Advanced Shortest Paths Algorithms on a Massively-Multithreaded Architecture

Joseph R. Crobak[1], Jonathan W. Berry[2], Kamesh Madduri[3], and David A. Bader[3]

[1]Rutgers University
Dept. of Computer Science
Piscataway, NJ 08854 USA
crobakj@cs.rutgers.edu

[2]Sandia National Laboratories
Albuquerque, NM USA
jberry@sandia.gov

[3]Georgia Institute of Technology
College of Computing
Atlanta, GA 30332 USA
{kamesh,bader}@cc.gatech.edu

Abstract

We present a study of multithreaded implementations of Thorup's algorithm for solving the Single Source Shortest Path (SSSP) problem for undirected graphs. Our implementations leverage the fledgling MultiThreaded Graph Library (MTGL) to perform operations such as finding connected components and extracting induced subgraphs. To achieve good parallel performance from this algorithm, we deviate from several theoretically optimal algorithmic steps. In this paper, we present simplifications that perform better in practice, and we describe details of the multithreaded implementation that were necessary for scalability.

We study synthetic graphs that model unstructured networks, such as social networks and economic transaction networks. Most of the recent progress in shortest path algorithms relies on structure that these networks do not have. In this work, we take a step back and explore the synergy between an elegant theoretical algorithm and an elegant computer architecture. Finally, we conclude with a prediction that this work will become relevant to shortest path computation on structured networks.

1. Introduction

Thorup's algorithm [15] solves the SSSP problem for undirected graphs with positive integer weights in linear time. To accomplish this, Thorup's algorithm encapsulates information about the input graph in a data structure called the *Component Hierarchy (CH)*. Based upon information in the *CH*, Thorup's algorithm identifies vertices that can be settled in arbitrary order. This strategy is well suited to a shared-memory environment since the component hierarchy can be constructed only once, then shared by multiple concurrent SSSP computations.

Thorup's SSSP algorithm and the data structures that it uses are complex. The algorithm has been generalized to run on directed graphs in $O(n + m \log w)$ time [8] (where w is word-length in bits) and in the pointer-addition model of computation in $O(m\alpha(m, n) + n \log \log r)$ time [13] (where $\alpha(m, n)$ is Tarjan's inverse-Ackermann function and r is the ratio of the maximum-to-minimum edge length).

In this paper, we perform an experimental study of Thorup's original algorithm. In order to achieve good performance, our implementation uses simple data structures and deviates from some theoretically optimal algorithmic strategies. Thorup's SSSP algorithm is complex, and we direct the reader to his original paper for a complete explanation.

In the following section, we summarize related work and describe in detail the Component Hierarchy and Thorup's algorithm. Next, we discuss the details of our multithreaded implementation of Thorup's algorithm and detail the experimental setup. Finally, we present experimental results and plans for future work.

2. Background and Related Work

The Cray MTA-2 and its successor, the XMT [4], are massively multithreaded machines that provide elaborate

1-4244-0910-1/07/$20.00

hardware support for latency tolerance, as opposed to latency mitigation. Specifically, a large amount of chip space is devoted to supporting many thread contexts in hardware rather than providing cache memory and its associated complexity. This architecture is ideal for graph algorithms, as they tend to be dominated by latency and to benefit little from cache.

We are interested in leveraging such architectures to solve large shortest paths problems of various types. Madduri, et al. [11] demonstrate that for certain inputs, *delta-stepping* [12], a parallel Dijkstra variant, can achieve relative speedups of roughly 30 in 40-processor runs on the MTA-2. This performance is achieved while finding single-source shortest paths on an unstructured graph of roughly one billion edges in roughly 10 seconds. However, their study showed that there is not enough parallelism in smaller unstructured instances to keep the MTA-2 busy. In particular, similar instances of roughly one million edges yielded relative speedups of only about 3 on 40 processors of the MTA-2. Furthermore, structured instances with large diameter, such as road networks, prove to be very difficult for parallel delta stepping regardless of instance size.

Finding shortest paths in these structured road network instances has become an active research area recently [1, 9]. When geographical information is available, precomputations to identify "transit nodes" [1] make subsequent s-t shortest path queries extremely fast. However, depending on the parameters of the algorithms, serial precomputation times range from 1 to 11 hours on modern 3Ghz workstations. We know of no work to parallelize these precomputations.

