DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem

Saeed Maleki[†‡], Donald Nguyen[§], Andrew Lenharth[§]
María Garzarán[†♮], David Padua[†], Keshav Pingali[§]
†: Department of Computer Science, University of Illinois at Urbana-Champaign
‡: Microsoft Research
§: Department of Computer Science, The University of Texas at Austin
♮: Intel Corp.
saemal@microsoft.com, {ddn@cs,lenharth@ices}.utexas.edu,
maria.garzaran@intel.com, padua@illinois.edu, pingali@cs.utexas.edu

ABSTRACT

The Single Source Shortest Path (SSSP) problem consists in finding the shortest paths from a vertex (the source vertex) to all other vertices in a graph. SSSP has numerous applications. For some algorithms and applications, it is useful to solve the SSSP problem in parallel. This is the case of Betweenness Centrality which solves the SSSP problem for multiple source vertices in large graphs. In this paper, we introduce the Dijkstra Strip Mined Relaxation (DSMR) algorithm, an efficient parallel SSSP algorithm for shared and distributed-memory systems. We also introduce a set of preprocessing optimization techniques that significantly reduce the communication overhead without increasing the total amount of work dramatically. Our results show that, DSMR is faster than the best previous algorithm, parallel Δ-Stepping, by up-to 7.38×.

1. INTRODUCTION

Parallel graph algorithms are becoming increasingly important in high performance computing, as evidenced by the numerous parallel graph libraries and frameworks in existence today [17, 24, 5, 22, 32]. The reason for this growing interest is that the input graphs are rapidly increasing in size and, as a result, their processing requires more computation power and memory space. *Scale-free networks* [3] such as Twitter's tweets graph [29] are among the many examples.

This paper presents DSMR (Dijkstra Strip Mined Relaxation), a new parallel algorithm for solving the Single Source Shortest Path (SSSP) problem that is particularly efficient on scale-free networks. Given a weighted graph G and a source vertex s in G, the SSSP problem computes the shortest distance from s to all vertices of G. SSSP is a classical problem that is used in numerous applications such as transportation and robotics. SSSP is also used in the computation of Betweenness Centrality [16], which in turn has multiple applications [31, 28].

Several sequential and parallel algorithms and implementations for SSSP have been proposed, including Dijkstra's algorithm [13], Bellman-Ford's algorithm [4], Chaotic Relaxation [9] and Δ-Stepping [35]. However, these algorithms target general graphs without any specific property. In this paper, we study the parallelization of SSSP for scale-free networks which satisfy the *power law* degree distribution property. This means that scale-free networks have few vertices with high-degrees and many vertices with low degrees [3]. Social networks in which celebrities are represented as high degree vertices and commoners as low degree vertices are examples of graphs that have this property. The skew in degree distribution is also seen in other graphs such as internet web-graphs, and network of citations in scientific articles.

The skew in degree distribution makes parallelization of SSSP more challenging in terms of data distribution, load balancing, and communication. On the other hand, it is possible to take advantage of the nonuniform degree distribution to optimize parallelization of SSSP. The contributions of this paper are:

1. **DSMR**: a partially asynchronous parallel algorithm for solving SSSP that reduces communication without excessively increasing the computation.

2. **Subgraph Extraction**: given an input graph G, a subgraph G' is extracted from G by selecting edges and vertices in the intersection of most shortest paths. SSSP is first solved for G' and then it is solved for $G \backslash G'$ where $G \backslash G'$ is the same graph as G with edges of G' removed.

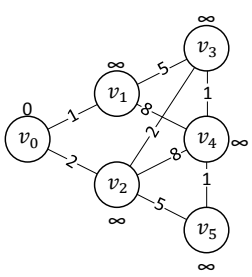
3. **Pruning**: this optimization identifies and removes those edges that are not used in any shortest path.

Our results show that DSMR is up-to 7.38× faster than one of the best shared-memory implementations of the Δ-Stepping algorithm and up-to 2.05× faster than our own implementation of Δ-Stepping on a distributed-memory machine. We also show that our optimization techniques improve the performance by up-to 13×.

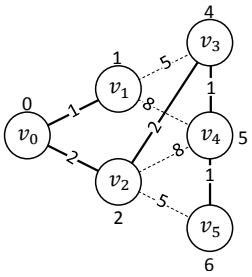
The rest of this paper is organized as follows: Section 2 presents the background, Section 3.1 gives an overview of our approach, Section 3 introduces DSMR and Sections 4

ICS '16, June 01-03, 2016, Istanbul, Turkey

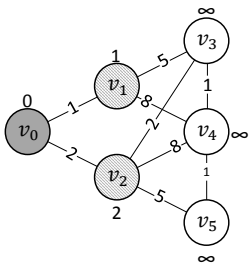
DOI: http://dx.doi.org/10.1145/2925426.2926287



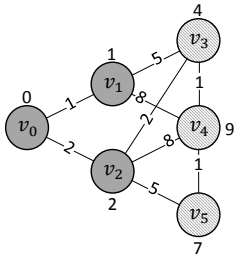
(a) Initial setup. $d(v_1) = \cdots = d(v_5) = \infty$ and $d(v_0) = 0$.



(b) Solution. The bold solid lines show the shortest paths.



(c) Vertex v_0 is relaxed and thus, vertices v_1 and v_2 are activated.



(d) Vertices v_0 and v_1 are relaxed and vertices v_2 and v_3 are active.

Figure 1: An instance of SSSP problem with v_0 as the source vertex. The values next to vertices are the current distances of vertices.

and 5 explain the subgraph extraction and pruning techniques, respectively. Section 6 describes the environmental setup, Section 7 shows the results, Section 8 discusses related works, and Section 9 presents the conclusion.

2. BACKGROUND

The SSSP problem computes the shortest distances in a weighted graph from a source vertex to every other vertex. In this paper, we consider only undirected graphs with non-negative edge weights, but the ideas introduced in this paper can also be applied to directed graphs. A graph G is a pair of (V, E) where V are the vertices and E the edges. Each edge is a pair of vertices (v_i, v_j). The edges and vertices of a graph G are denoted $V(G)$ and $E(G)$, respectively. Figure 1 shows a graph used to illustrate SSSP. We assume that v_0 is the source vertex. The values on the edges represent weights, and those on the vertices represent distances. We use the following notation: $w(v_iv_j)$ denotes the weight of edge v_iv_j. $d(v_i)$ is a dynamic value that changes as the algorithm advances in the computation. At each point in time, $d(v_i)$ is the shortest distance known from the source vertex to v_i. $d(v_i)$ is called the *current* distance of v_i. The value $d_f(v_i)$ denotes the shortest distance computed for v_i. That is, $d_f(v_i)$ is the final value of $d(v_i)$. Figure 1a shows the initialization of the problem: for the source vertex v_0, $d(v_0)$ is set to 0 and for the other vertices $d(v_i)$ is set to ∞. Given this initial setup, applying a set of *relaxation* operations (explained next) will ultimately compute the shortest distances for each vertex. Figure 1b shows the final distances and the shortest paths in solid bold lines.

Relaxation: Relaxation is the basic operation of every SSSP algorithm. There are two types of relaxations: 1) *Relaxing an edge:* relaxing v_iv_j updates $d(v_j)$ to $\min\{d(v_j), d(v_i)+w(v_iv_j)\}$. 2) *Relaxing a vertex:* relaxing v_i relaxes all of the edges connected to v_i (outgoing edges in the case of directed graphs). Relaxation of a vertex, v, becomes necessary when it becomes *active*, that is, when its distance, $d(v)$, is updated (updates always lower the distance). SSSP starts by relaxing the source vertex. This updates the distances of the neighbors of the source vertex which become active. In Figure 1c, relaxing vertex v_0 activates its neighbors v_1 and v_2. Each time the algorithm relaxes a vertex, it removes the vertex from the list of active vertices. Relaxing a vertex can produce new active vertices. When there are no active vertices left, the algorithm terminates and for all vertices, v, we will have that $d_f(v) = d(v)$. Since there could be multiple active vertices at a time, there are multiple possible orders of relaxation. This also means that active vertices can be relaxed in parallel.

Amount of Work: Relaxation of an edge vu requires accessing the distance of the destination vertex. For a parallel algorithm, the distance (d) of this destination vertex, most likely, will not be available in the local cache of the processor doing the relaxation due to the size of the graph and the unpredictability of memory accesses. Therefore, edge relaxation typically requires a long memory access time, which is the dominating factor in the execution time of the SSSP algorithms. For this reason, we use **number of edge relaxations** as a measurement of the amount of work.

Scheduling: Consider Figure 1c again where v_0 is relaxed and vertices v_1 and v_2 are activated by updating their distances. Active vertices v_1 and v_2 can be relaxed in any order or in parallel since they have reached their final shortest distance values. In Figure 1d, vertices v_0, v_1 and v_2 have already been relaxed and vertices v_3, v_4 and v_5 are active. If vertex v_4 is relaxed before vertex v_3, value $d(v_4) = 9$ would be used to relax v_4. Then, when vertex v_3 is relaxed, $d(v_4)$ is updated to 5 and, consequently, v_4 becomes active and needs to be relaxed *again*. A similar situation occurs when v_5 is relaxed before v_4. Therefore, relaxing a vertex whose current distance is not the shortest causes unnecessary work. The vertex relaxation order is the *schedule* of an algorithm and it is the basic difference of the SSSP algorithms considered in this paper.

