138426</td>
</tr>
<tr>
<td>HI</td>
<td>64892</td>
<td>153418</td>
</tr>
<tr>
<td>AK</td>
<td>69082</td>
<td>156200</td>
</tr>
<tr>
<td>VT</td>
<td>97975</td>
<td>215116</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance</th>
<th># Vertices</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-16384</td>
<td>16384</td>
<td>63456</td>
</tr>
<tr>
<td>Long-32768</td>
<td>32768</td>
<td>128944</td>
</tr>
<tr>
<td>Long-65536</td>
<td>65536</td>
<td>253920</td>
</tr>
<tr>
<td>Long-131072</td>
<td>131072</td>
<td>597872</td>
</tr>
<tr>
<td>Square-16384</td>
<td>16384</td>
<td>65024</td>
</tr>
<tr>
<td>Square-65536</td>
<td>65536</td>
<td>261120</td>
</tr>
</tbody>
</table>

Table 5.1: The summary of graphs we used in our experiments.

list [3] in main memory and we used single-precision (32-bit) floating point numbers to store edge lengths and path lengths. We used the binary heap [3, 22] for the priority queue implementation.

### 5.1 Test Instances

We performed our experiments with real-world road networks and synthetic graphs. The graphs we used in this evaluation are:

- **Geometric Graphs.** We obtained TIGER/Line files from the 9th DIMACS Implementation Challenge site\(^1\), which consist of road networks in the United States. Each vertex corresponds to an intersection and each edge corresponds to a road connecting two intersections. Since the edges in the graphs are undirected, we converted each undirected edge into two directed edges.

- **Grid Graphs.** A grid graph is a two-dimensional mesh with grid dimensions $X$ and $Y$. The vertices of grid graphs correspond to points on the plane with integer coordinates $[x, y]$, where $1 \leq x \leq X$, $1 \leq y \leq Y$. These points are connected by edges of the form $([x, y], [x + 1, y])$, $([x + 1, y], [x, y])$, $([x, y], [x, y + 1])$, and $([x, y + 1], [x, y])$ for $1 \leq x < X$ and $1 \leq y < Y$. Hence the number of vertices is $XY$ and the number of edges is $4XY - 2X - 2Y$. The length of each edge is an integer chosen uniformly and independently from the interval $[0, 1000]$. Note that the edges are not symmetric, that is the lengths of edges $(v, w)$ and $(w, v)$ are different with high probability. We generated two families of grid graphs: Long-$n$ ($X = n/16$, $Y = 16$) and Square-$n$ ($X = Y = \sqrt{n}$).

**Summary.** Table 5.1 gives a summary of the graph instances we used in our experiments. Note that the numbers of edges of the geometric graphs are twice the values shown on the DIMACS site, since we converted each undirected edge into two directed edges.

### 5.2 Results

We first tested the performances of sequential (single-thread) implementations, which consist of a scalar version and a SIMD version with SSE intrinsics. Then we tested the parallel implementations. Unless otherwise noted, we used a parameter $B = 128$ and the BFS strategy for the graph partitioning. All of the execution times are expressed in seconds and do not include the times for reading the input graphs or the times for writing the outputs.

**Sequential program.** We first compared our algorithm to the $n$-Dijkstra algorithm for solving the APSP problem and Table 5.2 shows the results. The first column shows the instance names, the second and third columns show the execution times of the scalar and SIMD versions of our algorithm respectively, and the fourth column shows the execution times of the $n$-Dijkstra algorithm. The fifth column shows the scan ratio for the number of scans of edges in the $n$-Dijkstra algorithm divided by the number of scans used by our algorithm. Large scan ratio means that our algorithm performed fewer scans than the $n$-Dijkstra algorithm. 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 results show that our algorithm is much faster than the $n$-Dijkstra algorithm for all of the instances. In particular, the SIMD version of our algorithm achieves 14.7–30.8x speedup compared to the $n$-Dijkstra’s algorithm. The large scan ratios indicate that our algorithm reduced the number of scans drastically compared to the $n$-Dijkstra algorithm and it contributes to the speedup. Because fewer scan operation indicates fewer operations on the priority queue, our algorithm is less sensitive on the performance of the priority queue implementation than the $n$-Dijkstra algorithm.

