An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances

Kamesh Madduri* David A. Bader* Jonathan W.Berry† Joseph R. Crobak‡

Abstract

We present an experimental study of the single source shortest path problem with non-negative edge weights (NSSP) on large-scale graphs using the Δ-stepping parallel algorithm. We report performance results on the Cray MTA-2, a multithreaded parallel computer. The MTA-2 is a high-end shared memory system offering two unique features that aid the efficient parallel implementation of irregular algorithms: the ability to exploit fine-grained parallelism, and low-overhead synchronization primitives. Our implementation exhibits remarkable parallel speedup when compared with competitive sequential algorithms, for low-diameter sparse graphs. For instance, Δ-stepping on a directed scale-free graph of 100 million vertices and 1 billion edges takes less than ten seconds on 40 processors of the MTA-2, with a relative speedup of close to 30. To our knowledge, these are the first performance results of a shortest path problem on realistic graph instances in the order of billions of vertices and edges.

1 Introduction

We present an experimental study of the Δ-stepping parallel algorithm [49] for solving the single source shortest path problem on large-scale graph instances. In addition to applications in combinatorial optimization problems, shortest path algorithms are finding increasing relevance in the domain of complex network analysis. Popular graph theoretic analysis metrics such as betweenness centrality [25, 9, 39, 41, 32] are based on shortest path algorithms. Our parallel implementation targets graph families that are representative of real-world, large-scale networks [7, 22, 12, 51, 50]. Real-world graphs are typically characterized by a low diameter, heavy-tailed degree distributions modeled by power laws, and self-similarity. They are often very large, with the number of vertices and edges ranging from several hundreds of thousands to billions. On current workstations, it is not possible to do exact in-core computations on these graphs due to the limited physical memory. In such cases, parallel computing techniques can be applied to obtain exact solutions for memory and compute-intensive graph problems quickly. For instance, recent experimental studies on Breadth-First Search for large-scale graphs show that a parallel in-core implementation is two orders of magnitude faster than an optimized external memory implementation [5, 2]. In this paper, we present an efficient parallel implementation for the single source shortest paths problem that can handle scale-free instances in the order of billions of edges. In addition, we conduct an experimental study of performance on several other graph families, also used in the 9th DIMACS Implementation Challenge [17] on Shortest Paths. Please refer to our technical report [43] for additional performance details.

Sequential algorithms for the single source shortest path problem with non-negative edge weights (NSSP) are studied extensively, both theoretically [20, 18, 23, 24, 54, 56, 33, 30, 46] and experimentally [19, 28, 27, 15, 59, 29]. Let m and n denote the number of edges and vertices in the graph respectively. Nearly all NSSP algorithms are based on the classical Dijkstra's [20] algorithm. Using Fibonacci heaps [23], Dijkstra's algorithm can be implemented in $O(m + n \log n)$ time. Thorup [56] presents an $O(m + n)$ RAM algorithm for undirected graphs that differs significantly different from Dijkstra's approach. Instead of visiting vertices in the order of increasing distance, it traverses a *component tree.* Meyer [47] and Goldberg [29] propose simple algorithms with linear average time for uniformly distributed edge weights.

Parallel algorithms for solving NSSP are reviewed in detail by Meyer and Sanders [46, 49]. There are no known PRAM algorithms that run in sub-linear time and $O(m + n \log n)$ work. Parallel priority queues [21, 11] for implementing Dijkstra's algorithm have been developed, but these linear work algorithms have a worst-case time bound of $\Omega(n)$, as they only perform edge relaxations in parallel. Several matrix-multiplication based algorithms [34, 26], proposed for the parallel All-Pairs Shortest Paths (APSP), involve running time and efficiency trade-offs. Parallel approximate NSSP algorithms [40, 16, 55] based on the randomized Breadth-First search algorithm of Ullman and Yannakakis [58]

*Georgia Institute of Technology

†Sandia National Laboratories

‡Rutgers University

run in sub-linear time. However, it is not known how to use the Ullman-Yannakakis randomized approach for exact NSSP computations in sub-linear time.

Meyer and Sanders give the Δ-stepping [49] NSSP algorithm that divides Dijkstra's algorithm into a number of *phases*, each of which can be executed in parallel. For random graphs with uniformly distributed edge weights, this algorithm runs in sub-linear time with linear average case work. Several theoretical improvements [48, 44, 45] are given for Δ-stepping (for instance, finding shortcut edges, adaptive bucket-splitting), but it is unlikely that they would be faster than the simple Δ-stepping algorithm in practice, as the improvements involve sophisticated data structures that are hard to implement efficiently. On a random d-regular graph instance (2^{19} vertices and $d = 3$), Meyer and Sanders report a speedup of 9.2 on 16 processors of an Intel Paragon machine, for a distributed memory implementation of the simple Δ-stepping algorithm. For the same graph family, we are able to solve problems three orders of magnitude larger with near-linear speedup on the Cray MTA-2. For instance, we achieve a speedup of 14.82 on 16 processors and 29.75 on 40 processors for a random d-regular graph of size 2^{29} vertices and d set to 3.

The literature contains few experimental studies on parallel NSSP algorithms [35, 52, 37, 57]. Prior implementation results on distributed memory machines resorted to graph partitioning [14, 1, 31], and running a sequential NSSP algorithm on the sub-graph. Heuristics are used for load balancing and termination detection [36, 38]. The implementations perform well for certain graph families and problem sizes, but in the worst case, there is no speedup.

Implementations of PRAM graph algorithms for arbitrary sparse graphs are typically memory intensive, and the memory accesses are fine-grained and highly irregular. This often leads to poor performance on cache-based systems. On distributed memory clusters, few parallel graph algorithms outperform the best sequential implementations due to long memory latencies and high synchronization costs [4, 3]. Parallel shared memory systems are a more supportive platform. They offer higher memory bandwidth and lower latency than clusters, and the global shared memory greatly improves developer productivity. However, parallelism is dependent on the cache performance of the algorithm [53] and scalability is limited in most cases.

We present our shortest path implementation results on the Cray MTA-2, a massively multithreaded parallel machine. The MTA-2 is a high-end shared memory system offering two unique features that aid considerably in the design of irregular algorithms: fine-grained parallelism and low-overhead word-level synchronization. The MTA-2 has no data cache; rather than using a memory hierarchy to reduce latency, the MTA-2 processors use hardware multithreading to tolerate the latency. The word-level synchronization support complements multithreading and makes performance primarily a function of parallelism. Since graph algorithms have an abundance of parallelism, yet often are not amenable to partitioning, the MTA-2 architectural features lead to superior performance and scalability. Our recent results highlight the exceptional performance of the MTA-2 for implementations of key combinatorial optimization and graph theoretic problems such as list ranking [3], connected components [3, 8], subgraph isomorphism [8], Breadth-First Search and *st*-connectivity [5].

The main contributions of this paper are as follows:

- *An experimental study of solving the single-source shortest paths problem in parallel using the Δ-stepping algorithm.* Prior studies have predominantly focused on running sequential NSSP algorithms on graph families that can be easily partitioned, whereas we also consider several arbitrary, sparse graph instances. We also analyze performance using machine-independent algorithmic operation counts.