Dijkstra's Algorithm: Dijkstra's algorithm [13] relaxes active vertices in current distance order meaning that, at each iteration, the active vertex with the minimum current distance is relaxed. For example, in Figure 1d, v_3 must be relaxed before v_4 because $d(v_3) = 4 < d(v_4) = 9$. The Dijkstra's schedule guarantees that each vertex is relaxed at most once (in non-negative edge-weight graphs) and therefore, it performs the minimum amount of work. However, the only source of parallelism in Dijkstra's algorithm is that the vertices with the same minimum current distance can be relaxed at the same time and this parallelism can be limited since typically not many vertices have equal current distances. We will refer to this algorithm as the *parallel Dijkstra's algorithm.*

Bellman-Ford and Chaotic Relaxation Algorithms: The Bellman-Ford's algorithm [4], on the other hand, relaxes all vertices $|V(G)|$ (number of vertices) times regardless of whether or not they are active. Chaotic Relaxation [9] is similar to the Bellman-Ford's algorithm except that it only relaxes the active vertices. Both algorithms are inefficient

in terms of the amount of work they perform. For example, in Figure 1d, these two algorithms allow v_3, v_4 and v_5 to be relaxed at the same time which, as discussed before, results in unnecessary work. On the other hand, they expose more parallelism than Dijkstra's algorithm. For instance, they allow v_1 and v_2 to be relaxed in parallel in Figure 1c.

Δ-Stepping: Δ-Stepping [35] is a SSSP algorithm whose schedule can be adjusted to fall between Dijkstra's and the Chaotic Relaxation algorithms. In Δ-Stepping, i iterates increasingly in $\{0, 1, 2, \dots\}$. For each i, the set of active vertices that can be relaxed are those vertices v that $i.\Delta \leq d(v) < (i+1).\Delta$ where Δ is a constant throughout the algorithm. For example, assume $\Delta = 3$ in Figure 1. For $i = 0$, the active vertices with distances between $[0 \dots 3)$ can be relaxed in any order or parallel. That means that for $i = 0$, v_0 is relaxed first and activates v_1 and v_2 as shown in Figure 1c. Since $d(v_1), d(v_2) < 3$, they can be relaxed in parallel when $i = 0$. Then for $i = 1$, vertices with distances between $[3 \dots 6)$ can be relaxed. In Figure 1d, at first, only v_3 is included in the range and when it is relaxed, $d(v_4)$ is updated to 5 and then v_4 is relaxed. Similarly, relaxing v_4 updates $d(v_5)$ to 6 and v_6 can be relaxed when $i = 2$. Therefore, Δ-Stepping provides two benefits: performing a close-to minimum amount of work while having a reasonable amount of parallelism. Note that Δ-Stepping with $\Delta = 1$ is equivalent to parallel Dijkstra's (assume that edge weights are integers) and with $\Delta = \infty$ is equivalent to Chaotic Relaxation. Thus, Δ is adjustable to balance between work efficiency and parallelism. However, as shown later, Δ-Stepping performs poorly when applied to scale-free graphs.

3. PARALLELIZING SSSP

This section first gives an overview of the Dijkstra Strip Mined Relaxation (DSMR) algorithm and then describes the details of it.

3.1 Overview of DSMR

Figure 2 shows the steps of our SSSP algorithm. Here, the steps in shaded boxes are optional, but the third (Subgraph Extraction) and fifth (Fix-Up) boxes represent a single step, broken into two parts. First the input graph is given to the **Distributor** engine which breaks the graph into P (total number of processors) subgraphs so that all subgraphs have approximately the same number of edges. Each subgraph is assigned to a different processor. The owner of each vertex computes the shortest distance to that vertex from the source. Each processor contains information on all edges incident on the vertices it owns. Therefore, the edges joining vertices assigned to different processors will be replicated.

After the distribution, the graph may be given to the *optional* preprocessing engines: **Pruning** and **Subgraph Extraction**. Pruning is an engine that detects edges that can be guaranteed not to be used for any shortest path from any source vertex. Subgraph Extraction extracts a subgraph of the input such that most shortest paths go through that subgraph. The output graph from the distributor or the preprocessing engines is given to **DSMR** which computes all the shortest paths from a given source vertex. Since the Subgraph Extraction ignores a portion of the graph, it may cause some incorrect computation which are fixed in the **Fix-Up** stage.

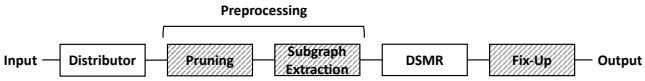


Figure 2: Overview of the engines in our algorithm.

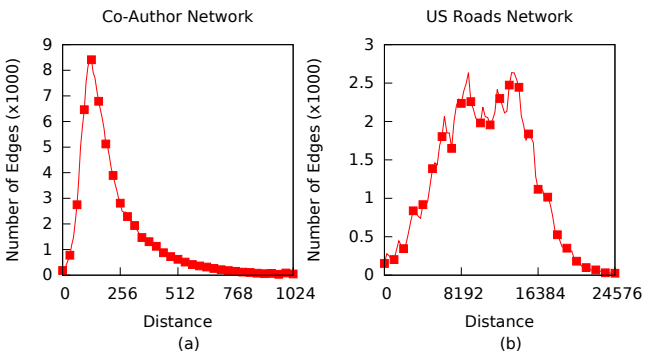


Figure 3: Degree-distance distribution for Co-Author and US Roads Networks.

3.2 Degree-Distance Distributions

Degree-Distance distribution is a characteristic measured after computing the shortest distances from a source vertex. The distribution function is $y(x) = \sum_{v:d_f(v)=x} degree(v)$ where $d_f(v)$ is the shortest distance of v. In other words, $y(x)$ is the total number of edges that are connected to vertices with shortest distance x. Figure 3 shows the degree-distance distribution from a random source vertex for Co-Author and US Roads networks (described in Section 6). Co-Author network is a scale-free network while US Roads network is not. Since distance values for the US Roads network are sparse, each point x represents the accumulation of the distribution function for range $[512x \dots 512(x+1))$.

The obvious difference between the degree-distance distributions of the two networks is that the Co-Author network's plot has a narrow Gaussian shape with a long tail at the end while the US Roads network's plot has a wide Gaussian shape with short head and tail. The US Road network's degree-distance distribution is to be expected since the degree of vertices in the US Roads network are typically small. On the other hand, it typically takes few edges to connect any pair of vertices in a scale-free network as the high-degree vertices can serve as hubs [10]. Therefore, the narrow Gaussian shape for the degree-distance distribution of the Co-Author network results from the fact that most vertices are reached by traversing few edges (narrow and tall part of the plot) and then, there are few vertices that require the traversal of numerous edges to be reached (long tail of the plot). Clearly, the details of the shape highly depends on the edge weights, the source vertex, and the size of a scale-free network, but it is safe to assume that the degree-distance distribution has a narrow shape in scale-free networks.

3.3 High Level Idea of DSMR

In this paper, the *superstep* notion of the BSP (Bulk Synchronous Parallel) [40] model is used. In every superstep, each processor asynchronously executes its local computation and the remote memory accesses are buffered locally. This continues until a global *synchronization* point is reached

and the buffers are exchanged.

The parallel Dijkstra's algorithm (explained in Section 2) could, in each superstep, concurrently relax all active vertices with the same minimum current distance. Thus, the degree-distance distribution of the Co-Author network shown in Figure 3a is also a plot of the amount of parallelism available in each superstep of the algorithm in terms of the total number of edge relaxations for the parallel Dijkstra's algorithm. Also, since this algorithm introduces no unnecessary relaxations, the area under the degree-distance distribution curve is the minimum amount of work needed to compute the shortest distances. As Figure 3a shows, the amount of parallelism in the Co-Author network is high during the first iterations but it drops for longer distances. Note that a synchronization is required after relaxing vertices for each distance value. Because of the long tail, numerous synchronizations are required for longer distances, making this algorithm inefficient. The Δ-Stepping algorithm can reduce the number of synchronizations by allowing relaxation of vertices in ranges of Δ distances as explained in Section 2. This may, however, cause unnecessary edge relaxations.

Figure 3b shows the amount of parallelism for the US Roads network assuming the relaxation schedule of the Δ-Stepping algorithm with $\Delta = 512$. Our experiments show that there are not many unnecessary edge relaxations when using this value of Δ. Therefore, the area under the degree-distance distribution for this network is close to the minimum amount of work. For this Δ, the degree-distance distribution shows that the amount of parallelism for US Roads network, unlike for the Co-Author network, is distributed roughly uniformly, making Δ-Stepping suitable for this graph.

