Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems

Venkatesan T. Chakaravarthy, Fabio Checconi, Prakash Murali, Fabrizio Petrini, and Yogish Sabharwal

Abstract—We consider the single-source shortest path (SSSP) problem: given an undirected graph with integer edge weights and a source vertex v, find the shortest paths from v to all other vertices. In this paper, we introduce a novel parallel algorithm, derived from the Bellman-Ford and Delta-stepping algorithms. We employ various pruning techniques, such as edge classification and direction-optimization, to dramatically reduce inter-node communication traffic, and we propose load balancing strategies to handle higher-degree vertices. These techniques are particularly effective on power-law graphs, as demonstrated by our extensive performance analysis. In the largest tested configuration, an R-MAT graph with 2^{38} vertices and 2^{42} edges on 32,768 Blue Gene/Q nodes, we have achieved a processing rate of three Trillion Edges Per Second (TTEPS), a four orders of magnitude improvement over the best published results.

Index Terms—Shortest path, parallel algorithm, delta stepping, graph 500 benchmark, distributed system

1 Introduction

The past decade has seen an exponential increase of data produced by online social networks, blogs, and micro-blogging tools. Many of these data sources are best modeled as graphs that can be analyzed to discover sociological processes and their temporal evolution through properties of the underlying edges. A growing number of applications work with web-scale graphs having billions of vertices and edges.

In this paper, we focus on developing scalable algorithms for the Single Source Shortest Path (SSSP) problem over large-scale graphs on massively parallel distributed systems. In addition to applications in combinatorial optimization (such as VLSI design and transportation), shortest path algorithms are increasingly relevant in complex network analysis [1].

Basic Algorithms. The proposed SSSP algorithm inherits important design aspects of three existing algorithms. We present a brief review of these algorithms in this section and defer a more detailed discussion on additional related work to a later section.

Two classical approaches for the SSSP problem are attributable to Dijkstra and the Bellman-Ford [2]. In a sequential implementation, Dijkstra's algorithm is efficient in terms of processing speed since it runs in time linear in the number of edges. However, the algorithm requires many phases (i.e., iterations) and it is less amenable to parallelization. In contrast, Bellman-Ford involves fewer phases and each phase is highly parallelizable. However, the algorithm may process each edge multiple times and is likely to incur high processing time.

Meyer and Sanders [3] proposed the Δ-stepping algorithm, a trade-off between the two extremes of Dijkstra's and Bellman-Ford. The algorithm involves a tunable parameter Δ. Setting $\Delta = \infty$ (a sufficiently large number) yields the Bellman-Ford algorithm. On the other hand, Dijkstra's algorithm is equivalent to setting $\Delta = 1$. By varying Δ in the range $[1, \infty]$, we get a spectrum of algorithms with varying degrees of processing time and parallelism.

Graph 500 Benchmark. The Graph 500 list[1] was introduced in 2010 as an alternative to the Top500 list to rank computer performance based on data-intensive computing applications. The current version of the benchmark includes Breadth First Search (BFS), with SSSP under evaluation. The benchmark uses power-law graphs generated using the Recursive MATrix (R-MAT) model [4], [5]. The performance of a Graph 500 implementation is typically measured in terms of Traversed Edges Per Second (TEPS), computed as m/t, where m is the number of edges in the input graph and t is the time taken in seconds. Graph 500 captures many essential features of data intensive applications, and has raised a lot of interest in the supercomputing community at large, both from the scientific and operational points of view. In fact, many supercomputer procurements are now including the Graph 500 in their collection of benchmarks.

Contributions. The paper provides several important contributions. We describe a new scalable SSSP algorithm for large-scale distributed-memory systems. The algorithm is obtained by augmenting the Δ-stepping algorithm with three important optimizations.

(a) Hybridization: We observe that the number of iterative steps performed in the Δ-stepping algorithm can be significantly reduced by running the algorithm only for its initial phases, then switching to the Bellman-Ford algorithm.

- *V.T. Chakaravarthy, P. Murali, and Y. Sabharwal are with IBM Research, Vasant Kunj, New Delhi 110070, India. E-mail: {vechakra, prakmura, ysabharwal}@in.ibm.com.*
- *F. Checconi is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. E-mail: fchecco@us.ibm.com.*
- *F. Petrini was with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. He is now with the Intel Parallel Computing Labs, Santa Clara, CA 95054-1549. E-mail: fabrizio.petrini@intel.com.*

Manuscript received 10 Feb. 2016; revised 8 Nov. 2016; accepted 10 Nov. 2016. Date of publication 1 Dec. 2016; date of current version 14 June 2017. Recommended for acceptance by U. V. Catalyurek.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2634535

1. http://www.graph500.org

Reference	Problem	Graph Type	Vertices	Edges	GTEPS	Processors	System
Bader, Madduri [6]	BFS	R-MAT	200 M	1 B	0.5	40	Cray MTA-2
Checconi et al. [7]	BFS	Graph 500	2^{32}	2^{36}	254	4,096 nodes/65,536 cores	IBM Blue Gene/Q (NNSA SC)
Graph 500 Nov 2013 [8]	BFS	Graph 500	2^{36}	2^{40}	1,427	4,096 nodes/65,536 cores	IBM Blue Gene/Q (Mira)
Graph 500 Nov 2013 [8]	BFS	Graph 500	2^{39}	2^{43}	14,328	32,768 nodes/524,288 cores	IBM Blue Gene/Q (Mira)
Graph 500 Nov 2015 [8]	BFS	Graph 500	2^{41}	2^{45}	23,751	98,304 nodes/1,572,864 cores	IBM Blue Gene/Q (Sequoia)
Graph 500 Nov 2015 [8]	BFS	Graph 500	2^{40}	2^{44}	38,621	82,944 nodes/663,552 cores	Fujitsu (K Computer)
Madduri et al. [9]	SSSP	R-MAT	2^{28}	2^{30}	0.1	40 nodes	Cray MTA-2
This paper	SSSP	R-MAT	2^{35}	2^{39}	650	4096 nodes/65,536 cores	IBM Blue Gene/Q (Mira)
This paper	SSSP	R-MAT	2^{38}	2^{42}	3100	32,768 nodes/524,288 cores	IBM Blue Gene/Q (Mira)

Fig. 1. Performance comparison.

(b) Pruning: We introduce a new model of direction optimization [6], in combination with edge classification, that relaxes only a fraction of the input edges and avoids redundant relaxations. In practical settings, the number of relaxations performed by the algorithm is significantly smaller than that of Dijkstra's (which relaxes all the edges). The model involves two different types of relaxation mechanisms called push and pull, and we have designed a near-optimal heuristic for determining the mechanism to be employed in different phases of the algorithm. The dramatic reduction in the number of relaxations obtained by the above techniques leads to reduced communication volume and processing time.

(c) Load Balancing: We observe that in very large graphs, several billions of vertices and beyond, performance tends to be affected by load imbalance, especially when processing vertices with high degree. We alleviate this issue by employing a two-tiered load balancing strategy. First, the neighborhood of heavy degree vertices is split across the processing nodes, and then, within a processing node, the load is evenly distributed among threads.

We validate the algorithm with an extensive performance evaluation on power-law graphs and a preliminary study involving real world graphs. In the largest graph configuration, a scale-38 R-MAT graph with 2^{38} vertices and 2^{42} undirected edges explored on 32,768 Blue Gene/Q nodes, we have obtained a processing rate of 3,000 billion TEPS (GTEPS).

We are not aware of any prior study on the SSSP problem involving graphs of such scale, system size and processing rate. Madduri et al. [7] presented an experimental evaluation of the Δ-stepping algorithm on graphs of size up to 2^{30} edges on 40 MTA-2 Cray nodes, achieving a performance number of about 100 million TEPS (MTEPS), approximately four orders of magnitude smaller than our results.[2] To understand the performance in a better context, we compare our results against some of the available performance figures for the closely related Breadth First Search problem. We note that while BFS shares certain common features with SSSP, BFS is a much simpler problem from a computational point of view, as discussed in the Related Work section below. Fig. 1 shows the performance results available in the literature and the latest Graph 500 submissions. It is worth noting that SSSP is only two to five times slower than BFS on the same machine configuration, graph type and level of optimization. This is a very promising result that proves that BFS levels of performance can also be achieved by more complex graph algorithms.

2. This an estimated upper bound derived from the experimental evaluation in [7].

Related Work. Considerable attention has been devoted to solving SSSP in sequential as well as parallel settings. The classical Dijkstra's algorithm is the most popular algorithm in the sequential world. The algorithm can be implemented in $O(n \log n + m)$ time using Fibonacci heaps [8]. There have been attempts to develop parallel versions of Dijkstra's algorithm and its variants (for example, in Parallel Boost Graph Library [9]). Efforts have also been made to parallelize the algorithm using transactional memory and helper threads [10] but with very modest speedups. Parallel implementations of the Bellman-Ford algorithm and its variants are also available (for example, using MapReduce framework [11]). However, these methods do not provide good scalability.

A body of prior work has dealt with designing parallel algorithms for SSSP from a theoretical perspective under the PRAM model. Under this model, the performance is measured in terms of parallel running time and work done (product of the number of processors and running time). Prior work present algorithms with different tradeoffs on the two parameters (e.g., [12], [13], [14]). The most relevant work in the context of this paper is due to Meyers and Sanders [3], who presented an algorithm that does well in the average-case on random graphs. For random directed graphs with edge probability d/n and uniformly distributed edge weights, they presented a PRAM version that runs in expected time $O(\log^3 n / \log\log n)$ using linear work. The algorithm can also be easily adapted for distributed memory systems. Madduri et al. showed that this algorithm scales well on the massively multi-threaded shared memory Cray system [7].

While our paper is particularly catered towards power-law graphs, prior work has considered other classes of graphs. These include graphs having geometric structure [15], [16] such as those arising in VLSI design. Another well-studied family is the class of road networks where hierarchical decomposition techniques have been shown to be effective [17]. For instance, PHAST [18] presents efficient algorithms for SSSP on multi-core systems and GPUs for the case of road networks.

Breadth First Search is a well-studied problem which is closely related to SSSP. However, a crucial aspect makes BFS computationally simpler than SSSP. In BFS, a vertex can be "settled" and added to the BFS tree the first time it is reached, whereas in SSSP the distance of a vertex may be revised multiple times. There has been considerable effort on developing parallel implementations of BFS on massive graphs on a variety of parallel settings: for example, shared memory architectures [6], [19], GPUs [20] and massively parallel distributed memory machines [21]. The best BFS implementation of the Graph 500 benchmark achieves 38,621 GTEPS over a graph of size 2^{40} vertices and 2^{44} edges on K Computer, a Fujitsu distributed memory system having 82,944 nodes and 663,552 cores.