- *Demonstration of the power of massive multithreading for graph algorithms on highly unstructured instances.* We achieve impressive performance on low-diameter random and scale-free graphs.

- *Solving NSSP for large-scale realistic graph instances in the order of billions of edges.* Δ-stepping on a synthetic directed scale-free graph of 100 million vertices and 1 billion edges takes 9.73 seconds on 40 processors of the MTA-2, with a relative speedup of approximately 31. These are the first results that we are aware of, for solving instances of this scale and also achieving near-linear speedup. Also, the sequential performance of our implementation is comparable to competitive NSSP implementations.

2 Review of the Δ-stepping Algorithm

Let $G = (V, E)$ be a graph with n vertices and m edges. Let $s \in V$ denote the source vertex. Each edge $e \in E$ is assigned a non-negative real weight by the length function $l : E \to \mathbb{R}$. Define the *weight of a path* as the sum of the weights of its edges. The single source shortest paths problem with non-negative edge weights (NSSP) computes $\delta(v)$, the weight of the *shortest* (minimum-weighted) path from s to v.

$\delta(v) = \infty$ if v is unreachable from s. We set $\delta(s) = 0$.

Most shortest path algorithms maintain a *tentative distance* value for each vertex, which are updated by *edge relaxations*. Let $d(v)$ denote the tentative distance of a vertex v. $d(v)$ is initially set to ∞, and is an upper bound on $\delta(v)$. *Relaxing* an edge $\langle v, w\rangle \in E$ sets $d(w)$ to the minimum of $d(w)$ and $d(v) + l(v, w)$. Based on the manner in which the tentative distance values are updated, most shortest path algorithms can be classified into two types: *label-setting* or *label-correcting*. Label-setting algorithms (for instance, Dijkstra's algorithm) perform relaxations only from *settled* ($d(v) = \delta(v)$) vertices, and compute the shortest path from s to all vertices in exactly m edge relaxations. Based on the values of $d(v)$ and $\delta(v)$, at each iteration of a shortest path algorithm, vertices can be classified into *unreached* ($d(v) = \infty$), *queued* ($d(v)$ is finite, but v is not settled) or *settled*. Label-correcting algorithms (e.g., Bellman-Ford) relax edges from unsettled vertices also, and may perform more than m relaxations. Also, all vertices remain in a *queued* state until the final step of the algorithm. Δ-stepping belongs to the label-correcting type of shortest path algorithms.

The Δ-stepping algorithm (see Alg. 1) is an "approximate bucket implementation of Dijkstra's algorithm" [49]. It maintains an array of buckets B such that $B[i]$ stores the set of vertices $\{v \in V : v \text{ is queued and } d(v) \in [i\Delta, (i + 1)\Delta)\}$. Δ is a positive real number that denotes the "bucket width".

In each *phase* of the algorithm (the inner *while* loop in Alg. 1, lines 9–14, when bucket $B[i]$ is not empty), all vertices are removed from the current bucket, added to the set S, and *light* edges ($l(e) \leq \Delta$, $e \in E$) adjacent to these vertices are relaxed (see Alg. 2). This may result in new vertices being added to the current bucket, which are deleted in the next phase. It is also possible that vertices previously deleted from the current bucket may be reinserted, if their tentative distance is improved. *Heavy* edges ($l(e) > \Delta$, $e \in E$) are not relaxed in a phase, as they result in tentative values outside the current bucket. Once the current bucket remains empty after relaxations, all heavy edges out of the vertices in S are relaxed at once (lines 15–17 in Alg. 1). The algorithm continues until all the buckets are empty.

Observe that edge relaxations in each phase can be done in parallel, as long as individual tentative distance values are updated atomically. The number of phases bounds the parallel running time, and the number of *reinsertions* (insertions of vertices previously deleted) and *rerelaxations* (relaxation of their out-going edges) costs an overhead over Dijkstra's algorithm. The performance of the algorithm also depends on the value of the bucket-width Δ. For $\Delta = \infty$, the algorithm is

Algorithm 1: Δ-stepping algorithm

```
   Input: $G(V, E)$, source vertex $s$, length function $l : E \rightarrow \mathbb{R}$
   Output: $\delta(v), v \in V$, the weight of the shortest path from $s$ to $v$

 1 foreach $v \in V$ do
 2     $heavy(v) \longleftarrow \{\langle v, w\rangle \in E : l(v, w) > \Delta\}$;
 3     $light(v) \longleftarrow \{\langle v, w\rangle \in E : l(v, w) \leq \Delta\}$;
 4     $d(v) \longleftarrow \infty$;
 5 relax($s$, 0);
 6 $i \longleftarrow 0$;
 7 while $B$ is not empty do
 8     $S \longleftarrow \phi$;
 9     while $B[i] \neq \phi$ do
10         $Req \longleftarrow \{(w, d(v) + l(v, w)) : v \in B[i] \wedge \langle v, w\rangle \in light(v)\}$;
11         $S \longleftarrow S \cup B[i]$;
12         $B[i] \longleftarrow \phi$;
13         foreach $(v, x) \in Req$ do
14             relax($v$, $x$);
15     $Req \longleftarrow \{(w, d(v) + l(v, w)) : v \in S \wedge \langle v, w\rangle \in heavy(v)\}$;
16     foreach $(v, x) \in Req$ do
17         relax($v$, $x$);
18     $i \longleftarrow i + 1$;
19 foreach $v \in V$ do
20     $\delta(v) \longleftarrow d(v)$;
```

Algorithm 2: The *relax* routine in the Δ-stepping algorithm

```
Input: $v$, weight request $x$
Output: Assignment of $v$ to appropriate bucket

if $x < d(v)$ then
    $B[\lfloor d(v)/\Delta \rfloor] \leftarrow B[\lfloor d(v)/\Delta \rfloor] \setminus \{v\}$;
    $B[\lfloor x/\Delta \rfloor] \leftarrow B[\lfloor x/\Delta \rfloor] \cup \{v\}$;
    $d(v) \leftarrow x$;
```

similar to the Bellman-Ford algorithm. It has a high degree of parallelism, but is inefficient compared to Dijkstra's algorithm. Δ-stepping tries to find a good compromise between the number of parallel phases and the number of re-insertions. Theoretical bounds on the number of phases and re-insertions, and the average case analysis of the parallel algorithm are presented in [49]. We summarize the salient results.

Let d_c denote the maximum shortest path weight, and P_Δ denote the set of paths with weight at most Δ. Define a parameter l_{max}, an upper bound on the maximum number of edges in any path in P_Δ. The following results hold true for any graph family.

- The number of buckets in B is $\lceil d_c/\Delta \rceil$.
- The total number of reinsertions is bounded by $|P_\Delta|$, and the total number of rerelaxations is bounded by $|P_{2\Delta}|$.
- The number of phases is bounded by $\frac{d_c}{\Delta} l_{max}$, i.e., no bucket is expanded more than l_{max} times.

