Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems

Venkatesan T. Chakaravarthy*, Fabio Checconi†, Fabrizio Petrini†, Yogish Sabharwal*
* IBM Research - India, New Delhi. {vechakra,ysabharwal}@in.ibm.com
† IBM T J Watson Research Center, USA. {fchecco,fpetrin}@us.ibm.com

***Abstract*—In the single-source shortest path (SSSP) problem, we have to find the shortest paths from a source vertex v to all other vertices in a graph. 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. The extensive performance analysis shows that our algorithms work well on scale-free and real-world graphs. 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.**

I. 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. For example, the Facebook social network has 800 million vertices with an average vertex degree of 130 edges,[1] and major search engines are laying out new infrastructure to support a web graph of trillions of vertices.

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], [2].

A. 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 Section I-D.

Two classical approaches for the SSSP problem are attributable to Dijkstra [3] and the Bellman-Ford [4], [5]. 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 iterations and it is less amenable to parallelization. In contrast, Bellman-Ford involves fewer iterations and each iteration is highly parallelizable. However, the algorithm may process each edge multiple times and is likely to incur high processing time.

Meyer and Sanders [6] 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 $\Delta = 1$ yields a variant of Dijsktra's algorithm [7], while setting $\Delta = \infty$ yields the Bellman-Ford algorithm. By varying Δ in the range $[1, \infty]$, we get a spectrum of algorithms with varying degrees of processing time and parallelism.

B. Graph 500 Benchmark

The Graph 500 list[2] 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 Recursive MATrix (R-MAT) scale-free graphs [8], [9], and 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. While still in its infancy, 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.

C. 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.
(b) Pruning: We introduce a new model of direction optimization [14], 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

[1] http://www.facebook.com/press/info.php?statistics

[2] http://www.graph500.org

Reference	Problem	Graph Type	Vertices	Edges	GTEPS	Processors	System
Bader, Madduri [10]	BFS	R-MAT	200 M	1 B	0.5	40	Cray MTA-2
Checconi et al. [11]	BFS	Graph 500	2^{32}	2^{36}	254	4,096 nodes/65,536 cores	IBM Blue Gene/Q (NNSA SC)
Graph 500 Nov 2013 [12]	BFS	Graph 500	2^{36}	2^{40}	1,427	4,096 nodes/65,536 cores	IBM Blue Gene/Q (Mira)
Graph 500 Nov 2013 [12]	BFS	Graph 500	2^{39}	2^{43}	14,328	32,768 nodes/524,288 cores	IBM Blue Gene/Q (Mira)
Graph 500 Nov 2013 [12]	BFS	Graph 500	2^{40}	2^{44}	15,363	65,536 nodes/1,048,576 cores	IBM Blue Gene/Q (Sequoia)
Madduri et al. [13]	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

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 using R-MAT graphs, and a preliminary study involving real world graphs (such as Friendster, Orkut and LiveJournal). 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. [13] 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.[3] 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 (BFS) 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. Figure 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.

D. Related Work

Considerable attention has been devoted to solving SSSP in sequential as well as parallel settings. The classical Dijkstra's algorithm [3] is the most popular algorithm in the sequential world. The algorithm can be implemented in $O(n \log n + m)$ time using Fibonacci heaps [15]. There have been attempts to develop parallel versions of Dijkstra's algorithm and its variants (for example, in Parallel Boost Graph Library [16]). Efforts have also been made to parallelize the algorithm using transactional memory and helper threads [17] but with very modest speedups. Parallel implementations of the Bellman-Ford algorithm [4], [5] and its variants are also available (for example, using MapReduce framework [18]). 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. In this model, there are two parameters of importance in measuring the performance of an algorithm. The first is the running time, and the second is the work done, which is determined by the product of the number of processors and its running time. The best known $O(n \log n+m)$ work solution is obtained by combining the techniques of Paige et al. [19] with Driscoll et al. [20] – this algorithm has running time $O(n \log n)$. Brodal et al. [21] presented an algorithm with a running time of $O(n)$ and $O((n + m) \log n)$ work. Thorup [22] presented an $O(n+m)$ time RAM algorithm but this was restricted to a special case of undirected graphs. Algorithms with poly-logarithmic running times are inefficient in terms of work done [23]. The most relevant work in the context of this paper is due to Meyers and Sanders [6], 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 [13].

Breadth First Search (BFS) 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 [10], [14], GPUs [24] and massively parallel distributed memory machines [11]. The best BFS implementation of the Graph 500 benchmark achieves $15,363$ GTEPS over a graph of size 2^{40} vertices and 2^{44} edges on a Blue Gene/Q distributed memory system having $65,536$ nodes and $1,048,576$ cores.

II. Basic Algorithms

In the SSSP problem, the input consists of a weighted undirected graph $G = (V, E, w)$, where the weight function w assigns an integer weight $w(e) > 0$ to each edge $e \in E$. Let the number of vertices be $|V| = N$ and the number of

[3]This an estimated upper bound derived from the experimental evaluation in [13]

edges be $|E| = m$. The input also specifies a vertex $\texttt{rt} \in V$ called the root. The goal is to compute the shortest distance from $\texttt{rt}$ to every vertex $v \in V$. For a vertex $v \in V$, let $d^*(v)$ denote the shortest distance. The graph scale *scale* is defined as $\log_2 N$.