2 Basic Algorithms

In the SSSP problem, the input consists of a weighted undirected graph $G = (V, E, w)$ over n vertices and m edges. The weight function w assigns an integer weight $w(e) \geq 0$ to each edge. The input also specifies a vertex $\texttt{rt} \in V$ called the root. The goal is to compute the shortest distance from rt to every vertex $v \in V$. For a vertex v, let $d^*(v)$ denote the shortest distance. *Scale* of a graph is defined as $\log_2 N$.

Our algorithm for SSSP builds on three well-known algorithms, which we refer to as the basic algorithms: Dijkstra's algorithm, the Bellman-Ford algorithm (see [2]) and the Δ-stepping algorithm [3]. Here, we will first introduce these algorithms and then analyze their characteristics. Based on the intuition derived from the above analysis, we will develop improved algorithms in the following section.

2.1 Basic Algorithms: Description

All the three algorithms maintain a *tentative distance* $d(v)$, for each vertex $v \in V$. At any stage of the algorithm, the value $d(v)$ is guaranteed to be an upper bound on the actual shortest distance $d^*(v)$. The tentative distance of the root vertex is initialized to 0 and for all the other vertices to ∞. As the algorithm proceeds, the tentative distance $d(v)$ is monotonically decreased. The algorithms guarantee that, at the end of the process, $d(v)$ matches the actual shortest distance $d^*(v)$. The tentative distances are modified by an operation called *relaxation* of an edge. When we reduce the tentative distance of a vertex u, we can possibly reduce the tentative distance of its neighbors as well. Given an edge $e = \langle u, v \rangle$, the operation $\texttt{Relax}(u, v)$ is defined as follows:

$$d(v) \leftarrow \min\{d(v), d(u) + w(\langle u, v \rangle)\}.$$

At any stage of the algorithm, we say that a vertex v is *settled*, if the algorithm can guarantee that $d(v) = d^*(v)$.

Dijkstra's Algorithm. The algorithm begins by declaring all the vertices to be unsettled and proceeds in multiple phases. In each phase, the unsettled vertex u having the minimum tentative distance is selected. We call u as the *active vertex* of this phase and declare u to be settled. Then, for each neighbor v of u given by an edge $e = \langle u, v \rangle$, the operation $\texttt{Relax}(u, v)$ is performed. The algorithm terminates when there are no more unsettled vertices.

Bellman-Ford Algorithm. Dijkstra's algorithm selects only one active vertex in any phase, whereas the Bellman-Ford algorithm selects multiple active vertices. The algorithm proceeds in multiple phases. In each phase, we declare a vertex u to be active, if its tentative distance $d(u)$ changed in the previous phase. For each such active vertex u, consider all its incident edges $e = \langle u, v \rangle$ and perform $\texttt{Relax}(u, v)$. The process terminates when there are no active vertices at the beginning of an phase.

Δ-Stepping Algorithm. Dijkstra's and the Bellman-Ford algorithms employ two contrasting strategies for selecting active vertices in each phase. The former chooses only one vertex (which is guaranteed to be settled), whereas the latter activates any vertex whose tentative distance was reduced in the previous phase. The Δ-stepping algorithm strikes a balance between these two extremes.

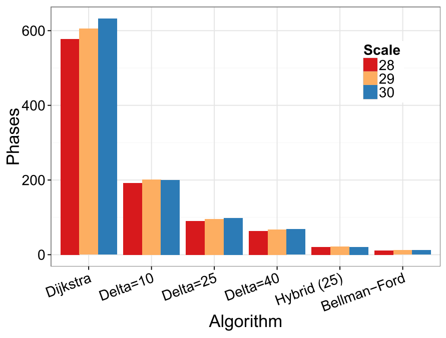
```
Initialization
  Set $d(\texttt{rt}) \leftarrow 0$; for all $v \neq \texttt{rt}$, set $d(v) \leftarrow \infty$.
  Set $B_0 \leftarrow \{\texttt{rt}\}$ and $B_\infty \leftarrow V - \{\texttt{rt}\}$.
  For $k = 1, 2, \ldots,$, set $B_k \leftarrow \emptyset$.
Δ-Stepping Algorithm
  $k \leftarrow 0$.
  Loop      // Epochs
    ProcessBucket($k$)
    Next bucket index : $k \leftarrow \min\{i > k : B_i \neq \emptyset\}$.
    Terminate the loop, if $k = \infty$.
ProcessBucket($k$)
  $A \leftarrow B_k$.      //active vertices
  While $A \neq \emptyset$      //phases
    For each $u \in A$ and for each edge $e = \langle u, v \rangle$
      Do $\texttt{Relax}(u, v)$
    $A' \leftarrow \{x : d(x)$ changed in the previous step$\}$
    $A \leftarrow B_k \cap A'$
Relax($u, v$):
  Old bucket: $i \leftarrow \lfloor \frac{d(v)}{\Delta} \rfloor$.
  $d(v) \leftarrow \min\{d(v), d(u) + w(\langle u, v \rangle)\}$.
  New bucket : $j \leftarrow \lfloor \frac{d(v)}{\Delta} \rfloor$.
  If $j < i$, move $v$ from $B_i$ to $B_j$.
```

Fig. 2. Δ-stepping algorithm.

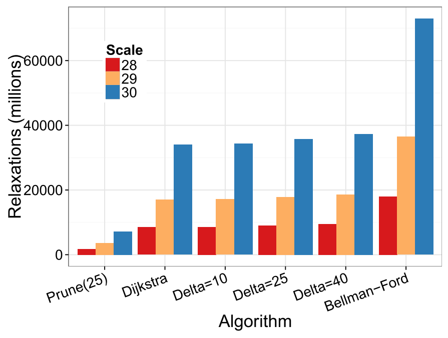
Fix an integer constant $\Delta \geq 1$. We partition the vertices into multiple *buckets*, based on their tentative distance. For an integer $k \geq 0$, the bucket B_k would consist of vertices v whose tentative distance falls in the range $[k\Delta, (k+1)\Delta - 1]$. The bucket index for a vertex v is given by $\lfloor \frac{d(v)}{\Delta} \rfloor$. Initially, the root vertex rt is placed in the bucket B_0 and all the other vertices are placed in the bucket B_∞. The algorithm works in multiple epochs. The goal of epoch k is to settle all the vertices whose actual shortest distance falls in the range of bucket k. The epoch works in multiple phases. In each phase, a vertex u is declared to be active if its tentative distance changed in the previous phase and the vertex is found in the bucket B_k (in the first phase of the epoch, all vertices found in the bucket are considered active). For each active vertex u and all its incident edges $e = \langle u, v \rangle$, we perform $\texttt{Relax}(u, v)$. When the bucket does not contain any active vertices, the epoch terminates and we proceed to the next non-empty bucket. The algorithm terminates when B_∞ is the only non-empty bucket of index higher than k. Notice that during a relax operation, it may happen that the tentative distance of a vertex reduces in a such a manner that the new value falls in the range of a bucket of lower index. In such a case, the vertex is moved from its current bucket to the new bucket. We will treat the above movement as a step within the relax process. The pseudocode for the algorithm is presented in Fig. 2.

Setting $\Delta = 1$ yields a variant of the Dijkstra's algorithm (due to Dial et al. [22]), whereas setting $\Delta = \infty$ yields the Bellman-Ford algorithm. In the rest of the paper, we will analyze Dijkstra's algorithm as Δ-stepping with $\Delta = 1$.

Distributed Implementation. Our distributed implementation of the Δ-stepping algorithm is briefly outlined below. We use the $1D$ distribution strategy: the vertices are partitioned among the processors, with each vertex being owned by some processor. The vertex ids are scrambled in a preprocessing step so that each processor owns a random set of vertices. The scrambling helps in load balancing. A processor would execute only the instructions of the algorithm



(a) Number of phases



(b) Number of relaxations

Fig. 3. Comparison of different algorithms.

that are pertinent to its vertices. Relax operations require communication among the processors: to perform Relax(u, v), the owner of the source vertex u will send $d(u)$ to the owner of the destination vertex v (if the two owners are distinct). For each vertex v, the owner performs all the updates received. The phases and epochs are executed in a bulk synchronous manner. Termination checks and computing the next bucket index require *Allreduce* operations.

An alternative strategy is the $2D$ distribution that distributes the edges among the processors, as against vertices. A prior work on the related breadth first search problem [23] presents a brief experimental comparison the two strategies on power-law graphs, wherein the results suggest that $1D$ offers better performance than $2D$. While our choice of $1D$ distribution is motivated by the above result, we note that a detailed study comparing the two strategies in the context of SSSP would be an interesting avenue of research.

2.2 Characteristics of the Basic Algorithms

In this section, we characterize and compare the three basic algorithms with respect to two key metrics.

Number of Relaxation. This metric not only determines a major portion of the processing time, but also the communication time in a parallel setting (since a relax operation typically involves communication between the owners of the endpoints of the relaxed edge). We shall use the term "work done" to refer to the number of relaxations.

Number of Buckets/Phases. The number of *phases* (i.e., iterations) taken by the algorithm is another important metric. Each phase is associated with overheads such as determining whether the algorithm can be terminated and testing whether we can proceed to the next bucket. These operations need bulk synchronization and communication among the processors. Furthermore, dividing the work across more phases tends to increase the load imbalance among the processors. A similar reasoning applies to the case of buckets as well. Consequently, minimizing the number of phases and buckets is beneficial.

Dijkstra's algorithm is very efficient in terms of work done. Each vertex is made active only once and so, each edge $\langle u, v\rangle$ is relaxed only twice (once along each direction (Relax(u, v) and Relax(v, u)). Hence, the total number of relaxations is $2m$. On the other hand, in the Bellman-Ford algorithm, a vertex may be made active multiple times and as a result, an edge may be relaxed several times. Therefore, Dijkstra's algorithm is better than the Bellman-Ford algorithm in terms of work-done.

Regarding the number of phases, Dijkstra's algorithm (implemented as Δ-stepping with $\Delta = 1$) settles only the vertices having the smallest tentative distance in each phase. As a result, the number of phases is exactly the number distinct shortest distances in the final output. For the Bellman-Ford algorithm, it can be shown that the number of phases is at most the depth of the shortest path tree. Even though both the above quantities can be as high as the number of vertices, the latter quantity is typically much smaller in practice. Hence, the Bellman-Ford algorithm is better than Dijkstra's algorithm , in terms of the number of phases.