For graph families with random edge weights and a maximum degree of d, Meyer and Sanders [49] theoretically prove that $\Delta = \theta(1/d)$ is a good compromise between work efficiency and parallelism. The sequential algorithm performs $O(dn)$ expected work divided between $O(\frac{d_c}{\Delta} \cdot \frac{\log n}{\log \log n})$ phases *with high probability*. In practice, in case of graph families for which d_c is $O(\log n)$ or $O(1)$, the parallel implementation of Δ-stepping yields sufficient parallelism for our parallel system.

3 Parallel Implementation of Δ-stepping

The bucket array B is the primary data structure used by the parallel Δ-stepping algorithm. We implement individual buckets as *dynamic arrays* that can be resized when needed and iterated over easily. To support constant time insertions and deletions, we maintain two auxiliary arrays of size n: a mapping of the vertex ID to its current bucket, and a mapping from the vertex ID to the position of the vertex in the current bucket (see Fig. 1 for an illustration). All new vertices are added to the end of the array, and deletions of vertices are done by setting the corresponding locations in the bucket and the mapping arrays to -1. Note that once bucket i is finally empty after a light edge relaxation phase, there will be no more insertions into the bucket in subsequent phases. Thus, the memory can be reused once we are done relaxing the light edges in the current bucket. Also observe that all the insertions are done in the relax routine, which is called once in each phase, and once for relaxing the heavy edges.

We implement a timed pre-processing step to *semi-sort* the edges based on the value of Δ. All the light edges adjacent to a vertex are identified in parallel and stored in contiguous virtual locations, and so we visit only light edges in a phase. The $O(n)$ work pre-processing step scales well in parallel on the MTA-2.

We also support fast parallel insertions into the request set R. R stores $\langle v, x \rangle$ pairs, where $v \in V$ and x is the requested tentative distance for v. We add a vertex v to R only if it satisfies the condition $x < d(v)$. We do not store duplicates in R. We use a sparse set representation similar to one used by Briggs and Torczon [10] for storing vertices in R. This sparse data structure uses two arrays of size n: a *dense* array that contiguously stores the elements of the set, and a *sparse* array that indicates whether the vertex is a member of the set. Thus, it is easy to iterate over the request set, and membership queries and insertions are constant time. Unlike other Dijkstra-based algorithms, we do not relax edges in one step. Instead, we inspect adjacencies (light edges) in each phase, construct a request set of vertices, and then relax *vertices* in the relax step.

All vertices in the request set R are relaxed in parallel in the relax routine. In this step, we first delete a vertex from the old bucket, and then insert it into the new bucket. Instead of performing individual insertions, we first determine the expansion factor of each bucket, expand the buckets, and add then all vertices into their new buckets in one step. Since there are no duplicates in the request set, no synchronization is involved for updating the tentative distance values.

To saturate the MTA-2 processors with work and to obtain high system utilization, we need to minimize the number of phases and non-empty buckets, and maximize the request set sizes. Entering and exiting a parallel phase involves a negligible running time overhead in practice. However, if the number of phases is $O(n)$, this overhead dominates the actual running time of the implementation. Also, we enter the relax routine once every phase. The number of implicit barrier synchronizations in the algorithm is proportional to the number of phases. Our implementation reduces the number of barriers. Our source code for the Δ-stepping implementation, along with the MTA-2 graph generator ports, is freely available online [42].

4 Experimental Setup

4.1 Platforms We report parallel performance results on a 40-processor Cray MTA-2 system with 160 GB uniform shared memory. Each processor has a clock speed of 220 MHz and support for 128 hardware threads. The Δ-stepping code is written in C with MTA-2 specific pragmas and directives for parallelization. We com-

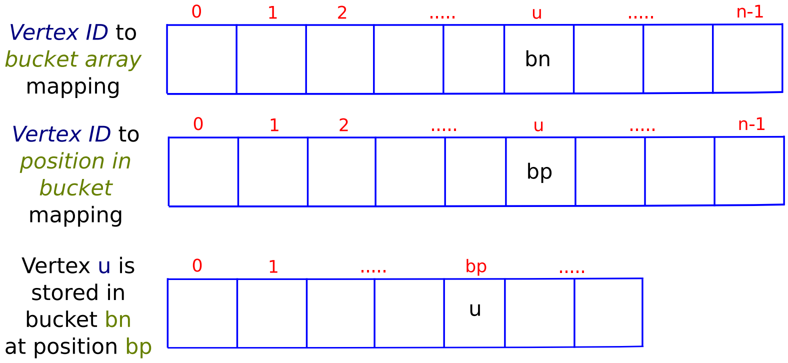

Vertex ID to bucket array mapping
0
1
2
.....
u
.....
n-1
bn
Vertex ID to position in bucket mapping
0
1
2
.....
u
.....
n-1
bp
Vertex u is stored in bucket bn at position bp
0
1
.....
bp
.....
u

Figure 1: Bucket array and auxiliary data structures

pile it using the MTA-2 C compiler (Cray Programming Environment (PE) 2.0.3) with -O3 and -par flags.

The MTA-2 code also compiles and runs on sequential processors without any modifications. Our test platform for the sequential performance results is one processor of a dual-core 3.2 GHz 64-bit Intel Xeon machine with 6GB memory, 1MB cache and running RedHat Enterprise Linux 4 (linux kernel 2.6.9). We compare the sequential performance of our implementation with the DIMACS reference solver [17]. Both the codes are compiled with the Intel C compiler (icc) Version 9.0, with the flags `-O3`.

4.2 Problem Instances We evaluate sequential and parallel performance on several graph families. Some of the generators and graph instances are part of the DIMACS Shortest Path Implementation Challenge benchmark package [17]:

- *Random graphs*: Random graphs are generated by first constructing a Hamiltonian cycle, and then adding $m - n$ edges to the graph at random. The generator may produce parallel edges as well as self-loops. We define the random graph family *Random4-n* such that n is varied, $\frac{m}{n} = 4$, and the edge weights are chosen from a uniform random distribution.

- *Grid graphs*: This synthetic generator produces two-dimensional meshes with grid dimensions x and y. *Long-n* ($x = \frac{n}{16}$, $y = 16$) and *Square-n* grid ($x = y = \sqrt{n}$) families are defined, similar to random graphs.

- *Road graphs*: Road graph families with transit time (*USA-road-t*) and distance (*USA-road-d*) as the length function.

In addition, we also study the following families:

- *Scale-free graphs*: We use the R-MAT graph model [13] for real-world networks to generate scale-free graphs. We define the family *ScaleFree4-n* similar to random graphs.

- *Log-uniform weight distribution*: The above graph generators assume randomly distributed edge weights. We report results for an additional *log-uniform* distribution also. The generated integer edge weights are of the form 2^i, where i is chosen from the uniform random distribution $[1, \log C]$ (C denotes the maximum integer edge weight). We define *Random4logUnif-n* and *ScaleFree4logUnif-n* families for this weight distribution.