Our algorithm for the SSSP problem builds on three well-known algorithms, which we refer to as the basic algorithms: Dijkstra's algorithm, the Bellman-Ford algorithm (see [25]) and the Δ-stepping algorithm [6]. In this section, 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.

A. 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:

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

Furthermore, 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 iterations. In each iteration, the unsettled vertex u having the minimum tentative distance is selected. We call u as the *active vertex* of this iteration 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 iteration, whereas the Bellman-Ford algorithm selects multiple active vertices. The algorithm proceeds in multiple iterations. In each iteration, we declare a vertex u to be active, if its tentative distance $d(u)$ changed in the previous iteration. 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 iteration.

Δ-Stepping Algorithm. Dijkstra's and the Bellman-Ford algorithms employ two contrasting strategies for selecting active vertices in each iteration. 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 iteration. The Δ-stepping algorithm strikes a balance between these two extremes.

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 $\texttt{rt}$ 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 iterations. In each iteration, a vertex u is declared to be active if its tentative distance changed in the previous iteration and the vertex is found in the bucket B_k (in the first iteration 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 Figure 2.

Setting $\Delta = 1$ yields a variant of the Dijkstra's algorithm (due to Dial et al. [7]), whereas setting $\Delta = \infty$ yields the Bellman-Ford algorithm. In the rest of the paper, we will analyze Dijkstra's algorithm as Δ-stepping algorithm with $\Delta = 1$.

Distributed Implementation. Our distributed implementation of the Δ-stepping algorithm is briefly outlined below. The vertices are equally distributed among the processors using block distribution, and each vertex is owned by some processor. A processor would execute only the instructions of the algorithm that are pertinent to its vertices. Relax operations require communication among the processors: to perform $\texttt{Relax}(u, v)$, 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). The iterations and epochs are executed in a bulk synchronous manner. Termination checks and computing the next bucket index require *Allreduce* operations.

B. Characteristics of the Basic Algorithms

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

Work done. We measure the amount of work done as the total number of relax operations performed. The above metric not only determines 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).

Number of buckets/phases. The number of *phases* (i.e., iterations) taken by the algorithm is another important metric in determining the efficacy of an algorithm. 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

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$\}$
$A \leftarrow B_k \cap A'$

$\texttt{Relax}(u, v)$**:**
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

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 ($\texttt{Relax}(u, v)$ and $\texttt{Relax}(v, u)$). 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.

Let us next analyze 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 iteration. As a result, the number of phases is exactly the number distinct shortest distances in the final output. Regarding the Bellman-Ford algorithm, it can be shown that the number of phases is at most the depth (number of levels) of the shortest path tree. Even though both the above quantities can be as high as the number of vertices, N, 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 strikes a balance between the two algorithms and offers a trade-off determined by the parameter Δ. The relationship among the three algorithms is given below:

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

Figures 3 provides an illustrative comparison of the different algorithms for both the metrics on sample graphs used in the experimental study. For the Δ-stepping algorithm, we have included three representative Δ values 10, 25 and 40; the reported statistics pertain to a refined version of the algorithm discussed in Section III-A. [4] We can see that the Δ-stepping algorithm offers a trade-off between the other two algorithms in terms of work-done and number of phases.

III. 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. Our experiments show that these techniques provide significant performance gains over the basic algorithms.

While discussing the proposed optimizations, we will utilize the synthetic graphs used in the experimental study for the purpose of motivation and illustration. These graphs are generated according to the Graph 500 specifications, with an R-MAT random graph generator [8]. The R-MAT 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.

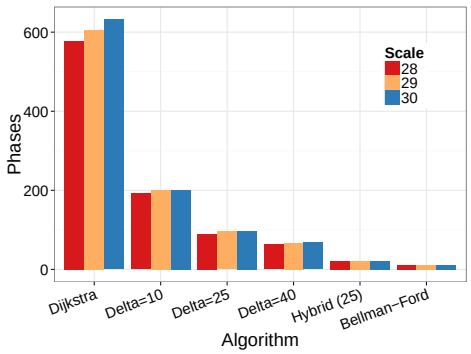
A. Edge Classification

The pruning heuristic utilizes the concept of edge classification, introduced by Meyer and Sanders [6], as a refinement of their Δ-stepping algorithm. We first describe the concept and then, propose an improvement.

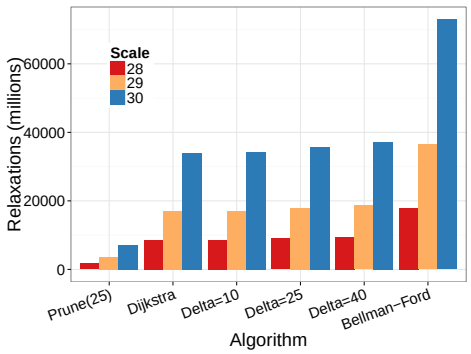
Meyer and Sanders [6] classify the edges into two groups: *long* and *short*. An edge e is said to be *short* if $w(e) < \Delta$, and it is said to be *long* otherwise. Suppose we are processing a bucket and let u be an active vertex with an incident edge $e = \langle u, v \rangle$ such that v belongs 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

[4] The figure also includes statistics for two other algorithms, called Hybrid and Prune, that are discussed later in the paper



(a) Number of phases



(b) Number of relaxations

Fig. 3. Comparison of different algorithms

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., $d'(v) \leq (k+1)\Delta - 1$). 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 ($d'(v) \geq (k+1)\Delta$). 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%, on the benchmark graphs.