The Δ-stepping algorithm offers a trade-off between the two algorithms, as determined by the parameter Δ. The three algorithms are related as follows:

- Work-done: Dijkstra $\leq$ Δ-stepping $\leq$ Bellman-Ford.
- # phases: Bellman-Ford $\leq$ Δ-stepping $\leq$ Dijkstra.

Fig. 3 illustrates the above relationships on a sample power-law graphs used in our experimental study. Three representative Δ values 10, 25 and 40 are included (the statistics pertain to a refined version of Δ-stepping discussed in Section 3.1). The figure also includes statistics for two other algorithms, called Hybrid and Prune, discussed later in the paper. We can observe the tradeoff offered by Δ-stepping on both the metrics.

3 Our Algorithm

Our algorithm builds on the three basic algorithms and obtains improved performance by employing three classes of optimizations: (i) Pruning; (ii) Hybridization; (iii) Load balancing. The first strategy reduces the number of relaxations, whereas the second reduces the number of phases/buckets. The third optimization improves load sharing among the processors at high scales.

While discussing the proposed optimizations, we utilize sample Graph500 RMAT graphs used in the experimental study for the purpose of motivation and illustration. The RMAT process generates sparse graphs having marked skew in the degree distribution, where a sizeable fraction of the vertices exhibit very high degree. These vertices induce a dense sub-graph and tend to have smaller shortest distances. In contrast, the vertices of low degree tend to have larger shortest distances.

3.1 Edge Classification

The pruning heuristic utilizes the concept of edge classification, introduced by Meyer and Sanders [3], as a refinement of their Δ-stepping algorithm. We first describe the concept and then, propose an improvement.

Meyer and Sanders [3] classify the edges into two groups: *long* and *short*. An edge e is said to be *short* if $w(e) < \Delta$, and it is said to be *long* otherwise. Suppose we are processing a bucket and let u be an active vertex with an incident edge $e = \langle u, v\rangle$ such that v belongs to a bucket of higher index. When the edge is relaxed, the tentative distance of v may decrease. However, observe that if e is a long edge, the decrease would not be sufficient to cause v to move to the current bucket. Therefore, it is redundant to relax a long edge in every phase of the current bucket; instead, it is enough to relax long edges once at the end of the epoch. Intuitively, cross-bucket long edges are irrelevant while processing the current bucket, because we are settling only the vertices present in the current bucket (including the ones that move into the bucket while the epoch is in progress).

Based on this observation, the algorithm is modified as follows. The processing of each bucket is split into two stages. The first stage involves multiple *short edge phases*, wherein only the short edges incident on the active vertices are relaxed. Once all the vertices in the current bucket are settled and no more active vertices are found, we proceed to the second stage, called the *long-edge phase*. In this phase, all the long edges incident on the vertices in the current bucket are relaxed. Thus the long edges get relaxed only once.

We improve the above idea by additionally classifying the short edges into two groups, as discussed below. Consider the first stage of processing a bucket B_k and let u be an active vertex with an incident short edge $e = \langle u, v\rangle$ such that v belongs to a bucket of higher index. Notice that when the edge is relaxed, the vertex v can potentially move to the current bucket only if the tentative distance $d'(v) = d(u) + w(e)$ falls within the current bucket (i.e., $d'(v) \le (k+1)\Delta - 1$). Thus, if the above criterion is not satisfied, we can ignore the edge in the first stage. Based on the above observation, the heuristic works as follows. Let u be an active vertex in any phase of the first stage. We relax an edge $e = \langle u, v\rangle$ only if the newly proposed tentative distance $d'(v)$ falls within the current bucket; such edges are called *inner short edges*. As before, the first stage is completed when all the vertices in the current bucket are settled and there are no more active vertices. Finally, in the long edge phase, we relax all the long edges, as well as all the short edges $e = \langle u, v\rangle$ satisfying the property that $d'(v) = d(u) + w(e)$ falls outside the current bucket range ($d'(v) \ge (k+1)\Delta$). The short edges relaxed in the long edge phase are called *outer short edges*. We call the above heuristic as the *inner-outer short* heuristic (IOS).

We can see that the IOS heuristic aims at reducing the number of short edge relaxations, leaving the number of long edge relaxations unaffected. Our experiments suggest that the number of short edge relaxations decreases by about 10 percent, on Graph 500 RMAT graphs.

3.2 Pruning: Beating Dijkstra's Algorithm

Among the algorithms discussed so far, Dijkstra's algorithm performs the best in terms of the number of relaxations: it relaxes every edge only twice, once along each direction. The Δ-stepping algorithm (equipped with edge classification) relaxes the long edges only twice and the IOS heuristic reduces the number of short edge relaxations. Consequently, the total number of relaxations performed by the above algorithm is nearly on par with that of Dijkstra's, as shown in Fig. 3b. In this section, we discuss the *pruning heuristic* which focuses on the long edges and relaxes only a fraction of such edges while ensuring correctness. The heuristic is inspired by the direction optimization technique proposed by Beamer et al. [6] in the context of BFS. In our SSSP context, the presence of weights on the edges warrants more sophistication. For the sake of clarity, we explain the heuristic with respect to the basic edge classification strategy (short and long), ignoring the refined notion of IOS.

Any epoch involves a first stage consisting of multiple short edge phases and a second stage consisting of a single long edge phase. We observe that among the two types of phases, the long edge phases tend to involve more relaxations: the intuitive reasoning is that if Δ is sufficiently small compared to the maximum edge weight $w_{\max}$, then more edges are likely to have weights in the range $[\Delta, w_{\max}]$ and fall into the long edge category. Our experimental results provide empirical validation for the above intuition.

The pruning heuristic targets the dominant long edge relaxations. Let us first discuss a natural method for implementing the long edge phases. Consider the long edge phase associated with a bucket B_k. Each processor scans all its active vertices in B_k. For each such vertex u and for each long edge $e = \langle u, v\rangle$ incident on u, the processor computes a new tentative distance for the destination vertex v given by $d'(v) = d(u) + w(e)$ and sends (pushes) the information to the processor owning v. We call this method the *push model*.

The main observation behind the prune heuristic is that a large fraction of relaxations performed by the push model is likely to be redundant, as explained next. With respect to the current bucket B_k, we classify the other buckets into two types: buckets B_i with $i < k$ are called *previous buckets* and bucket B_i with $i > k$ are called *later buckets*. Consider a vertex u present in the current bucket B_k and let $e = \langle u, v\rangle$ be a long edge incident on u. Let B_i be the bucket to which v belongs. We classify each edge into one of three categories: (i) *self edge*, if v belongs to the current bucket ($i = k$); (ii) *backward edge*, if v belongs to a previous bucket ($i < k$); (iii) *forward edge*, if v belongs to a later bucket ($i > k$). Observe that all the vertices in the previous buckets are already settled and their shortest distances are known. As a result, it is redundant to relax the self and backward edges, since such relaxations cannot result in any change in the tentative distance of v.

Illustration. Fig. 4a shows the three types of edges and the direction of communication under the push model.

A natural idea for eliminating the redundant relaxations is to determine the category of all edges, and to relax only the forward edges. However, such a method is difficult to implement in a distributed environment, because the category of an edge $\langle u, v\rangle$ is dependent on the index of the bucket to which the vertex v belongs. The issue is that the bucket index of v is known only to the owner of destination vertex v. The source may obtain the above information by communicating with the destination. However, such a strategy defeats the purpose of avoiding redundant relaxations.

The pruning heuristic overcomes the issue by employing a *pull model* of communication, in addition to the natural push model. In the pull model, the destination side owner pulls the new tentative distance from the source side owner.

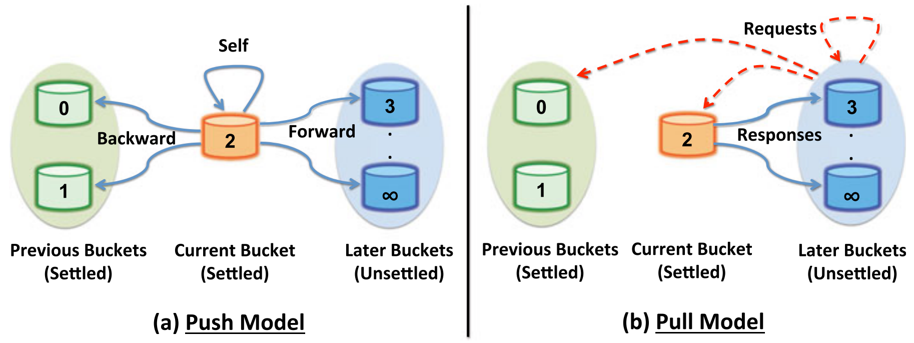


Fig. 4. Push and pull models.

Consequently, relaxing an edge $\langle u, v\rangle$ involves two communications: a *request* being sent from the owner of v to the owner of u, and a *response* in return.

The implementation of the pull model is described next. Each processor scans all the vertices owned by it and contained in the later buckets. For each such vertex v and each long edge $\langle u, v\rangle$ incident on v, a request is sent to the processor owning u. Upon receiving the request, the owner of u sends a a response containing the new distance $d'(v)$.

Illustration. We next illustrate the advantage of the pull model using a simple example graph, shown in Fig. 5. Consider running the Δ-stepping algorithm with $\Delta = 5$, using only the push model. The algorithm takes three phases (where non-zero communication takes place). Initially, the root is placed in bucket B_0 and made active. In the first phase (corresponding to the long edge phase of B_0), all the edges incident on the root are relaxed and the clique vertices move to B_2. In the second phase (corresponding to the long edge phase of B_2), all the edges incident on the clique vertices are relaxed, resulting in the isolated vertices moving to bucket B_4. Finally, in the third phase (corresponding to the long edge phase of B_4), all the edges incident on the isolated vertices are relaxed. The cost (number of relaxations) per phase is shown in the figure; the total is 40. In contrast, consider running the same algorithm, but applying the pull model in the second phase. In this case, the owners of isolated vertices send requests and the owners of clique vertices respond. For the second phase, the push model has cost 30, whereas the pull model has cost only 10.