4.3 Methodology For sequential runs, we report the execution time of the reference DIMACS NSSP solver (an efficient implementation of Goldberg's algorithm [30], which has expected-case linear time for some inputs) and the baseline Breadth-First Search (BFS) on every graph family. The BFS running time is a natural lower bound for NSSP codes and is a good indicator of how optimized the shortest path implementations are. It is reasonable to directly compare the execution times of the reference code and our implementation: both use a similar adjacency array representation for the graph, are written in C, and compiled and run in identical experimental settings. Note that our implementation is optimized for the MTA-2 and we make no modifications to the code before running on a sequential machine. The time taken for semi-sorting and mechanisms to reduce memory contention on the MTA-2 both constitute overhead on a sequential processor. Also, our implementation assumes real-weighted edges, and we cannot use fast bitwise operations. By default, we set the value of Δ to $\frac{n}{m}$ for all graph instances. We will show that this choice of Δ may not be optimal for all graph classes and weight distributions.

On a sequential processor, we execute the BFS and shortest path codes on all the core graph families, for the recommended problem sizes. However, for parallel runs, we only report results for sufficiently large graph instances in case of the synthetic graph families. We parallelize the synthetic core graph generators and port them to run on the MTA-2.

Our implementations accept both directed and undirected graphs. For all the synthetic graph instances, we report execution times on directed graphs in this paper. The road networks are undirected graphs. We also assume the edge weights to be distributed in $[0, 1]$ in the Δ-stepping implementation. So we have a pre-processing step to scale the integer edge weights in the core problem families to the interval [0, 1], dividing the integer weights by the maximum edge weight.

On the MTA-2, we compare our implementation running time with the execution time of a multithreaded level-synchronized breadth-first search [6], optimized for low-diameter graphs. The multithreaded BFS scales as well as Δ-stepping for all the graph instances considered, and the execution time serves as a lower bound for the shortest path running time.

The first run on the MTA-2 is usually slower than subsequent ones (by about 10% for a typical Δ-stepping run). So we report the average running time for 10 successive runs. We run the code from three randomly chosen source vertices and average the running time. We found that using three sources consistently gave us execution time results with little variation on both the MTA-2 and the reference sequential platform. We tabulate the sequential and parallel performance metrics in [43], and report execution time in seconds.

5 Results and Analysis

5.1 Sequential Performance First we present the performance results of our implementation on the reference sequential platform, experimenting with various graph families. Fig. 2 compares the execution time across graph instances of the same size, but from different families. The DIMACS reference code is about 1.5 to 2 times faster than our implementation for large problem instances in each family. The running time on the *Random4-n* is slightly higher than the rest of the families. For additional details such as performance as we vary the problem size for BFS, Δ-stepping, and the DIMACS implementation execution times, please refer to Section B.1 of [43]. Our key observation is that the ratio of the Δ-stepping execution time to the BFS time varies between 3 and 10 across different problem instances.

5.2 Δ-stepping analysis To better understand the algorithm performance across graph families, we use machine-independent algorithm operation counts. The parallel performance is dependent on the value of Δ, the number of phases, the size of the request set in each phase. Fig. 3 plots the size of the light request set in each phase, for different graph families. By default, Δ is set to 0.25 for all runs. If the request set size is less than 10, it is not plotted. Consider the random graph family (Fig. 3(a)). It executes in 84 phases, and the request set sizes vary from 0 to 27,000. Observe the recurring pattern of three bars stacked together in the plot. This indicates that all the light edges in a bucket are relaxed in roughly three phases, and the bucket then becomes empty. The size of the relax set is relatively high for several phases, which provides scope for exploiting multithreaded parallelism. The relax set sizes of a similar problem instance from the Long grid family (Fig. 3(b)) stands in stark contrast to that of the random graph. It takes about 200,000 phases to execute, and the maximum request size is only 15. Both of these values indicate that our implementation would fare poorly on long grid graphs (e.g. meshes with a very high aspect ratio). On square grids (Fig. 3(c)), Δ-stepping takes fewer phases, and the request set sizes go up to 500. For a road network instance (NE USA-road-d, Fig. 3(d)), the algorithm takes 23,000 phases to execute, and only a few phases (about 30) have request sets greater than 1000 in size.

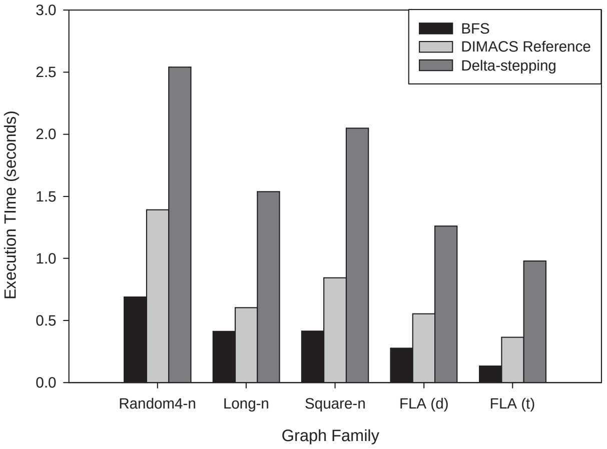
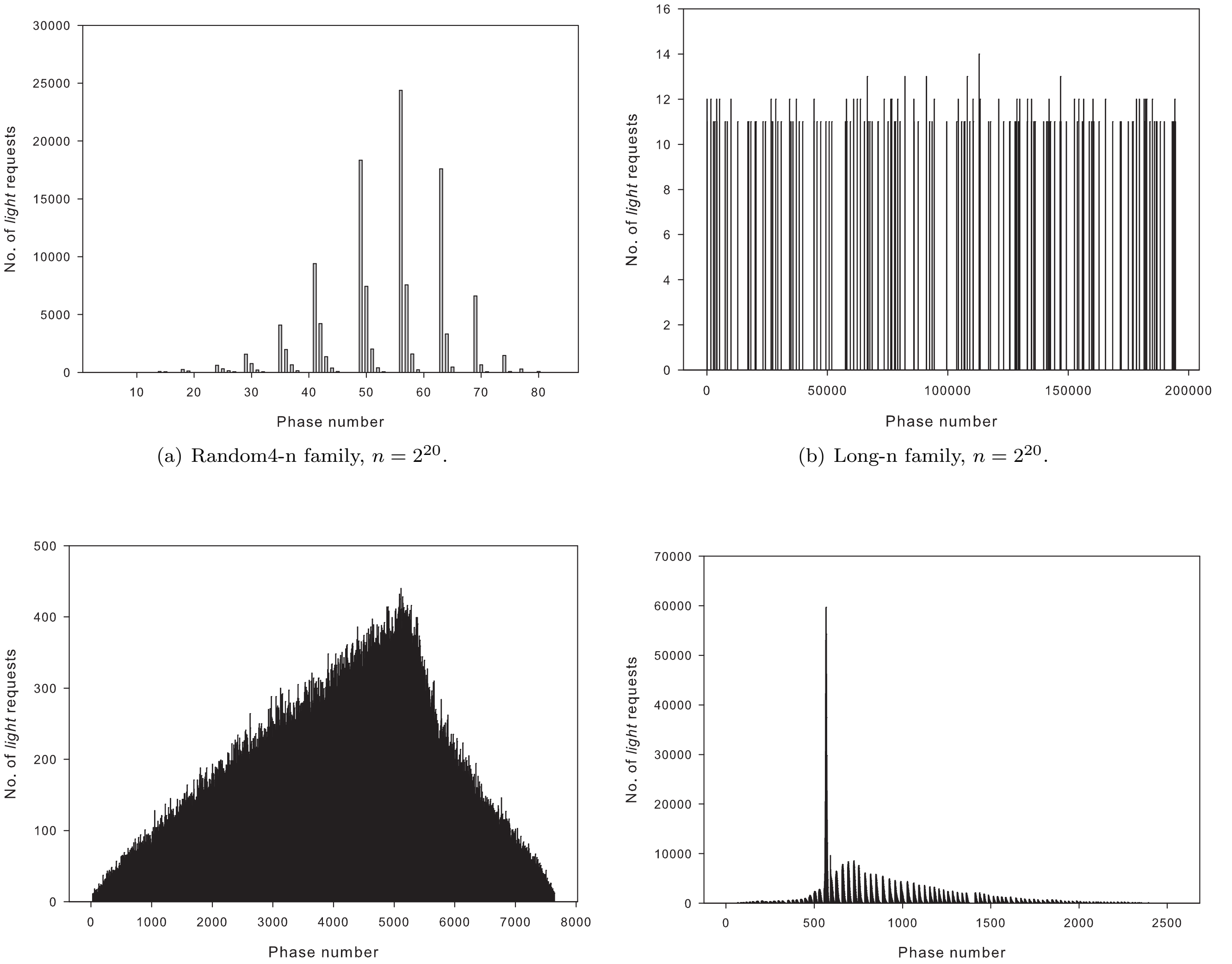