Although we do not explicitly address that challenge in this paper, we do note that the precomputations tend to consist of Dijkstra-like searches through hierarchical data. These serial searches could be batched trivially into parallel runs, but we conjecture that this process could be accelerated even further by the basic idea of allowing multiple searches to share a common component hierarchy. In this paper, we explore the utility of this basic idea.

2.1. The Component Hierarchy

The *Component Hierarchy (CH)* is a tree structure that encapsulates information about a graph G. The *CH* of an undirected graph with positive edge weights can be computed directly, but preprocessing is needed if G contains zero-weight edges. Each *CH-node* represents a subgraph of G called a *component*, which is identified by a vertex v and a level i. *Component*(v,i) is the subgraph of G composed of vertex v, the set S of vertices reachable from v when traversing edges with weight $< 2^i$, and all edges adjacent to $\{v\} \cup S$ of weight less than 2^i. Note that if w $\in$*Component*(v,i), then *Component*(v,i) = *Component*(w,i).

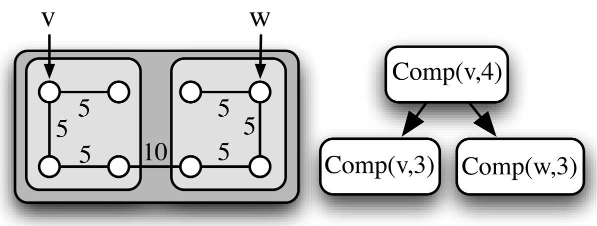


Figure 1. An example component hierarchy. *Component(v,4)*, **the root of this hierarchy, represents the entire graph.**

The root *CH-node* of the *CH* is a component containing the entire graph, and each leaf represents a singleton vertices. The children of *Component*(v,i) in the *CH* are components representing the connected components formed when removing all the edges with weight $> 2^{i-1}$ from *Component*(v,i). See Figure 1 for an example *CH*.

2.2. Thorup's SSSP Algorithm

Given an undirected graph $G = (V, E)$, a source vertex $s \in V$, and a length function $\ell : E \to \mathbb{Z}^+$, the Single Source Shortest Path (SSSP) problem is to find $\delta(v)$ for $v \in V \backslash s$. The value $\delta(v)$ is the length of the shortest path from s to v in G. By convention, $\delta(v) = \infty$ if v is unreachable from s.

Most shortest path problems maintain a *tentative distance* value, $d(v)$, for each $v \in V$. This value is updated by *relaxing* the edges out of a vertex v while *visiting* v. Relaxing an edge $e = (u, v)$ sets $d(v) = \texttt{min}(d(v), d(u) + \ell(e))$. Dijkstra [6] noted in his famous paper that the problem can be solved by visiting vertices in nondecreasing order of their d-values. Dijkstra's algorithm maintains three sets of vertices: *unreached, queued,* and *settled.* A vertex v is settled when $d(v) = \delta(v)$ (initially only s is settled), is queued when $d(v) < \infty$, and is unreached when a path to v has not yet been found ($d(v) = \infty$). Dijkstra's algorithm repeatedly selects vertex v such that $d(v)$ is minimum for all queued vertices and visits v.

Thorup's algorithm uses the *CH* to identify vertices that can be visited in arbitrary order ($d(v) = \delta(v)$). His major insight is presented in the following Lemma.

Lemma 1 (From Thorup [15])**.** *Suppose the vertex set V divides into disjoint subsets $V_1, \ldots, V_k$ and that all edges between subsets have weight at least Δ. Let S be the set of settled vertices. Suppose for some i such that $v \in V_i \backslash S$, that $d(v) = min\{d(x)|x \in V_i \backslash S\} \leq min\{d(x)|x \in V \backslash S\} + \delta$. Then $d(v) = \delta(v)$ (see Figure 2).*

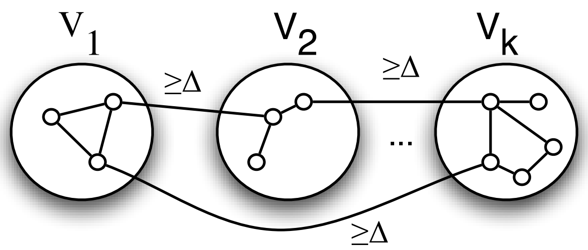


Figure 2. The vertex set V divided into k subsets.