In the above model, we can reduce both the number of requests and responses based on the observation that all the previous buckets have already been processed and the pertinent long edges are taken care of. Consequently, it suffices to send a response only when the source vertex u belongs to the current bucket B_k. Furthermore, given the above refinement, it suffices to send a request only when there is a possibility of getting a response. For an edge $e = \langle u, v\rangle$, the owner of the source vertex u sends a response only if u belongs to the current bucket B_k and the shortest distance $d(u)$ of such a vertex is at least $k\Delta$. Therefore, the newly proposed tentative distance $d'(v) = d(u) + w(e)$ would be at least $k\Delta + w(e)$. If the current tentative distance $d(v)$ happens to be smaller than the above quantity, then the relaxation is useless (since it cannot decrease the tentative distance of v). Hence, a request needs to be sent only if $d(v) > k\Delta + w(e)$ or alternatively,

$$w(e) < d(v) - k\Delta. \qquad (1)$$

Illustration. Fig. 4b illustrates the pull model, along with the refinement. Requests may be sent on a long edge whose destination vertex is in a later bucket. However, responses are sent only if the source vertex falls in the current bucket.

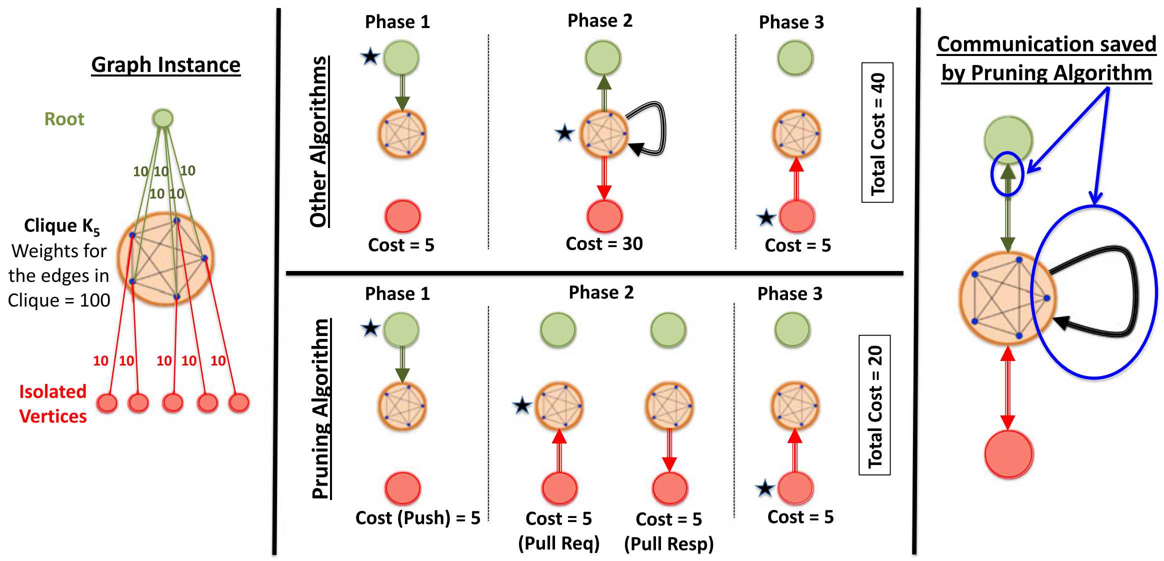


Fig. 5. Illustration of benefit of pull mechanism (the current bucket is marked with an asterix).

3.3 Push versus Pull Heuristic

While the pull model is useful, applying it for all the buckets may be lead to performance degradation. Our algorithm is based on a careful combination of the push and pull models. Namely, we use the push model for certain buckets, while applying the pull model for others. In this section, we present a comparison of the two models and discuss a heuristic for making the push-pull decision at the end of each bucket.

Intuitively, the aggregate degree of vertices present in the bucket is the main factor that determines the most suitable model for the bucket. The push model must relax all the long edges incident on the current bucket, whereas the pull model must process a subset of the long edges incident on the later buckets, as determined by equation (1). Consequently, the push model would be a better choice if the bucket contains low degree vertices; in contrast, the pull model is likely to outperform if the bucket contains high degree vertices.

The above discussion shows that applying the same model uniformly across all buckets is not a good strategy. Our algorithm works by selecting a suitable model for each bucket. The model selection is performed via a heuristic that estimates the communication volume.

Let B_k be the current bucket. For the push model, the volume is given by the total number of long edges incident on the vertices contained in the bucket B_k. Assuming that the parameter Δ is fixed *a priori*, the number of long edges for all vertices can be computed in a preprocessing stage and the volume can be determined by aggregating across the relevant vertices. Regarding the pull model, the volume is given by the number of requests and responses. For each vertex v belonging to a later bucket and any long edge e incident on v, a request is sent if $w(e) < d(v) - k\Delta$; equivalently, $w(e) \in [\Delta, d(v) - k\Delta - 1]$ (see (1)). Hence, for computing the number of requests, we need a procedure that takes as input a vertex v and a range $[a, b]$, and returns an estimate for the number of edges incident on v whose weight falls in the given range.

Multiple strategies are possible for implementing the procedure. For instance, assuming that the edge list of each vertex is sorted according to weights, the quantity can be computed via a binary search. Our implementation uses a strategy based on histograms. In a preprocesing stage, we build an histogram by scanning input graph and computing the frequency of different edge weights. Then, while computing the shortest paths from a given root, we use the histogram to estimate the number of edges belonging to a given weight range.

The number of responses is difficult to measure since it requires the knowledge of bucket indices of both the endpoints of the relevant edges. However, the number of responses can be at most the number of requests, and we have determined experimentally that this upper bound works well in practice. Using the above ideas we can derive estimates on the communication volume for both the push and the pull models. A natural heuristic is to choose the model with the lower estimated communication volume.

We evaluated the heuristic by designing an offline procedure that executes the algorithm under all possible push/pull decision choices. We found that the heuristic is highly effective and achieves near-optimal results on all the configurations considered. The details are deferred to the experimental section.

Bucket	1	2	3	4	5	6	7
Relax opr.	0.006	78211	4911	298	96	48	29
Settled vertices	0.001	1791	2071	882	536	372	279

Fig. 6. Bucket-wise distribution of relaxation operations and number of settled vertices (in thousands).

Our experimental study also showed that the pruning strategy is particularly effective on graph having higher skew in the degree distribution. The reason is that in such graphs, the high degree vertices tend to have small shortest distances to the root and settle in early buckets. Moreover, these vertices tend to form a dense cluster. The two factors lead to a large number of self-edges in the early buckets. This offers opportunities for applying the pull model and avoiding relaxations. The phenomenon gets more pronounced with increase in skew. Fig. 3 provides a sample illustration of the efficacy of the pruning strategy, wherein we can see that on the RMAT graphs considered in the figure, the the strategy offers significant reduction in the number of relaxations.

3.4 Hybridization

Our next optimization called *hybridization*, attempts to reduce the number of phases and epochs (buckets). As discussed earlier, the Δ-stepping algorithm is better in terms of number of relax operations, whereas the Bellman-Ford algorithm is better in terms of the number of phases/buckets. The hybridization method strikes a trade-off between the two. The idea is to execute the Δ-stepping algorithm for an initial set of buckets and then switch to the Bellman-Ford algorithm. The strategy is motivated by the observation that Δ-stepping may involve a large number of buckets, but most of the relax operations are concentrated in the initial set of buckets, especially in power-law graphs. The reason is that vertices having higher degree tend to have smaller shortest distances and get settled in the buckets having lower index. In contrast, the vertices having lower degree tend to have larger shortest distances and get settled in buckets of higher index. Consequently, the initial epochs dealing with high degree vertices involve more relaxations.

Exploiting the above phenomenon, we can reduce the overall number of epochs/phases by applying the Δ-stepping algorithm only for the first few buckets and then switching to the Bellman-Ford algorithm. Application of the Bellman-Ford algorithm can potentially increase the number of relaxations. However, the increase is not significant in practice, since the number of relaxations performed in the high index buckets is relatively low. The strategy is implemented as follows. If we decide to switch to the Bellman-Ford algorithm after ℓ buckets, the buckets $B_{\ell+1}$, $B_{\ell+2}, \ldots, B_\infty$ will not be processed individually. Instead, we group these buckets into a single bucket B and apply the Bellman-Ford algorithm.

An interesting issue is to design a heuristic for deciding when to make the switch. Our analysis on power-law graphs revealed that the number of relaxations in different buckets exhibits a bell-curve. Furthermore, the number vertices settled in the buckets also tend to exhibit a bell-curve and it is nearly aligned with the number of relaxations. For an illustration, Fig. 6 provides the above data for an

Scale	28	29	30	31	32
Graph 500	2.4 M	3.8 M	5.9 M	9.4 M	14.4 M
SSSP Proposal	31126	41237	54652	72158	95482

Fig. 7. Maximum degree for two graph families.

example power-law graph from our experimental study; only intial buckets are shown, with the actual number of buckets being 30. Motivated by the above observations, our heuristic is as follows: we compute the number of settled vertices after each bucket is processed and switch to Bellman-Ford once a local maxima has been observed. In the case of Fig. 6, we switch after bucket 4.

Fig. 3a illustrates the efficacy of the hybridization strategy on the sample power-law graph. In terms of number of phases, we can see that the Bellman-Ford algorithm performs the best and the hybrid strategy is nearly as good.

Hybridization can be rephrased as follows: initially, Δ is set to a fixed number (Δ-stepping stage) and then changed to ∞ (Bellman-Ford stage). A more generic strategy would dynamically adjust Δ according to bucket statistics and further investiagation of the idea would be of interest.

3.5 Load Balancing

By augmenting the Δ-stepping algorithm with the heuristics of edge classification, IOS, pruning and hybridization, we get an algorithm that we refer to as `OPT` (optimized heuristic). We implemented the algorithms on a distributed memory system, wherein each processing node is multi-threaded with shared memory. The vertices owned by a node are further distributed among the threads, so that each vertex is owned by a thread.

Our experimental evaluation of the `OPT` algorithm on RMAT graphs revealed that `OPT` scales poorly due to load imbalance. The aggregate degree of the vertices owned by a thread serves a simple metric to estimate the load on the thread. Any skew in the degree distribution leads to load imbalance among the threads. The maximum degree provides a useful yardstick for measuring the skew. Fig. 7 presents the maximum degree for two families of RMAT graphs, generated according to the Graph 500 benchmark specification and a proposed SSSP benchmark [24] (see experimental analysis for more details). The average degree for all graphs was fixed to be a constant (32 edges). We can see that the maximum degree is very high and increases dramatically especially for the first family, leading to load imbalance.

In order to overcome the issue, we devised an intra-node thread-level load balancing strategy. We classify the vertices into two groups based on their degree. A suitable threshold π is chosen and all vertices having degree higher than π are declared *heavy*; the other vertices are called *light*. We retain the concept of ownership for both the type of vertices. However, whenever a heavy vertex needs to be processed, the owner thread does not relax all the incident edges by itself. Instead, the edges are partitioned among the threads and all the threads participate in the process.