Figure 2: Sequential performance of our Δ-stepping implementation on the core graph families. All the synthetic graphs are directed, with 2^{20} vertices and $\frac{m}{n} \approx 4$. FLA(d) and FLA(t) are road networks corresponding to Florida, with 1070376 vertices and 2712768 edges

Fig. 4 plots several key Δ-stepping operation counts for various graph classes. All synthetic graphs are roughly of the same size. Fig. 4(a) plots the average



(a) Random4-n family, $n = 2^{20}$.

(b) Long-n family, $n = 2^{20}$.

(c) Square-n family, $n = 2^{20}$.

(d) USA-road-d family, Northeast USA (NE). $n = 1524452$, $m = 3897634$.

Figure 3: Δ-stepping algorithm: Size of the light request set at the end of each phase, for the core graph families. Request set sizes less than 10 are not plotted.

shortest path weight for various graph classes. Scale-free and Long grid graphs are on the two extremes. A log-uniform edge weight distribution also results in low average edge weight. The number of phases (see Fig. 4(b)) is highest for Long grid graphs. The number of buckets shows a similar trend as the average shortest path weight. Fig. 4(d) plots the total number of insertions for each graph family. The number of vertices is 2^{20} for all graph families (slightly higher for the road network), and so Δ-stepping results in roughly 20% overhead in insertions for all the graph families with random edge weights. Note the number of insertions for graphs with log-uniform weight distributions. Δ-stepping performs a significant amount of excess work for these families, because the value of Δ is quite high for this particular distribution.

We next evaluate the performance of the algorithm as Δ is varied (tables in Section B.2). Fig. 5 plots the execution time of various graph instances on a sequential machine, and one processor of the MTA-2. Δ is varied from 0.1 to 10 in each case. We find that the absolute running times on a 3.2 GHz Xeon processor and the MTA-2 are comparable for random, square grid and road network instances. However, on long grid graphs (Fig. 5(b)), the MTA-2 execution time is two orders of magnitude greater than the sequential time. The number of phases and the total number of relaxations vary as Δ is varied (See Section B.2 in [43]). On the MTA-2, the running time is not only dependent on the work done, but also on the number of phases and the average number of relax requests in a phase. For instance, in the case of long grids (see Fig. 5(b), with execution time plotted on a log scale), the running time decreases significantly as the value of Δ is decreased, as the number of phases reduce. On a sequential processor, however, the running time is only dependent on the work done (number of insertions). If the value of Δ is greater than the average shortest path weight, we perform excess work and the running time noticeably increases (observe the execution time for $\Delta = 5, 10$ on the random graph and the road network). The optimal value of Δ (and the execution time on the MTA-2) is also dependent on the number of processors. For a particular Δ, it may be possible to saturate a single processor of the MTA-2 with the right balance of work and phases. The execution time on a 40-processor run may not be minimal with this value of Δ.

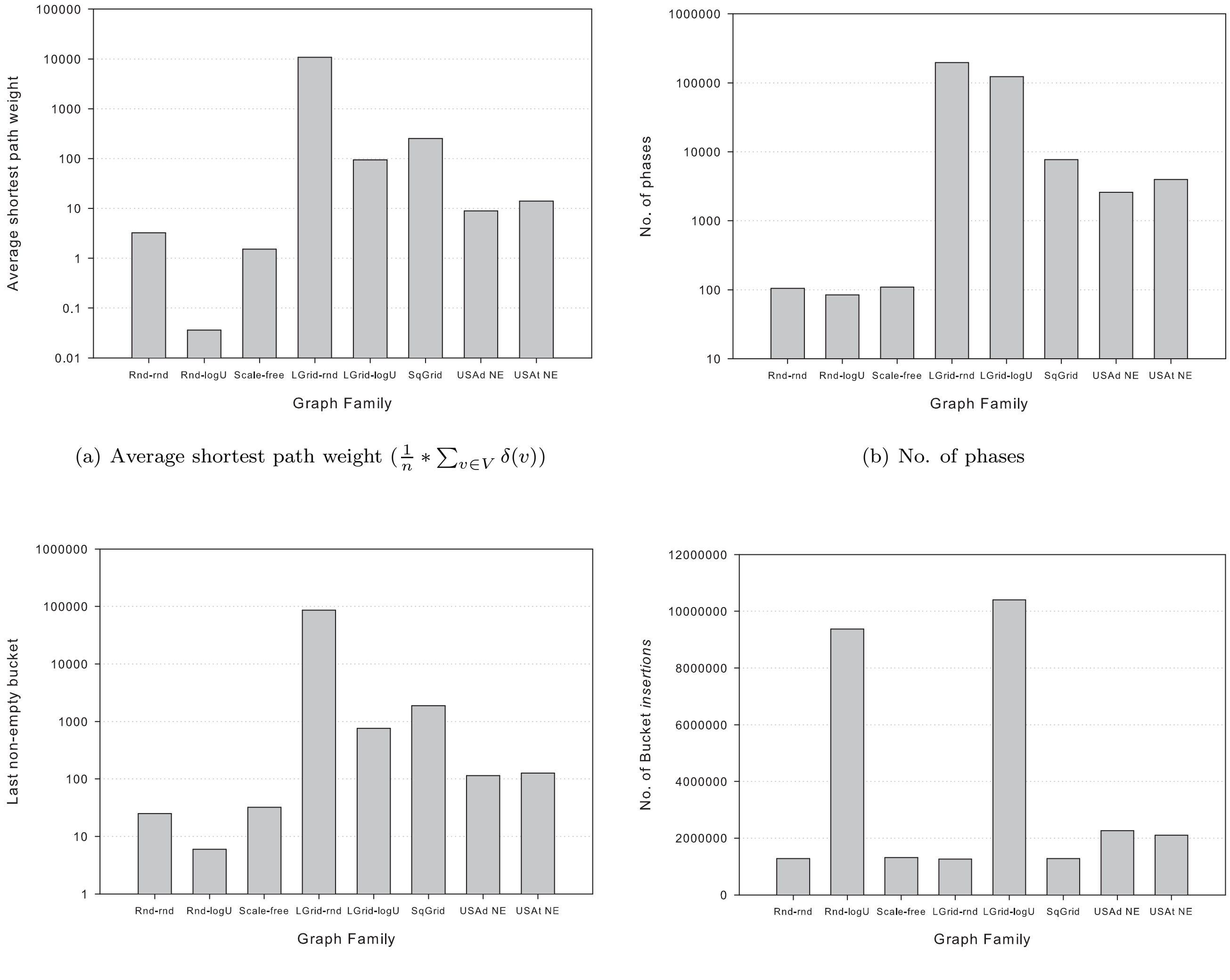
5.3 Parallel Performance We present the parallel scaling of the Δ-stepping algorithm in detail. We ran Δ-stepping and the level-synchronous parallel BFS on all graph instances described in Section 4.2 (see [43] for complete tabular results from all experiments). We define the speedup on p processors of the MTA-2 as the ratio of the execution time on 1 processor to the execution time on p processors. In all graph classes except long grids, there is sufficient parallelism to saturate a single processor of the MTA-2 for reasonably large problem instances.