Based upon this Lemma, Thorup's algorithm identifies vertices that can be visited in arbitrary order. Let $\alpha = \log_2 \Delta$. Component V buckets each of it's children $V_1 \ldots V_k$ according to $\mathtt{min}\{d(x)|x \in V_i \backslash S\} \gg \alpha$. Note that $(\mathtt{min}\{d(x)|x \in V_i \backslash S\} \gg \alpha) \leq (\mathtt{min}\{d(x)|x \in V \backslash S\} \gg \alpha)$ implies that $(\mathtt{min}\{d(x)|x \in V_i \backslash S\}) \leq (\mathtt{min}\{d(x)|x \in V \backslash S\} + \Delta)$. Consider bucket $B[j]$ such that j is the smallest index of a non-empty bucket. If $V_i \in B[j]$ then $\mathtt{min}\{d(x)|x \in V_i \backslash S\} \gg \alpha = \mathtt{min}\{d(x)|x \in V \backslash S\} \gg \alpha$. This implies that $\mathtt{min}\{d(x)|x \in V_i \backslash S\} \leq \mathtt{min}\{d(x)|x \in V \backslash S\} + \Delta$. Thus, each $v \in V_i \backslash S$ minimizing $D(v)$ can be visited by Lemma 2.2.

This idea can be applied recursively for each component in the *CH*. Each *component(v,i)* buckets each child V_j based upon $\mathtt{min}\{d(x)|x \in V_j \backslash S\}$. Beginning at the root, Thorup's algorithm *visits* its children recursively, starting with those children in the bucket with the smallest index. When a leaf component l is reached, the vertex v represented by l is visited (all of its outgoing edges are relaxed). Once a bucket is empty, the components in the next highest bucket are visited and so on. We direct the reader to Thorup [15] for details about correctness and analysis.

3. Implementation Details

Before computing the shortest path, Thorup's algorithm first constructs the Component Hierarchy. We developed a parallel algorithm to accomplish this. For each component c in the Component Hierarchy, Thorup's algorithm maintains *minD*$(c) = \mathtt{min}(d(x)|x \in c \backslash S)$. Additionally, c must bucket each child c_i according to the value of the *minD*(c_i). When visiting c, children in the bucket with smallest index are visited recursively and in parallel.

Our algorithm to generate the Component Hierarchy is described in Section 3.1. The implementation strategies for maintaining *minD*-values and proper bucketing are described in section 3.2. Finally, our strategies for visiting components in parallel are described in Section 3.3.

Input: $G(V, E)$, length function $\ell : E \rightarrow \mathbb{Z}^+$
Output: *CH*(G), the Component Hierarchy of G

foreach $v \in V$ **do**
    Create leaf *CH-node* n and set *component*(v) to n
$G' \leftarrow G$
for $i = 1$ *to* $\lceil \log C \rceil$ **do**
    Remove edges of weight $\geq 2^i$ from G'
    Find the connected components of G'
    Create a new graph G^*
    foreach *connected component* c *of* G' **do**
        Create a vertex x in G^*
        Create new *CH-node* n for x
        component$(x) \leftarrow n$
        foreach $v \in c$ **do**
            rep$(v) \leftarrow x$
            parent(component$(v)) \leftarrow n$
    foreach *edge* $(u, v) \in G$ **do**
        Create an edge (*rep*(u),*rep*(v)) in G^*
    $G' \leftarrow G^*$

Algorithm 1: Generate Component Hierarchy

3.1. Generating the Component Hierarchy

Thorup [15] presents a linear-time algorithm for constructing the component hierarchy from the minimum spanning tree. Rather than using this approach, we build the *CH* naively in $\lceil \log C \rceil$ phases, where C is the length of the largest edge. Our algorithm is presented in Algorithm 1.

Constructing the minimum spanning tree is pivotal in Thorup's analysis. However, we build the *CH* from the original graph because this is faster in practice than first constructing the MST and then constructing the *CH* from it. This decision creates extra work, but it does not greatly affect parallel performance because of the data structures we use, which are described in Section 3.2.