The results also show that our SIMD implementation is faster than the scalar version. The degree of acceleration with SIMD instructions is in the range of 2.3–3.7x. This is smaller than the degree of parallelism available from the SIMD instructions (4x), because our SIMD implementation accelerates only the scan operations and the other parts of our algorithm (such as the operations for the priority queue) are still scalar.

\(^1\)http://www.dis.uniroma1.it/~challenge9/
between a vertex and a large $B$ takes more time but fewer scan operations are needed. Table 5.4 gives an experimental results for various values of partitionings. The results show that the BFS and

The scalar version scales 3.6–3.9x with four threads, while the SIMD version scales only 2.4–3.1x with four threads.

5.3). Fig. 5.1 shows the parallel speedup of the scalar and the SIMD versions of our algorithm for three instances.

Parallel program. We then compared the graph partitioning strategies. Table 5.3 gives the running times and the scan ratios for each strategy. Here we used the SIMD version. The times in the parentheses show the times used for the graph partitionings. The results show that the BFS and $k$NN strategies clearly outperform the DFS strategy. While the $k$NN is faster on instance Square-65536, the BFS strategy would be the best choice. We also confirmed that the times for the graph partitionings are not dominant, especially in the BFS strategy.

Next we compared various settings of the parameter $B$. As the parameter $B$ becomes large, the scan operation takes more time but fewer scan operations are needed. Table 5.4 gives an experimental results for various values of the parameter $B$, where we used SIMD version of our algorithm. Each column shows the execution times and the scan ratios, where the lowest execution times are in bold. We see that a small $B$ value is better for a small graph and a large $B$ value is better for a large graph, and that the parameter $B = 128$ seems to be the best choice for instances we tested.

Finally, we compared other definitions of the keys (the closeness). We can define various measures of closeness between a vertex $v$ and a set of vertices $S$. Here we consider the following three simple definitions: $PQ_{\min}$, $PQ_{\max}$, and $PQ_{\text{avg}}$ that are respectively the minimum, the maximum, and the average distance between a vertex $v$ and a vertex in $S$. Hence we use the following value as the key $k(v)$ in each definition:

- $PQ_{\min}: k(v) = \min\{d(s,v) \mid s \in S\}$,
- $PQ_{\max}: k(v) = \max\{d(s,v) \mid d(s,v) \neq \infty, s \in S\}$,
- $PQ_{\text{avg}}: k(v) = \sum_{s \in S, d(s,v) \neq \infty} d(s,v)$.

Table 5.5 gives an experimental result, which shows the running times and the scan ratios for each definition of closeness. We used the SIMD version with a parameter $B = 128$ and the BFS graph partitioning in this experiment. This results shows that $PQ_{\min}$ is the best.