As expected, Δ-stepping performs best for low-diameter random and scale-free graphs with randomly distributed edge weights (see Fig. 6(a) and 6(b)). We achieve a speedup of approximately 31 on 40 processors for a directed random graph of nearly a billion edges, and the ratio of the BFS and Δ-stepping execution time is a constant factor (about 3-5) throughout. The implementation performs equally well for scale-free graphs, that are more difficult for partitioning-based parallel computing models to handle due to the irregular degree distribution. The execution time on 40 processors of the MTA-2 for the scale-free graph instance is within 9% (a difference of less than one second) of the running time for a random graph and the speedup is approximately 30 on 40 processors. We have already shown that the execution time for smaller graph instances on a sequential machine is comparable to the DIMACS reference implementation, a competitive NSSP algorithm. Thus, achieving a speedup of 30 for a realistic scale-free graph instance of one billion edges (Fig. 6(b)) is a substantial result. To our knowledge, these are the first results to demonstrate near-linear speedup for such large-scale unstructured graph instances.

In case of all the graph families, the relative speedup increases as the problem size is increased (for e.g., on 40 processors, the speedup for a *Random4-n* instance with $n = 2^{21}$ is just 3.96, whereas it is 31.04 for 2^{28} vertices). This is because there is insufficient parallelism in a problem instance of size 2^{21} to saturate 40 processors of the MTA-2. As the problem size increases, the ratio of Δ-stepping execution time to multithreaded BFS running time decreases. On an average, Δ-stepping is 5 times slower than BFS for this graph family.

For random graphs with a log-uniform weight distribution, Δ set to $\frac{n}{m}$ results in a significant amount of additional work. The Δ-stepping to BFS ratio is typically 40 in this case, about 8 times higher than the corresponding ratio for random graphs with random edge weights. However, the execution time scales well with the number of processors for large problem sizes.

In case of *Long-n* graphs and Δ set to $\frac{n}{m}$, there is insufficient parallelism to fully utilize even a single processor of the MTA-2. The execution time of the level-synchronous BFS also does not scale with the number of processors. In fact, the running time goes up in case of multiprocessor runs, as the parallelization overhead becomes significant. Note that the execution time on a



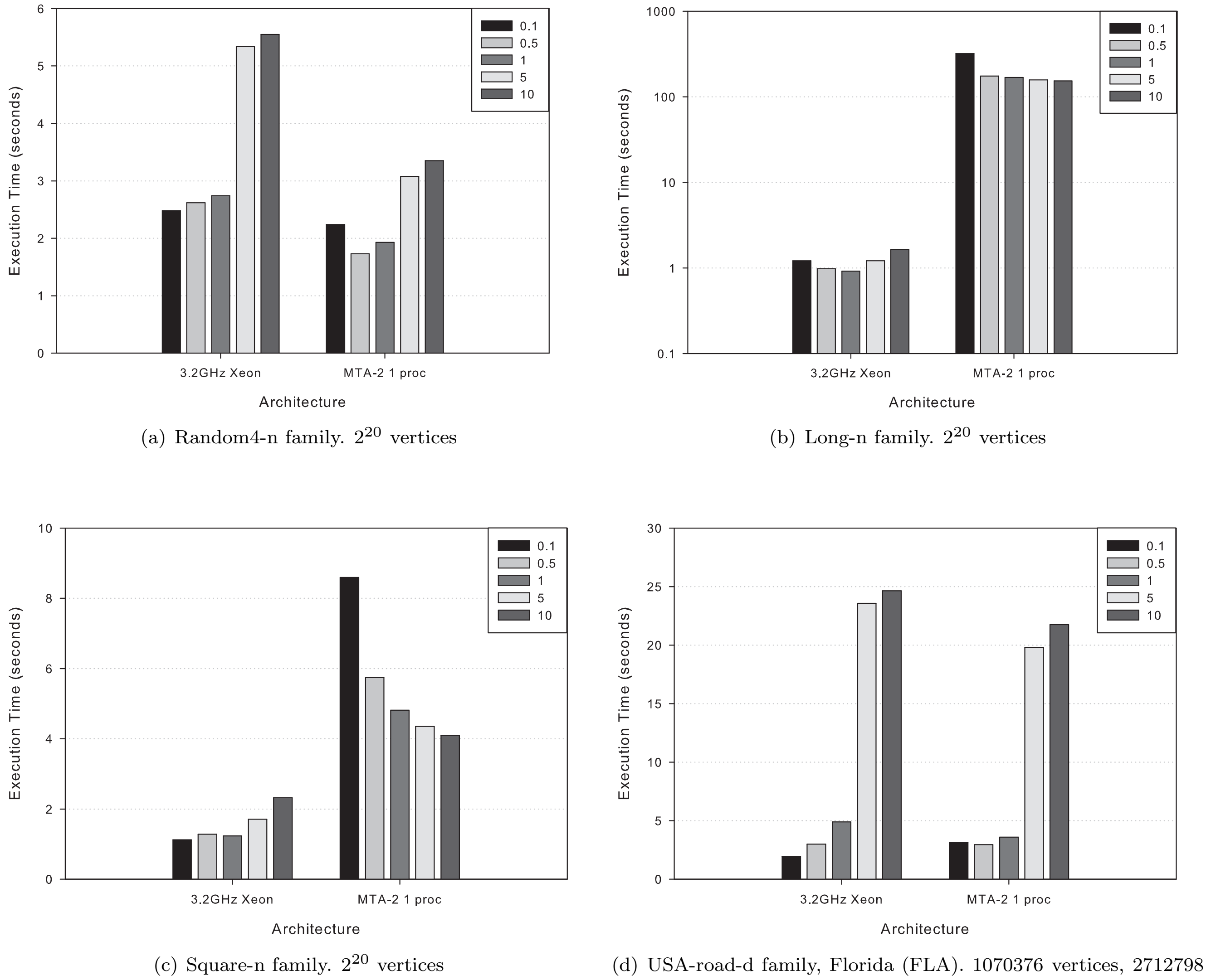
(a) Average shortest path weight ($\frac{1}{n} * \sum_{v \in V} \delta(v)$)