Our implementation relies on repeated calls of a connected components algorithm, and we use the "bully algorithm" for connected components available in the MultiThreaded Graph Library (MTGL) [2]. This algorithm avoids hot spots inherent in the Shiloach-Vishkin algorithm [14] and demonstrates near-perfect scaling through 40 MTA-2 processors on the unstructured instances we study.

3.2. Run-time Data Structures

We define *minD*(c) for component c as $\mathtt{min}(d(x)|x \in c \backslash S)$. The value of *minD*$(c)$ can change when the $d(v)$ decreases for vertex $v \in c$, or it can change when a vertex $v \in c$ is *visited* (added to S). Changes in a component's *minD*-value might also affect ancestor component's in the *CH*. Our implementation updates *minD* values by propagating values from leaves towards the root. Our implementation must lock the value of *minD* during an update since multiple vertices are visited in parallel. Locking on *minD* does not create contention between threads because *minD* values are not propagated very far up the *CH* in practice.

Conceptually, each component c at level i has an array of buckets. Each child c_k of c is in the bucket indexed *minD*$(c_k) \gg i$. Buckets are bad data structures for a parallel machine because they do not support simultaneous insertions. Rather that explicitly storing an array of buckets, each component c stores *index*(c), which is c's index into its parents buckets. Child c_k of component c is in bucket j if *index*$(c_k) = j$. Thus, inserting a component into a bucket is accomplished by modifying *index*(c). Inserting multiple components into buckets and finding the children in a given bucket can be done in parallel.

3.3. Traversing the Component Hierarchy in parallel

The Component Hierarchy is an irregular tree, in which some nodes have several thousand children and others only two. Additionally, it is impossible to know how much work must be done in a subtree because as few as one vertex might be visited during the traversal of a subtree. These two facts make it difficult to efficiently traverse the *CH* in parallel. To make traversal of the tree efficient, we have split the process of recursively visiting the children of a component into a two step process. First, we build up a list of components to visit. Second, we recursively visit these nodes.

Throughout execution, Thorup's algorithm maintains a *current bucket* for each component (in accordance with Lemma 2.2). All of those children (virtually) in the *current bucket* compose the list of children to be visited, called the *toVisit* set. To build this list, we look at all of node n's children and add each child that is (virtually) in the current bucket to an array. The MTA supports automatic parallelization of such a loop with the *reduction* mechanism. On the MTA, code to accomplish this is shown in Figure 3.

Executing a parallel loop has two major expenses. First, the runtime system must setup for the loop. In the case of a *reduction*, the runtime system must fork threads and divides the work across processors. Second, the body of the loop is executed and the threads are abandoned. If the number of iterations is large enough, then the second expense far outweighs the first. Yet, in the case of the *CH*, each node can have between two and several hundred thousand children. In the former case, the time spent setting up for the loop far outweighs the time spent executing the loop body. Since the *toVisit* set must be built several times for each node in the *CH* (and there are $O(n)$ nodes in the *CH*), we designed a

```
int index=0;
#pragma mta assert nodep
for (int i=0; i<numChildren; i++) {
  CHNode *c = children_store[i];
  if (bucketOf[c->id] == thisBucket) {
    toVisit[index++] = child->id;
  }
}
```

Figure 3. Parallel code to populate the toVisit set with children in the current bucket.

more efficient strategy for building the *toVisit* set.

Based upon the number of iterations, we either perform this loop on all processors, a single processor, or in serial. That is, if *numChildren* $>$ *multi_par_threshold* then we perform the loop in parallel on all processors. Otherwise, if *numChildren* $>$ *single_par_threshold* then we perform the loop in parallel on a single processor. Otherwise, the loop is performed in serial. We determined the thresholds experimentally by simulating the *toVisit* computation. In Section 5.4, we present a comparison of the naive approach and our approach.

4. Experimental Setup

4.1. Platform

The Cray MTA-2 is a massively multithreaded supercomputer with slow, 220Mhz processors and a fast, 220Mhz network. Each processor has 128 hardware threads, and the network is capable of processing a memory reference from every processor at every cycle. The run-time system automatically saturates the processors with as many threads are as available. We ran our experiments on a 40 processor MTA-2, the largest one ever built. This machine has 160Gb of RAM, of which 145Gb are usable. The MTA-2 has support for primitive locking operations, as well as many other interesting features. An overview of the features is beyond the scope of this discussion, but is available as Appendix A of [10].