B. 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 Figure 3 (b). In this section, we discuss the *pruning heuristic* which focuses on the long edges and provides a drastic reduction in the number of relaxations. The pruning heuristic would relax only a fraction of the edges in the graph, while ensuring correctness. Our experimental study shows that the heuristic causes a reduction by a factor of five of the number of relaxations and a comparable improvement in terms of performance (GTEPS). The heuristic is inspired by the direction optimization technique adopted by Beamer el al. [14] 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. Figure 4 illustrates the above phenomenon using a sample run of the Δ-stepping algorithm. The figure shows the phase-wise distribution of the number of relaxations. Between the short and the long edge phases, we see that the latter phases dominate.

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 would scan all its active vertices in B_k. For each such vertex u and for each long

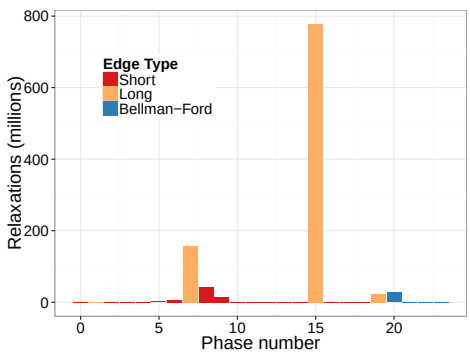


Fig. 4. Dominance of long phases

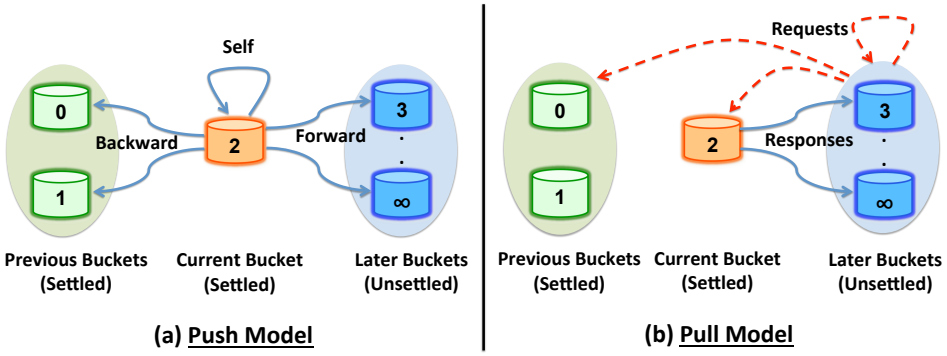


Fig. 5. Push and Pull Models

edge $e = \langle u, v \rangle$ incident on u, the processor would compute a new tentative distance for the destination vertex v given by $d'(v) = d(u) + w(e)$ and send (push) 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 would be redundant. 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. Figure 5 (a) illustrates the 3 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 would defeat 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 will pull the new tentative distance from the source side owner. 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 would scan 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 would send a a response containing the new tentative distance $d'(v)$.

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 will send a response only if u belongs to the current bucket B_k and the shortest distance $d(u)$ of such a vertex would be 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 above 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,

$$w(e) \quad < \quad d(v) - k\Delta \tag{1}$$

Illustration. Figure 5(b) illustrates the communications involved in the pull model, along with the refinement regarding the responses. 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. □

We next illustrate the advantage of the pull model using a simple example graph, shown in Figure 6. Consider running the Δ-stepping algorithm with $\Delta = 5$, using only the push model. The algorithm would take three iterations (where non-zero communication takes place). Initially, the root will be placed in bucket B_0 and made active. In the first iteration (corresponding to the long edge phase of B_0), all the edges incident on the root will be relaxed and the clique vertices will move to B_2. In the second iteration (corresponding to the long edge phase of B_2), all the edges incident on the clique vertices will be relaxed, resulting in the isolated vertices moving to bucket B_4. Finally, in the third iteration (corresponding to the long edge phase of B_4), all the edges incident on the isolated vertices will be relaxed. The cost (number of relaxations) per iteration is shown in the figure; the total is 40. In contrast, consider running the same algorithm, but applying the pull model in the second iteration. In this case, the owners of isolated vertices would send requests and the owners of clique