(b) No. of phases

(c) Last non-empty bucket

(d) Number of relax requests

Figure 4: Δ-stepping algorithm performance statistics for various graph classes. All synthetic graph instances have n set to 2^{20} and $m \approx 4n$. Rand-rnd: Random graph with random edge weights, Rand-logU: Random graphs with log-uniform edge weights, Scale-free: Scale-free graph with random edge weights, Lgrid: Long grid, SqGrid: Square grid, USA NE: 1524452 vertices, 3897634 edges. Plots (a), (b), (c) are on a log scale, while (d) uses a linear scale.



(a) Random4-n family. 2^{20} vertices

(b) Long-n family. 2^{20} vertices

(c) Square-n family. 2^{20} vertices

(d) USA-road-d family, Florida (FLA). 1070376 vertices, 2712798 edges

Figure 5: A comparison of the execution time on the reference sequential platform and a single MTA-2 processor, as the bucket-width Δ is varied.

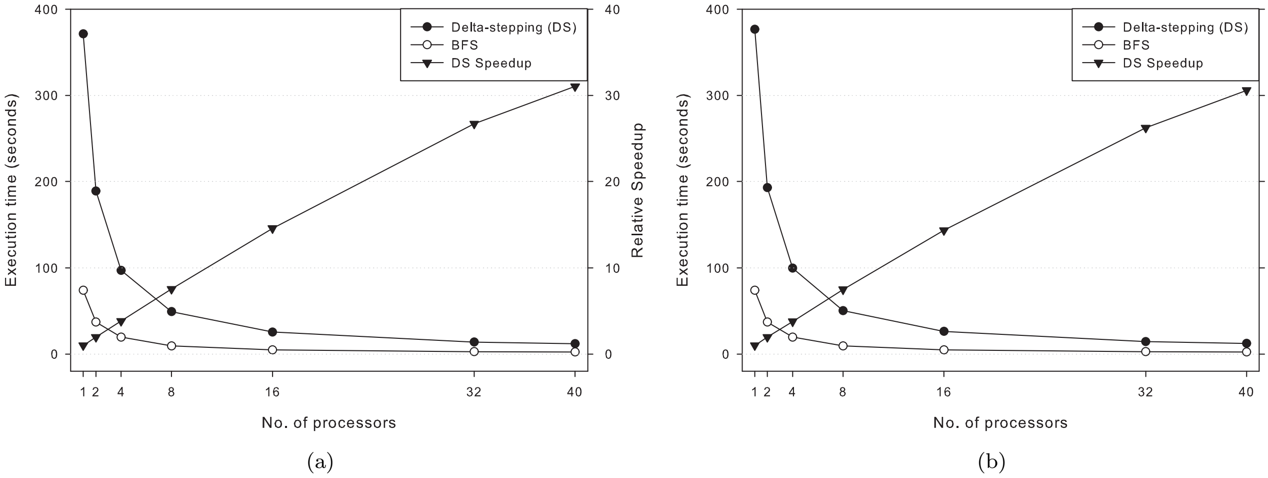


Figure 6: Δ-stepping execution time and relative speedup on the MTA-2 for Random4-n (left) and ScaleFree4-n (right) graph instances (directed graph, n=2^{28} vertices and $m = 4n$ edges, random edge weights).

single processor of the MTA-2 is two orders of magnitude slower than the reference sequential processor. In case of square grid graphs, there is sufficient parallelism to utilize up to 4 processors for a graph instance of 2^{24} vertices. For all smaller instances, the running time does not scale for multiprocessor runs. The ratio of the running time to BFS is about 5 in this case, and the Δ-stepping MTA-2 single processor time is comparable to the sequential reference platform running time for smaller instances. For the road networks, we note that the execution time does not scale well with the number of processors, as the problem instances are quite small. We observe better performance (lower execution time, better speedup) on USA-road-d graphs than on USA-road-t graphs.

6 Conclusions and Future Work

In this paper, we present an efficient implementation of the parallel Δ-stepping NSSP algorithm along with an experimental evaluation. We study the performance for several graph families on the Cray MTA-2, and observe that our implementation execution time scales very well with number of processors for low-diameter sparse graphs. Few prior implementations achieve parallel speedup for NSSP, whereas we attain near-linear speedup for several large-scale low-diameter graph families. We also analyze the performance using platform-independent Δ-stepping algorithm operation counts such as the number of *phases* and the *request set sizes* to explain performance across graph instances.

We intend to further study the dependence of the bucket-width Δ on the parallel performance of the algorithm. For high diameter graphs, there is a trade-off between the number of phases and the amount of work done (proportional to the number of bucket insertions). The execution time is dependent on the value of Δ as well as the number of processors. We need to reduce the number of phases for parallel runs and increase the system utilization by choosing an appropriate value of Δ. Our parallel performance studies have been restricted to the Cray MTA-2 in this paper. We have designed and have a preliminary implementation of Δ-stepping for multi-core processors and symmetric multiprocessors (SMPs), and for future work we will analyze its performance.

Acknowledgments

This work was supported in part by NSF Grants CNS-0614915, CAREER CCF-0611589, ACI-00-93039, NSF DBI-0420513, ITR ACI-00-81404, ITR EIA-01-21377, Biocomplexity DEB-01-20709, ITR EF/BIO 03-31654, and DARPA Contract NBCH30390004. We acknowledge the algorithmic inputs from Bruce Hendrickson of Sandia National Laboratories. We would also like to thank John Feo of Cray for helping us optimize the MTA-2 implementation. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

References

[1] P. Adamson and E. Tick. Greedy partitioned algorithms for the shortest path problem. *Internat. J. Parallel Program*, 20(4):271–278, 1991.

[2] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory BFS algorithms. In *Proc. 17th Ann. Symp. Discrete Algorithms (SODA-06)*, pages 601–610, Miami, FL, January 2006. ACM Press.

[3] D.A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proc. 34th Int'l Conf. on Parallel Processing (ICPP)*, Oslo, Norway, June 2005. IEEE Computer Society.

[4] D.A. Bader, A.K. Illendula, B. M.E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G.S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[5] D.A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*, Columbus, OH, August 2006. IEEE Computer Society.