In addition to the MTA-2, our implementation compiles on sequential processors without modification. We used a Linux workstation to evaluate the sequential performance of our Thorup implementation. Our results were generated on a 3.4GHz Pentium 4 with 1MB of cache and 1GB of RAM. We used the Gnu Compiler Collection, version 3.4.4.

4.2. Problem Instances

We evaluate the parallel performance on two graph families that represent unstructured data. The two families are among those defined in the 9th DIMACS Implementation Challenge [5]:

- *Random graphs*: These are generated by first constructing a cycle, and then adding $m - n$ edges to the graph at random. The generator may produce parallel edges as well as self-loops.
- *Scale-free graphs (RMAT)*: We use the R-MAT graph mode [3] to generate Scalefree instances. This algorithm recursively fills in an adjacency matrix in such a way that the distribution of vertex degrees obeys an inverse power law.

For each of these graph classes, we fix the number of undirected edges, m by $m = 4n$. In our experimental design, we vary two factors: C, the maximum edge weight, and the weight distribution. The latter is either uniform in $[1, ..., C]$ (UWD) or *poly-logarithmic* (PWD). The *poly-logarithmic* distribution generates integer weights of the form 2^i, where i is chosen uniformly over the distribution $[1, \log C]$.

In the following figures and tables, we name data sets with the convention: `<class>-<dist>-<n>-<C>`.

4.3. Methodology

We first explore the sequential performance of the Thorup code on a Linux workstation. We compare this to the serial performance of the "DIMACS reference solver," an implementation of Goldberg's multilevel bucket shortest path algorithm, which has an expected running time of $O(n)$ on random graphs with uniform weight distributions [7]. We compare these two implementations to establish that our implementation is portable and that it does not perform much extra work. It is reasonable to compare these implementations because they operate in the same environment, use the same compiler, and use the similar graph representation. Because these implementations are part of different packages, the only graph class we are able to compare is Random-UWD.

We collected data about many different aspects of the Component Hierarchy generation. Specifically, we measured number of components, average number of children, memory usage, and instance size. These numbers give a platform independent view of the structure of the graph as represented by the Component Hierarchy.

On the MTA-2, we first explore the relative speedup of our multithreaded implementation of Component Hierarchy construction and Thorup's algorithm by varying the number of processors and holding the other factors constant. We also show the effectiveness of our strategy for building the *toVisit* set. Specifically, we compare the theoretically optimal approach to our approach of selecting from three loops with different levels of parallelism. Our time measurements for Thorup's algorithm are an average of 10 SSSP runs.

Family	Thorup	DIMACS
Rand-UWD-2^{20}-2^{20}	4.31s	1.66s
Rand-UWD-2^{20}-2^{2}	2.66s	1.24s

Table 1. Thorup sequential performance versus the DIMACS reference solver.

Family	Comp.	Children	Instance
Rand-UWD-2^{24}-2^{24}	20.79	5.18	4.01GB
Rand-UWD-2^{24}-2^{2}	17.24	37.02	3.49GB
Rand-PWD-2^{24}-2^{24}	17.25	36.63	3.20GB
RMAT-UWD-2^{24}-2^{24}	19.98	6.23	3.83GB
RMAT-UWD-2^{24}-2^{2}	17.58	21.88	3.54GB
RMAT-PWD-2^{24}-2^{24}	17.66	19.92	3.29GB

Table 2. Statistics about the CH. "Comp" is total components in the CH (millions). "Children" is average number of children per component. "Instance" is memory required for a single instance.

Conversely, we only measure a single run of the Component Hierarchy construction.

After verifying that our implementation scales well, we compare it to the multithreaded delta stepping implementation of [11]. Finding our implementation to lag behind, we explore the idea of allowing many SSSP computations to share a common component hierarchy and its performance compared to a sequence of parallel (but single-source) runs of delta stepping.

