Parallel Shortest Path Algorithms for Solving Large-Scale Instances

Kamesh Madduri*
Georgia Institute of Technology
kamesh@cc.gatech.edu

David A. Bader
Georgia Institute of Technology
bader@cc.gatech.edu

Jonathan W. Berry
Sandia National Laboratories
jberry@sandia.gov

Joseph R. Crobak
Rutgers University
crobakj@cs.lafayette.edu

August 30, 2006

Abstract

We present an experimental study of parallel algorithms for solving the single source shortest path problem with non-negative edge weights (NSSP) on large-scale graphs. We implement Meyer and Sander's Δ-stepping algorithm and report performance results on the Cray MTA-2, a multithreaded parallel architecture. The MTA-2 is a high-end shared memory system offering two unique features that aid the efficient implementation of irregular parallel graph algorithms: the ability to exploit fine-grained parallelism, and low-overhead synchronization primitives. Our implementation exhibits remarkable parallel speedup when compared with a competitive sequential algorithm, 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 parallel NSSP problem on realistic graph instances in the order of billions of vertices and edges.

1 Introduction

This paper primarily discusses parallel algorithms and implementations 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

*Contact author

metrics such as betweenness centrality [27, 9, 41, 43, 34] are based on shortest path algorithms. Our parallel implementations target the *scale-free* graph family, a well-studied model [7, 24, 12, 53, 52] for real-world large-scale graphs that captures features such as a low diameter, heavy-tailed degree distributions modeled by power laws, and self similarity. We also conduct an experimental study of performance on several other graph families, and this work is our submission to the 9th DIMACS Implementation Challenge [18] on Shortest Paths.

Sequential algorithms for the single source shortest path problem with non-negative edge weights (NSSP) are studied extensively, both theoretically [22, 20, 25, 26, 56, 58, 35, 32, 48] and experimentally [21, 30, 29, 15, 61, 31]. 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 [22] algorithm. Using Fibonacci heaps [25], Dijkstra's algorithm can be implemented in $O(m + n \log n)$ time. Thorup [58] 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 [49] and Goldberg [31] propose simple algorithms with linear average time for uniformly distributed edge weights.

In this paper, we primarily focus on parallel implementations of NSSP. Prior parallel NSSP algorithms have been reviewed in detail by Meyer and Sanders [48, 51]. There are no known PRAM algorithms that run in sub-linear time and $O(m + n \log n)$ work. Parallel priority queues [23, 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 [36, 28], proposed for the parallel All-Pairs Shortest Paths (APSP), involve running time and efficiency tradeoffs. Parallel approximate NSSP algorithms [42, 16, 57] based on the randomized Breadth-First search algorithm of Ullman and Yannakakis [60] 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 [51] NSSP algorithm that divides Dijkstra's algorithm into a number of *phases*, each of which could 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 [50, 46, 47] are given for Δ-stepping.

The literature contains few experimental studies on parallel NSSP algorithms [37, 54, 39, 59]. Prior implementation results on distributed memory machines resorted to graph partitioning [14, 1, 33], and then running a sequential NSSP algorithm on the sub-graph. Heuristics are used for load balancing and termination detection [38, 40]. The implementations perform well for certain graph families and problem sizes, but no speedup may be possible in the worst case.

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 and scalability is limited in most cases. While it may be possible to improve the cache performance to a certain degree for some classes of graphs [55], there are no known general techniques for cache optimization because the memory access pattern is largely dependent on the structure of the graph.

We present our shortest path implementation results on the Cray MTA-2, a massively multithreaded parallel architecture. 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 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 low-overhead synchronization support complements multithreading and makes performance primarily a function of parallelism. Since graph algorithms often have an abundance of parallelism, these 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 and subgraph isomorphism [8], Breadth-First Search and *st*-connectivity [5].

Our main contributions in this paper are as follows:

- We conduct an experimental study of *parallel* shortest path algorithms designed for shared memory architectures. Prior studies have predominantly focused on running sequential SSSP algorithms on graph families that can be easily partitioned, whereas we also consider several arbitrary, sparse graph instances.

- We present parallel performance results of the Δ-stepping algorithm on the Cray MTA-2, a multithreaded architecture. We attain impressive performance on low-diameter graphs. We also analyze performance using machine-independent algorithmic operation counts.

- On the MTA-2, we are able to solve NSSP for large-scale realistic graph instances in the order of billions of edges. For instance, 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. Also, the sequential performance of our implementation is comparable to competitive NSSP implementations (such as the DIMACS reference solver).

- We also have a parallel implementations of Thorup's algorithm for NSSP, and the Bellman-Ford algorithm for solving the single-source shortest paths allowing negative edge weights. For All-pairs computations using the component hierarchy, we expect the Thorup implementation to outperform Δ-stepping.

This paper is organized as follows. Section 2 provides a brief overview of Δ-stepping. Our parallel implementation of Δ-stepping is discussed in Section 3. Section 4 and 5 describe our experimental setup, performance results and analysis. We conclude with a discussion on implementation improvements and future plans in Section 6. Applendix A describes the MTA-2 architecture.

2 Review of the Δ-stepping Algorithm

2.1 Preliminaries

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 $l : E \rightarrow \mathbb{R}$. 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 $\delta(s) = 0$.

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.

2.2 Algorithmic Details

The Δ-stepping algorithm (see Alg. 1) is an "approximate bucket implementation of Dijkstra's algorithm" [51]. 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 9–13, 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

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

```
1  foreach v ∈ V do
2      heavy(v) ← {⟨v, w⟩ ∈ E : l(v, w) > Δ};
3      light(v) ← {⟨v, w⟩ ∈ E : l(v, w) ≤ Δ};
4      d(v) ← ∞;
5  relax(s, 0);
6  i ← 0;
7  while B is not empty do
8      S ← φ;
9      while B[i] ≠ φ do
10         Req ← {(w, d(v) + l(v, w)) : v ∈ B[i] ∧ ⟨v, w⟩ ∈ light(v)};
11         S ← S ∪ B[i];
12         foreach (v, x) ∈ Req do
13             relax(v, x);
14     Req ← {(w, d(v) + l(v, w)) : v ∈ S ∧ ⟨v, w⟩ ∈ heavy(v)};
15     foreach (v, x) ∈ Req do
16         relax(v, x);
17     i ← i + 1;
18 δ(v) ← d(v);
```

Algorithm 1: Δ-stepping algorithm

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

```
1  if x < d(v) then
2      B[⌊d(v)/Δ⌋] ← B[⌊d(v)/Δ⌋] \ {v};
3      B[⌊x/Δ⌋] ← B[⌊x/Δ⌋] ∪ {v};
4      d(v) ← x;
```

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

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 14–16 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 similar to the Belman-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 [51]. 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 [51] theoretically show 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

See Appendix A for details of the MTA-2 architecture and parallelization primitives.

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. 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 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 only add a vertex v to R 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.

On the MTA-2, accessing the same memory location concurrently by several threads incurs a performance penalty. We call these high-contention memory locations *hot spots*, and need to minimize these to ensure good scalability. For instance, in the relax routine, the bucket size counter may become a hot spot if a significant number of vertices in the current request set are inserted into the same bucket. This is particularly true for low-diameter graph families such as random and scale-free graphs. However, this leads to a performance penalty only in the case of very large problem instances (random graphs with 500 million to 2 billion edges) using over 30 processors.

To saturate the MTA-2 processors with work and to attain 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. There are several implicit barrier synchronizations in the algorithm that are 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 [44].

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 compile 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 [19]. Both the codes are compiled with the Intel C compiler (icc) Version 9.0, with the flags `-O3`. The source code is freely available online [44].

4.2 Problem Instances

We evaluate the sequential and parallel performance on the core graph families provided in the DIMACS benchmark package [19]. The core package includes the following graph families:

- *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. Two random graph families are defined: *Random4-n* (n grows, and the maximum weight is set to n. m is set to $4n$) and *Random4-C* (n is fixed, and C grows).

- *Grid graphs*: This synthetic generator produces grid-like graphs. The grid contains $x \cdot y$ vertices, (i, j) for $0 \leq i < x$ and $0 \leq j < y$. A vertex (i, j) is connected to adjacent vertices in the same layer. If $i < x - 1$, each vertex (i, j) is also connected to the vertex $(i + 1, j)$. Long ($x = \frac{n}{16}$, $y = 16$) and Square 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-t*) as the length function.

In addition to the core families, we study graph instances from the scale-free graph family (denoted by *ScaleFree4-n*). These are low-diameter graphs with an unbalanced degree distribution, and the structural properties modeled by these graphs are been observed in several real-world complex networks [7, 2, 24]. We use the R-MAT graph model [13] to generate Scalefree4-n graph instances. The R-MAT scale-free generator is included in our GTgraph [45] synthetic graph generator package.

All the synthetic core 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 *ScaleFree4 logUnif-n* families for this weight distribution.

4.3 Methodology

For sequential runs, we report the execution time of the reference DIMACS NSSP solver and the baseline Breadth-First Search (BFS) on every core 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, 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. Unless otherwise specified, assume that the value of Δ is set 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 the MTA-2, it would be meaningless to compare the performance of our optimized parallel implementation with the sequential DIMACS solver. Parallelizing the reference solver to run on a massively multithreaded machine such as the MTA-2 is a non-trivial task. So, we report the execution time of a multithreaded level-synchronized breadth-first search [5], optimized for low-diameter graphs. The multithreaded BFS scales as well as δ-stepping for the core graph families, and the execution time serves as a lower bound for the shortest path running time.