[6] D.A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proc. 35th Int'l Conf. on Parallel Processing (ICPP)*, Columbus, OH, August 2006. IEEE Computer Society.

[7] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[8] J.W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Graph software development and performance on the MTA-2 and Eldorado. In *Proc. Cray User Group meeting (CUG 2006)*, Lugano, Switzerland, May 2006. CUG Proceedings.

[9] U. Brandes. A faster algorithm for betweenness centrality. *J. Mathematical Sociology*, 25(2):163–177, 2001.

[10] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Lett. Program. Lang. Syst.*, 2(1-4):59–69, 1993.

[11] G.S. Brodal, J.L. Träff, and C.D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.

[12] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM)*, Orlando, FL, April 2004. SIAM.

[14] K.M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM*, 25(11):833–837, 1982.

[15] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

[16] E. Cohen. Using selective path-doubling for parallel shortest-path computation. *J. Algs.*, 22(1):30–56, 1997.

[17] C. Demetrescu, A. Goldberg, and D. Johnson. 9th DIMACS implementation challenge – Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/`, 2006.

[18] R.B. Dial. Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM*, 12:632–633, 1969.

[19] R.B. Dial, F. Glover, D. Karney, and D. Klingman. A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks*, 9:215–248, 1979.

[20] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[21] 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*, 31(11):1343–1354, 1988.

[22] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *Proc. ACM SIGCOMM*, pages 251–262, Cambridge, MA, August 1999. ACM.

[23] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[24] M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48:533–551, 1994.

[25] L.C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[26] A.M. Frieze and L. Rudolph. A parallel algorithm for all-pairs shortest paths in a random graph. In *Proc. 22nd Allerton Conference on Communication, Control and Computing*, pages 663–670, 1985.

[27] G. Gallo and P. Pallottino. Shortest path algorithms. *Ann. Oper. Res.*, 13:3–79, 1988.

[28] F. Glover, R. Glover, and D. Klingman. Computational study of an improved shortest path algorithm. *Networks*, 14:23–37, 1984.

[29] A.V. Goldberg. Shortest path algorithms: Engineering aspects. In *ISAAC 2001: Proc. 12th Int'l Symp. on Algorithms and Computation*, pages 502–513, London, UK, 2001. Springer-Verlag.

[30] A.V. Goldberg. A simple shortest path algorithm with linear average time. In *9th Ann. European Symp. on Algorithms (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 230–241, Aachen, Germany, 2001. Springer.

[31] D. Gregor and A. Lumsdaine. Lifting sequential graph

algorithms for distributed-memory parallel computation. In *Proc. 20th ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 423–437, New York, NY, USA, 2005. ACM Press.

[32] R. Guimerà, S. Mossa, A. Turtschi, and L.A.N. Amaral. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. *Proceedings of the National Academy of Sciences USA*, 102(22):7794–7799, 2005.

[33] T. Hagerup. Improved shortest paths on the word RAM. In *27th Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *Lecture Notes in Computer Science*, pages 61–72, Geneva, Switzerland, 2000. Springer-Verlag.

[34] Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing the all pair shortest paths in directed graphs. *Algorithmica*, 17(4):399–415, 1997.

[35] M.R. Hribar and V.E. Taylor. Performance study of parallel shortest path algorithms: Characteristics of good decomposition. In *Proc. 13th Ann. Conf. Intel Supercomputers Users Group (ISUG)*, 1997.

[36] M.R. Hribar, V.E. Taylor, and D.E. Boyce. Parallel shortest path algorithms: Identifying the factors that affect performance. Report CPDC-TR-9803-015, Northwestern University, Evanston, IL, 1998.

[37] M.R. Hribar, V.E. Taylor, and D.E. Boyce. Reducing the idle time of parallel shortest path algorithms. Report CPDC-TR-9803-016, Northwestern University, Evanston, IL, 1998.

[38] M.R. Hribar, V.E. Taylor, and D.E. Boyce. Termination detection for parallel shortest path algorithms. *Journal of Parallel and Distributed Computing*, 55:153–165, 1998.

[39] H. Jeong, S.P. Mason, A.-L. Barabási, and Z.N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41–42, 2001.

[40] P.N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algs.*, 25(2):205–220, 1997.

[41] F. Liljeros, C.R. Edling, L.A.N. Amaral, H.E. Stanley, and Y. Åberg. The web of human sexual contacts. *Nature*, 411:907–908, 2001.

[42] K. Madduri. 9th DIMACS implementation challenge: Shortest Paths. Δ-stepping C/MTA-2 code. `http://www.cc.gatech.edu/~kamesh/research/DIMACS-ch9`, 2006.

[43] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak. Parallel shortest path algorithms for solving large-scale instances. Technical report, Georgia Institute of Technology, September 2006.

[44] U. Meyer. Heaps are better than buckets: parallel shortest paths on unbalanced graphs. In *Proc. 7th International Euro-Par Conference (Euro-Par 2001)*, pages 343–351, Manchester, United Kingdom, 2000. Springer-Verlag.

[45] U. Meyer. Buckets strike back: Improved parallel shortest-paths. In *Proc. 16th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–8, Fort Lauderdale, FL, April 2002. IEEE Computer Society.

[46] U. Meyer. *Design and Analysis of Sequential and Parallel Single-Source Shortest-Paths Algorithms*. PhD thesis, Universität Saarlandes, Saarbrücken, Germany, October 2002.

[47] U. Meyer. Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds. *J. Algs.*, 48(1):91–134, 2003.

[48] U. Meyer and P. Sanders. Parallel shortest path for arbitrary graphs. In *Proc. 6th International Euro-Par Conference (Euro-Par 2000)*, volume 1900 of *Lecture Notes in Computer Science*, pages 461–470, Munich, Germany, 2000. Springer-Verlag.

[49] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *J. Algs.*, 49(1):114–152, 2003.

[50] M.E.J. Newman. Scientific collaboration networks: II. shortest paths, weighted networks and centrality. *Phys. Rev. E*, 64:016132, 2001.

[51] M.E.J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[52] M. Papaefthymiou and J. Rodrigue. Implementing parallel shortest-paths algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 30:59–68, 1997.

[53] J. Park, M. Penner, and V.K. Prasanna. Optimizing graph algorithms for improved cache performance. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2002)*, Fort Lauderdale, FL, April 2002. IEEE Computer Society.

[54] R. Raman. Recent results on single-source shortest paths problem. *SIGACT News*, 28:61–72, 1997.

[55] H. Shi and T.H. Spencer. Time-work tradeoffs of the single-source shortest paths problem. *J. Algorithms*, 30(1):19–32, 1999.

[56] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[57] J. L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21(9):1505–1532, 1995.

[58] J. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. In *Proc. 2nd Ann. Symp. Parallel Algorithms and Architectures (SPAA-90)*, pages 200–209, Crete, Greece, July 1990. ACM.

[59] F.B. Zhan and C.E. Noon. Shortest path algorithms: an evaluation using real road networks. *Transp. Sci.*, 32:65–73, 1998.