We used the following strategy for determining the threshold π. Let the number of vertices and edges be N and m. Let n and p be the number of processing nodes and the number of threads per node. Then, the number of vertices owned by each thread is $N/(np)$. A thread handles not only its own vertices, but also the heavy vertices, and hence, we should not have too many heavy vertices. A natural idea is to ensure that the number of heavy vertices is no more than $N/(np)$. If we set $\pi = (2mn/N)$, then the number of heavy vertices can be at most N/n (because the total degree is $2m$). Assuming that the heavy vertices get uniformly distributed across the nodes, then the number of heavy vertices handled by each thread is at most $N/(np)$, as desired.

As it turns out, the intra-node load balancing is not sufficient for very high scales (beyond 35) in the case of the Graph 500 family. At such scales, the skew in the degree distribution becomes so extreme that inter-node load balancing becomes necessary. We address the issue by employing an inter-node *vertex splitting* strategy. The idea is to split the vertices having extreme degree and distribute their incident edges among other processing nodes, a concept used in prior work as well [25].

The SSSP framework provides a simple and elegant mechanism for accomplishing the above goal. Consider a vertex u that we wish to split. We modify the input graph by creating ℓ new vertices (called *proxies*) $u_1, u_2, \ldots, u_\ell$, for a suitable value ℓ. All the proxies are connected to the vertex u via edges with zero weight. The set of edges originally incident on u is partitioned into ℓ groups $E_1, E_2, \ldots, E_\ell$. The edges in each group E_i are modified so that they become incident on the proxy u_i (instead of u). Observe that solving the SSSP problem on the original and the new graph are equivalent. The vertices are selected for splitting based on a degree threshold π' (similar to the intra-node balancing procedure). Higher number of proxies not only lead to more vertices, but also the relaxation involving such vertices require two communications. We fix the threshold by considering the tradeoff between spitting high degree vertices, while keeping the number of proxies small. For this purpose, we use a strategy similar to that of the intra-node threshold. The splitting of vertices is carried out as part of a preprocessing phase of graph construction.

3.6 Tuning the Δ Parameter

Our experimental study shows that the value of Δ influences the performance of our algorithm. A simple heuristic to select a good Δ is to consider different candidate Δ values and estimate the performance of each value. This can be achieved by selecting r random roots (for some parameter r)) and measuring the average GTEPs. A naive implementation of the heuristic considers each root `rt` and computes the GTEPS for all the Δ values. This involves d shortest path computations the root `rt`. We design an faster strategy that approximates the above heuristic, while using only a single shortest path computation. The idea is to use a single Δ value and compute the actual shortest path distances from `rt`. Using these distances, we next estimate the number of relaxations for each candidate Δ value, as follows. Consider any edge e. If e is a short edge, then it will get relaxed at least twice (once along each direction); we take the estimated relaxation to be two. On the other hand suppose $e = (u, v)$ is a long edge. The edge will get relaxed in the direction u to v, if the condition (1) is satisfied, where $d(v)$ is the tentative distance of v at the end of epoch k. In our estimate, we use the actual shortest distance from `rt` to v as a proxy for $d(v)$. This way, we can estimate the total number of relaxations for all the candidate Δ across the r roots. The Δ having the minimum estimate is chosen.

4 Experimental Analysis

In this section, we present an experimental evaluation of our algorithms on synthetic R-MAT and real world graphs. The experiments were conducted on a Blue Gene/Q system.

4.1 Architecture Description and Implementation

Blue Gene/Q (BG/Q) [26] is the third generation of highly scalable, power efficient supercomputers of the IBM Blue Gene family, following Blue Gene/L and Blue Gene/P. The two largest Blue Gene/Q supercomputers are Sequoia, a 96 rack system installed at the Lawrence Livermore National Laboratory, and Mira, a 48 rack configuration installed at the Argonne National Laboratory.

In order to get the best performance out of Blue Gene/Q, we have utilized three important optimizations. 1) A light-weight threaded model where each thread has complete access to all memory on a node. 2) Direct access to the System Processing Interface (SPI) communication layer. Inter-node communication is implemented at the SPI level, a thin software layer allowing direct access to the "metal" injection and reception DMA engines of the network interface. Each thread is guaranteed private injection and reception queues and communication does not require locking. Threads can communicate with very little overhead, less than a hundred nanoseconds, with a base network latency of half a microsecond in the absence of contention. The SPI interface is also capable of delivering several tens of millions of messages per second per node [26]. 3) L2 Atomics. We rely on the efficient implementation of a set of atomic operations in the nodes' L2 caches to implement the relaxations. Each core can issue an atomic operation every other clock cycle, providing a considerable aggregate update rate.

Each Blue Gene/Q node has 16 cores supporting four-way SMT and our implementation uses 64 threads per node. The implementation is entirely written in C and uses Pthreads for on-node threading and SPI for communication; the compiler used is GCC 4.4.6. The data in the experimental section was collected on Mira.

4.2 Graph Configurations

We conducted an extensive experimental evaluation of the different graph algorithms discussed in the paper on synthetic graphs, and a preliminary study on real world graphs.

The synthetic graphs are generated using the R-MAT model [4]. Each edge is determined using a random process governed by four parameters (probabilities) A, B, C and D satisfying $A+B+C+D=1$. An edge is generated by choosing the endpoints u and v via a random bit fixing mechanism directed by the four probabilities. Typically, the parameters B and C are set to the same value so that a family of graphs get described by two parameters, A and B. The R-MAT parameters determine the skew in the degree distribution. When all the parameters are equal (1/4), then all pairs of vertices are equally likely to be selected as the edge and the graph is expected to have a uniform degree distribution. Otherwise, there is a skew in the degree distribution, determined by the deviation of the parameters from the mean $1/4$. As we shall see, the skew in the graph has a major impact on the performance of the algorithms.

Our experimental study consists of four parts. The goal of the first part is to study the different algorithms on benchmark graphs suggested by the Graph500 specification [27], namely sparse RMAT graphs with parameters $A=0.57$ and $B=0.19$ with edge weights assigned according to a Graph500 SSSP benchmark proposal [24] (randomly selecting integers in the range $[0, w_{\max}]$, where $w_{\max}=255$). In the second part, we explore a spectrum of RMAT graphs by focusing on a representative set of 14 configurations that have different values of A and B parameters. These configuration represent different levels of skew in the degree distribution. Furthermore, we also study the effect of weight selection by varying the weight range and also by picking weights according to non-uniform distributions. In the third part, we evaluate the algorithms on eight real-world graphs. This is followed by the fourth part, a brief report on the performance of the `OPT` algorithm on larger systems (32,768 nodes).

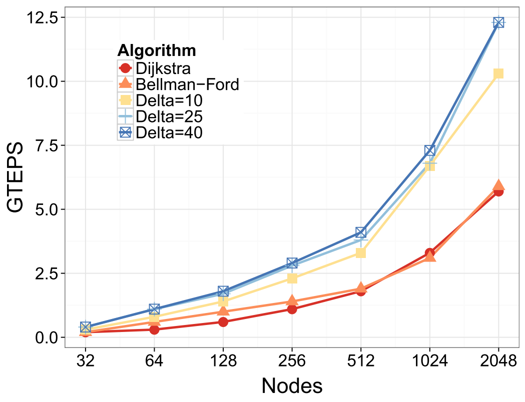


Fig. 8. Graph500: Performance of Δ-stepping algorithm.

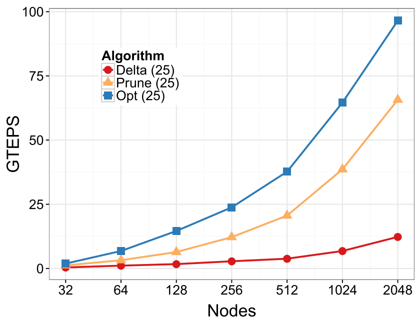
For the RMAT graphs, we used an edge factor of 16 (i.e., the number of undirected edges is $m=16 \cdot N$), as suggested in the benchmark. The experiments are based on weak scaling: the number of vertices per Blue Gene/Q node was fixed at 2^{23} and the number of nodes was varied from 32 to 32,768.

We assume a setting where the graph is given a priori and linear time preprocessing is allowed. Then, given any source vertex as a query, we utilize the pre-processed graph to compute the shortest distances. We ignore the graph construction and related preprocessing routines (such as splitting of heavy vertices), and evaluate the performance by measuring the time taken to compute the shortest path distances once the source vertex is revealed. In all the experiments, we randomly selected 12 roots and the average statistics are reported.

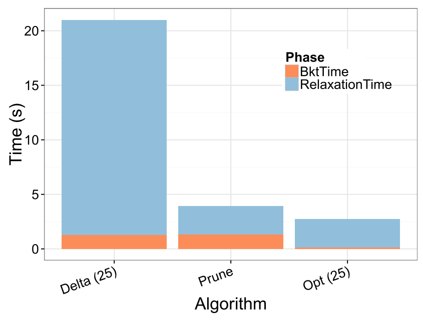
4.3 Graph500 Benchmark

Here, we consider RMAT graphs generated according to the Graph500 specification ($A=0.57$ and $B=0.19$) with edge weights being randomly selected integers in the range $[0, 255]$. We evaluate the algorithms on systems of size up to 4,096 nodes.