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.

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 Appendix B, and report execution time in seconds. If the execution time is less than 10^{-3} seconds, we round the time to four decimal digits. If it is less than 10^{-2} seconds, we round it to three digits. In all other cases, the reported running time is rounded to two decimal digits.

5 Results and Analysis

5.1 Sequential Performance

First we present the performance results of our implementation on the reference sequential platform for the core graph families. The BFS, Δ-stepping, and reference DIMACS implementation execution times on the recommended core graph instances are given in Section B.1. We observe that the ratio of the Δ-stepping execution time to the Breadth-First Search time varies between 3 and 10 across different problem instances. Also, the DIMACS reference code is about 1.5 to 2 times faster than our implementation for large problem instances in each family.

Table 1 summarizes the performance for random graph instances. For the Random4-n family, n is varied from 2^{11} to 2^{21}, the maximum edge weight is set to n, and the graph density is constant. For the largest instance, Δ-stepping execution time is 1.7 times slower than the reference implementation and 5.4 times the BFS execution time. For the Random4-C family, we normalize the weights to the maximum integer weight. We do not observe any trend similar to the reference implementation, where the execution time gradually rises as the maximum weight increases.

The sequential performance of Δ-stepping on Long grid graphs (Table 2) is similar to that on Random graphs. However, the reference implementation is slightly faster on long grids. For square grids and road networks, the Δ-stepping to BFS ratio is comparatively higher (for e.g., BFS to Δ-stepping ratio is 4.71 for the largest Square-n graph, and 3.74 for the largest Random4-n graph) than the Random and Long grid families.

Figs. 1 and 2 summarize the key observations from the tables in Section B.1. Comparing execution time across graphs of the same size in Fig. 1, we find that the Δ-stepping running time for the Random4-n graph instance is slightly higher than the rest of the families. The Δ-stepping running time is also comparable to the execution time of the reference implementation for all graph families. Fig. 2 plots the execution time normalized to the problem size (or the running time per edge) for Random4-n and Long-n families. Observe that the Δ-stepping implementation execution time scales with problem size at a faster rate compared to BFS or the DIMACS reference implementation.

5.2 Δ-stepping analysis

To better understand the algorithm performance across graph families, we study 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

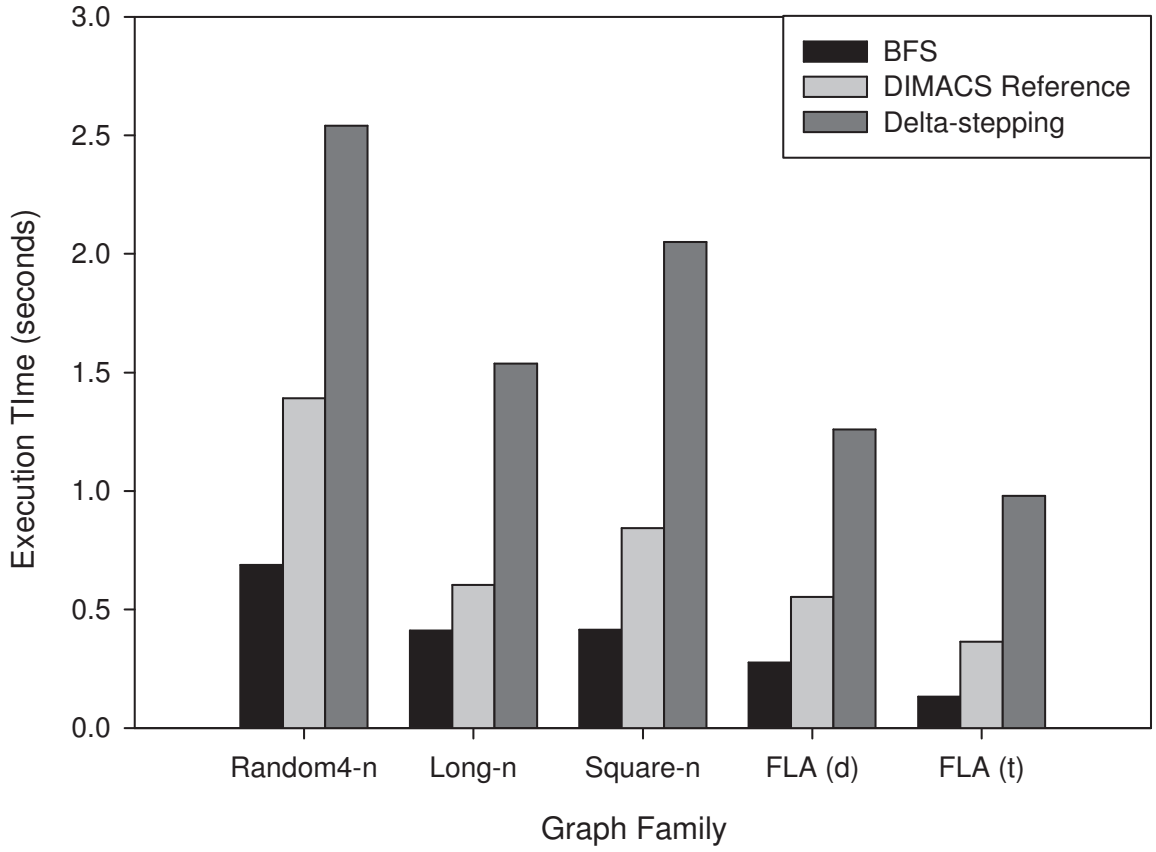
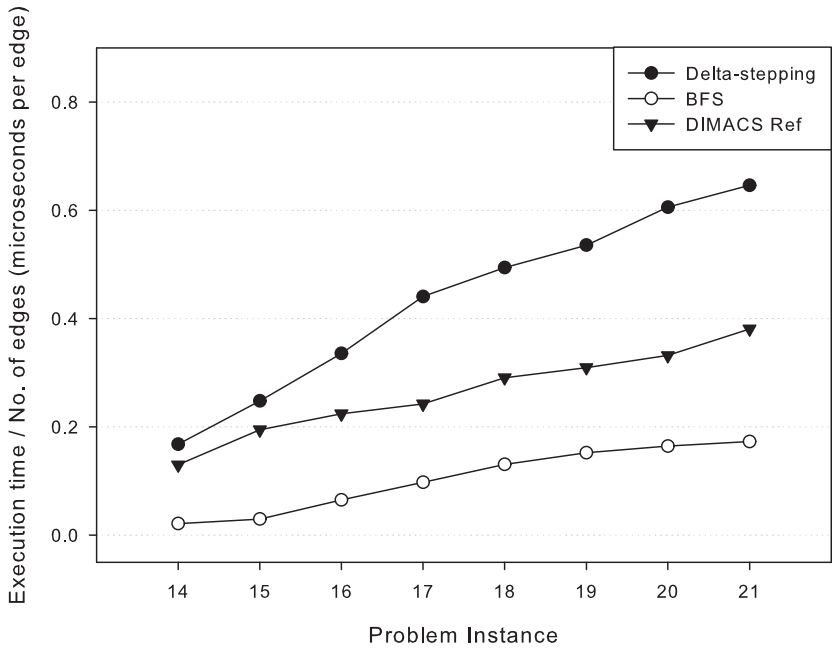
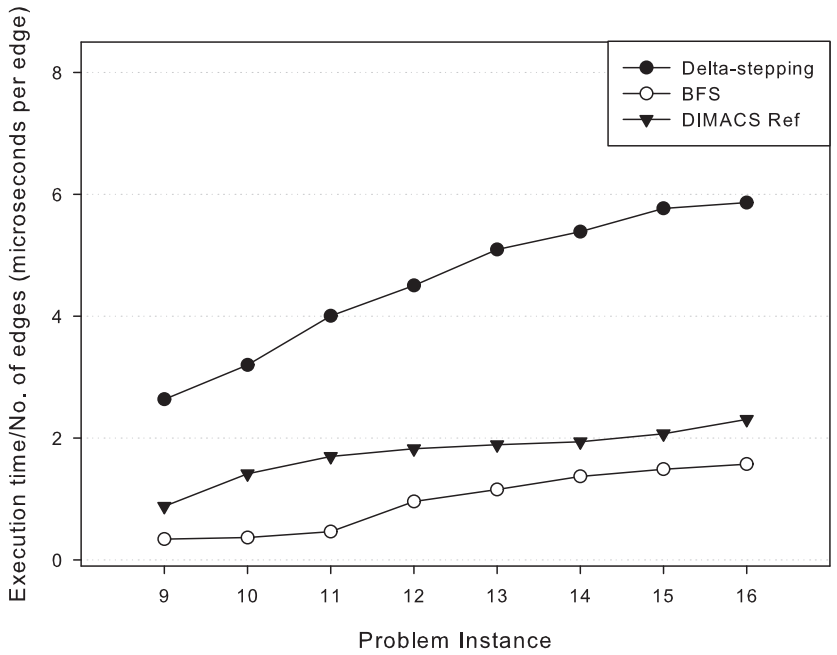


Figure 1: Sequential performance of our Δ-stepping implementation, on the core graph families. All the synthetic graphs are directed, with 2^{20} vertices and $\frac{m}{n} \approx 4$. FLA(d) and FLA(t) are road networks corresponding to Florida, with 1070376 vertices and 2712768 edges