5. Results and Analysis

5.1. Sequential Results

We present the performance results of our implementation of Thorup's algorithm on two graph families: *Random-UWD*-2^{20}-2^{20} and *Random-UWD*-2^{20}-2^{2}. Our results are presented in Table 1. In addition to the reported time, Thorup requires a preprocessing step that takes 7.00s for both graph families. The results show that there is a large performance hit for generating the Component Hierarchy, but once generated the execution time of Thorup's algorithm is within 2-4x of the DIMACS reference solver. Our code is not optimized for serial computation, especially the code to generate the Component Hierarchy. Regardless, our Thorup computation is reasonably close to the time of the DIMACS reference solver.

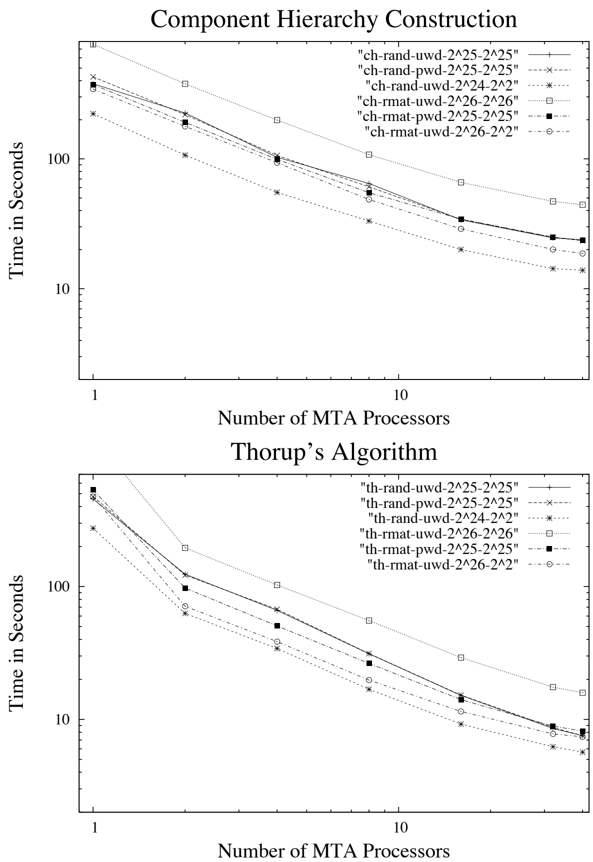


Figure 4. Scaling of Thorup's algorithm on the MTA-2.

5.2. Component Hierarchy Analysis

Several statistics of the *CH* across different graph families are shown in Table 2. All graphs have about the same number of vertices and edges and thus require about the same amount of memory– namely 5.76GB. It is more memory efficient to allocate a new instance of the *CH* than it is to create a copy of the entire graph. Thus, multiple Thorup queries using a shared *CH* is more efficient than several Δ-Stepping queries each with a separate copy of the graph.

The most interesting categories in Table 2 are the number of components and the average number of children. Graphs favoring small edge weights ($C = 2^2$ and *PWD*) have more children on average and a fewer number of components. In Section 5.3, we find that graphs favoring small edge weights have faster running times.

5.3. Parallel Performance

We present the parallel performance of constructing the Component Hierarchy and computing SSSP queries in detail. We ran Thorup's algorithm on graph instances from the Random and RMAT graph families, with uniform and polylog weight distributions, and with small and large maximum edge weights. We define the speedup on p processors of the MTA-2 as the ratio of the execution time on one processor to the execution time on p processors. Note that since the MTA-2 is thread-centric, single processor runs are also parallel. In each instance, we computed the speedup based upon the largest graph that fits into the RAM of the MTA-2.

Graph Family	CH	CH Speedup
Rand-UWD-2^{25}-2^{25}	23.85s	15.89
Rand-PWD-2^{25}-2^{25}	23.41s	18.27
Rand-UWD-2^{24}-2^{2}	13.87s	16.04
RMAT-UWD-2^{26}-2^{26}	44.33s	17.19
RMAT-PWD-2^{25}-2^{25}	23.58s	15.83
RMAT-UWD-2^{26}-2^{2}	18.67s	18.45

Table 3. Running time and speedup for generating the CH on 40 processors.