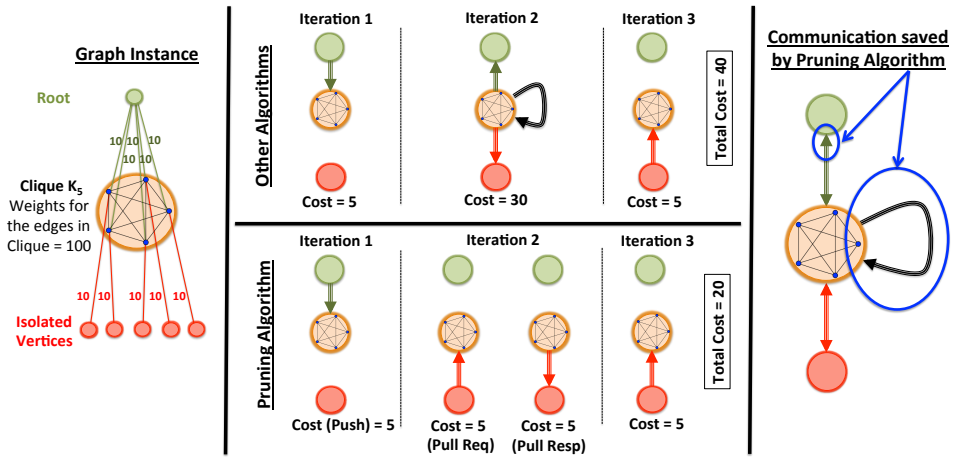


Fig. 6. Illustration of benefit of pull mechanism

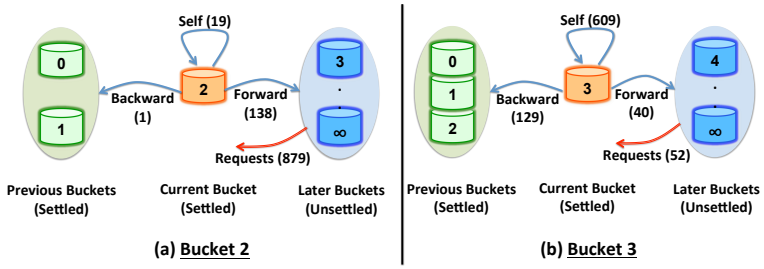


Fig. 7. Pruning: push vs pull model

vertices would respond. For the second iteration, the push model has cost 30, whereas the pull model has cost only 10.

C. Push vs. Pull Heuristic

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.

Illustration. Figure 7 illustrates the need for using a combination of both the models. The figure shows the relevant statistics for two different buckets on an illustrative benchmark graph. The counts of self, backward and forward edges are shown (in millions). The figure also provides the number of requests that would be sent with the pull model. Consider bucket 2. With the push model, then the number of relaxations is the total number of self, backward and forward edges and turns out to be 158 million. On the other hand, under the pull model, the number of requests is 879 million. Even if we ignore the communication involved in sending responses, we see that the push model is better suited for this bucket. In contrast, consider bucket 3. Under the push model, the number of relaxations is 778 millions, whereas under the pull model, the number of requests is 52 million with a comparable number of replies. The number of messages is at most 104 million; thus, the pull model is better suited for bucket 3. □

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 employs a suitable model on a per bucket basis. Towards that goal, we next present a heuristic for selecting the model for each bucket.

Push-Pull Decision Heuristic. The heuristic works by estimating the communication volume and the processing time involved under the push and the pull models.

First, consider the case of 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 the number of edges incident on v whose weight falls in the given range. Multiple strategies are possible for implementing the procedure. Assuming that the edge list of each vertex is sorted according to weights, the quantity can be computed via a binary search. Alternatively, histograms could be used for deriving approximate estimates. The effectiveness of such strategies in terms of running time and memory overhead needs to be investigated further. In our experimental setting, the edges are assigned random weights uniformly distributed in the range $[0, w_{\max}]$, where $w_{\max}$ is the maximum weight. Exploiting the above fact, our implementation computes an estimate by taking the expectation: $\deg(v) \times \frac{d(v)-(k+1)\Delta}{w_{\max}}$. 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.

Our evaluation revealed that the above heuristic accurately determines the best model, with respect to the communication volume. However, in terms of the overall running time, the heuristic fails to make the best choice for about 15% of the test cases considered. As it turns out, the heuristic only considers one parameter, the overall communication volume, and ignores the load imbalance among the processors when sending requests. We fine-tuned the heuristic by taking into account the maximum number of requests sent by the processors. The details are omitted in the interest of space. We evaluated the final heuristic by designing an offline procedure that executes the algorithm under all possible push/pull decision choices. The study shows that the final heuristic is highly effective and achieves near-optimal results on all the configurations considered. The details are deferred to the experimental section.

The Δ-stepping algorithm extended with the combination of push and pull strategies, along with the decision heuristic, is defined as the *pruning algorithm*. The efficacy of the pruning algorithm is illustrated in Figure 3 (b) on sample benchmark runs. In terms of the total number of relaxations, we can see that among the basic algorithms, Dijkstra's performs the best and the pruning algorithm (with $\Delta = 25$) provides a remarkable 5x factor improvement with synthetic graphs. In order to ensure a fair comparison, we included both requests and responses in the overall count (contributing two times) for any edge relaxed via the pull mechanism.

D. 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 strategy 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 the Δ-stepping algorithm may involve a large number of buckets, but most of the relax operations are concentrated in the initial set of buckets. The underlying reasoning is as follows. The 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, we determined that the increase is not significant in practice, since the overall work done 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.

Based on experimental observations, we determined that the number of settled vertices provides a good metric for determining the switching point. Namely, we fix a threshold $\tau \in [0, 1]$. At the end of processing a bucket B_k, we compute the total number of vertices settled so far (this is given by the aggregate number of vertices present in buckets $B_0, B_1, \ldots, B_k$). If the fraction of settled vertices exceeds τ, then we switch to the Bellman-Ford algorithm. Our experiments suggest that setting $\tau = 0.4$ is a good choice.

Figure 3 (a) illustrates the efficacy of the hybridization strategy on sample benchmark runs. 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. Our experimental evaluation on two different benchmark family of graphs shows that the hybridization strategy (with Δ=25) provides significant improvement in overall performance (GTEPS) when applied to the pruning algorithm: up to 30% improvement for one family and 2x improvement for the second one.