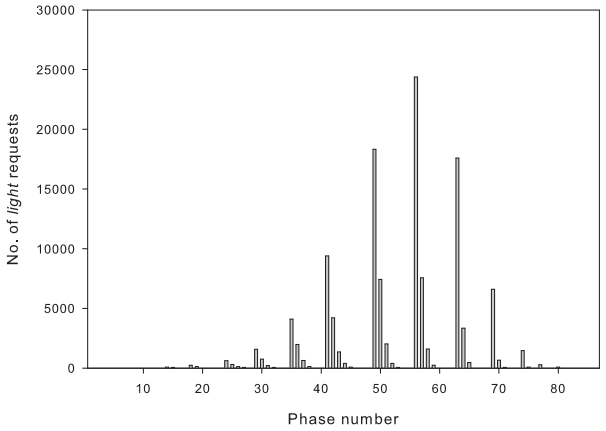


(a) Random4-n family. Problem instance i denotes a directed graph of 2^i vertices, $m = 4n$ edges, and maximum weight $C = n$

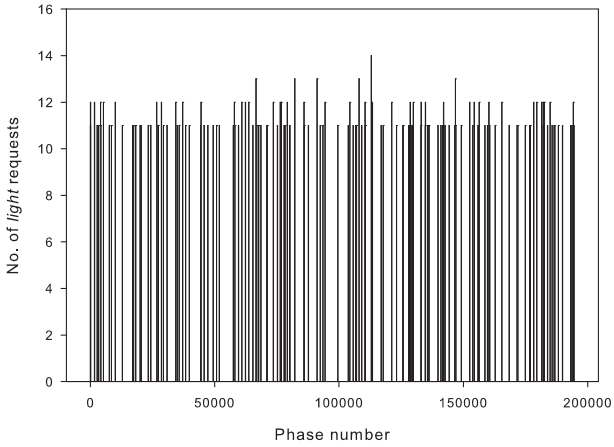


(b) Long-n family. Problem instance i denotes a grid with $x = 2^i$ and $y = 16$. $n = xy$ and $\frac{m}{n} \simeq 4$

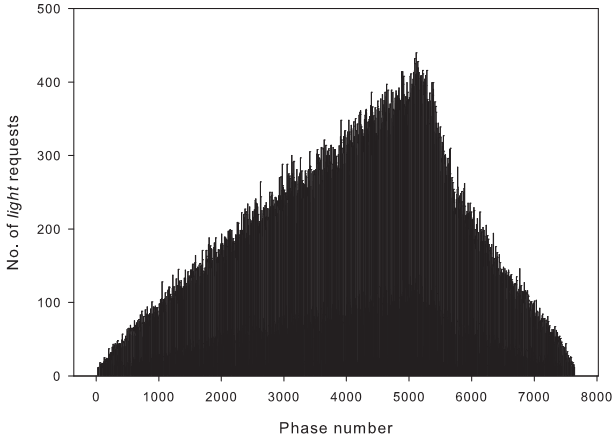
Figure 2: Δ-stepping sequential execution time as a function of problem size



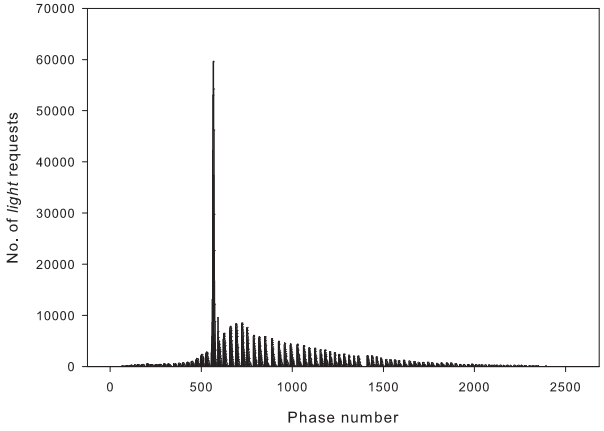
(a) Random4-n family, $n = 2^{20}$.



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



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



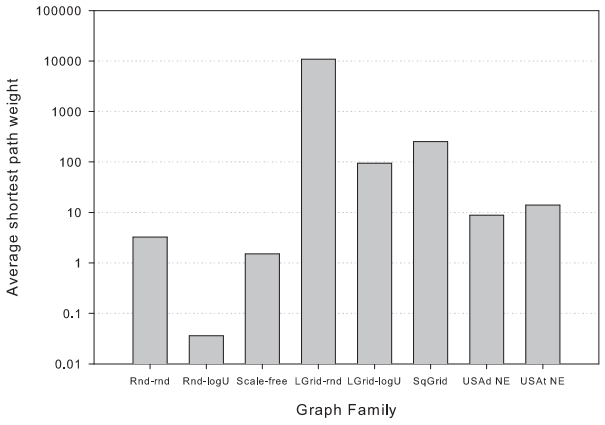
(d) USA-road-d family, Northeast USA (NE). $n = 1524452$, $m = 3897634$.

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.

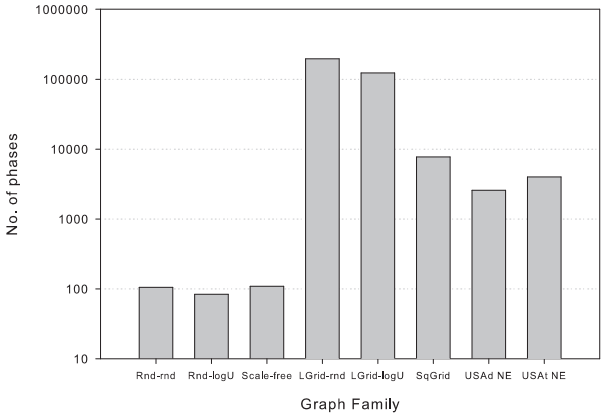
the light request set in each phase, for each core graph family. Δ is set to 0.25 for all runs. If the request set size is less than 10, it is not plotted. Consider the random graph family (Fig. 3(a)). It executes in 84 phases, and the request set sizes vary from 0 to 27,000. Observe the recurring pattern 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 size plot of a similar problem instance from the Long grid family (Fig. 3(b)) stands in stark contrast to the random graph plot. 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. 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 set counts greater than 1000.

Fig. 4 plots several key Δ-stepping operation counts for various graph classes. Along with the core graph families, we include ScaleFree4-n, RandomlogUnif4-n, and LonglogUnif4-n graph classes. All synthetic graphs are roughly of the same size. Fig. 4(a) plots the average 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 lot of excess work for these families, because the value of Δ is quite high for this particular distribution.

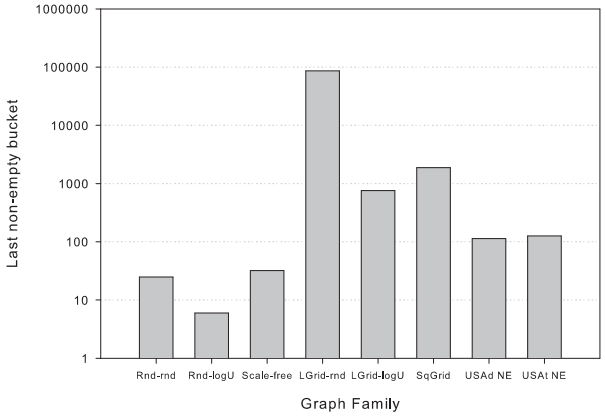
We next evaluate the performance of the algorithm as Δ is varied (tables in Section B.2). Fig. 5 plots the execution time of various graph instances on a sequential machine, and one processor of the MTA-2. Δ is varied from 0.1 to 10 in each case. We find that the absolute running times on a 3.2 GHz Xeon 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 (Tables 5 and 6). 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



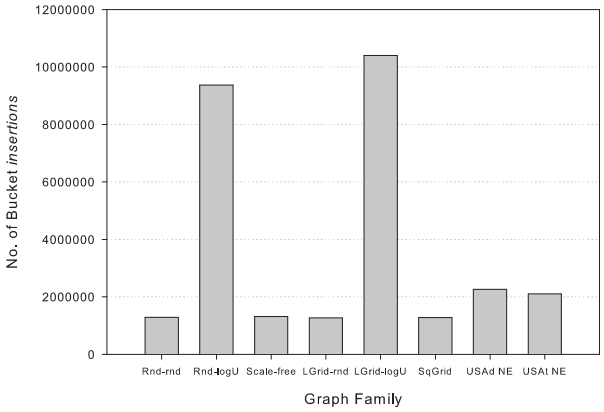
(a) Average shortest path weight ($\frac{1}{n} * \sum_{v \in V} \delta(v)$)



(b) No. of phases



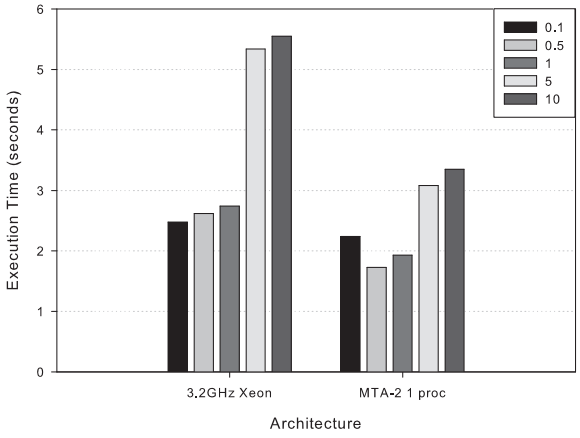
(c) Last non-empty bucket



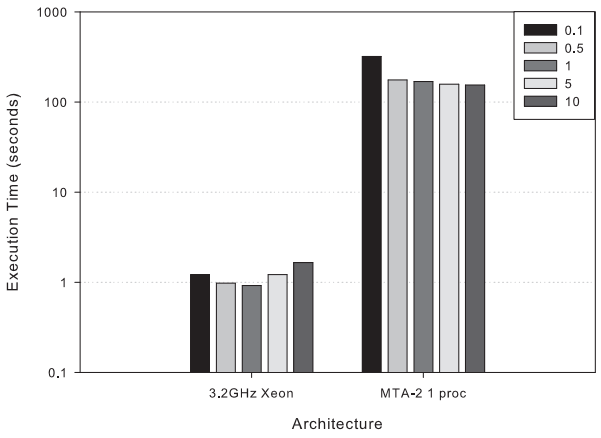
(d) Number of relax requests

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) and (c) are on a log scale, while (d) uses a linear scale.