DSMR (Dijkstra Strip Mined Relaxation), our SSSP algorithm, consists of a sequence of supersteps each organized into three stages: 1) Each processor applies Dijkstra's algorithm to the subgraph it owns relaxing its vertices in *distance order* until it has *processed* exactly D edges. D is a parameter of the algorithm. Processing an edge means that the edges with local destinations are relaxed immediately and the edge relaxations that require access to vertices stored in another processor's memory are buffered. This process happens asynchronously and consequently, different processors may work on different distances during the same superstep. 2) After D edges have been processed, the processors rendezvous with all other processors in an all-to-all communication that exchanges the buffered relaxations. 3) We call a vertex v assigned to processor p a *boundary vertex* if there is an edge vu with u assigned to processor $q \neq p$. After the all-to-all, relaxations update the distances of vertices and activate them. These 3-stage supersteps continue until there are no more active vertices.

We call *work overhead* the number of relaxations that an algorithm does in excess of those that Dijkstra's algorithm would have done. Recall that the number of relaxations done by Dijkstra's algorithm is the minimum necessary to compute the shortest distances. Large D values in the DSMR algorithm cause late distance updates and work overhead and small D values cause frequent synchronizations increasing communication cost. To study how DSMR's overhead compares with Δ-Stepping's, we studied the Overhead Distribution and the number of synchronizations for both algorithms.

Overhead Distribution and Synchronization: The cause of overhead is premature vertex relaxation, i.e. a vertex v is relaxed with a $d(v)$ that is greater than the length of the shortest path to v. In other words, vertex v is relaxed prematurely. For example, in Figure 1d, relaxing vertex v_5 would be premature since the final shortest path to v_5 has not been computed, that is, the current value of $d(v_5)$ is not that of the shortest path (Figure 1b). This premature relaxation causes unnecessary relaxations because $d(v_5)$ will be updated at a later time and then, the edges incident on v_5 will have to be relaxed again. The reason for the premature vertex relaxation of v_5 is the order of relaxations. If v_3 and v_4 had been relaxed before v_5, then $d(v_5)$ would be relaxed only once which is not premature. In general, assume that there is a premature vertex relaxation for vertex v at time t. We denote the current shortest distances at time t by $d_t(u)$ for all $u \in V(G)$. If the final shortest path from s to v (that will be eventually computed) is $(s, v_1, v_2, \ldots, v_k, v)$, we say that the premature vertex relaxation of v at time t is due to the first v_i such that $d_t(v_i)$ equals the final shortest distance $(d_f(v_i))$ and v_i has not been relaxed yet at time t. In other words, v_i is the first vertex that should have been relaxed before v. For a premature vertex relaxation of v at time t, we denote culprit vertex v_i by $C_t(v)$.

Each premature vertex relaxation performs unnecessary edge relaxations on each of the edges incident on the vertex. As discussed in Section 2, the number of edge relaxations is a measure of the amount of work. We define *Overhead Distribution* as follows: for distance x, $y(x)$ is $\sum_{v:d_f(v)=x} \sum_{u,t:C_t(u)=v} degree(u)$. In other words, for each vertex v, the number of unnecessary edge relaxations at different times due to v are counted and this number is accumulated to $y(x)$ where x is the final shortest distance of v. Therefore, overhead distribution of an SSSP algorithm shows the amount of overhead associated with each distance.

Figure 4 shows the degree-distance distribution (minimum amount of work) compared with the overhead distribution of Δ-Stepping (DS) and DSMR algorithms for the Co-Author and US Roads networks. The Δ-Stepping algorithm used to obtain this figure and throughout the rest of the paper is our implementation of the original algorithm [35]. In Δ-Stepping, edges vu with $w(vu) \geq \Delta$ are relaxed at most once (because of a technique explained in [35]) and therefore, they are excluded from the overhead distribution. Our Distributor engine distributes the data for both DSMR and the Δ-Stepping algorithm. The values of Δ for Δ-Stepping and D for DSMR algorithms are chosen to facilitate this discussion. Table 1 shows the number of synchronizations (Syncs column), the work overhead ratio which is $(R_{\text{algo}} - R_{\text{Dijkstra}})/R_{\text{Dijkstra}}$ where R_{algo} and R_{Dijkstra} are respectively the number of relaxations done by the algorithm and Dijkstra's algorithm (OH column), and the parameters (D and Δ) for both algorithms.

As Figure 4a shows, the overhead distribution for the Co-Author network with Δ-Stepping is skewed towards the shorter distances. Vertices with long distances cause negligible overhead and can be relaxed in parallel. This result indicates why Δ-Stepping does not perform well with scale-free networks unless Δ is dynamically adjusted (small Δs for shorter distances and large Δs for longer distances). On the other hand, the DSMR overhead distribution is much more uniform and the overall overhead is much smaller than that of Δ-Stepping, while the total number of synchronizations are almost equal in both algorithms, as Table 1 shows. This

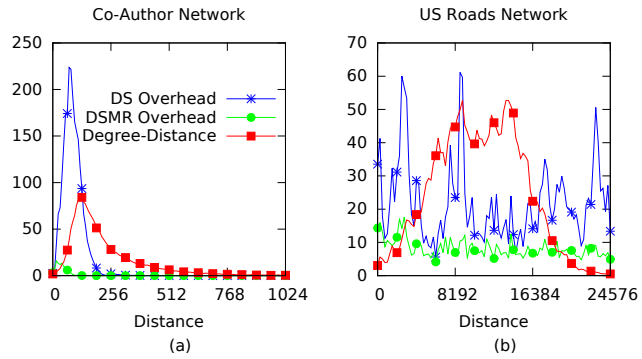


Figure 4: Overhead distribution of Δ-Stepping (DS) algorithm and DSMR algorithm compared with degree-distance distribution.

Graphs	DSMR			Δ-Stepping		
	D	OH	Syncs	Δ	OH	Syncs
Co-Author	2^8	4.8%	62	2^8	115%	64
US Roads	2^5	5.4%	29,932	2^{17}	219%	40,855

Table 1: Comparison of the overhead and number of synchronizations for Δ-Stepping and DSMR for the configurations in Figure 4. **OH:** Overhead, **Syncs:** Number of synchronizations.

is because DSMR relaxes D edges in each superstep avoiding the restrictions on the order of vertex relaxations imposed by Δ-Stepping.

Figure 4b shows the overhead distributions for the US Roads network which reveals a spiky overhead distribution for Δ-Stepping. To explain this behavior, consider range $[\Delta i \dots \Delta(i+1))$. A vertex v with distance close to $\Delta(i+1)$ is unlikely to update any vertices' distance from this range, while vertices with distances close to Δi are. Therefore, the spiky behavior is usually due to premature relaxations of vertices with distances close to Δi. On the other hand, the overhead distribution of DSMR is roughly uniform. The results in Table 1 illustrates that compared to Δ-Stepping, DSMR incurs in significantly less work overhead and requires fewer synchronizations.

3.4 Impact of Parameter D in the DSMR Algorithm

Figure 5 shows the impact of parameter D in the DSMR algorithm. For different values of D from $\{2^6, 2^7, \dots, 2^{18}\}$, the blue line shows the total number of edge relaxations by the DSMR algorithm using 32 processors on the primary Y axis. The red line shows the total number of synchronizations on the secondary Y axis. As it can be seen and is expected, as D increases, the work overhead increases and the number of synchronizations decreases. The best performing D value is the one that obtains a balance between the work overhead and the number of synchronizations. As it is shown, for almost the first half of the D values, the work overhead does not change significantly while for the second half, the number of synchronizations stay almost constant. Therefore, we empirically search for the best performing D value in a small set of values (typically $\{2^7, 2^8, \dots, 2^{14}\}$). We experimentally found that the best D value is usually

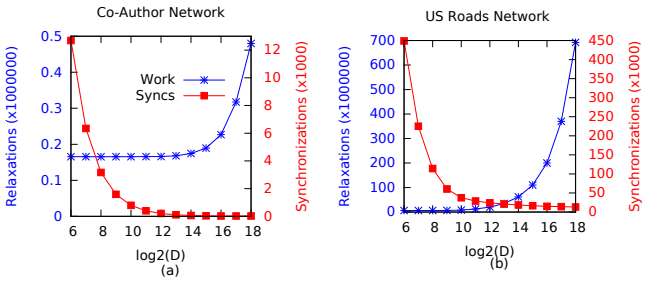


Figure 5: Impact of parameter D in the DSMR algorithm with 32 processors on the work overhead and number of synchronizations for the Co-Author and US Roads networks. The X axis represents different values for D, the primary Y axis shows the total number of relaxations, and the secondary Y axis shows the number of synchronizations.

consistent across different source vertices. Therefore, as will be discussed in Section 7, for the evaluation of DSMR for different graphs, we searched for the best D value from a random source vertex and then measured the average running time across 100 different random source vertices with the same D value.

3.5 The Distributor

Graph distribution has a major impact on the performance of parallel SSSP algorithms. However, this paper focuses on the DSMR algorithm and how its schedule is better than Δ-Stepping. Therefore, we leave the study of graph distribution impact on SSSP algorithms for future work since it is an independent factor in the performance. For this paper, we found that the distribution algorithm described next performs the best among all the distribution strategies we considered.