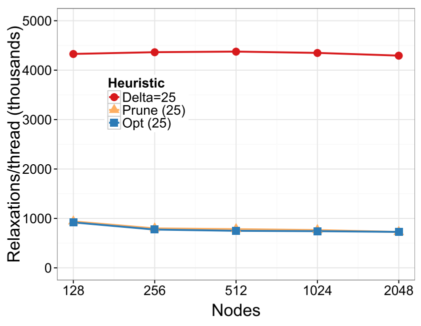
Δ-*Stepping*. The first experiment evaluates our implementation of the Δ-stepping algorithm (with short and long edge classification) for different values of Δ. We tested various values of Δ ranging from 1 (Dijkstra's) to ∞ (Bellman-Ford). The results are shown in Fig. 8. As discussed earlier, the conflicting aspects of number of phases/buckets and work-done play a key role in determining the overall performance. Dijkstra's algorithm performs poorly, because it



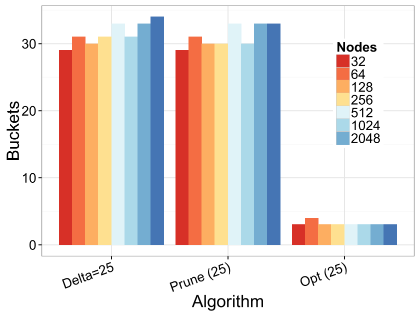
(a) GTEPS of different algorithms



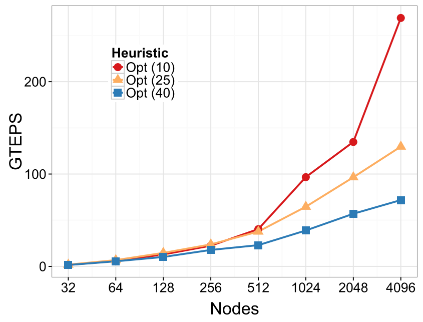
(b) Time breakdown



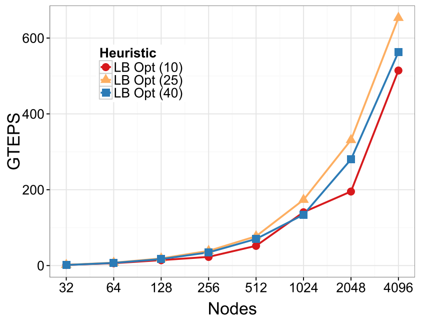
(c) Relaxations



(d) Number of buckets



(e) Effect of Δ on `OPT`



(f) Effect of load balancing on `OPT`

Fig. 9. Graph500: Analysis.

utilizes a large number of buckets. Bellman-Ford is equally bad. Δ values between 10 and 50 offer the best performance.

Evaluation of the Proposed Heuristics. Given the above results, we fixed $\Delta = 25$ and evaluated our heuristics by comparing them against the baseline Δ-stepping algorithm. Three algorithms are considered in the evaluation: (i) Del-25: the baseline Δ-stepping algorithm (along with short-long edge classification); (ii) Prune-25: the Del-25 algorithm augmented with the pruning and IOS heuristics; (iii) `OPT`-25 : Prune-25 augmented with the hybridization heuristic.

The GTEPS performance of the algorithms is shown in Fig. 9a. Considering the case of 2,048 nodes (scale-34 graphs), we see that the pruning strategy is very effective and provides a factor five improvement, and the hybridization strategy offers an additional improvement of about 30 percent. Combining the above two heuristics, the `OPT`-25 algorithm improves the baseline Δ-stepping algorithm by a factor of about eight.

We performed a detailed analysis of the algorithms by dividing the overall time taken into two groups. (i) *Bucket processing overheads (denoted BktTime)*: we must identify the current bucket vertices at the beginning of each epoch, and the set of active vertices at the beginning of each phase. The index of the next (non-empty) bucket must be computed at the end of each epoch. (ii) *Relaxation time*: This includes processing and communication involved in the relaxations.

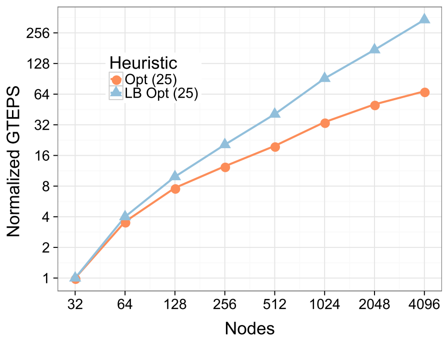
Fig. 9b presents the breakdown of the running time of the three algorithms on scale-34 graphs (2,048 nodes). We see that compared to the baseline Del-25 algorithm, pruning targets the relaxation time and achieves a reduction by a factor of about seven (while incurring the same bucket processing time, as expected). However, the hybridization strategy reduces the bucket processing time and nearly eliminates the overhead.

The above phenomenon can be explained by considering the two underlying statistics: the number of relaxations and number of buckets. Fig. 9c presents the number of relaxations (expressed as an average over all the threads) and we see that the pruning strategy obtains a reduction by a factor of about 6. In Fig. 9d, we see that Del-25 uses about 30 buckets, whereas the hybridization strategy converges in at most five buckets. It is interesting to note that the number of buckets is insensitive to the graph scale.

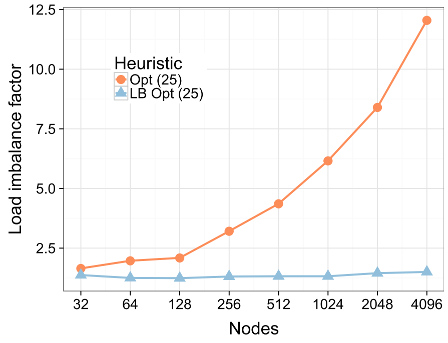
Impact of Load Balancing. We analyzed the effect of the parameter Δ on the `OPT` algorithm, by considering values in the range 10 to 50. For the sake of clarity, the GTEPS performance is reported for three representative values of $\Delta = 10, 25$ and 40, as shown in Fig. 9e. We see that `OPT`-10 performs the best. However, all the versions suffer from poor scaling. We analyzed the above phenomenon and observed that the load imbalance is the root cause. As discussed in Section 3.5, the above effect can be attributed to the remarkable skew in the degree distribution of Graph500 graphs. The skew is highlighted by the maximum vertex degrees, shown in Fig. 7.

We evaluated the effectiveness of our load balancing strategies. As it turns out, on graphs of scale up to 35 (or system size up to 4,096 nodes), the skew in the degree distribution is not high enough to warrant the inter-node load balancing technique of vertex splitting. The simpler intra-node thread level load balancing is sufficient. Fig. 9f presents the GTEPS for the load balanced version, denoted $\mathtt{LB} - \mathtt{Opt}$. Comparing with Fig. 9e, we can see that load balancing improves the performance by a factor of two to eight, depending upon the value of Δ.

Further analysis of load balancing is presented in Fig. 10. Fig. 10a shows GTEPS of `OPT` and $\mathtt{LB} - \mathtt{Opt}$ normalized with respect to GTEPS at 32 nodes. We can see that devoid of load balancing, `OPT` scales poorly, whereas $\mathtt{LB} - \mathtt{Opt}$ scales near perfectly. The behavior can be understood by tracking the load imbalance factor, which is the ratio of the maximum



(a) Normalized GTEPS



(b) Load imbalance factor

Fig. 10. Analysis of load balancing.

load across threads to the average load (by the taking number of relaxations as the load). Under the weak scaling model, the factor should stay close to one for an ideal algorithm. The figure shows the factor for both OPT and $\texttt{LB} - \texttt{Opt}$: while the factor increases for OPT, it stays close to one for $\texttt{LB} - \texttt{Opt}$, as desired.

4.4 Study of the RMAT Parameter Spectrum

RMAT graphs serve as a abstraction for real world networks and model the properties of these networks in terms of the parameters A, B, C and D. Given a real world graph, one can derive the corresponding RMAT parameters to generate synthetic graphs with similar degree distribution, community structure and diameter [4]. The parameters corresponding to such graphs typically show that $A \geq B, A \geq C, A \geq D$ [4]. We use this property to systematically explore the parameter space by setting $B = C$ and varying A and B across a range of values. In this fashion, we generated 14 RMAT families (see Fig. 11). We first study the algorithms on these families with uniform weigths distributed in the range $[0, 255]$. Then we consider the effect of other weight ranges and non-uniform distributions.

4.4.1 *Comparison of* $\texttt{LB} - \texttt{Opt}$ *and* Δ*-Stepping Algorithms*

We compared the performance of the $\texttt{LB} - \texttt{Opt}$ and the baseline Δ-stepping algorithms on the 14 RMAT graphs. For this purpose, we fixed the number of nodes to be 1,024. For the baseline algorithm, we considered a range of Δ values and found that $\Delta = 25$ was near-optimal across all the 14 graphs.

A	B	Skew (A-D)	DEL (GTEPS)	$\texttt{LB} - \texttt{Opt}$ (GTEPS)	GTEPS improv. factor
0.4	0.1	0	37.4	76.6	2.0
0.5	0.1	0.2	25	73.8	3.0
0.6	0.1	0.4	25.5	120.4	4.7
0.7	0.1	0.6	12	176.2	14.7
0.35	0.15	0	37.4	78.2	2.1
0.45	0.15	0.2	27.5	77	2.8
0.55	0.15	0.4	26.7	125.3	4.7
0.65	0.15	0.6	5.5	178.2	32.4
0.3	0.2	0	37.8	77.4	2.0
0.4	0.2	0.2	29.3	81.7	2.8
0.5	0.2	0.4	28.2	123.2	4.4
0.25	0.25	0	37.8	80.2	2.1
0.35	0.25	0.2	31.2	82.2	2.6
0.45	0.25	0.4	29.9	109.8	3.7

Fig. 11. Performance on 14 RMAT graphs.

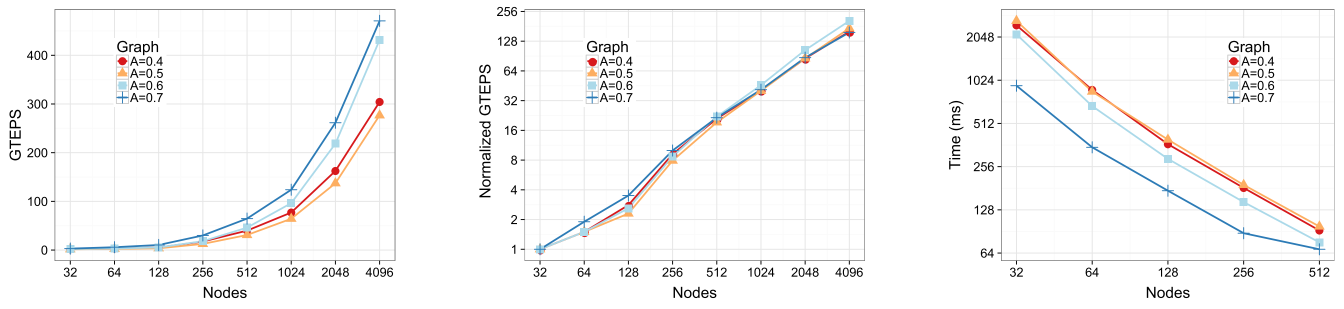
Hence, we fixed $\Delta = 25$ in our experiments. The results are summarized in Fig. 11. For each graph, the difference between A and D serves as an indicator of the skew in the degree distribution, with a larger value indicating a higher skew. For each graph, computed the ratio of the GTEPS of the $\texttt{LB} - \texttt{Opt}$ to the Δ-stepping algorithm (denoted GTEPS improvement factor). We see that the $\texttt{LB} - \texttt{Opt}$ algorithm outperforms the baseline in all the 14 graphs with $2\times$ to $32\times$ improvement in GTEPS, touching a maximum of 178.2 GTEPS for the highly skewed graph with $A = 0.65$ and $B = 0.15$. For any fixed value of B, we can see that increase in skew results in increase in improvement factor. For example, for $B = 0.1$, when A is varied from 0.4 to 0.7, the GTEPS obtained by the $\texttt{LB} - \texttt{Opt}$ algorithm increases from 76.6 to 176.2, and the improvement factor increases from 2.05 to 14.68. Across the 14 graphs, we also see that graphs with similar skew exhibit similar GTEPS and improvement factors.