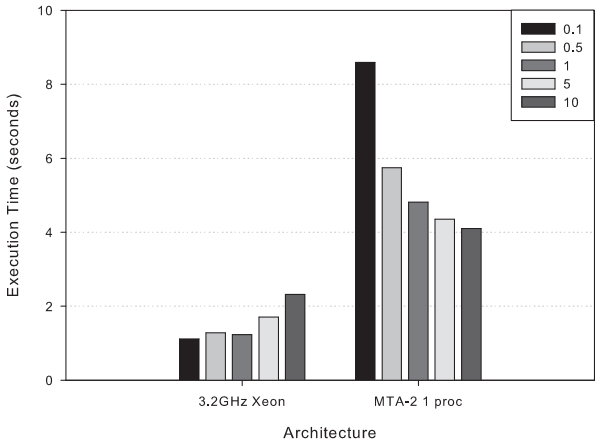
a particular Δ, it may be possible to saturate a single processor of the MTA-2 with the right balance of work and phases. The execution time on a 40-processor run may not be minimal with this value of Δ.



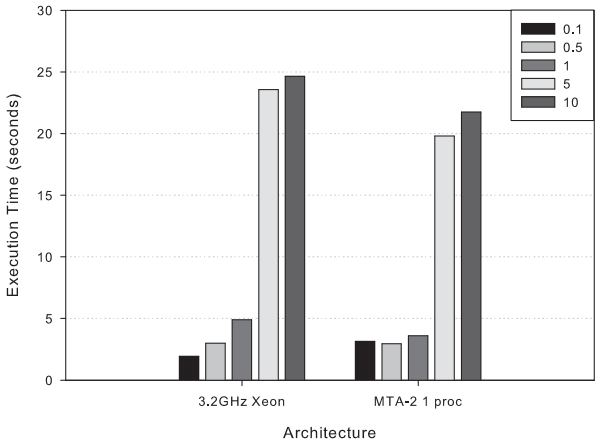
(a) Random4-n family. 2^{20} vertices



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



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

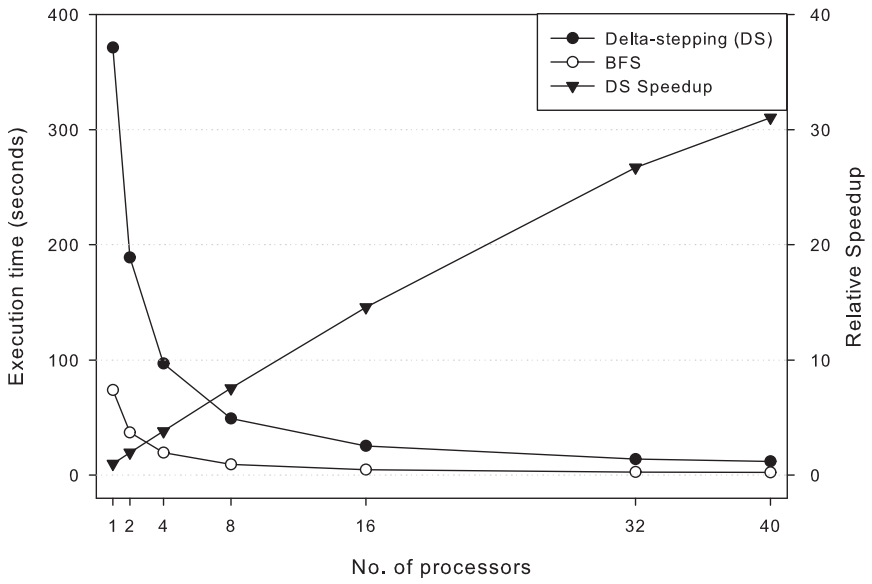


(d) USA-road-d family, Florida (FLA). 1070376 vertices, 2712798 edges

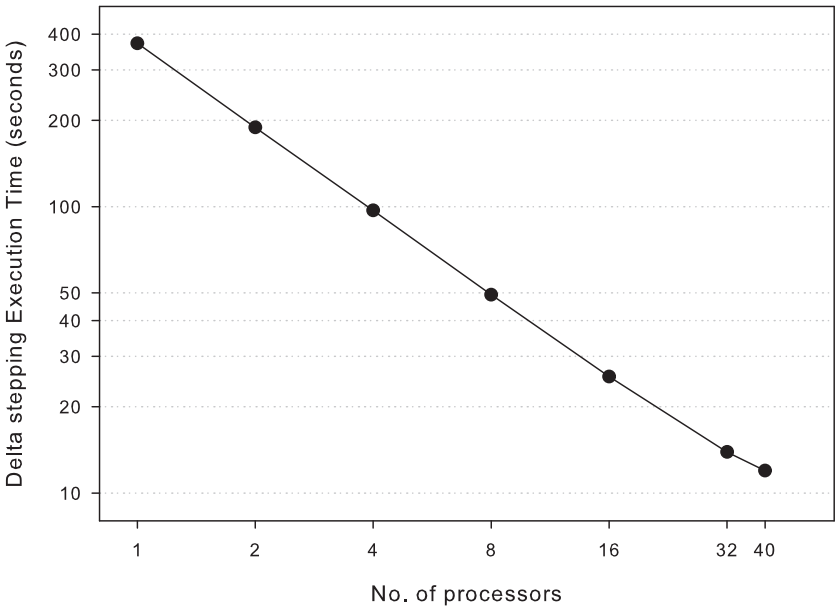
Figure 5: A comparison of the execution time on the reference sequential platform and a single MTA-2 processor, as the bucket-width Δ is varied.

5.3 Parallel Performance

We present the parallel scaling of the Δ-stepping algorithm in detail (see Section B.3). We ran Δ-stepping and the level-synchronous parallel BFS on graph instances from the core families, scale-free graphs and graphs with log-uniform edge weight distributions. Define the

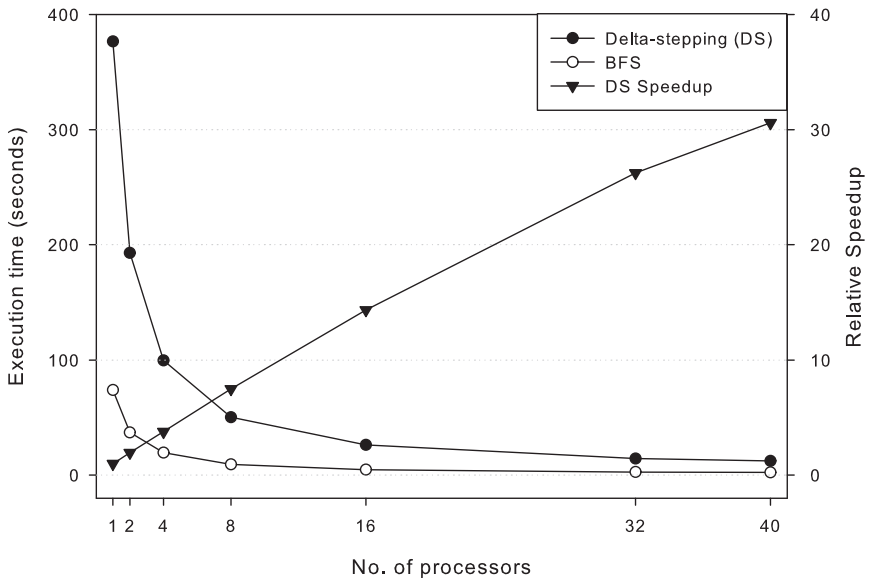


(a) Execution time and Relative Speedup (linear scale)

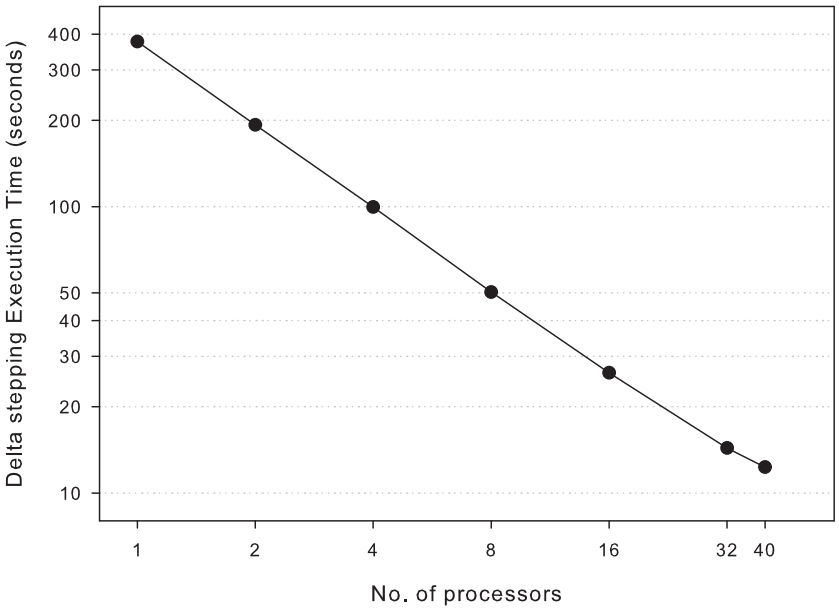


(b) Execution time vs. No. of processors (log-log scale)

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



(a) Execution time and Relative Speedup (linear scale)



(b) Execution time vs. No. of processors (log-log scale)

Figure 7: Δ-stepping execution time and relative speedup on the MTA-2 for a ScaleFree4-n graph instance (directed graph, n=2^{28} vertices and $m = 4n$ edges, random edge weights).