Graph Family	Thorup	Thorup Speedup
Rand-UWD-2^{25}-2^{25}	7.53s	60.51
Rand-PWD-2^{25}-2^{25}	7.54s	63.09
Rand-UWD-2^{24}-2^{2}	5.67s	48.45
RMAT-UWD-2^{26}-2^{26}	15.86s	85.55
RMAT-PWD-2^{25}-2^{25}	8.16s	65.42
RMAT-UWD-2^{26}-2^{2}	7.39s	64.36

Table 4. Running time and speedup for Thorup's algorithm on 40 processors.

Both the Component Hierarchy construction and SSSP computations scale well on the instances studied (see Figure 4). Running times and speedups on 40 processors are detailed in Tables 3 and 4. For a RMAT graph with 2^{26} vertices, 2^{28} undirected edges, and edge weights in the range $[1, 4]$, Thorup takes 7.39 seconds after 18.67 seconds of preprocessing on 40 processors. With the same number of vertices and edges, but edge weights in the range $[1, 2^{26}]$, Thorup takes 15.86 seconds. On random graphs, we find that graphs with PWD and UWD distributions have nearly identical running times on 40 processors (7.53s for UWD and 7.54s for PWD).

For all graph families, we attain a relative speedup from one to forty processors that is greater than linear. We attribute this contradiction to an anomaly present when running Thorup's algorithm on a single processor. Namely, we see speedup of between three and seven times when going from one to two processors. This is unexpected, since the optimal speedup should be twice that of one processor. On a single processor, loops with a large amount of work only receive a single thread of execution in some cases because the

Family	Δ-Stepping	Thorup	CH
Rand-UWD-2^{25}-2^{25}	4.95s	7.53s	23.85s
Rand-PWD-2^{25}-2^{25}	4.95s	7.54s	23.41s
Rand-UWD-2^{24}-2^{2}	2.34s	5.67s	13.87s
RMAT-UWD-2^{26}-2^{26}	5.74s	15.86s	44.33s
RMAT-PWD-2^{25}-2^{25}	5.37s	8.16s	23.58s
RMAT-UWD-2^{26}-2^{2}	4.66s	7.39s	18.67s

Table 5. Comparison of Delta-Stepping and Thorup's algorithm on 40 processors. "CH" is the time taken to construct the CH.

Family	Thorup A	Thorup B
RMAT-UWD-2^{26}-2^{26}	28.43s	15.86s
RMAT-PWD-2^{25}-2^{25}	14.92s	8.16s
RMAT-UWD-2^{25}-2^{2}	9.87s	7.57s
Rand-UWD-2^{25}-2^{25}	13.29s	7.53s
Rand-PWD-2^{25}-2^{25}	13.31s	7.54s
Rand-UWD-2^{24}-2^{2}	4.33s	5.67s

Table 6. Comparison of naive strategy (Thorup A) to our strategy (Thorup B) for building *toVisit* set on 40 processors.

remainder of the threads are occupied visiting other components. This situation does not arise for more than two processors on the inputs we tested.

Madduri et al. [11] present findings for shortest path computations using Delta-Stepping on directed graphs. We have used this graph kernel to conduct Delta-Stepping tests for undirected graphs so that we can directly compare Delta-Stepping and Thorup. The results are summarized in Table 5. Delta-Stepping performs better in all of the single source runs presented. Yet, in Section 5.5, we show that Thorup's algorithm can processor simultaneous queries more quickly than Delta-Stepping.

5.4. Selective parallelization

In Section 3.3, we showed our strategy for building the *toVisit* set. This task is executed repeatedly for each component in the hierarchy. As a result, the small amount of time that is saved by selectively parallelizing this loop translates to an impressive performance gain. As seen in Table 6, the improvement is nearly two-fold for most graph instances.

In the current programming environment, the programmer can only control if a loop executes on all processors, on a single processor, or in serial. We conjecture that better control of the number of processors for a loop would lead to a further speedup in our implementation.

5.5. Simultaneous SSSP runs

Figure 5 presents results of simultaneous Thorup SSSP computations that share a single Component Hierarchy. We ran simultaneous queries on random graphs with a uniform weight distribution. When computing for a modest number of sources simultaneously, our Thorup implementation outpaces the baseline delta-stepping computation.