As explained earlier (Section 3.3), the skew in the graph is an important factor in determining the effectiveness of the pruning strategy. When the skew is higher, the graph tends to contain dense regions that settle in the early buckets, leading to a larger number of self-edges. As a result, the pulling strategy can be employed, yielding more pruning opportunities. Furthermore, load balancing strategies are more effective on skewed graphs. Consequently, the optimized algorithm $\texttt{LB} - \texttt{Opt}$ provides better improvement over the baseline Δ-stepping on such graphs.

We next present more detailed analysis on graphs with $B = 0.1$ and study the factors that influence the performance of $\texttt{LB} - \texttt{Opt}$ and the baseline Δ-stepping algorithms. Fig. 12 shows the ratio of $\texttt{LB} - \texttt{Opt}$ to the baseline under three different metrics: (i) bucket improvement factor (ratio

						Load Imbalance Factor	
A	S	$d_{\max}$	BIF	LIF	GIF	DEL	OPT
0.4	0	75	1.8	3.1	2.0	1.0	1.0
0.5	0.2	12K	15.4	2.2	3.0	1.0	1.0
0.6	0.4	895K	42.5	3.7	4.7	1.2	1.1
0.7	0.6	1.1M	19.0	5.5	14.7	4.6	1.2

Fig. 12. Class of RMAT graphs with $B = 0.1$. S—Skew $(A - D)$. $d_{\max}$—maximum degree. BIF—bucket time improvement factor (ratio of bucket processing time of Δ-stepping to $\texttt{LB} - \texttt{Opt}$). LIF—load improvement factor (ratio of load of Δ-stepping to $\texttt{LB} - \texttt{Opt}$). GIF—GTEPS improvement factor (ratio of GTEPS of $\texttt{LB} - \texttt{Opt}$ to Δ-stepping). Load imbalance factor—ratio of maximum load across threads to the average load.



(a) Weak Scaling (GTEPS) (b) Weak Scaling (Normalized GTEPS) (c) Strong Scaling (Execution time)

Fig. 13. Scaling behavior with $B = 0.1$.

of number of buckets); (ii) load improvement factor (ratio of number of relaxations); (iii) GTEPS improvement factor (ratio of GTEPS). The figure also presents load imbalance factor (ratio of maximum load to the average load) for the two algorithms. We see that increase in skew results in higher load improvement (more pruning) and bucket improvement factors. Furthermore, $\texttt{LB}-\texttt{Opt}$ is better load balanced with the load imbalance factor remaining close to one (ideal value), whereas the factor tends to increase for the baseline algorithm. These factors collectively enable the $\texttt{LB}-\texttt{Opt}$ algorithm to outperform the baseline Δ-stepping algorithm across the spectrum of RMAT graphs with GTEPS improvements ranging from $2.0\times$ to $14.7\times$.

4.4.2 Evaluation of Push-Pull Decision Heuristic

We evaluated the efficacy of the push-pull decision heuristic by designing a routine that checks whether the heuristic makes the correct sequence of decisions. Consider an input graph and a root, and suppose the Δ-stepping algorithm (in conjunction with the hybridization strategy) involves k buckets. At the end of each epoch, the pruning algorithm needs to make a decision on whether to use the push or the pull mechanism, and thus, 2^k different sequences of decisions are possible. The validation routine considers all the possible decision sequences and computes the corresponding running time. The time taken by the best of these sequences is compared against the running time obtained by applying our push-pull decision heuristic.

We fixed the number of nodes to be 32, $\Delta = 25$ and conducted the validation process on all the 14 graphs for 12 random roots. We found that across these test cases, the sequence of decisions made by our heuristic was near-optimal in most cases. On an average, the heuristic is sub-optimal only be a factor of $1.07\times$ with the factor becoming $1.26\times$ in the worse case on the RMAT graph with $A = 0.35$ and $B = 0.25$, which has very less skew.

4.4.3 Evaluation of Δ Prediction Heuristic

In this section, we present an evaluation of our Δ prediction heuristic. We fixed the number of nodes at 1,024 and computed the optimal Δ for all the 14 RMAT graphs via an exhaustive search. We then compared the performance of the $\texttt{LB}-\texttt{Opt}$ algorithm with the optimal and the predicted Δ values. In nine out of the 14 graphs, we observed that the heuristic predicted the optimal value. In the remaining cases, the loss in GTEPS was at most $1.15\times$.

4.4.4 Scaling Study

We studied the weak scaling behavior of the $\texttt{LB}-\texttt{Opt}$ algorithm on different RMAT graphs by fixing $B = 0.1$. For this purpose, we varied the number of nodes from 32 to 4,096, and measured the GTEPS, shown in Fig. 13a. For the ease of visualization, we have also provided the GTEPS normalized with respect to the GTEPS at 32 nodes. We can see that while the performance differs depending on the graph, the algorithm scales well on all the graphs of varying skew.

We also studied the strong scaling behavior of the $\texttt{LB}-\texttt{Opt}$ algorithm on different RMAT graphs by fixing $B = 0.1$ and the scale of the graph to be 28. We varied the number of nodes from 32 to 512 nodes, and measured the execution time. The results are shown in Fig. 13c. We can see that the algorithm exhibits good strong scaling behavior.

4.4.5 Effect of Weight Range and Distribution

The goal of this experiment is to study the effect of range and non-uniformity in weight selection on the Δ-stepping and the $\texttt{LB}-\texttt{Opt}$ algorithms. We fix $B = 0.1$ and focus on four classes of RMAT graphs with $A = 0.4, 0.5, 0.6$ and 0.7. We first study the effect of weight range by considering four values for $w_{\max} = 255, 511, 767$ and 1,023, over the uniform distribution. Second, we fix $w_{\max} = 255$ and study the effect of non-uniformity in weight selection by using the triangular distribution (http://mathworld.wolfram.com/TriangularDistribution.html). The triangular distribution is similar in spirit to the normal distribution, with the probability mass being concentrated around a given point c. Namely, given a *center* $c \in [0, w_{\max}]$, the distribution picks a weight $x \in [0, w_{\max}]$ with probability, as follows: $\frac{x}{cw_{\max}}$, if $x \leq c$ and $\frac{w_{\max}-x}{w_{\max}(w_{\max}-c)}$, otherwise. We considered three values for the center $c = 63, 127$ and 191. For $\texttt{LB}-\texttt{Opt}$, we choose Δ as given by the prediction heuristic, whereas for Δ-stepping the parameter is fixed at 25.

The GTEPS for the algorithms on the two settings are given in Fig. 14. We can see that $\texttt{LB}-\texttt{Opt}$ outperforms Δ-stepping on all settings, with varying levels of improvement. Furthermore, the GTEPS of $\texttt{LB}-\texttt{Opt}$ is largely independent of the weight range and distribution, but it is primarily dependent on the skew in the degree distribution (parameter A). We also studied the underlying metrics and noticed that the number of buckets does not vary much, while the optimal Δ value increases with increase in $w_{\max}$ in the first setting, and increase in c in the second setting.

	w_{max}				center c		
	255	511	767	1023	63	127	191
A = 0.4	44.7	41.9	41.0	40.7	39.7	40.6	45.7
	21.4	20.5	18.7	17.2	22.4	21.5	20.7
A = 0.5	38.2	37.9	36.4	36.6	40.4	40.1	40.2
	16.2	13.8	11.5	9.9	18.7	18.3	17.5
A = 0.6	56.4	54.5	52.9	51.0	50.7	50.2	50.2
	16.0	13.3	10.9	9.3	17.5	17.5	17.1
A = 0.7	76.9	75.3	74.8	74.7	70.1	71.3	66.9
	7.0	6.7	6.2	5.6	7.1	7.2	7.4

Fig. 14. Effect of weight range and distribution. $w_{\max}$—maximum weight. c—center for the triangular distribution. For each A value, the first row gives the GTEPS of $\texttt{LB}-\texttt{Opt}$ and the second row that of Δ-stepping.

4.5 Real Life Graphs

In this section, we present a preliminary evaluation of our algorithm on eight real life graphs from social, web, collaboration and road networks. The edge weights are assigned as in the benchmark study (randomly selected in the range $[0, 255)$. The graphs are shown in the Fig. 15. We compared the $\texttt{LB}-\texttt{Opt}$ algorithm with the baseline Δ-stepping algorithm using 256 nodes at $\Delta = 25$. Fig. 15 shows the GTEPS of the two algorithms, and the ratio of GTEPS of $\texttt{LB}-\texttt{Opt}$ to that of Δ-stepping, termed improvement factor.

It can be seen that OPT provides a 1.8-50× improvement in performance over the baseline algorithm and achieves a maximum GTEPS of 28.7. However, across the graphs, we see wide variation in the performance of the OPT algorithm. This variation arises due to the different factors: the number of buckets, the amount of pruning obtained by the algorithm and the load imbalance among the different threads. We measure the load imbalance factor as the ratio of the maximum load to the average load and the effectiveness of pruning as the ratio of number of relaxations to the number of edges in the graph (termed normalized relaxation). These statistics are shown in Fig. 16. The algorithm performs well on graphs such as `friendster` and `orkut` because it achieves significant pruning. We also see, for example in the case of the `sk2005` graph, that although our algorithm performs lesser number of relaxations, high load imbalance can dampen the GTEPS. The performance is poor on the `Hollywood` and the `USA road` graph because of large number of buckets and very high average relaxations per edge. We believe that the lower performance is because of structural properties such as high diameter, which leads to a large number of refinements to the tentative distances. This requires further study.

These results show that while the heuristics employed by the $\texttt{LB}-\texttt{Opt}$ algorithm provide significant improvements over the Δ stepping algorithm, further tuning can enhance the performance on real world graphs. We remark that in terms of absolute running time, the algorithm was able to process the biggest real life graph in our study, `uk2007`, in less than a second, using 256 nodes of BlueGene/Q.