There are two major concerns in data distribution of a scale-free graph on a distributed-memory system: existence of high-degree vertices and vertices assignment to processors. As discussed before in Section 1, scale-free networks have a few high-degree vertices and many low degree vertices. Assigning a high-degree vertex to a single processor increases the likelihood of load imbalance, which can be handled by a technique known as *Vertex Splitting* [22]. The Distributor engine accepts as an input a threshold and the vertices with degree higher than that are copied on each of the P processors and $\frac{1}{P}$th of edges of the original vertex are assigned to each of the processors. These P copies are connected to a unique copy by P edges with weight 0 which guarantees equal shortest distances for all $P+1$ copies. The low degree vertices are shuffled using a random permutation. Then, they are assigned to processors in consecutive chunks such that the number of edges for each processor is roughly the same.

The output of the distributor will be P subgraphs with disjoint vertex sets. However, the edges joining these subgraphs are shared by the processors containing the subgraphs. Each subgraph has an equal number of edges and, consequently, the number of vertices are not necessarily equal. The owner of each vertex is responsible for computing the shortest distance to that vertex.

3.6 Implementation of DSMR Algorithm

Figure 6 shows a pseudo code for our DSMR algorithm.

The algorithm is written in an SPMD model and uses MPI for communication. Therefore, all of the variables are private. The codes related to the control of the number of relaxations per superstep (which is equal to `D`) are shown in magenta. Array `d` contains the current distance of each vertex. For any vertex `u`, `d(u)` is initially ∞. Variable `relaxed`, declared in line 2, tracks the number of edge relaxations in each superstep and whenever it reaches the threshold `D`, an all-to-all communication is executed. The worklist `wl`, declared in line 5, is a vector of sets where each set corresponds to a distance value and contains all active vertices whose current distance is that of the set. In this algorithm, we assume that all the edge values (and consequently distance values) are integers. Therefore going through vertices in distance order locally in each processor is straightforward. Function `RelaxEdge` in line 7 relaxes an edge and updates `d(u)` and `wl` in lines 9-12. `active(u)` specifies if vertex `u` is active, which means that it is in the worklist (as checked before erasing it in line 9) and needs to be relaxed using the `RelaxVertex` function in line 15. Relaxation of an edge `vu` for which the processor owns both ends occurs immediately in line 20 but the remote relaxations are buffered in line 19. Line 22 enforces to not have more than `D` edge relaxations in a superstep.

The main DSMR algorithm's function is in line 24 which takes a source vertex v_{src} that is only set for the processor which owns it. The initialization of v_{src} occurs in line 25. In line 29, the set with minimum distance in `wl` is found and active vertices from it are relaxed in line 31. Eventually, after `D` edge relaxations, an `MPI_Alltoall` routine exchanges the buffers in line 33 and the received remote relaxations from the buffers are relaxed in the loop in line 35. At the end, `relaxed` is reset in line 37. The algorithm terminates when `wl`s in all processors are empty.

We are omitting a few parts of the algorithm for the sake of simplicity. This includes the code for when the break in line 22 occurs in the middle of the relaxation of a vertex. DSMR completes the relaxation of that vertex at the beginning of the next superstep. The other part of the algorithm that we are omitting is determining that the `wl`s are empty. This test is performed during the `MPI_Alltoall` communication without requiring additional communication.

The implementation of Dijkstra's algorithm which is executed by each processor can be replaced by other implementation to obtain better efficiency or be able to work with non-integer weights. However, our experiments show that the implementation we used for the work reported in this paper works well with most of the graphs that we evaluated.

Since each of the P processor processes D edges and the graph is randomized, it could be expected that $\frac{D}{P}$ remote edge relaxations be typically buffered and then sent to each other processor (line 19). However, we have seen some uneven behavior with a large number of processors. Thus, to maintain a constant amount of work for the loop in line 35, each processor only sends maximum of $1.25 \times \frac{D}{P}$ remote edge relaxations to each other processor. As a result, no processor receives more than $1.25 \times D$ edges to relax at the beginning of each superstep.

3.7 Computational Complexity of DSMR

In this section, we will study the computational complexity of the parallel DSMR algorithm.

```
1  // Number of relaxations in each superstep
2  int relaxed = 0;
3  // Worklist for active vertices
4  // Each vector index represents a distance
5  Vector<Set<Vertex> > wl;
6
7  void RelaxEdge(Vertex u, int newDist){
8    if (d(u) > newDist){
9      if (active(u)) // Remove u from old set
10       wl[d(u)].erase(u);
11     d(u) = newDist;
12     wl[d(u)].insert(u); // Insert u to new set
13     active(u) = true; }}
14
15 void RelaxVertex(Vertex v){
16   active(v) = false;
17   foreach Edge vu in edges(v) {
18     relaxed++;
19     if (IsRemote(vu)) Buffer(<u,d(v)+w(vu)>);
20     else RelaxEdge(u, d(v)+w(vu));
21     // When threshold is reached, break
22     if (relaxed >= D) break; }}
23
24 void DSMR(Vertex v_src){
25   if (v_src) RelaxEdge(v_src, 0); // Initialization
26   do {
27     do {
28       // Find the minimum non-empty set
29       int ind = min i: !IsEmpty(wl[i]);
30       while (!IsEmpty(wl[ind]) && relaxed < D){
31         RelaxVertex(wl[ind].pop()); }
32     } while (ind < ∞ || relaxed < D);
33     MPI_Alltoall(buffer); // Exchange buffers
34     // Relax received requests
35     foreach <u,dist> in buffer:
36       RelaxEdge(u, dist);
37     relaxed = 0; // Reset
38   } while (all IsEmpty(wl)); }
```

Figure 6: Pseudo code for our DSMR algorithm.

A popular implementation of Dijkstra's algorithm requires $O(|E| + |V| \cdot \log |V|)$ operations assuming that a Fibonacci heap [15] is used to store the priority queue ordering the vertices by their current distance. The $|V| \cdot \log |V|$ term represents operations to maintain the heap. However, in the implementation of DSMR, we used an approach that is similar to the one proposed by Goldberg [20]. To maintain the active vertices, DSMR uses a set for each discrete distance value. An active vertex with the minimum distance can be found by looking at the first non-empty set. If there are a few distinct distance values and they are close to each other, there would be very few sets to maintain and, consequently, the computational cost to maintain the sets would be small. In [20], Goldberg shows that if the edge weights are uniformly distributed, the average running time of the algorithm would be $O(|E|)$. As we will discuss in Section 6, most of the graphs we studied satisfy this requirement. The only exceptions are the US Roads and Co-Author networks but we found that the cost of maintaining the sets for these graphs is negligible.

Each processor in DSMR locally performs Dijkstra's algorithm as described above. Since the edges are distributed uniformly, each processor owns approximately $|E|/P$ edges where P is the total number of processors. We call the total work overhead (additional edge relaxations over those required by the sequential Dijkstra's algorithm) of DSMR L_D where D is the number of edge relaxations between communications. We assume that L_D is distributed uniformly across processors. Therefore, each processor performs $O((|E| + L_D)/P)$ edge relaxations.

The only communication routine that we use in DSMR is personalized all-to-all whose cost is $O((\alpha + \beta m)P)$ where m is the size of the message from each processor to another. As discussed in Section 3.6, $m = O(D/P)$ and therefore, the cost of each all-to-all is $O(\alpha P + \beta D) = O(P + D)$. Now, if we assume that the number of processors is large, most of the edges will be remote and therefore, require communication to relax. As discussed above, there are $O((|E| + L_D)/P)$ edge relaxations per processor that are mostly remote. Each all-to-all routine exchanges $(P-1) \times D/P = O(D)$ edges from each processor. Therefore, $O((|E| + L_D)/(P \cdot D))$ all-to-alls are required. Adding it all together, the overall complexity of DSMR algorithm is $O((|E| + L_D)/(P \cdot D))O(P + D) + O((|E| + L_D)/P)$ which is equivalently $O((|E| + L_D)/P + (|E| + L_D)/D)$.

The $O(L_D)$ work overhead can range from 0 (with a very small D value effectively DSMR is the same as the Dijkstra's algorithm) to $O(|V| \cdot |E|)$ (with a very large D value DSMR has the work complexity of the Bellman-Ford's algorithm [4]).

4. GRAPH EXTRACTION

Graph extraction is a preprocessing technique that **extracts** a subgraph $G' \subseteq G$ from the input graph G, such that most of the shortest paths in G go through G'. Once G' is computed, the shortest paths are computed in two phases: First, DSMR is executed with G' to compute the shortest distances in G'. After this is done, the shortest distances for most vertices of G would have been computed correctly. Then, for the rest of the graph, $G \backslash G'$, the Fix-Up engine corrects the distances of the vertices in G computed incorrectly in the first phase. Using the edges in $G \backslash G'$ the Fix-up algorithm updates the distances of only a few vertices and consequently, relaxing these vertices in any order will not cause significant work overhead. Therefore, the fix-up phase uses Chaotic Relaxation to minimize the number of synchronizations. Next, we will discuss the input characteristic that impacts the profitability of graph extraction.