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. Since the computation on the MTA-2 is thread-centric rather than processor-centric, note that the single processor run is also parallel. 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 from the discussion in the previous section, Δ-stepping performs best for low-diameter random and scale-free graphs with randomly distributed edge weights (see Fig. 6 and 7). We attain 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 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 only 1 second slower than 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, attaining a speedup of 30 for a realistic scale-free graph instance of one billion edges (Fig. 7) is a remarkable result.

Table 7 gives the execution time of Δ-stepping on the Random4-n family, as the number of vertices is increased from 2^{21} to 2^{28}, and the number of processors is varied from 1 to 40. Observe that the relative speedup increases as the problem size is increased (for e.g., on 40 processors, the speedup for $n = 2^{21}$ 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.

Table 8 gives the execution time for random graphs with a log-uniform weight distribution. With Δ set to $\frac{n}{m}$, we do a lot 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.

Table 9 summarizes the execution time for the Random4-C family. The maximum edge weight is varied from 4^0 to 4^{15} while keeping m and n constant. We do not notice any trend in the execution time in this case, as we normalize the edge weights to fall in the interval [0, 1]. Similarly, there is no noticeable trend in case of the Long-C family (Table 11).

Tables 10 and 12 give the execution time for Δ-stepping for the long and square grid graphs respectively, as the problem size and number of processors are varied. For Long-n graphs with 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 single processor of the MTA-2 is two orders of magnitude slower than the reference sequential processor (Fig. 5(b)). 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 other 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. Tables 13 and 14 summarize the running times on the USA road networks. The execution time does not scale well with problem size, as the problem instances are 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 experimentally evaluate the parallel Δ-stepping NSSP algorithm for the 9th DIMACS Shortest Paths Challenge. We study the algorithm performance for core challenge graph instances on the Cray MTA-2, and observe that our implementation execution time scales impressively with number of processors for low-diameter sparse graphs. 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 families.

We would like 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.

We are currently optimizing parallel implementations of Thorup's algorithm for NSSP, and the Bellman-Ford algorithm for solving the single-source shortest paths allowing negative edge weights. For all-pairs shortest path computations, we expect our optimized Thorup implementation to outperform Δ-stepping. We intend to repeat this NSSP experimental study on the MTA-2 with Thorup's implementation in the near future.

Our parallel performance studies have been restricted to the Cray MTA-2 in this paper. We have a preliminary implementation of Δ-stepping designed for multi-core processors and symmetric multiprocessors (SMPs) that we wish to add to this study. We expect the SMP implementation would also perform well on low-diameter graphs.

Acknowledgements

This work was supported in part by NSF Grants 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.

A The Cray MTA-2

This section is excepted from [6].

A.1 Architecture

The Cray MTA-2 [17] is a novel multithreaded architecture with no data cache, and hardware support for synchronization. The computational model for the MTA-2 is *thread-centric*, not processor-centric. A thread is a logical entity comprised of a sequence of instructions that are issued in order. An MTA-2 processor consists of 128 hardware *streams* and one instruction pipeline. A stream is a physical resource (a set of 32 registers, a status word, and space in the instruction cache) that hold the state of one thread. An instruction is three-wide: a memory operation, a fused multiply-add, and a floating point add or control operation. Each stream can have up to 8 outstanding memory operations. Threads from the same or different programs are mapped to the streams by the runtime system. A processor switches among its streams every cycle, executing instructions from non-blocked streams. As long as one stream has a ready instruction, the processor remains fully utilized. No thread is bound to any particular processor. System memory size and the inherent degree of parallelism within the program are the only limits on the number of threads used by a program.

The interconnection network is a partially connected 3-D torus capable of delivering one word per processor per cycle. The system has 4 GBytes of memory per processor. Logical memory addresses are hashed across physical memory to avoid stride-induced hot spots. Each memory word is 68 bits: 64 data bits and 4 tag bits. One tag bit (the full-empty bit) is used to implement synchronous load and store operations. A thread that issues a synchronous load or store remains blocked until the operation completes; but the processor that issued the operation continues to issue instructions from non-blocked streams.

The MTA-2 is closer to a theoretical PRAM machine than a shared memory symmetric multiprocessor system. Since the MTA-2 uses parallelism to tolerate latency, algorithms must often be parallelized at very fine levels to expose sufficient parallelism. However, it is not necessary that all parallelism in the program be expressed such that the system can exploit it; the goal is simply to saturate the processors. The programs that make the most effective use of the MTA-2 are those which express the parallelism of the problem in a way that allows the compiler to best exploit it.

A.2 Expressing Parallelism on the MTA-2

The MTA-2 compiler automatically parallelizes *inductive* loops of three types: parallel loops, linear recurrences and reductions. A loop is inductive if it is controlled by a variable that is incremented by a loop-invariant stride during each iteration, and the loop-exit test compares this variable with a loop-invariant expression. An inductive loop has only one exit test and can only be entered from the top. If each iteration of an inductive loop can be executed completely independently of the others, then the loop is termed parallel. To attain the best

performance, we need to write code (and thus design algorithms) such that most of the loops are implicitly parallelized.

There are several compiler directives that can be used to parallelize various sections of a program. The three major types of parallelization schemes available are

- single-processor (*fray*) parallelism: The code is parallelized in such a way that just the 128 streams on the processor are utilized.
- multi-processor (*crew*) parallelism: This has higher overhead than single-processor parallelism. However, the number of streams available is much larger, bounded by the size of the whole machine rather than the size of a single processor. Iterations can be statically or dynamically scheduled.
- future parallelism: The *future* construct (detailed below) is used in this form of parallelism. This does not require that all processor resources used during the loop be available at the beginning of the loop. The runtime growth manager increases the number of physical processors as needed. Iterations are always dynamically scheduled.

A *future* is a powerful construct to express user-specified explicit parallelism. It packages a sequence of code that can be executed by a newly created thread running concurrently with other threads in the program. Futures include efficient mechanisms for delaying the execution of code that depends on the computation within the future, until the future completes. The thread that spawns the future can pass information to the thread that executes the future via parameters. Futures are best used to implement task-level parallelism and the parallelism in recursive computations.

A.3 Synchronization support on the MTA-2

Synchronization is a major limiting factor to scalability in the case of practical shared memory implementations. The software mechanisms commonly available on conventional architectures for achieving synchronization are often inefficient. However, the MTA-2 provides hardware support for fine-grained synchronization through the full-empty bit associated with every memory word. The compiler provides a number of generic routines that operate atomically on scalar variables. We list a few useful constructs that appear in the algorithm pseudo-codes in subsequent sections.

- The `int_fetch_add` routine (`int_fetch_add`$(\&v, i)$) atomically adds integer i to the value at address v, stores the sum at v, and returns the original value at v (setting the full-empty bit to full). If v is an empty sync or future variable, the operation blocks until v becomes full.
- `readfe`$(\&v)$ returns the value of variable v when v is full and sets v empty. This allows threads waiting for v to become empty to resume execution. If v is empty, the read blocks until v becomes full.

- `writeef`$(\&v, i)$ writes the value i to v when v is empty, and sets v back to full. The thread waits until v is set empty.
- `purge`$(\&v)$ sets the state of the full-empty bit of v to empty.

B Tables

B.1 Sequential performance of Δ-stepping implementation on the reference platform

(a) Random4-n core family. Problem instance denotes the log of the number of vertices. A directed random graph of n vertices, $m = 4n$ edges, and maximum weight $C = n$.

Problem Instance	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
BFS	.0001	.0003	.0006	.001	.004	.02	.05	.14	.32	.69	1.45
Δ-stepping	.0007	.002	.004	.01	.03	.09	.23	.52	1.12	2.54	5.42
Normalized to BFS	7.00	6.67	6.67	10.00	7.50	4.50	4.60	3.71	3.50	3.68	3.74
DIMACS Reference	.0003	.0008	.002	.008	.02	.06	.13	.30	0.65	1.39	3.19
Normalized to BFS	3.00	2.67	3.33	8.00	5.00	3.00	2.60	2.14	2.03	2.01	2.20

(b) Random4-C core family. Problem instance denotes the log of the maximum edge weight. $n = 2^{20}$ vertices and $m = 4n$ edges.