### Table 5.2: Comparison against the $n$-Dijkstra algorithm.

<table>
<thead>
<tr>
<th>Instance</th>
<th>BFS time ratio</th>
<th>DFS time ratio</th>
<th>$k$NN time ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>0.984 (0.000)</td>
<td>51.6</td>
<td>3.875 (0.032)</td>
</tr>
<tr>
<td>DE</td>
<td>15.437 (0.047)</td>
<td>101.9</td>
<td>42.515 (0.797)</td>
</tr>
<tr>
<td>RI</td>
<td>18.812 (0.047)</td>
<td>98.2</td>
<td>39.140 (0.782)</td>
</tr>
<tr>
<td>HI</td>
<td>10.781 (0.032)</td>
<td>87.8</td>
<td>14.640 (0.235)</td>
</tr>
<tr>
<td>AK</td>
<td>16.906 (0.047)</td>
<td>103.6</td>
<td>31.594 (0.547)</td>
</tr>
<tr>
<td>VT</td>
<td>57.906 (0.063)</td>
<td>110.3</td>
<td>139.547 (2.281)</td>
</tr>
<tr>
<td>Long-65536</td>
<td>3.125 (0.016)</td>
<td>54.9</td>
<td>3.640 (0.156)</td>
</tr>
<tr>
<td>Long-32768</td>
<td>9.797 (0.000)</td>
<td>77.4</td>
<td>11.125 (0.593)</td>
</tr>
<tr>
<td>Long-65536</td>
<td>33.844 (0.032)</td>
<td>96.2</td>
<td>37.796 (2.438)</td>
</tr>
<tr>
<td>Long-131072</td>
<td>124.703 (0.063)</td>
<td>109.9</td>
<td>138.704 (10.766)</td>
</tr>
<tr>
<td>Square-65536</td>
<td>3.282 (0.016)</td>
<td>54.6</td>
<td>14.125 (0.000)</td>
</tr>
<tr>
<td>Square-65536</td>
<td>38.875 (0.032)</td>
<td>95.3</td>
<td>94.047 (0.015)</td>
</tr>
</tbody>
</table>

We then compared the graph partitioning strategies. Table 5.3 gives the running times and the scan ratios for each strategy. Here we used the SIMD version. The times in the parentheses show the times used for the graph partitionings. The results show that the BFS and $k$NN strategies clearly outperform the DFS strategy. While the $k$NN is faster on instance Square-65536, the BFS strategy would be the best choice. We also confirmed that the times for the graph partitionings are not dominant, especially in the BFS strategy.

Next we compared various settings of the parameter $B$. As the parameter $B$ becomes large, the scan operation takes more time but fewer scan operations are needed. Table 5.4 gives an experimental results for various values of the parameter $B$, where we used SIMD version of our algorithm. Each column shows the execution times and the scan ratios, where the lowest execution times are in bold. We see that a small $B$ value is better for a small graph and a large $B$ value is better for a large graph, and that the parameter $B = 128$ seems to be the best choice for instances we tested.

Finally, we compared other definitions of the keys (the closeness). We can define various measures of closeness between a vertex $v$ and a set of vertices $S$. Here we consider the following three simple definitions: $PQ_{\min}$, $PQ_{\max}$, and $PQ_{\text{avg}}$ that are respectively the minimum, the maximum, and the average distance between a vertex $v$ and a vertex in $S$. Hence we use the following value as the key $k(v)$ in each definition:

- $PQ_{\min}: k(v) = \min\{d(s,v) \mid s \in S\}$,
- $PQ_{\max}: k(v) = \max\{d(s,v) \mid d(s,v) \neq \infty, s \in S\}$,
- $PQ_{\text{avg}}: k(v) = \sum_{s \in S, d(s,v) \neq \infty} d(s,v)$.

Table 5.5 gives an experimental result, which shows the running times and the scan ratios for each definition of closeness. We used the SIMD version with a parameter $B = 128$ and the BFS graph partitioning in this experiment. This results shows that $PQ_{\min}$ is the best.

### Parallel program. We then tested the parallel implementations of our algorithm. We used multi-thread to solve the MSSP problems in parallel, because each MSSP computation is independent. The graph partitioning is implemented as a single-thread since the execution time required for the graph partitioning is negligible (see Table 5.3). Fig. 5.1 shows the parallel speedup of the scalar and the SIMD versions of our algorithm for three instances. The scalar version scales 3.6–3.9x with four threads, while the SIMD version scales only 2.4–3.1x with four threads.
With negative length edges. In addition, although our algorithm computes only the shortest path
the multi-source shortest path lengths for each set. We showed that our implementation with SIMD instructions
We gave a parallel algorithm for the APSP problem, which divides vertices into sets of vertices and computes
Since each task for computing MSSP is independent of the other tasks, this may be due to the memory-cache
bandwidth and the decrease in the cache available per processor.