4.1 Graph Characteristics

Artificial or unweighted scale-free networks are typically weighted by assigning pseudorandom values with a uniform distribution in the interval $[1, C)$ where C is a constant. This approach is widely used [7, 19, 36, 33] and also adopted by the DARPA SSCA#2 benchmark [2]. One of the properties of the weighted graphs generated in this manner is that heavy-weight edges are unlikely to be used in shortest paths. There are other edge-weight distribution such as log-uniform [33] for which it is even less likely for heavy-edge weights to appear in shortest paths. To study this property, we measured the *HE (Heaviest Edge)* distribution of the vertices. Assume that SSSP is computed for a graph from a random source vertex s and that $(s, v_0, v_1 \ldots, v_k, v)$ is the shortest path from s to v. We define $HE(v)$ to be the heaviest edge weight in the shortest path from s to v: $HE(v) = \max\{w(sv_0), w(v_0v_1), \ldots, w(v_kv)\}$. The key idea is that if all the edges with weight $> HE(v)$ are ignored, the shortest path for v can still be computed correctly.

Figure 7 shows two different distributions for a type-2 RMAT graph with *scale* = 21 (graph description in Section 6) with edge weight distributed uniformly from $[1 \ldots 256]$. The horizontal X axis represents weight values and the two distributions are: 1) Cumulative HE (CHE) distribution of the vertices: $CHE(x)$ is the percentage of vertices v with $HE(v) \leq x$. 2) Cumulative edge weight (CEW) distribution: $CEW(x)$ shows the percentage of edges uv with $w(uv) \leq x$. This distribution is linear because of the uniform edge weight distribution. Now, consider the vertical dashed purple line ($x = 48$) of Figure 7. If subgraph G' is extracted from G with all edges uv where $w(uv) \leq 48$, G' contains less than 20% of edges of G (the edge-weight distribution at the vertical dashed line). However, short distances will be computed correctly for almost 80% of the vertices in G' (HE distribution at the dashed vertical line in the figure). This means that by considering a small part of $E(G)$, shortest distance will be computed for a large set of $V(G)$. For the remaining 80% of the edges, fix-up phase will correct the distances for the remaining 20% of the vertices. The subtle difference between the first phase and fix-up phase is that, fix-up phase can have less synchronizations than DSMR while the total amount of work is not increased significantly.

Note that for the experiment shown in Figure 7, the source vertex s was chosen from G'. In the cases where s is not in G', DSMR starts by processing the whole graph, G, and relaxes vertices and edges in G. Once all active vertices are in G', DSMR continues working in G' only. Afterwards, as before, the fix-up phase will take care of the rest of the graph.

4.2 Implementation of Graph Extraction and Fix-Up

Subgraph G' is extracted from input graph G by simply specifying a threshold T and assigning edges with weight less than T to G'. This process is completely hidden by the I/O while the graph is being loaded from the disk. Therefore,

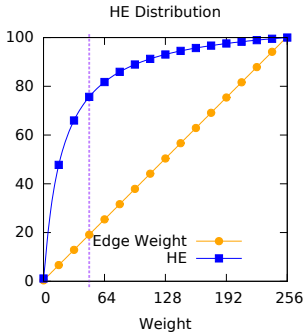


Figure 7: HE distributions for an RMAT graph with uniformly random edge weight distribution from $[1 \ldots 256]$.

```
1  Set<Vertex> wl; // Set of active vertices
2  void RelaxEdge(Vertex u, int newDist){
3    if (d(u) > newDist){
4        d(u) = newDist;
5        wl.insert(u); }}
6
7  void FixUp(){
8    foreach vu in G\G'
9      if (d(u) > w(vu)) // Optional if
10       RelaxEdge(u, d(v)+w(vu));
11   while (!IsEmpty(wl)){
12     Vertex v = wl.pop();
13     foreach vu in G
14       RelaxEdge(u, d(v)+weight(vu));  }}
```

Figure 8: Pseudo code for Fix-Up engine.

the overhead for this process is negligible as a part of loading time.

Figure 8 shows the pseudo code for the Fix-Up engine. This code is sequential (we discuss later how to parallelize it). The algorithm is similar to the one from the DSMR algorithm in Figure 6 with a few exceptions. `wl` in line 1 is just a set instead of a vector of sets. This is because the fix-up executes Chaotic-Relaxation, where relaxations can occur in any order whereas in Figure 6, DSMR relaxes vertices in distance order. Consequently, `RelaxEdge` in line 2 is simpler than the one in Figure 6. The `FixUp` function has two major loops. The first loop in line 8 goes through all the edges in `G\G'` and relaxes them. In this loop, there is an optional condition in magenta in line 9 whose raison d'etre is discussed below. The second loop in line 11, goes through all vertices of `wl` and relaxes all of their incident edges in `G`. The major difference between these two loops are the graphs: `G\G'` for the loop in line 8 and `G` for the loop in line 11. Before the Fix-Up engine, DSMR computes shortest distances in `G'`. The loop in line 8 relaxes edges in `G\G'` and the incorrect distances are updated. Updating distances of vertices causes activating them and adding them to `wl` in line 5. Later, vertices in `wl` are relaxed in the whole graph, `G`, in the loop in line 11. This loop, itself, may activate other vertices in `G` in line 14.

The parallelization of the fix-up algorithm is straight forward and similar to the parallelization of DSMR. The remote edge relaxations are buffered and after `wl` is empty in all processors, the buffers are exchanged via an `MPI_Alltoall` and then the remote relaxations are performed. The processors continue until no more relaxations are left.

The magenta condition in line 9 is an optional branch and can be removed. However, it makes a great difference in performance. As discussed before, an edge relaxation requires a memory look up of the distance of the target vertex. Now assume that edge `vu` is accessed from the destination vertex `u` where `w(vu)`, `d(u)` are close in memory (spatial locality). Edge `vu` would update `d(u)` only if the magenta condition is true: `d(u)>w(vu)`. Otherwise, it is not required to access `d(v)` which in turn improves the performance. Surprisingly, the condition is seldom true and that comes from the fact that the shortest distances in scale-free networks with uniform edge-weight distribution are even shorter than the length of most heavy-weight edges. Our experiments show that this is independent of the constant C in the uniform distribution $[1 \ldots C]$. The idea behind this condition originated from the pull model in the SSSP algorithm discussed in [7], but it is used in a different way in this paper. The idea is used in the Fix-Up engine in this paper while in [7], it is used within their SSSP algorithm and there is no post fix-up phase. However, the benefit of this condition disappears with the Pruning engine as discussed in Section 5. Section 7 presents the performance gains with graph extraction and pruning.

5. PRUNING

Pruning is another preprocessing technique that identifies edges in a graph G which can be guaranteed not to be used in any shortest path from any source vertex. Figure 9 shows a pruning scenario where edge vu, shown as a dotted line, is a candidate for pruning. Our pruning engine chooses a random source vertex s (not necessarily the one used by the DSMR engine) and computes the shortest distances for all vertices. Figure 9 shows the shortest paths from the chosen source vertex s to u and v by solid wavy lines. Assume that these two shortest path diverge at vertex x. We call x the first common ancestor of v and u. If the condition $(d(u) - d(x)) + (d(v) - d(x)) < w(vu)$ is true, vu is marked useless. We call this the *the pruning test*. The reason why this edge can be marked as useless and ignored when computing the shortest path from any source vertex is that the paths from x to v and from x to u are of distance $d(v) - d(x)$ and $d(u) - d(x)$, respectively. Therefore, if the distances of these two paths together are less than $w(vu)$, the edge vu will not be used in any shortest path since when going from u to v (or from v to u) is always shorter to go through x.

A useless edge seems illogical in a road network because a long segment of a road will be useless if there is a faster way around connecting the two points. However, in social networks, edge weights do not represent the distance between vertices, but rather the strength of their connectivity. For example, in the Co-Author network, the weight of an edge between two vertices is a function of the number of articles two authors had together and the number of participants in those publications. Therefore, it is likely to find useless edges in scale-free networks.

Our pruning algorithm, even for only one source vertex, takes longer than running SSSP itself, but when SSSP is executed for multiple source vertices on the same graph, running pruning could be profitable.

5.1 Algorithm

Figure 10 shows the sequential pseudo code for our pruning algorithm which is optimized for the amount of memory used. The algorithm starts by executing DSMR in line 4.

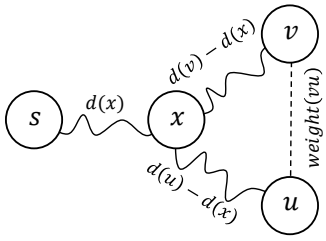


Figure 9: Pruning idea. x is the first common ancestor of v and u.

```
1  // Root vertices of subtrees
2  Set<Vertex> st;
3  void Prune(Vertex v_src){
4    DSMR(v_src); // Run SSSP
5    subtrees.insert(v_src);
6    do {
7      foreach Vertex w in st
8        foreach v in subtree(w)
9          foreach Edge vu in edges(v)
10           if u in subtree(w) && !useless(vu)
11             // Pruning test
12             if d(v)+d(u)-2*d(w)<w(vu)
13               useless(vu) = true;
14     // Go to the subtrees of st
15     foreach Vertex v in sb {
16       sb.remove(v); sb.insert(succ(v)); }
17   } while(!IsEmpty(st)); }
```

Figure 10: Pseudo code for Prune engine.