Problem Instance	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BFS	0.69	0.69	0.69	0.69	0.69	0.69	0.69	0.69
Δ-stepping	2.31	2.55	2.53	2.55	2.54	2.54	2.54	2.54
Normalized to BFS	3.35	3.70	3.67	3.70	3.68	3.67	3.67	3.67
DIMACS Reference	0.87	0.89	0.92	1.21	1.26	1.31	1.38	1.36
Normalized to BFS	1.26	1.29	1.33	1.75	1.83	1.90	2.00	1.97

Problem Instance	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
BFS	0.69	0.69	0.69	0.69	0.69	0.69	0.69	0.69
Δ-stepping	2.55	2.54	2.54	2.55	2.54	2.54	2.54	2.54
Normalized to BFS	3.70	3.68	3.68	3.70	3.68	3.68	3.68	3.68
DIMACS Reference	1.37	1.37	1.38	1.37	1.37	1.38	1.37	1.38
Normalized to BFS	1.98	1.98	2.00	1.98	1.98	2.00	1.98	2.00

Table 1: Sequential performance (execution time in seconds, and normalized performance with reference to the baseline BFS) of our implementation for the core random graph families.

(a) Long-n core family. Problem instance i denotes a grid with $x = 2^i$ and $y = 16$. $n = xy$ and $\frac{m}{n} \simeq 4$.

Problem Instance	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
BFS	.0001	.0002	.0003	.0007	.001	.004	.02	.04	.09	.19	.41
Δ-stepping	.0005	.001	.002	.005	.01	.03	.07	.17	.35	.76	1.54
Normalized to BFS	5.00	5.00	6.67	7.14	10.00	7.50	3.50	4.25	3.89	4.00	3.76
DIMACS Reference	.0002	.0003	.0007	.002	.006	0.01	.03	0.06	.13	.27	.60
Normalized to BFS	2.00	1.50	2.33	2.86	6.00	2.50	1.50	1.50	1.44	1.42	1.46

(b) Long-C core family. Problem instance denotes the log of the maximum edge weight. The grid dimensions are set to $x = 2^{16}$ and $y = 16$.

Problem Instance	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BFS	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
Δ-stepping	0.68	0.75	0.78	0.88	1.02	1.07	1.09	1.09
Normalized to BFS	2.75	3.00	3.12	3.52	4.08	4.28	4.36	4.36
DIMACS Reference	0.50	0.54	0.57	0.59	0.57	0.58	0.60	0.60
Normalized to BFS	2.00	2.16	2.28	2.36	2.28	2.32	2.40	2.40

Problem Instance	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
BFS	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
Δ-stepping	1.08	1.09	1.09	1.09	1.08	1.09	1.08	1.09
Normalized to BFS	4.32	4.36	4.36	4.36	4.32	4.36	4.32	4.36
DIMACS Reference	0.59	0.60	0.61	0.59	0.61	0.60	0.60	0.60
Normalized to BFS	2.36	2.40	2.44	2.36	2.44	2.40	2.40	2.40

Table 2: Sequential performance (execution time in seconds, and normalized performance with reference to the baseline BFS) of our implementation for the core long grid graph families.

(a) Square-n core family. Problem instance denotes the log of the grid x dimension. $x = y$ and $\frac{m}{n} \simeq 4$.

Problem Instance	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**
BFS	.0001	.0003	.0007	.001	.003	.01	.04	.08	.20	.42	.93
Δ-stepping	.0008	.002	.004	.008	.03	.07	.20	.36	.81	2.05	4.38
Normalized to BFS	8.00	6.67	5.71	8.00	10.00	7.00	5.00	4.00	4.05	4.88	4.71
DIMACS Reference	.0003	.0007	.002	.006	.01	.03	.06	0.14	.36	.84	2.01
Normalized to BFS	3.00	2.33	2.86	6.00	3.33	3.00	1.50	1.75	1.80	2.00	2.16

(b) Square-C core family. Problem instance denotes the log of the edge weight. The grid dimensions are set to $x = y = 2^{10}$, and $n = xy$.

Problem Instance	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
BFS	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42
Δ-stepping	1.99	2.06	2.03	2.09	2.05	2.07	2.06	2.01
Normalized to BFS	4.74	4.90	4.83	4.89	4.88	4.93	4.90	4.79
DIMACS Reference	0.56	0.68	0.71	0.79	0.78	0.76	0.81	0.80
Normalized to BFS	1.33	1.62	1.69	1.88	1.86	1.81	1.93	1.90

Problem Instance	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
BFS	0.42	0.42	0.42	0.42	0.42	0.42	0.42	0.42
Δ-stepping	2.04	2.09	2.05	2.06	2.01	2.08	2.09	2.08
Normalized to BFS	4.86	4.98	4.88	4.90	4.78	4.95	4.98	4.95
DIMACS Reference	0.82	0.80	0.83	0.79	0.77	0.79	0.78	0.77
Normalized to BFS	1.95	1.90	1.98	1.88	1.83	1.88	1.86	1.83

Table 3: Sequential performance (execution time in seconds, and normalized performance with reference to the baseline BFS) of our implementation for the core square grid graph families.

(a) Core graphs from the USA road network, with the transit time as the length function.

Problem Instance	**CTR**	**W**	**E**	**LKS**	**CAL**	**NE**	**NW**	**FLA**	**COL**	**BAY**	**NY**
BFS	4.16	1.49	.65	.39	.26	.17	.16	.13	.04	0.03	.02
Δ-stepping	25.24	9.87	4.97	2.52	1.95	1.43	.89	.97	.30	.21	.15
Normalized to BFS	6.07	6.63	7.65	6.46	7.50	8.41	5.56	7.46	7.5	7.00	7.50
DIMACS Reference	9.06	3.12	1.65	1.14	.72	.58	.45	.36	.13	.09	.07
Normalized to BFS	2.18	2.09	2.54	2.92	2.77	3.41	2.81	2.77	3.25	3.00	3.50

(b) Core graphs from the USA road network, with the distance as the length function.

Problem Instance	**CTR**	**W**	**E**	**LKS**	**CAL**	**NE**	**NW**	**FLA**	**COL**	**BAY**	**NY**
BFS	4.32	1.89	1.05	.80	.54	.34	.31	.28	.05	.03	.02
Δ-stepping	21.63	10.34	7.02	3.52	3.67	1.06	1.26	1.17	0.15	0.11	0.08
Normalized to BFS	5.01	5.47	6.69	4.40	6.80	3.11	4.06	4.18	3.00	3.67	4.00
DIMACS Reference	15.52	4.91	3.12	2.24	1.41	0.86	0.71	0.55	0.13	0.08	0.07
Normalized to BFS	3.59	2.60	2.97	2.80	2.61	2.53	2.29	1.96	2.60	2.67	3.50

Table 4: Sequential performance (execution time in seconds, and normalized performance with reference to the baseline BFS) of our implementation for the core road networks.

B.2 Algorithm performance as a function of Δ

(a) Random4-n graph instance ($n = C = 2^{28}$, $m = 4n$).

Δ	0.1	0.5	1	5	10
No. of phases	328	122	89	58	50
Last non-empty bucket	93	19	9	1	0
Average distance	4.94	4.94	4.94	4.94	4.94
Avg. no. of light relax requests per phase	161K	1.84M	4.43M	8.28M	20.00M
Avg. no. of heavy relax requests per bucket	3.12M	5.46M	0	0	0
Total number of relaxations	343.20M	328.70M	394.30M	480.30M	1.00B
Execution Time (40 processors MTA-2, seconds)	14.03	11.64	13.57	16.15	27.14

(b) ScaleFree4-n graph instance ($n = C = 2^{25}$, $m = 4n$).

Δ	0.1	0.5	1	5	10
No. of phases	312	117	83	51	39
Last non-empty bucket	131	26	13	2	1
Average distance	1.68	1.68	1.68	1.68	1.68
Avg. no. of light relax requests per phase	22.40K	267.00K	667.00K	2.40M	3.15M
Avg. no. of heavy relax requests per bucket	278.00K	455.80K	0	0	0
Total number of relaxations	43.78M	43.63M	55.40M	122.68M	122.76M
Execution Time (40 processors MTA-2, seconds)	4.23	2.55	2.79	5.48	6.38

(c) RandomLogUnif4-n instance ($n = C = 2^{20}$, $m = 4n$).

Δ	0.001	0.05	0.1	0.5	1
No. of phases	460	115	93	77	71
Last non-empty bucket	134	17	8	4	2
Average distance	0.04	0.04	0.04	0.04	0.04
Avg. no. of light relax requests per phase	1.50K	50.80K	84.80K	132.00K	150.01K
Avg. no. of heavy relax requests per bucket	6.50K	3.47K	3.97K	2.18K	1.42K
Total number of relaxations	1.59M	5.91M	7.92M	10.17M	10.74M
Execution Time (40 processors MTA-2, seconds)	2.15	1.18	0.96	0.80	0.75

Table 5: Performance of the Δ-stepping algorithm as a function of the bucket width Δ. K denotes 10^3, M denotes 10^6 and B denotes 10^9.

(a) Long grid instance ($n = C = 2^{20}$).

Δ	0.1	0.5	1	5	10
No. of phases	295.27K	151.43K	124.76K	97.38K	92.70K
Last non-empty bucket	216.38K	43.28K	21.64K	4.33K	2.16K
Average distance	10805	10805	10805	10805	10805
Avg. no. of light relax requests per phase	0.65	5.49	11.29	22.99	37.22
Avg. no. of heavy relax requests per bucket	5.21	9.65	0	0	0
Total number of relaxations	1.32M	1.25M	1.40M	2.23M	3.45M
Execution Time (40 processors MTA-2, seconds)	858.75	465.14	443.05	369.57	274.23

(b) Square grid instance ($n = C = 2^{20}$).