6 Concluding Remarks
We gave a parallel algorithm for the APSP problem, which divides vertices into sets of vertices and computes
the multi-source shortest path lengths for each set. We showed that our implementation with SIMD instructions
achieves an order of magnitude speedup compared to the $n$-Dijkstra algorithm. Our algorithm works on a graph
with negative length edges. In addition, although our algorithm computes only the shortest path lengths, extending
it for computing the shortest paths is easy by maintaining a predecessor for each vertex $v$ (see [3] for details).
Since we used simple heuristics for the key on the priority queue and the graph partitioning, improving heuristics
would be a natural research focus. In addition, assuming that our algorithm runs in a massively parallel and
distributed environment, the sequential graph partitioning algorithm may become a bottleneck for our algorithm
and therefore its parallelisation is also interesting. Since our algorithm is easy to modify for solving the many-to-many
shortest paths problem, its computational study is another research topic.

References
shortest-paths in a directed graph,” In Proc. of the 20th IEEE International Parallel and Distributed Processing
Symposium (IPDPS), 2006.

Table 5.4: Comparison of the various definitions of keys.

<table>
<thead>
<tr>
<th>Instance</th>
<th>DC</th>
<th>DE</th>
<th>RI</th>
<th>HI</th>
<th>AK</th>
<th>VT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B = 64$</td>
<td>0.922</td>
<td>18.907</td>
<td>23.438</td>
<td>11.765</td>
<td>20.203</td>
<td>74.594</td>
</tr>
<tr>
<td>$B = 128$</td>
<td>0.984</td>
<td>15.437</td>
<td>18.812</td>
<td>10.781</td>
<td>16.906</td>
<td>57.906</td>
</tr>
<tr>
<td>$B = 192$</td>
<td>1.359</td>
<td>10.047</td>
<td>18.578</td>
<td>11.328</td>
<td>16.625</td>
<td>53.906</td>
</tr>
<tr>
<td>$B = 256$</td>
<td>2.094</td>
<td>15.468</td>
<td>18.859</td>
<td>11.812</td>
<td>16.828</td>
<td>51.594</td>
</tr>
</tbody>
</table>

Table 5.5: Comparison for various parameter $B$.

<table>
<thead>
<tr>
<th>Instance</th>
<th>PQ$_{\min}$</th>
<th>PQ$_{\max}$</th>
<th>PQ$_{\text{avg}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B = 64$</td>
<td>time</td>
<td>ratio</td>
<td>time</td>
</tr>
<tr>
<td>DC</td>
<td>0.984</td>
<td>51.6</td>
<td>2.297</td>
</tr>
<tr>
<td>DE</td>
<td>18.907</td>
<td>101.9</td>
<td>41.079</td>
</tr>
<tr>
<td>RI</td>
<td>23.438</td>
<td>97.977</td>
<td>56.000</td>
</tr>
<tr>
<td>HI</td>
<td>11.765</td>
<td>87.8</td>
<td>10.781</td>
</tr>
<tr>
<td>AK</td>
<td>20.203</td>
<td>103.6</td>
<td>28.360</td>
</tr>
<tr>
<td>VT</td>
<td>74.594</td>
<td>110.3</td>
<td>130.125</td>
</tr>
<tr>
<td>$B = 128$</td>
<td>0.984</td>
<td>51.6</td>
<td>2.297</td>
</tr>
<tr>
<td>DE</td>
<td>15.437</td>
<td>101.9</td>
<td>41.079</td>
</tr>
<tr>
<td>RI</td>
<td>23.438</td>
<td>97.977</td>
<td>56.000</td>
</tr>
<tr>
<td>HI</td>
<td>11.765</td>
<td>87.8</td>
<td>10.781</td>
</tr>
<tr>
<td>AK</td>
<td>20.203</td>
<td>103.6</td>
<td>28.360</td>
</tr>
<tr>
<td>VT</td>
<td>74.594</td>
<td>110.3</td>
<td>130.125</td>
</tr>
</tbody>
</table>

Since each task for computing MSSP is independent of the other tasks, this may be due to the memory-cache bandwidth and the decrease in the cache available per processor.
Figure 5.1: Parallel performance of the scalar and SIMD implementations.