The shortest paths in a graph from a source vertex creates a tree which we denote by T. T is computed along with our DSMR algorithm by storing $succ(v)$, the successor list of v for all v in T. Therefore, subtrees of T can be accessed by their root and following the *succ* lists. We denote the subtree of a vertex v as its root by $subtree(v)$. Set `st` in line 2 keeps the root of subtrees for the computation. In line 5, v_{src} is added to the set `st`. At anytime, `st` holds non-overlapping subtrees. The loop in line 7 goes through each root `w` in `st`. `w` is a common ancestor for all vertices in `subtree(w)`. Therefore, all edges among pairs of vertices of `subtree(w)` are traversed in the loops in lines 8 and 9 and the pruning test is executed for them with `w` as their common ancestor in line 12.

After the test is done for all subtrees in `st`, the loop in line 15 goes through the roots in `st` and replaces them with their `succ`s. The new list of subtrees will be used for pruning in loop in line 7. This process continues until `st` is empty (line 17). This is a BFS traversal of the main subtree from v_{src}. Note that in our algorithm `subtree(w)` is never stored anywhere but is accessed through `succ` lists as discussed before. Therefore, our algorithm requires at most $O(|V(G)|)$ memory since T has at most $|V(G)|$ edges, which is equal to the sum of the length of the successor lists and the maximum size of set `st` is $|V(G)|$. Parallelizing pruning is similar to parallelizing DSMR algorithm in Figure 6. Operations requiring remote memory accesses are buffered and communicated once there is no more local work.

6. ENVIRONMENTAL SETUP

Machines: Two experimental machines were used for the evaluation: a shared-memory machine with 40 cores (4 10-core Intel® Xeon™ E7-4860) and 128GB of memory; the distributed memory machine Mira, a supercomputer at Argonne National Lab. Mira has 49152 nodes and each node has 16 cores (PowerPC A2) with 16GB of memory.

Compilers: For the shared-memory machine, we used Intel® C/C++ compiler [27] version 14.0.3 and MPICH MPI library 3.1.4 [37]. For Mira, we used IBM MPI and XL C/C++ compiler for Blue Gene version 12.1 [26].

Graphs: Most of large scale-free networks are unweighted and they are weighted by assigning pseudorandom values uniformly distributed in interval $[1, C)$ where C is a constant [7, 19, 36, 33, 2]. We adopted this approach for our unweighted graphs.

Co-Author Network: The Co-Author network represents the connectivity of authors publishing in the American Mathematical Society. It is considered a scale-free network. It has $391,529$ vertices and $873,775$ edges. Vertices represents authors and an article with N authors increases the edge weight between each pairs of authors by $1/(N-1)$ [38]. Consequently, heavier edge weights in this graph represents stronger connectivity. In the experiments, each edge weight w is replaced with $100/w$ so that stronger connections represent shorter distances.

US Roads Network: US Roads network is the map of the United States roads. Each edge weight represents the distance between a pair of vertices. It has $23,947,347$ vertices and $58,333,344$ edges and it is not a scale-free network [12].

RMAT: RMAT graph model is an artificial scale-free graph generator [8]. An instance of an RMAT graph has the following parameters: 1) *scale*: determines the vertex set size: $|V(G)| = 2^{scale}$, 2) edge factor: determines $|E(G)|/|V(G)|$ ratio, 3) a, b, c and d: determines the skewness of the degree distribution. Edge factor 16 was used as proposed by Graph500 [23]. For a, b, c and d there are two configurations: type-1 which is Graph500 setup ($a = .57$, $b = c = .19$ and $d = .05$) and type-2 which is SSCA#2 [2] benchmark setup ($a = .55$, $b = c = .1$ and $d = .25$). Edge weights for type-1 and type-2 RMAT graphs are integers chosen uniformly random from $[0 \ldots 256)$ and $[1 \ldots 256]$, respectively.

Orkut: Orkut is a scale-free social network website and the graph represents its users and their friendship. This network has $3,072,441$ nodes and $117,185,083$ edges. It is originally unweighted and was weighted by distributing edge weights uniformly random from interval $[1, 256]$.

Twitter: Twitter graph represents the follower/following relationship among the users [29]. Each vertex is a user and each edge vu between two users shows v is following u. This graph has 41.7 million users and 1.47 billion edges. This graph is unweighted and edge weights were distributed uniformly random from $[1 \ldots 256]$.

7. RESULTS

This Section evaluates the performance of our algorithms and compares it with the best existing SSSP algorithms. First, we discuss the engines involved in our computation.

7.1 Running Time Discussion

As discussed in Section 3.1, there are multiple engines involved in the execution of our algorithm: Loading the input (I/O), Distributor, Pruning and Subgraph Extraction (preprocessing engines), DSMR, and Fix-up. The first four engines are executed once while the last two are executed for each source vertex. Notice that computing SSSP for only one source vertex is significantly faster than loading the

Sources	I/O+Extraction	Distributor	Initialization	Pruning	DSMR+Fix Up
1	$\sim 10\%$	$\sim 66\%$	$\sim 6\%$	$\sim 16\%$	$\sim 2\%$
1024	$\sim 0.5\%$	$\sim 3.0\%$	$\sim 0.3\%$	$\sim 0.7\%$	$\sim 95.4\%$

Table 2: Engines relative running time for 1 and 1024 source vertices on the shared-memory machine using 32 processors.

graph. However, for a scale-free network, the desired computation is usually computing SSSP from multiple source vertices. For example, the SSCA#2 benchmark [2] suggests evaluating Betweenness Centrality [16], a metric computed by shortest distances from multiple sources, for at least 1024 sources. In this scenario, the running time of the last two engines executed for several times is more time-consuming than the running time of the first four engines executed only once. For comparison, we measured the running time of our engines for 1 and 1024 source vertices for the Orkut network on the shared-memory machine using 32 processors. Table 2 compares the fraction of the total running time consumed by each engine for 1 and 1024 source vertices. As it can be seen, the engines which are executed once consume $\sim 98\%$ of the total computation time for 1 source vertex but only $\sim 5\%$ when SSSP is solved for 1024 source vertices. Therefore, it is important to focus on the running time of the last two engines, DSMR and Fix-Up, since the impact of the others become insignificant with just 1024 source vertices. We found a similar result for other networks on the shared-memory machine and Mira as well. Therefore, for the rest of this Section, we only report the running time for these two engines.

Also note that in this paper, we studied the parallelism for a single SSSP computation (intra-SSSP parallelism). However multiple SSSP executions can be executed in parallel with each other. We did not study this type of parallelism, which could in theory complement the intra-SSSP parallelism. We should also point out that the execution of this embarrassingly parallel approach is limited by the need to replicate graph information which would increase memory requirements, perhaps beyond what the target machine can support.

7.2 Shared-Memory Results

Figure 11 compares DSMR, our implementation of Δ-Stepping (DS), the Δ-Stepping from the Elixir collection [39] implemented in the Galois system [17], and the performance of the sequential solver from DIMACS challenge [1]. The DIMACS solver is an efficient sequential SSSP algorithm that we use as a baseline. All of the lines in Figure 11 (expect DIMACS) represent a strong scaling comparison in which the same input graph across all processor numbers is used. The networks for this evaluation are: Co-Author, US Roads, Orkut and a type-2 RMAT graph with *scale* 22. The experimental machine is the 40-core shared-memory machine. For each algorithm, the best parameters (Δ in Δ-Stepping is searched from $\{2^0, 2^1, \ldots, 2^{13}\}$ and D in DSMR is searched from $\{2^7, 2^8, \ldots, 2^{14}\}$) were searched from a random source vertex. These parameters are stable when changing the source vertex and, therefore, we executed the three algorithms with them for 100 other random source vertices. The performance results are presented in TEPS (Traversed Edges Per Second) which is $|E(G)|/T$, where T is the running time in seconds. Note that when computing TEPS, we are not considering the number of edge relaxations but the number of existing edges.

Figure 11 shows the result for this evaluation. The X axis represents different number of processors and the Y axis shows the average MTEPS (Mega TEPS) of the 100 random source vertices. As the figure shows, the DIMACS solver is faster than the sequential DSMR: 1.09×, 1.83×, 4.60× and 2.26× for Co-Author, US Roads, RMAT 22 and Orkut networks, respectively. However, unlike DIMACS solver, DSMR is a parallel algorithm and eventually it becomes significantly faster: 14.03×, 3.75×, 12.06× and 12.93×, respectively. Also, as it can be seen from the figure, DSMR is faster and scales better than both Δ-Stepping algorithms, except for the US Roads network where DSMR is slower than the Elixir Δ-Stepping with 32 processors. Note that Elixir is a shared-memory implementation while DSMR and our Δ-Stepping algorithms are implemented using MPI. This explains why DSMR is slower than the Elixir Δ-Stepping in Orkut network on less than 16 processors. The speed up of DSMR over Elixir and our Δ-Stepping with 32 processors, respectively, are: for Co-Author 3.59× and 1.64×, for US Roads 0.75× (slow down) and 1.50×, for RMAT22 7.38× and 3.27×, and for Orkut 1.74× and 3.19×. Table 3 (shared-memory part) shows D, Δ, overhead (with respect to the minimum amount of work) and the number of synchronizations for the experiments in Figure 11 with 32 processors. As shown, both DSMR and our Δ-Stepping algorithms have similar overhead but the number of synchronizations are significantly different. This explains the difference in performance.