Δ	0.1	0.5	1	5	10
No. of phases	12795	5489	4188	2769	2504
Last non-empty bucket	4691	938	469	93	46
Average distance	251.86	251.86	251.86	251.86	251.86
Avg. no. of light relax requests per phase	15.51	155.11	340.50	785.18	1248.72
Avg. no. of heavy relax requests per bucket	242.22	437.44	0	0	0
Total number of relaxations	1.33M	1.26M	1.43M	2.17M	3.13M
Execution Time (40 processors MTA-2, seconds)	48.77	20.04	13.92	9.17	8.32

(c) Central USA road instance (distance).

Δ	0.1	0.5	1	5
No. of phases	3129	2017	1669	1300
Last non-empty bucket	105	21	10	2
Average distance	3.93	3.93	3.93	3.93
Avg. no. of light relax requests per phase	6.34K	26.04K	57.89K	82.60K
Avg. no. of heavy relax requests per bucket	320.60	1.27	0	0
Total number of relaxations	19.87M	52.50M	96.60M	107.00M
Execution Time (40 processors MTA-2, seconds)	7.83	5.84	5.62	8.88

(d) NE USA road instance (transit time).

Δ	0.1	0.5	1	5
No. of phases	437	3542	3126	2220
Last non-empty bucket	315	63	31	6
Average distance	14.06	14.06	14.06	14.06
Avg. no. of light relax requests per phase	369.90	783.80	1.38K	8.66K
Avg. no. of heavy relax requests per bucket	168.40	1.85	0	0
Total number of relaxations	1.76M	2.78M	4.31M	19.21M
Execution Time (40 processors MTA-2, seconds)	12.92	9.25	8.39	6.84

Table 6: Performance of the Δ-stepping algorithm as a function of the bucket width Δ. K denotes 10^3, M denotes 10^6 and B denotes 10^9.

B.3 Parallel Performance on the Cray MTA-2

p	Problem Instance	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
1	BFS time (sec)	0.62	1.24	3.39	4.91	9.70	18.90	37.30	73.94
	Δ-stepping time (sec)	3.21	6.34	12.05	23.61	46.63	93.77	187.84	371.27
	Ratio to BFS	5.18	5.11	3.55	4.81	4.81	4.96	5.04	5.02
2	BFS time (sec)	0.31	0.61	1.19	2.34	4.65	9.29	18.66	37.06
	Δ-stepping time (sec)	1.92	3.44	6.57	12.72	24.88	48.40	96.15	187.99
	Ratio to BFS	6.25	5.66	5.52	5.43	5.35	5.21	5.15	5.07
	Relative Speedup	1.67	1.84	1.83	1.86	1.87	1.94	1.95	1.99
4	BFS time (sec)	0.16	0.31	0.61	1.19	2.37	4.71	9.38	19.59
	Δ-stepping time (sec)	1.23	2.07	3.77	7.07	13.63	25.40	50.08	96.89
	Ratio to BFS	7.69	6.68	6.18	5.94	5.75	5.39	5.34	4.95
	Relative Speedup	2.61	3.06	3.20	3.34	3.42	3.69	3.75	3.83
8	BFS time (sec)	0.09	0.16	0.31	0.60	1.18	2.35	4.73	9.37
	Δ-stepping time (sec)	0.96	1.40	2.39	4.28	8.04	13.81	27.29	49.18
	Ratio to BFS	10.67	8.48	7.74	7.13	6.81	6.88	5.77	5.25
	Relative Speedup	3.34	4.53	5.04	5.52	5.80	6.79	6.88	7.55
16	BFS time (sec)	0.06	0.10	0.17	0.32	0.62	1.20	2.39	4.73
	Δ-stepping time (sec)	0.84	1.24	1.84	3.06	5.45	8.34	15.91	25.47
	Ratio to BFS	7.55	12.40	10.60	9.22	8.83	6.95	6.66	5.38
	Relative Speedup	3.82	5.11	6.55	7.71	8.55	11.24	11.81	14.58
32	BFS time (sec)	0.05	0.07	0.11	0.19	0.36	0.69	1.36	2.68
	Δ-stepping time (sec)	0.78	1.047	1.52	2.42	4.12	5.70	10.31	13.90
	Ratio to BFS	15.60	15.00	13.81	12.74	11.44	15.83	7.58	5.19
	Relative Speedup	4.12	6.04	7.93	9.76	11.32	16.45	18.22	26.71
40	BFS time (sec)	0.04	0.06	0.10	0.17	0.32	0.61	1.20	2.37
	Δ-stepping time (sec)	0.81	1.05	1.53	2.35	3.98	5.15	9.51	11.96
	Ratio to BFS	18.41	16.41	15.30	13.82	12.44	8.44	7.92	5.04
	Relative Speedup	3.96	6.04	7.88	10.05	11.72	11.11	19.75	31.04

Table 7: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS, relative speedup) of our implementation on Random4-n graphs. Problem instance denotes log of the number of vertices. p denotes the number of processors. $m = 4n$ edges, and maximum weight $C = n$.

p	Problem Instance	**21**	**22**	**23**	**24**	**25**	**26**	**27**
1	BFS time (sec)	0.62	1.24	3.39	4.91	9.70	18.90	37.30
	Δ-stepping time (sec)	20.43	41.72	85.10	173.96	378.80	878.86	1687.59
	Ratio to BFS	32.95	33.64	25.10	35.43	39.05	46.50	45.24
4	BFS time (sec)	0.16	0.31	0.61	1.19	2.37	4.71	9.38
	Δ-stepping time (sec)	6.03	11.17	22.90	45.38	97.63	224.46	426.02
	Ratio to BFS	37.69	36.03	37.54	38.13	41.19	47.65	45.52
	Relative Speedup	3.38	3.73	3.72	3.83	3.88	3.91	3.96
16	BFS time (sec)	0.06	0.10	0.17	0.32	0.62	1.20	2.39
	Δ-stepping time (sec)	2.47	3.94	7.43	13.96	26.50	60.82	113.12
	Ratio to BFS	41.17	39.40	43.70	43.62	42.74	50.68	47.33
	Relative Speedup	8.27	10.59	11.45	12.46	14.29	14.45	14.92
40	BFS time (sec)	0.04	0.06	0.10	0.17	0.32	0.61	1.20
	Δ-stepping time (sec)	1.99	2.61	4.27	7.23	12.86	29.58	51.89
	Ratio to BFS	49.17	43.50	42.70	42.53	40.19	48.49	43.24
	Relative Speedup	10.27	15.98	19.93	24.06	29.46	29.71	32.52

Table 8: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS, relative speedup) of our implementation for RandomLogUnif4-n graphs. Problem instance denotes the log of the number of vertices. p denotes the number of processors. $m = 4n$ edges and maximum weight $C = n$.

p	Problem Instance	**0**	**3**	**6**	**9**	**12**	**15**
1	BFS time (sec)	19.07	19.07	19.07	19.07	19.07	19.07
	Δ-stepping time (sec)	93.66	93.74	94.34	93.22	95.76	94.11
	Ratio to BFS	4.91	4.91	4.89	4.89	5.01	4.83
2	BFS time (sec)	9.38	9.38	9.38	9.38	9.38	9.38
	Δ-stepping time (sec)	48.24	48.15	48.78	48.5	49.25	48.63
	Ratio to BFS	5.14	5.13	5.20	5.17	5.25	5.18
	Relative Speedup	1.94	1.95	1.91	1.92	1.94	1.93
4	BFS time (sec)	4.73	4.73	4.73	4.73	4.73	4.73
	Δ-stepping time (sec)	25.81	25.43	25.47	25.81	25.39	25.35
	Ratio to BFS	5.46	5.38	5.38	5.46	5.37	5.36
	Relative Speedup	3.63	3.69	3.66	3.61	3.77	3.71
8	BFS time (sec)	2.36	2.36	2.36	2.36	2.36	2.36
	Δ-stepping time (sec)	14.06	13.67	13.86	13.85	14.07	13.85
	Ratio to BFS	5.96	5.79	5.87	5.87	5.96	5.87
	Relative Speedup	6.66	6.86	6.73	6.73	6.80	6.79
16	BFS time (sec)	1.21	1.21	1.21	1.21	1.21	1.21
	Δ-stepping time (sec)	8.37	8.38	8.4	8.37	8.42	8.38
	Ratio to BFS	6.92	6.92	6.94	6.92	6.96	6.92
	Relative Speedup	11.19	11.19	11.11	11.14	11.37	11.23
32	BFS time (sec)	0.69	0.69	0.69	0.69	0.69	0.69
	Δ-stepping time (sec)	5.66	5.65	5.66	5.68	5.66	5.67
	Ratio to BFS	8.20	8.19	8.20	8.23	8.20	8.21
	Relative Speedup	11.42	11.45	11.38	11.32	11.67	11.45
40	BFS time (sec)	0.61	0.61	0.61	0.61	0.61	0.61
	Δ-stepping time (sec)	5.23	5.27	5.22	5.23	5.21	5.26
	Ratio to BFS	8.52	8.58	8.50	8.52	8.48	8.57
	Relative Speedup	17.91	17.79	17.88	17.82	18.38	17.89

Table 9: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS, relative speedup) of our implementation on Random4-C graphs. Problem instance denotes the log of the maximum edge weight. p denotes the number of processors. $n = 2^{26}$ vertices, $m = 4n$ edges.