E. 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`. 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.

We conducted an extensive evaluation of the algorithm `OPT` on graphs generated according to the R-MAT model. The model is governed by certain parameters that determine the characteristics of the generated graphs. Our evaluation considers two different families of graphs by setting the parameters as per the specifications given in the Graph 500 BFS benchmark and a (proposed) Graph 500 SSSP benchmark[5];

[5]http://www.cc.gatech.edu/~jriedy/tmp/graph500/

Scale	28	29	30	31	32
RMAT − 1	2.4 M	3.8 M	5.9 M	9.4 M	14.4 M
RMAT − 2	31126	41237	54652	72158	95482

Fig. 8. Maximum degree

the two families are denoted RMAT − 1 and RMAT − 2 (see experimental evaluation section for more details).

Our experiments revealed that the scaling behavior of the OPT algorithm is not satisfactory, especially on the first family of graphs, due to load imbalance. The load on a thread can be measured by the aggregate degree of the vertices owned by it. Consequently, any skew in the degree distribution leads to load imbalance among the threads. The maximum degree of the graphs provides a useful yardstick for measuring the skew. Figure 8 presents the above quantity for both the family of graphs for different scales. The average degree for all graphs was fixed to be a constant (32 edges). However, we can see that the maximum degree is very high and increases with scale. As a result, our algorithm suffers from load imbalance, especially for the RMAT − 1 family.

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.

As it turns out, the intra-node load balancing is not sufficient for very high scales (beyond 35) in the case of RMAT − 1 family. At such scales, the skew in the degree distribution becomes so extreme that intra-node load balancing techniques prove insufficient and inter-node load balancing becomes necessary. We overcome 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. 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 create 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 (namely, the shortest distances in the new graphs are the same as that of the original graph). The vertices are selected for splitting based on a degree threshold π' (similar to the intra-node balancing procedure).

We have determined robust heuristics to determine the thresholds π and π', and the number of proxies and partitioning of the neighborhood (namely, the sets $E_1, E_2, \ldots, E_\ell$). The details are omitted for brevity.

IV. 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.

A. Architecture Description and Implementation

Blue Gene/Q (BG/Q) [26], [27], [28] 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 lightweight 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 [28]. 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.

B. 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 were generated using the R-MAT model [8]. 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.If all parameters equal (1/4), then all pairs of vertices are equally likely to be selected as the edge. Otherwise, there is a skew in the distribution, determined by the deviation from the mean $1/4$.

The experimental study considers two families of synthetic R-MAT graphs generated according to two different benchmark specifications that use different R-MAT parameters: (i) RMAT − 1 : this family uses the latest Graph 500 [12] BFS benchmark specification, wherein $A = 0.57$, $B = C = 0.19$ and $D = 1 - A - 2B = 0.05$; (ii) RMAT − 2 : this family

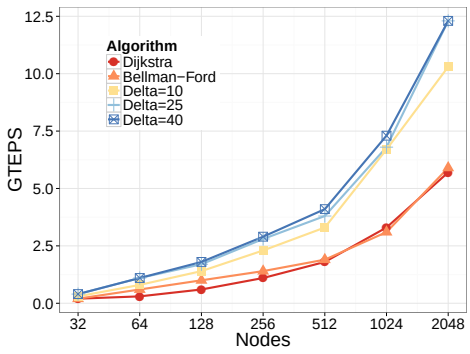


Fig. 9. RMAT − 1: Performance of Δ-stepping algorithm

uses the Graph 500 SSSP benchmark (proposed) specification[6], where $A = 0.50$, $B = C = 0.1$ and $D = 0.3$. We assign edge weights according to the SSSP benchmark proposal for both families: the weights are selected independently at random by picking an integer in the range $[0, 255]$. Both families generate sparse graphs using an edge factor of 16, namely the number of undirected edges is $m = 16 \cdot N$. The differences in the R-MAT parameters have an impact on the characteristics of the two families and the behavior of our algorithms. We first discuss the experimental evaluation for both families and then present a brief comparison.

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 present a detailed analysis of the different algorithms on both RMAT-families on systems of size up to 4,096 nodes, followed by a brief report on the performance of the OPT algorithm on larger systems.

C. RMAT − 1 *Family*

Here, we consider the RMAT − 1 family and 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 Figure 9. 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 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 Figure 10 (a). 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%. 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 (denoted OtherTime)*: This includes processing and communication involved in the relaxations.

Figure 10 (b) 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. Figure 10 (c) 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 Figure 10 (d), we see that Del-25 uses about 30 buckets, whereas the hybridization strategy converges in at most 5 buckets. It is interesting to note that the number of buckets is insensitive to the graph scale.