Now, consider the subgraph extraction results in Figure 11. These are shown only for RMAT22 and Orkut since subgraph extraction is not beneficial for the US Roads and Co-Author networks. The HE Extraction column in Table 3 shows the thresholds used for the HE property (Section 4.1) and the value of $|G'|/|G|$. As it can be seen, graph extraction significantly accelerates DSMR (1.88× in RMAT22 and 2.41× in Orkut with 32 processors). Note that DSMR+Extract is greatly faster than DS+Extract even though, as Table 3 shows, G' is a small part of G and the same fix-up code was used for $G \backslash G'$ (a large part of G). Finally, consider the pruned results in Figure 11 for Co-Author, RMAT22 and Orkut. Last column of Table 3 shows what percentage of each graph was pruned. For Co-Author, it took 84 iterations for the pruning algorithm (Section 5) to converge. For RMAT22 and Orkut, only one iteration was enough. As it can be seen, DSMR+Pruned is better than DSMR+Extract since it removes most of the useless edges. The improvement of DSMR with pruning over DSMR is: 1.22× for Co-Author, 3.12× for RMAT22 and 3.87× for Orkut networks. We excluded DS+Pruned since its difference with DSMR+Pruned is similar to the difference between DS+Extract and DSMR+Extract.

7.3 Distributed-Memory Results

Figure 12 shows similar results to those in Figure 11 for the distributed-memory machine, Mira, with larger graphs. Plots a, b and c show results for three fixed-size graphs (strong scaling): a type-2 RMAT with *scale* 26, Orkut, and Twitter, respectively. Plots d and e show weak scaling result of a type-1 RMAT graph compared with the results reported in [7]. Similarly, the best D and Δ values were searched for DSMR and Δ-Stepping from a random source vertex and

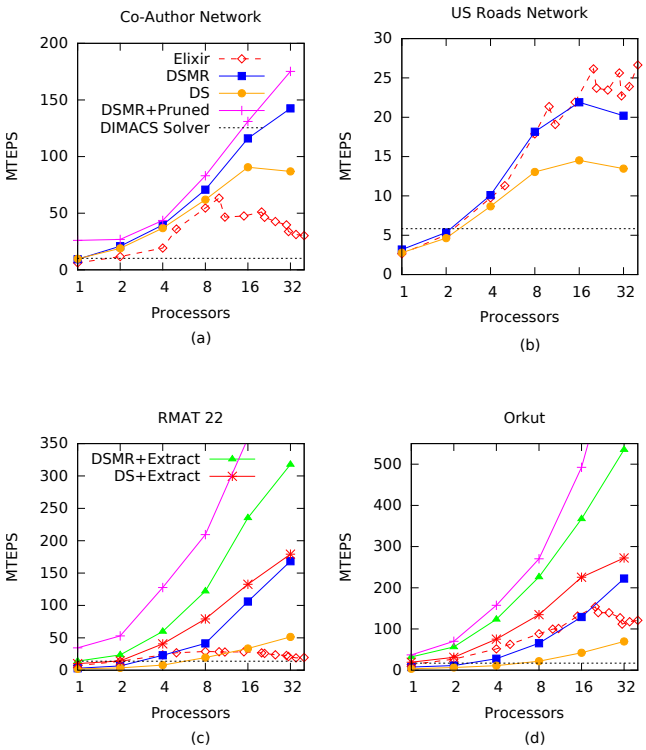


Figure 11: Evaluation of DSMR, Δ-Stepping and Elixir algorithms on the shared-memory machine. For readability, we do not show some data points for Pruned Orkut

Graph	DSMR			Δ-Stepping			*HE* Extraction		Pruned
	D	OH	Syncs	Δ	OH	Syncs	TH	$\|G'\|/\|G\|$	
Shared-Memory Results									
Co-Author	2^9	19%	38	2^7	23%	93	N/A		21.5%
US Roads	2^5	220%	28831	2^6	221%	47391	N/A		0%
RMAT22	2^{12}	5%	262	2^2	5%	556	44	0.17	89.5%
Orkut	2^{14}	4%	120	2^3	4%	187	40	0.17	88%
Distributed-Memory Results									
RMAT26	2^{12}	11%	45	2^5	40%	153	44	0.17	90.5%
Orkut	2^{12}	40%	19	2^7	101%	33	40	0.17	87.3%
Twitter	2^{14}	14%	28	2^6	34%	93	26	0.10	91.8%
Weak	2^{16}	11%	27	2^5	15%	79	32	0.125	97.1%

Table 3: Details of the performance evaluation in Figure 11 and 12. **OH:** Overhead, **Syncs:** Synchronizations, **TH:** Threshold.

used for 100 different random source vertices for our experiments. The distributed-memory part of Table 3 shows data for the maximum number of processors that DSMR scales: 4096 for RMAT26, 2048 for Orkut, 4096 for Twitter and 8192 for the weak scaling results.

First, consider the strong scaling results in plots a, b and c in Figure 12. As was the case for shared-memory, DSMR scales better than our Δ-Stepping. It is better by a factor of 2.05 in plot a, 1.60 in plot b and 1.78 in plot c. Unlike the shared-memory results, as Table 3 shows, the overhead of DSMR is noticeably less than that of Δ-Stepping (2.5 times less on average). DSMR also has significantly fewer synchronizations than Δ-Stepping. While the overhead of DSMR and Δ-Stepping are similar in shared memory, in distributed memory the overhead of Δ-Stepping is significantly larger. The reason is that the communication cost in distributed-memory machines is high and the value of Δ that obtains the best performance reduces the number of synchronizations. However, it does that at the expense of doing useless work. This explains why DSMR performs better than Δ-Stepping.

Now, consider the subgraph extraction optimization results for plots a, b and c in Figure 12. As in the shared-memory results, this technique improves the performance of DSMR significantly: up-to a factor of 2.90 in RMAT26, 4.03 in Orkut and 3.02 in Twitter. Table 3 shows the thresholds used for the subgraph extraction with the HE property. DSMR+Extract scales better than DS+Extract. This shows the impact of DSMR on performance, in spite of the small ratio of G' over G. Finally, consider the pruned results in Figure 12. As in the shared-memory results, pruning improves the performance of the algorithm significantly since many edges are identified as useless by the pruning algorithm, as Table 3 shows. Only one iteration of the pruning algorithm was executed for all three graphs in Figure 12. The speed ups of DSMR+Pruned over DSMR are up-to : 5.46× for RMAT26, 6.33× for Orkut, and 5.59× for Twitter.

Plots d and e in Figure 12 show weak scaling results for type-1 RMAT graphs. The RMAT *scale* is $17 + k$ for 2^k processors. Plot d compares the performance per processor in MTEPS for DSMR, our Δ-Stepping (DS) and the version of Δ-Stepping described in [7] (IPDPS-DS). This is a descending plot since the communication costs increase with the number of processors. As before, the ratio of DSMR over our Δ-Stepping increases with the number of processors and DSMR runs up-to a factor of 1.37 faster. On the the other hand, IPDPS-DS is faster than DSMR with 1024 processors (1.04×) but the decreasing slope of IPDPS-DS is faster than that of DSMR, which makes it 1.66× slower than DSMR for 8192 processors.

Plot e in Figure 12 compares the absolute performance of DSMR with and without the optimizations. The subgraph extraction for this plot includes the graph extraction discussed in Section 4, and it improves the performance of DSMR by up-to 4.76×. Lastly, as it can be seen from the last row of Table 3, pruning removes around 97% of the edges from type-1 RMAT graphs. Consequently, DSMR+Pruned in Figure 12 provides a speed up of up-to 13× over DSMR. Authors of [7] applied a set of optimizations to their implementation of Δ-Stepping. IPDPS-OPT in Plot e in Figure 12 shows their best result. As it can be seen, our pruning results improve upon IPDPS-OPT by factors between