p	Problem Instance	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
1	BFS time (sec)	0.54	1.22	1.54	4.19	7.60	14.30	34.90	55.62
	Δ-stepping time (sec)	3.99	8.57	13.77	32.11	57.16	123.73	243.53	404.91
	Ratio to BFS	7.39	7.02	8.94	7.66	7.52	8.65	6.97	7.28
4	BFS time (sec)	0.74	1.43	2.12	4.52	9.27	19.80	39.48	71.49
	Δ-stepping time (sec)	5.36	11.20	17.92	42.06	73.93	158.72	306.69	567.63
	Ratio to BFS	7.24	7.83	8.45	9.30	7.97	8.01	7.77	7.94
16	BFS time (sec)	1.04	1.85	3.09	6.72	13.56	25.44	57.71	107.00
	Δ-stepping time (sec)	7.10	15.07	23.50	56.08	97.99	212.51	503.33	967.70
	Ratio to BFS	6.83	8.14	7.60	8.34	7.23	8.35	8.72	9.04
40	BFS time (sec)	1.31	2.43	4.00	8.29	18.14	32.33	72.99	132.36
	Δ-stepping time (sec)	12.53	23.64	40.02	90.59	171.13	330.72	812.02	1534.05
	Ratio to BFS	9.56	9.73	10.00	10.97	9.43	10.23	11.12	11.59

Table 10: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS) of our implementation on Long-n graphs. Problem instance denotes the log of the rectangular grid x dimension. p denotes the number of processors, $y = 16$, $n = xy$, $m \simeq 4n$ edges, and maximum weight $C = n$.

p	Problem Instance	**0**	**3**	**6**	**9**	**12**	**15**
1	BFS time (sec)	7.60	7.60	7.60	7.60	7.60	7.60
	Δ-stepping time (sec)	57.24	56.88	57.13	57.89	58.11	56.97
	Ratio to BFS	7.53	7.48	7.52	7.62	7.65	7.50
4	BFS time (sec)	9.27	9.27	9.27	9.27	9.27	9.27
	Δ-stepping time (sec)	74.02	73.88	73.92	74.68	75.17	75.49
	Ratio to BFS	7.98	7.97	7.97	8.06	8.10	8.14
16	BFS time (sec)	13.56	13.56	13.56	13.56	13.56	13.56
	Δ-stepping time (sec)	96.76	97.11	97.45	98.82	98.30	98.61
	Ratio to BFS	7.14	7.16	7.19	7.24	7.25	7.27
40	BFS time (sec)	18.14	18.14	18.14	18.14	18.14	18.14
	Δ-stepping time (sec)	172.00	171.34	173.43	172.84	172.49	173.19
	Ratio to BFS	9.48	9.44	9.56	9.53	9.51	9.55

Table 11: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS) of our implementation on Long-C graphs. Problem instance denotes the log of the maximum edge weight. p denotes the number of processors. The grid dimensions are given by $x = 2^{14}$ and $y = 16$.

p	Problem Instance	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	BFS time (sec)	0.05	0.12	0.24	0.57	1.32	3.22	8.55
	Δ-stepping time (sec)	0.20	0.52	1.28	2.80	7.84	20.56	68.33
	Ratio to BFS	4.00	4.33	5.33	4.91	5.94	6.38	7.99
4	BFS time (sec)	0.09	0.16	0.33	0.72	1.51	3.19	6.59
	Δ-stepping time (sec)	0.23	0.60	1.41	2.84	6.85	14.29	38.62
	Ratio to BFS	2.55	3.75	4.27	3.94	4.54	4.48	5.86
16	BFS time (sec)	0.11	0.22	0.41	0.95	1.99	3.93	7.68
	Δ-stepping time (sec)	0.28	0.73	1.64	3.3	7.83	14.93	35.51
	Ratio to BFS	2.54	3.32	4.00	3.47	3.93	3.80	4.62
40	BFS time (sec)	0.12	0.23	0.44	1.00	2.05	4.01	7.90
	Δ-stepping time (sec)	0.35	0.84	1.91	3.59	8.35	15.29	35.46
	Ratio to BFS	2.92	3.65	4.34	3.59	4.07	3.81	4.49

Table 12: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS) of our implementation on Square-n graphs. Problem instance denotes the log of the number of the grid dimension x. p denotes the number of processors. $x = y$, $n = xy$, $m \simeq 4n$ edges, and maximum weight $C = n$.

p	Instance	**CTR**	**W**	**E**	**LKS**	**CAL**	**NE**	**NW**	**FLA**	**COL**	**BAY**	**NY**
1	BFS time	3.58	2.05	1.30	1.14	0.80	0.96	0.84	0.83	1.40	0.70	0.70
	Δ-stepping time	19.89	12.91	7.12	5.08	3.09	4.33	3.80	2.76	6.52	3.16	2.70
	Ratio to BFS	5.56	6.30	5.48	4.46	3.86	4.51	4.52	3.32	4.66	4.51	3.86
4	BFS time	1.61	1.21	0.83	0.79	0.56	0.79	0.76	0.73	1.61	0.77	0.80
	Δ-stepping time	8.58	7.12	3.29	3.59	1.92	4.21	3.41	2.28	7.90	3.70	3.39
	Ratio to BFS	5.33	5.88	3.96	4.54	3.43	5.33	4.49	3.12	4.91	4.80	4.24
16	BFS time	1.28	1.16	0.84	0.84	0.56	0.87	0.85	0.81	1.91	0.95	0.89
	Δ-stepping time	6.83	6.74	2.91	3.94	1.96	5.03	4.06	5.28	9.81	4.62	4.08
	Ratio to BFS	5.34	5.81	3.46	4.69	3.50	5.78	4.78	6.52	5.14	4.86	4.58
40	BFS time	1.30	1.21	0.86	0.93	0.68	0.95	0.98	0.91	2.19	1.06	1.00
	Δ-stepping time	6.75	7.07	2.86	4.33	2.14	5.92	5.01	3.13	12.99	5.63	5.06
	Ratio to BFS	5.19	5.84	3.32	4.66	3.15	6.23	5.11	3.44	5.93	5.31	5.06

Table 13: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS) of our implementation on USA core road graphs (distance is the length function). p denotes the number of processors.

p	Instance	**CTR**	**W**	**E**	**LKS**	**CAL**	**NE**	**NW**	**FLA**	**COL**	**BAY**	**NY**
1	BFS time	7.69	5.19	3.95	3.38	2.39	2.01	1.52	1.98	1.51	0.72	0.67
	Δ-stepping time	37.06	24.01	15.12	14.51	11.32	6.66	7.06	9.22	5.03	2.34	1.69
	Ratio to BFS	4.82	4.63	3.83	4.29	4.74	3.31	4.64	4.66	3.33	3.24	2.52
4	BFS time	6.48	4.95	4.09	3.6	2.53	2.14	1.57	2.15	1.76	0.83	0.76
	Δ-stepping time	34.38	23.75	15.73	15.36	12.03	7.18	8.07	10.45	5.95	2.73	1.91
	Ratio to BFS	5.31	4.80	3.84	4.27	4.75	3.31	5.14	4.86	3.38	3.29	2.51
16	BFS time	7.26	5.85	4.94	4.32	3.02	2.53	1.86	2.56	2.13	1.01	0.94
	Δ-stepping time	39.95	27.86	18.53	18.21	14.25	8.54	9.64	12.41	7.14	3.25	2.31
	Ratio to BFS	5.50	4.76	3.75	4.21	4.72	3.75	5.18	4.85	3.35	3.33	2.46
40	BFS time	7.50	6.56	5.47	4.43	3.37	2.92	2.18	2.95	2.19	1.05	0.95
	Δ-stepping time	42.94	30.24	21.15	20.09	16.29	9.96	12.05	14.54	9.09	3.95	2.82
	Ratio to BFS	5.72	4.61	3.87	4.53	4.83	3.41	5.53	4.93	7.64	3.76	2.97

Table 14: MTA-2 performance (execution time in seconds, normalized performance with reference to the baseline BFS) of our implementation on USA core road graphs (transit time is the length function).

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] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, 2002.

[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.

[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.

[6] D.A. Bader, K. Madduri, G. Cong, and J. Feo. Design of multithreaded algorithms for combinatorial problems. In S. Rajasekaran and J. Reif, editors, *Handbook of Parallel Computing: Models, Algorithms, and Applications*. CRC Press, 2006. to appear.

[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)*, May 2006.

[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.

[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] Cray, Inc. The MTA-2 multithreaded architecture. `http://www.cray.com/products/systems/mta/`.

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

[19] C. Demetrescu, A. Goldberg, and D. Johnson. 9th DIMACS implementation challenge – Shortest Paths: Reference benchmark package. `http://www.dis.uniroma1.it/~challenge9/download.shtml`.

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

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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. Springer-Verlag, 2000.

[36] 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.

[37] 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.

[38] M.R. Hribar, V.E. Taylor, and D.E. Boyce. Parallel shortest path algorithms: Identifying the factors that affect performance. Report CPDC-TR-9803-01i5, Northwestern University, Evanston, IL, 1998.

[39] 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.

[40] 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.

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

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

[43] 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.

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

[45] K. Madduri and D.A. Bader. GTgraph: A suite of synthetic graph generators. `http://www.cc.gatech.edu/~kamesh/GTgraph`.

[46] 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. Springer-Verlag, 2000.

[47] 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.

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

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

[50] 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. Springer-Verlag, 2000.

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

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

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

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

[55] 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.

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

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

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

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

[60] 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.

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