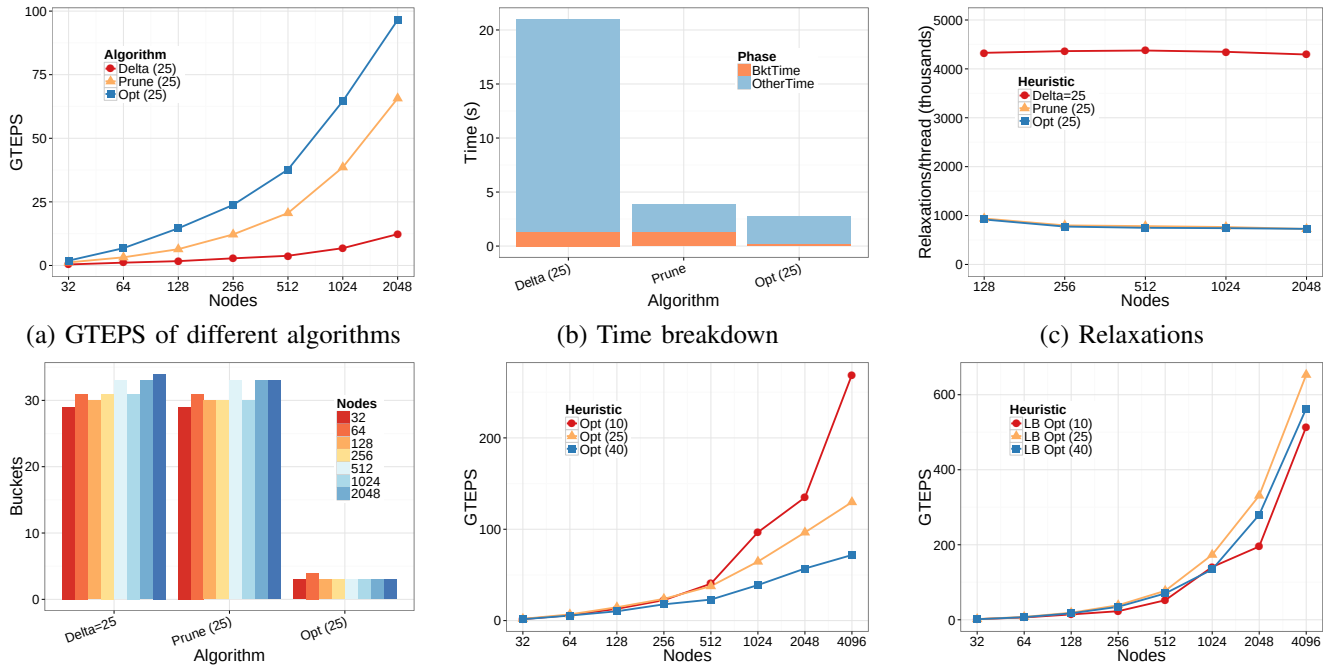
Impact of Load Balancing. We analyzed the effect of the parameter Δ on the OPT algorithm, by considering values in the range 10 to 50. For the sake of clarity, the GTEPS performance is reported for three representative values of $\Delta = 10, 25$ and 40, as shown in Figure 10 (e). We see that OPT-10 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 III-E, the above effect can be attributed to the remarkable skew in the degree distribution of RMAT − 1 graphs. The skew is highlighted by the maximum vertex degrees, shown in Figure 8.

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. Figure 10 (f) presents the GTEPS for the load balanced version LB − Opt. Comparing with Figure 10 (e), we can see that load balancing improves the performance by a factor of two to eight, depending upon the value of Δ. Furthermore, the load balanced version achieves near perfect scaling across all values of Δ, with $\Delta = 25$ performing the best.

D. RMAT − 2 *Family*

Here, we present a brief description of our experimental evaluation of the RMAT − 2 family. As in the case of

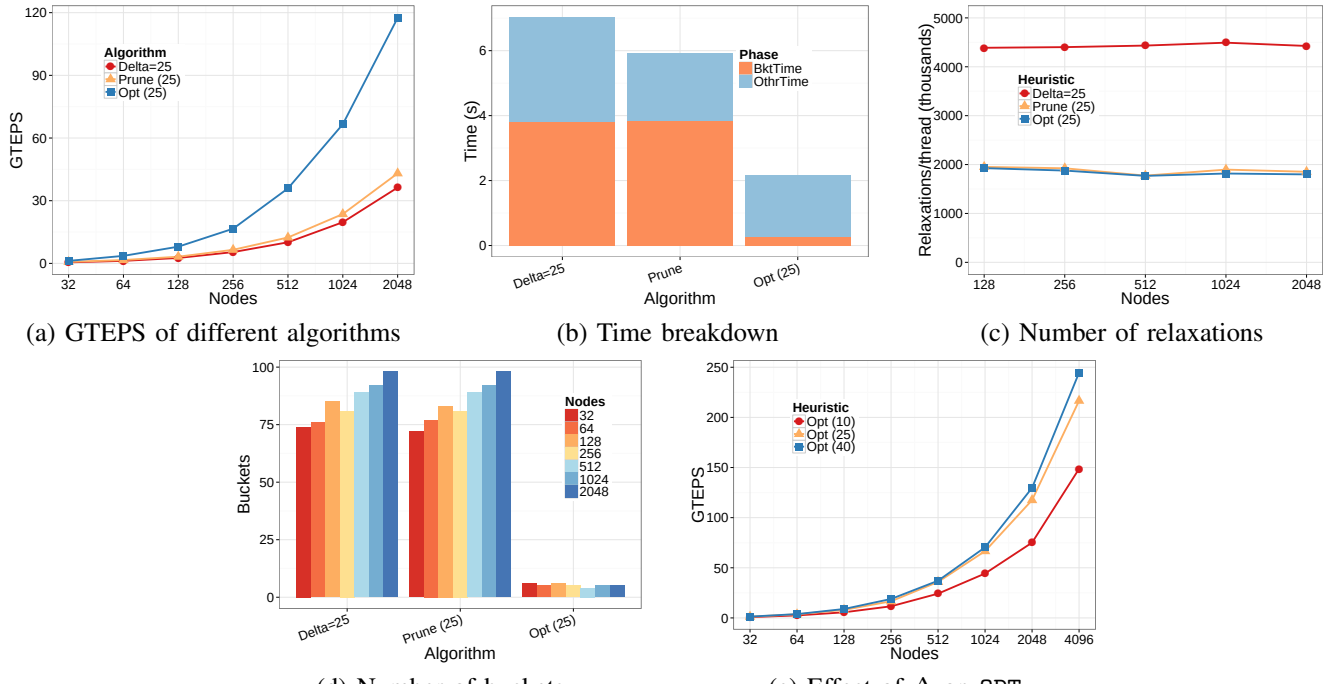
[6]http://www.cc.gatech.edu/~jriedy/tmp/graph500/