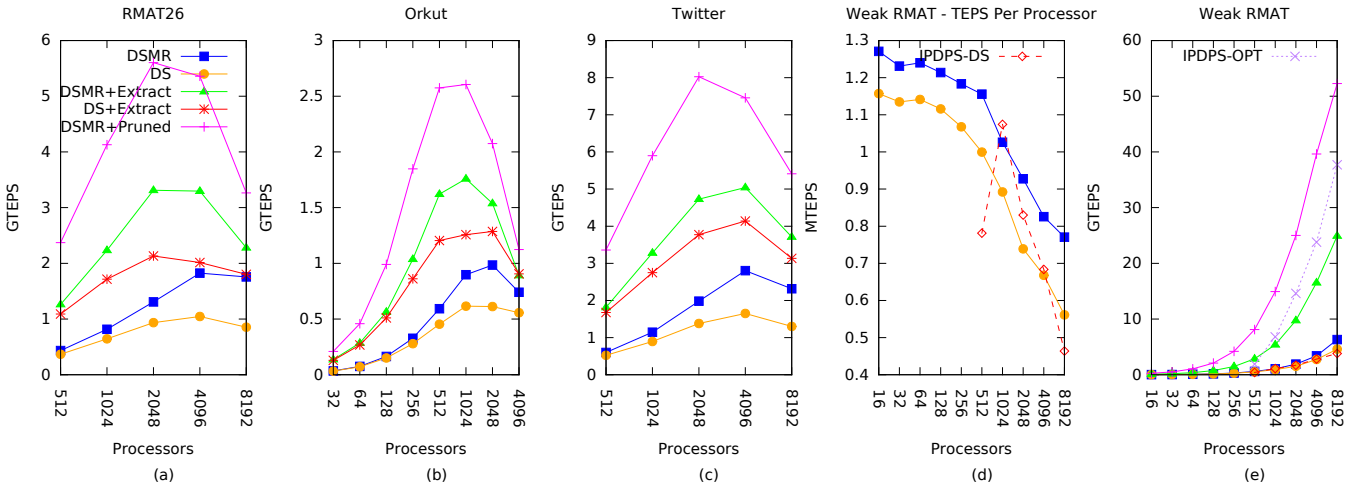


Figure 12: Performance comparison of Δ-Stepping, DSMR, Graph Extraction Optimization and Pruning. Plots a, b and c show strong scaling results, e show weak scaling results for RMAT graphs. Plot d is the same as plot e divided by the number of processors.

$1.38 - 4.26$.

8. RELATED WORK

Multiple algorithms for SSSP problem have been developed. Bellman-Ford [4], Chaotic Relaxation [9], Dijkstra [13], and Δ-Stepping [35] have been discussed through this paper. We have also shown that Δ-Stepping does not perform well for scale-free networks.

There are multiple implementations of Δ-Stepping available. Chakaravarthy et al. [7] studied parallelization of SSSP on large clusters. In this paper, we compare our results with theirs using the values reported in [7]. To make an accurate comparison, we used the same graph generation algorithm (through private communication with the authors) they used and the same machine. SSSP from the Elixir [39] benchmark is a shared-memory implementation of Δ-Stepping that we have run and compared with. SSSP from Parallel Boost Graph Library [14] is an implementation of Dijkstra and Δ-Stepping for distributed-memory systems. We found PBGL slower than the implementations considered in this paper and because of that we do not show results for it. There is also an implementation of Δ-Stepping on Cray MTA-2 by Madduri and Bader [33]. However, since the code was written for that machine, we could not do a comparison. Finally, there are implementations of Chaotic Relaxation in CombBLAS, GraphLab and PowerGraph [6, 32, 22] but this algorithm performs too many unnecessary relaxations [34].

There are a number of algorithms that use preprocessing techniques to speed up the process of finding the shortest path between a pair of vertices. However, all of these approaches are for road type of graphs and they require auxiliary information. Arc-flag [30] is a well-known technique which assigns a flag (label) to each arc and for each shortest path computation, it smartly only searches a subset of the graph. The strategy in [25] projects the vertices into an Euclidean space and assigns an attribute to the vertices to avoid unnecessary searches. The algorithm discussed in [21] also introduces shortcuts (extra edges) to improve the performance of point-to-point shortest path computation. These approaches are very effective for road types of network and point-to-point shortest path problem. However, these techniques cannot be used for the problem studied in this paper, computation of the shortest path in scale-free networks, as scale-free networks do not have the same characteristics as road networks. Finally, our preprocessing approaches introduce almost negligible space overhead. Pruning reduces significantly the size of the graph, while the other existing techniques require auxiliary information.

The SSSP algorithm in PHAST [11] includes a preprocessing phase similar to DSMR. PHAST is faster than Dijkstra's algorithm for graphs with a low highway dimension [18] (road type of networks). In this algorithm, in a preprocessing phase, vertices are selected in an order and for each selected vertex v and for all pairs of neighbor vertices of v, say x and y, an edge with weight $w(xv) + w(vy)$ is added to the graph. The authors show that doing so enables performing SSSP in two phases, where the first phase is very short and the second phase has a significant amount of parallelism, arguably more than Δ-Stepping. PHAST also explores the coarse grain parallelism that is across multiple sources. Authors report performance improvement of PHAST on GPUs and shared-memory systems. An important difference of PHAST with DSMR is that our algorithm works on a large distributed-memory system as well as on a shared-memory system. Also, the focus of DSMR is scale-free networks while PHAST is designed for road type of networks. Moreover, DSMR, unlike PHAST, does not explore the coarse grain parallelism across multiples source due to space limitation of large graphs. DSMR without any of the preprocessing engines is already a well-performing parallel algorithm while PHAST requires preprocessing. Finally, our preprocessing techniques do not introduce space overhead as opposed to the PHAST's preprocessing.

9. CONCLUSIONS

In this paper, we introduced DSMR, a new SSSP algorithm. We discuss why it performs better than Δ-Stepping on scale-free networks. Our results show that, on a shared-memory system, DSMR is faster than our own implementation of Δ-Stepping in all cases and only slower than Elixir

Δ-Stepping in the case of US Roads network (25% slower). However, DSMR is faster than Elixir Δ-Stepping on all the other graphs by up-to 7.38×. For distributed-memory systems, DSMR is faster than our Δ-Stepping implementation by up-to 2.05× and by up-to 1.66× faster than the best existing SSSP algorithm for distributed-memory systems. We also introduced subgraph extraction and pruning techniques, which improved performance by up-to 4.76× and 13×, respectively.

10. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS 1111407. The authors thank the Argonne Leadership Computing Facility at Argonne National Lab for providing computation time on the Mira cluster.

11. REFERENCES

[1] 9th dimacs implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/.

[2] David A. Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing*, HiPC'05, pages 465–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[4] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[5] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–14, March 2007.

[6] Aydin Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011.

[7] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 889–901, May 2014.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Fourth SIAM International Conference on Data Mining*, April 2004.

[9] D. Chazan and W. Miranker. Chaotic Relaxation. *Linear Algebra and Its Applications'69*, 2(7):199–222, 1969.

[10] Reuven Cohen and Shlomo Havlin. Scale-free networks are ultrasmall. *Phys. Rev. Lett.*, 90:058701, Feb 2003.

[11] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. Phast: Hardware-accelerated shortest path trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 921–931, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. Implementation challenge for shortest paths. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 1–99. Springer US, 2008.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[14] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine. Single-source shortest paths with the parallel boost graph library.

[15] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[16] Linton C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, March 1977.

[17] Galois system. http://iss.ices.utexas.edu/?p=projects/galois.

[18] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Andrew Goldberg. Shortest path algorithms: Engineering aspects. In *In Proc. ESAAC âĂŹ01, Lecture Notes in Computer Science*, pages 502–513. Springer-Verlag, 2001.

[20] Andrew V. Goldberg. A practical shortest path algorithm with linear expected time. *SIAM J. Comput.*, 37(5):1637–1655, February 2008.

[21] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *IN WORKSHOP ON ALGORITHM ENGINEERING & EXPERIMENTS*, pages 129–143, 2006.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[23] Graph 500. http://www.graph500.org/.

[24] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[25] Ron Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 100–111. SIAM, 2004.

[26] IBM XL C/C++ Compiler for Blue Gene/Q. http://www-03.ibm.com/software/products/en/xlcc+forbluegene.

[27] Intel C/C++ Compiler. http://software.intel.com/en-us/c-compilers.

[28] Valdis Krebs. Mapping networks of terrorist cells. *CONNECTIONS*, 24(3):43–52, 2002.

[29] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[30] Ekkehard KÃűhler, Rolf H. MÃűhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In *IN: 9TH DIMACS IMPLEMENTATION CHALLENGE [29*, 2006.

[31] F. Liljeros, C. R. Edling, L. A. N. Amaral, H. E. Stanley, and Y. Åberg. The Web of Human Sexual Contacts. *Nature*, 411:907–908, 2001.

[32] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[33] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 23–35, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[34] Saeed Maleki, G. Carl Evans, and David A. Padua. *Languages and Compilers for Parallel Computing: 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers*, chapter Tiled Linear Algebra a System for Parallel Graph Algorithms, pages 116–130. Springer International Publishing, Cham, 2015.

[35] U. Meyer and P. Sanders. Delta-stepping: A parallelizable shortest path algorithm. *J. Algorithms'03*, 49(1):114–152, October 2003.

[36] Ulrich Meyer. Average-case complexity of single-source shortest-paths algorithms: Lower and upper bounds. *J. Algorithms*, 48(1):91–134, August 2003.

[37] MPICH: High-Performance Portable MPI. http://www.mpich.org/.

[38] Gergely Palla, IllÃl's J Farkas, PÃl'ter Pollner, Imre DerÃl'nyi, and TamÃąs Vicsek. Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics*, 10(12):123026, 2008.

[39] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A system for synthesizing concurrent graph programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '12, 2012.

[40] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.