Graph	Vertices	Edges	OPT GTEPS	DEL GTEPS	Imp. fac.
`friendster`	63 M	1.8 B	28.7	8	3.5
`twitter`	41 M	1.4 B	19.7	0.7	28.1
`uk2007`	105 M	3.7 B	10	0.2	50
`sk2005`	50 M	1.9 B	6.3	0.5	12.6
`orkut`	3 M	117 M	4.7	2.6	1.8
`livejournal`	4.8 M	68 M	2	0.7	2.8
`hollywood`	2 M	228 M	0.04	0.01	4
`USA road`	2.7 M	6.8 M	0.005	0.001	5

Fig. 15. Comparison on real world graphs. Imp. fac.—improvement factor—ratio of GTEPS of $\texttt{LB}-\texttt{Opt}$ to that of Δ-stepping.

Graph	GTEPS	Buckets	Normalized relaxations	Load imb. factor
`friendster`	28.7	3	0.08	2.1
`twitter`	19.7	3	0.05	2.6
`uk2007`	10	7	0.12	4.4
`sk2005`	6.3	4	0.05	10.6
`orkut`	4.7	6	0.53	1.9
`livejournal`	2	7	0.9	1.8
`hollywood`	0.04	2671	3.8	3.9
`USA road`	0.005	3065	238.9	1.1

Fig. 16. Statistics for the OPT algorithm. Normalized relaxations—ratio of number of relaxations to the number of edges. Load imbalance factor—ratio of maximum load across threads to the average load.

4.6 Massive Graphs and Systems

The experiments described earlier deal with RMAT graphs of scale up to 35 and system sizes up to 4,096 nodes. Here, we present a brief experimental evaluation using larger RMAT graphs, up to scale 39, and systems, up to 32,768 nodes. The study is based on two families of RMAT graphs, with different parameters. The first is the Graph500 family, with $A = 0.57$ and $B = 0.19$, and the second is the SSSP Proposal family that sets $A = 0.55$ and $B = 0.1$, as suggested in an SSSP benchmark proposal [24].

At such large scales, the thread level load balancing strategy is not sufficient for processing Graph500 family and we employ the two-tiered strategy involving the vertex-splitting technique. On the other hand, the skew in the degree distribution for the SSSP Proposal family is sufficiently small so that we did not require load balancing procedures.

Fig. 17 presents the performance (GTEPS) achieved by the $\texttt{LB}-\texttt{Opt}$ algorithm on the two families for system sizes ranging from 1,024 to 32,768 nodes, scale 33 to 39. We see that in the largest configuration of 32,768 nodes and scale-39 graphs, the algorithms achieve about 3,100 and 1,500 GTEPS, respectively. The results show that the combination of the different heuristics proposed in this paper can achieve high performance and good scaling on large graphs and massive system sizes.

4.7 Effect of the Communication Layer

As mentioned earlier (Section 4.1), our implementation uses a communication interface that directly accesses the SPI communication layer. The interface avoids overheads and provides performance gains compared to using the standard MPI interface. A prior work [23] compares the two interfaces on the same BG/Q platform for the related BFS problem. The experimental study therein shows that while the scaling behavior remains the same in the both the cases, SPI offers a constant factor (about five factor) increase in performance across different system sizes. While our experimental study did not include a similar comparison, we note that all the algorithms considered in the experiments (from the baseline Δ-stepping to $\texttt{LB}-\texttt{Opt}$) use the same SPI interface. Based on the above discussion and the fact that

Nodes	1024	2,048	4,096	8,192	16,384	32768
Graph 500	173	331	653	1102	1870	3107
SSSP Proposal	70	129	244	460	840	1480

Fig. 17. Performance on large systems for two graph families.

the heuristics proposed in the paper improve fundamental machine-independent characteristics (number of relaxations, number of buckets and maximum load), we expect the heuristics to offer similar performance gains even for MPI-based implementations.

5 Conclusions and Future Work

We studied the SSSP problem on distributed systems and proposed heuristics that provide significant performance gains over the baseline Δ-stepping algorithm, especially on power-law graphs. The experimental study involved large parallel systems. On the largest system considered, 32,768 nodes of Mira, the leadership supercomputer at Argonne National Laboratory, the optimized algorithm achieves an impressive processing rate of 3,100 GTEPS on a scale 39 RMAT graph with 2^{39} vertices and 2^{43} undirected edges.

The optimization techniques are effective on real-life graphs as well, but the improvements are less pronounced: the improvement in GTEPS ranges from a factor of 2 to 50, depending on the characteristics of the input graph. Similarly, the GTEPS achieved by the algorithm varies widely ranging from 0.005 to 28.7 on 256 nodes, with the lowest performance achieved on road networks. Our preliminary analysis shows that the diameter and skew of the graph play a role.

We conclude the paper by highlighting some potential avenues for future work. The paper focuses on distributed memory systems and a study on other platforms such as shared memory systems, GPU and NUMA multicores would be of interest. We believe that while the pruning technique is more amenable for distributed memory settings, hybridization may be applicable in other platforms as well.

Based on our experiments on the triangular weight distribution, we surmise that the performance of LB $-$ Opt is more dependent on the structure of the graphs, rather than the weight distribution. A detailed study involving real world graphs with real edge weights would shed more light on the effect of weights on the performance of the algorithms.

The techniques proposed in this paper are potentially applicable to other graph problems in a distributed setting. The two-tiered load balancing strategy can be used for implementations based on 1D distribution for other problems as well. The pruning technique may be employed for problems wherein the vertices send update to their neighbors. Some potential candidates are densest subgraph problems [28] and truss computations [29]. Exploring these optimization techniques for other problems is an interesting avenue of future work.

Acknowledgments

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. We express our sincere thanks to Kalyan Kumaran, Ray Loy, Susan Coghlan and the other members of the ALCF support team for their help and support.

References

[1] U. Brandes, "A faster algorithm for betweenness centrality," *J. Math. Sociology*, vol. 25, no. 2, pp. 163–177, 2001.
[2] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley, 2006.
[3] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.
[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.
[5] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, 2010.
[6] S. Beamer, K. Asanovic, and D. A. Patterson, "Direction-optimizing breadth-first search," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. no. 12.
[7] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak, "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances," in *Proc. Meeting Algorithm Eng. Experiments*, 2007, pp. 23–35.
[8] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, 1987.
[9] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine, "Single source shortest paths with the parallel boost graph library," presented at the 9th DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ, USA, Nov. 2006.
[10] K. Nikas, N. Anastopoulos, G. I. Goumas, and N. Koziris, "Employing transactional memory and helper threads to speedup Dijkstra's algorithm," in *Proc. Int. Conf. Parallel Process.*, 2009, pp. 388–395.
[11] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," *Parallel Comput.*, vol. 37, no. 9, pp. 610–632, 2011.
[12] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis, "A parallel priority data structure with applications," in *Proc. 11th Int. Parallel Process. Symp.*, 1997, pp. 689–693.
[13] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *J. ACM*, vol. 46, no. 3, pp. 362–394, 1999.
[14] Y. Han, V. Y. Pan, and J. H. Reif, "Efficient parallel algorithms for computing all pair shortest paths in directed graphs," *Algorithmica*, vol. 17, no. 4, pp. 399–415, 1997.
[15] S. Peyer, D. Rautenbach, and J. Vygen, "A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing," *J. Discr. Algorithms*, vol. 7, no. 4, pp. 377–390, 2009.
[16] D. Wagner and T. Willhalm, "Geometric speed-up techniques for finding shortest paths in large sparse graphs," in *Proc. 11th Annu. Eur. Symp.*, 2003, pp. 776–787.
[17] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Proc. 7th Int. Conf. Exp. Algorithms*, 2008, pp. 319–333.
[18] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "PHAST: Hardware-accelerated shortest path trees," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 921–931.
[19] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2," in *Proc. Int. Conf. Parallel Process.*, 2006, pp. 523–530.
[20] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. 14th Int. Conf. High Perform. Comput.*, 2007, pp. 197–208.
[21] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed memory machines," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. no. 13.
[22] R. Dial, F. Glover, D. Karney, and D. Klingman, "A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees," *Networks*, vol. 9, no. 3, pp. 215–248, 1979.
[23] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 425–434.
[24] (2012). [Online]. Available: http://www.cc.gatech.edu/~jriedy/tmp/graph500/
[25] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[26] D. Chen, et al., "The IBM Blue Gene/Q interconnection network and message unit," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 26.

[27] (2016). [Online]. Available: http://www.graph500.org

[28] B. Bahmani, R. Kumar, and S. Vassilvitskii, “Densest subgraph in streaming and MapReduce,” *Proc. VLDB Endowment*, vol. 5, no. 5, pp. 454–465, 2012.

[29] X. Huang, H. Cheng, L. Qin, W. Tian, and J. Yu, “Querying k-truss community in large and dynamic graphs,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 1311–1322.

Venkatesan T. Chakaravarthy received the PhD degree in computer science from the University of Wisconsin-Madison. He is a researcher with IBM Research—India. He studied at Anna University and Indian Institute of Technology, Chennai. His current research focuses on designing high performance algorithms for tensor analysis and graph theoretic problems. His research interests include approximation algorithms and computational complexity theory.

Fabio Checconi received the BS and MS degrees in computer engineering from the University of Pisa, Italy, and the PhD degree in computer engineering from Scuola Superiore S. Anna, Pisa, Italy. He has been with IBM Research since 2010. His research interests include real-time operating systems and parallel graph algorithms. He won the best paper award at SOCA 2010 and IPDPS 2014.

Prakash Murali received the master’s degree from the Indian Institute of Science, Bangalore (IISc). He is a software engineer with IBM Research, India. He is interested in parallel algorithms and high performance computing.

Fabrizio Petrini received the PhD degree in computer science from the University of Pisa. He is a principal engineer with the Intel Parallel Computing Labs, Santa Clara, California. His research interests include data-intensive algorithms, exascale computing, high performance interconnection networks, and novel architectures for graph analytics and sparse linear algebra.

Yogish Sabharwal received the PhD degree from the Indian Institute of Technology (IIT) Delhi, in 2007, in the area of approximation algorithms in computational geometry. He has more than 19 years of experience in the areas of high performance computing, networking, and telecom. He has been with IBM Research for 13 years where he currently manages the High Performance Computing and Modelling group working on applying HPC to areas in deep learning, graph algorithms, and environmental science. He also holds adjunct faculty positions at Delhi University and Indian Institute of Technology Delhi, where he teaches courses in algorithms, combinatorial optimization, and parallel computing.