(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. 10. RMAT – 1: Analysis



(a) GTEPS of different algorithms (b) Time breakdown (c) Number of relaxations

(d) Number of buckets (e) Effect of Δ on OPT

Fig. 11. RMAT – 2: Analysis

RMAT − 1, we compared three algorithms Del-25, Prune-25 and OPT-25 considering GTEPS, number of buckets, number of relaxations and time breakdown. The results are shown in Figure 11. We see that compared to Del-25, the pruning strategy cuts the number of relaxations in half (Figure c) and improves the relaxation time (OthrTime) by 30% (Figure b). However, due to the high bucket processing time (BktTime), the overall improvement in GTEPS is only about 12%. The hybridization strategy yields higher gains. It reduces the number of buckets by a factor of 20 and achieves a corresponding reduction in the bucket processing time. As a result, OPT-25 is three times faster.

The performance of the OPT algorithm for different Δ values is shown in Figure 11 (e). For all values of Δ, the OPT algorithm scales reasonably well and so our load balancing strategies did not yield significant improvement for this family of graphs. Among these versions, OPT-40 provides the best results.

E. Comparison of the Two Families

The two graph families are generated using different R-MAT parameters, resulting in different graph characteristics. Both families show a strong skew in the degree distribution, as shown in Figure 8. While the average degree remains a constant at 32 directed edges, the maximum degree is substantially larger and increases with the scale. Moreover, the two families exhibit contrasting behavior: the maximum degree for RMAT − 1 is in the order of million of edges, whereas the growth is slower in the RMAT − 2 family.

The distinction between the two families has an impact on the effectiveness of our heuristics. Consider the pruning strategy. With the RMAT − 1, the heuristic avoids processing high degree vertices by employing the pull mechanism and achieves a higher rate of pruning. In contrast, the degree distribution for the RMAT − 2 is more uniform and therefore the push and pull mechanisms do not differ much in terms of relaxations. Consequently, the pruning factor is less dramatic. Compared to the first family, the shortest distances are distributed over a larger range of values, requiring more buckets. Consequently, the hybridization strategy can identify opportunities for optimization and is more effective. In a similar vein, the skew in the degree distribution determines load imbalance, therefore our load balancing techniques are more effective for the RMAT − 1 family. We conclude that the performance on RMAT − 1 graphs is better than that of RMAT − 2 graphs primarily because more effective pruning is achieved in the former family.

F. Massive Graphs and Systems

All the experiments described earlier deal with graphs of scale up to 35 and system sizes up to $4,096$ nodes. Here, we present a brief experimental evaluation using larger graphs, up to scale 39, and systems, up to $32,768$ nodes.

At such large scales, the thread level load balancing strategy is not sufficient for processing RMAT − 1 graphs and we employ the two-tiered strategy involving the vertex-splitting technique. The skew in the degree distribution for RMAT − 2 is sufficiently small so that we do not need to utilize load balancing procedures.

Nodes	1024	2,048	4,096	8,192	16,384	32768
RMAT − 1	173	331	653	1102	1870	3107
RMAT − 2	70	129	244	460	840	1480

Fig. 12. Performance on large systems

Figure 12 presents the performance (GTEPS) achieved by the final algorithms on the two families for system sizes ranging from $1,024$ to $32,768$ nodes, scale 33 to 39. We used the Δ values of 25 and 40 for the two families, respectively. 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 system sizes, and is also robust in terms of graph imbalances.

G. Evaluation of Push-Pull Decision Heuristic.

In this section, we discuss our validation of the efficacy of the push-pull decision heuristic. We designed an evaluation routine for checking 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 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 conducted the validation process on both graph families with $\Delta = 25$. We fixed the number of vertices per rank to be 2^{23} and varied the number of ranks from 32 to $1,024$. For each configuration, 16 random roots were generated and the validation was performed. We found that our heuristic made the best sequence of decisions on all the test cases.

H. Real Life Graphs.

In this section, we present a preliminary evaluation of our algorithm on three real life graphs related to social networks[7], shown in the table below. We compared our OPT algorithm with the baseline Δ-stepping algorithm on system sizes up to 128 nodes. For both algorithms, setting $\Delta = 40$ turned out to be the best configuration. The performance of the two algorithms is shown in the table below.

GTEPS	Vertices	Edges	Del-40	Opt-40
Friendster	63 million	1.8 billion	1.8	4.3
Orkut	3 million	117 million	2.1	4.6
Live Journal	4.8 million	68 million	1.1	2.2

It can be seen that OPT provides a factor two improvement in performance over the baseline algorithm. In the case of the Friendster graph, we also performed a scaling study up to 1024 nodes. The experiment showed that OPT scales well and yields 40 GTEPS on $1,024$ nodes, vs 20 GTEPS provided by the baseline algorithm.

[7] http://snap.stanford.edu

V. Conclusion

In this paper we presented a collection of parallel algorithms for the Single Source Shortest Path (SSSP) problem on distributed memory machines. We have used an arsenal of algorithmic techniques to reduce redundant communication and computation, minimizing the number of relaxations – the distance updates of each vertex. We have identified three main classes of optimizations: hybridization of Bellman-Ford and Delta stepping; various pruning techniques based on the edge weights, topological properties of the edges and direction optimization; in order to scale to thousands of nodes, we have applied careful thread-level and inter-node level load balancing and used lightweight communication libraries that provide direct thread-to-thread communication and optimized active messaging. The results obtained on $32,768$ nodes of Mira, the leadership supercomputer at Argonne National Laboratory, show an impressive processing rate of $3,100$ GTEPS on a scale 39 R-MAT graph with 2^{39} vertices and 2^{42} undirected edges.

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," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[2] L. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, March 1977.

[3] E. W. Dijkstra, "A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[4] R. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[5] L. A. Ford, "Network Flow Theory," Report P-923, The Rand Corporation, 1956.

[6] U. Meyer and P. Sanders, "Δ-stepping: A Parallelizable Shortest Path Algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, October 2003.

[7] 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.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proc. of the Fourth SIAM International Conference on Data Mining, (SDM '04), Lake Buena Vista, Florida, USA, April 2004*. SIAM, 2004, pp. 442–446.

[9] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker Graphs: An Approach to Modeling Networks," *Journal of Machine Learning Research*, vol. 11, pp. 985–1042, 2010.

[10] D. A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-Connectivity on the Cray MTA-2," in *Proc. of Intl. Conf. on Parallel Processing (ICPP '06), Columbus, Ohio, USA, August 2006*. IEEE Computer Society, 2006, pp. 523–530.

[11] 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. of Intl. Conf. on High Performance Computing Networking, Storage and Analysis, (SC '12), Salt Lake City, UT, USA - November 2012*. IEEE/ACM, 2012.

[12] http://www.graph500.org.

[13] 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. of the Ninth Workshop on Algorithm Engineering and Experiments, (ALENEX '07), New Orleans, Louisiana, USA, January 2007*. SIAM, 2007.

[14] S. Beamer, K. Asanovic, and D. A. Patterson, "Direction-optimizing Breadth-first Search," in *Proc. of Intl. Conf. on High Performance Computing Networking, Storage and Analysis, (SC '12), Salt Lake City, UT, USA - November 2012*. IEEE/ACM, 2012.

[15] M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, 1987.

[16] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine, "Single Source Shortest Paths with the Parallel Boost Graph Library," *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ*, November 2006.

[17] K. Nikas, N. Anastopoulos, G. I. Goumas, and N. Koziris, "Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm," in *Proc. of Intl Conf. on Parallel Processing, (ICPP '09), Vienna, Austria, September 2009*. IEEE Computer Society, 2009, pp. 388–395.

[18] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for Large-scale Graph algorithms," *Parallel Computing*, vol. 37, no. 9, pp. 610–632, September 2011.

[19] R. Paige and C. Kruskal, "Parallel Algorithms for Shortest Path Problems," in *Proc. of Intl. Conf. on Parallel Processing (ICPP '85), University Park, PA, USA, August 1985*. IEEE Computer Society, 1985, pp. 14–20.

[20] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation," *Communications of the ACM*, vol. 31, no. 11, pp. 1343–1354, November 1988.

[21] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis, "A Parallel Priority Data Structure with Applications," in *Proc. of the 11th Intl Parallel Processing Symposium (IPPS '97), April 1997, Geneva, Switzerland*. IEEE Computer Society, 1997, pp. 689–693.

[22] M. Thorup, "Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time," *Journal of the ACM*, vol. 46, no. 3, pp. 362–394, May 1999.

[23] 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, April 1997.

[24] P. Harish and P. J. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *Proc. of the 14th Intl Conf. on High Performance Computing, (HiPC '07), Goa, India, December 2007*, ser. Lecture Notes in Computer Science, vol. 4873. Springer, 2007, pp. 197–208.

[25] J. Kleinberg and E. Tardos, *Algorithm design*. Addison-Wesley, 2006.

[26] D. Chen, N. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker, "The IBM Blue Gene/Q Interconnection Fabric," *IEEE Micro*, vol. 32, no. 1, pp. 32–43, March-April 2012.

[27] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A., G. Chiu, P. Boyle, N. Christ, and C. Kim, "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, March-April 2012.

[28] D. Chen, N. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. D. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Network and Message Unit," in *Proc. of Intl. Conf. on High Performance Computing Networking, Storage and Analysis, (SC '11), Seattle, WA, USA, November 2011*. ACM, 2011.