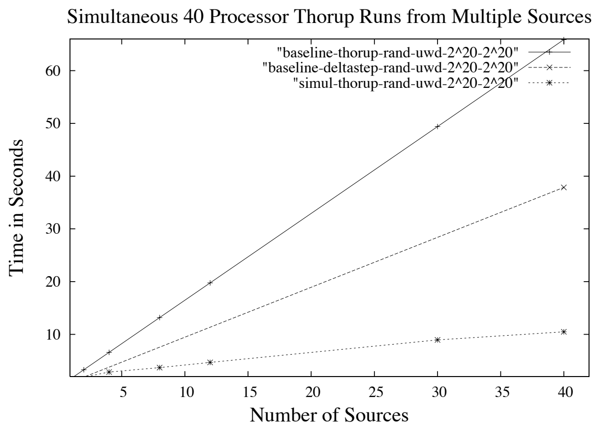
We note that Delta-Stepping stops scaling with more than four processors for small graphs. Thus, Delta-Stepping could run ten simultaneous four processor runs to process the graph in parallel. Preliminary tests suggest that this approach might beat Thorup, but this is yet to be proven.

6. Conclusion

We have presented a multithreaded implementation of Thorup's algorithm for undirected graphs. Thorup's algorithm is naturally suited for multithreaded machines since many computations can share a data structure within the same process. Our implementation uses functionality from the MTGL [2] and scales well from 2 to 40 processors on the MTA-2. Although our implementation does not beat the existing Delta-Stepping [11] implementation for a single source, it does beat Delta-Stepping for simultaneous runs on 40 processors. These runs must be computed in sequence with Delta-Stepping.

During our implementation, we created strategies for traversing the Component Hierarchy, an irregular tree structure. These strategies include selectively parallelizing a loop with an irregular number of iterations. Performing this optimization translated to a large speedup in practice. Yet, the granularity of this optimization was severely limited by the programming constructs of the MTA-2. We were only able to specify if the code operated on a single processor or on all processors. In the future, we would like to see the compiler or the runtime system automatically choose the number of processors for loops like these. In the new Cray XMT [4], we foresee this will be an important optimization since the number of processors is potentially much larger.

We would like to expand our implementation of Thorup's algorithm to compute shortest paths on road networks. We hope to overcome the limitation of our current implementation, which exhibits trapping behavior that severely limits performance on road networks. After this, the Component Hierarchy approach might potential contributed speedup of the precomputations associated with cutting-edge road network shortest path computations based-upon transit nodes [1, 9]. Massively multithreaded architectures should be contributing to this research, and this is the most promising avenue we see for that.



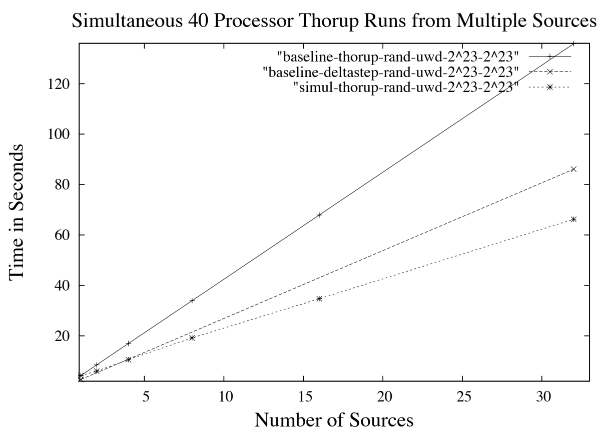


Figure 5. Simultaneous Thorup SSSP runs from multiple sources using a shared CH.

7. Acknowledgments

This work was supported in part by NSF Grants CAREER CCF-0611589, NSF DBI-0420513, ITR EF/BIO 03-31654, and DARPA Contract NBCH30390004. 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. We acknowledge the algorithmic inputs from Bruce Hendrickson of Sandia National Laboratories.

References

[1] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, New Orleans, LA, January 2007.

[2] J. 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.

[3] 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.

[4] Cray, Inc. The XMT platform. `http://www.cray.com/products/xmt/`, 2006.

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

[6] E. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1(4):269–271, 1959.

[7] A. V. Goldberg. A simple shortest path algorithm with linear average time. *Lecture Notes in Computer Science*, 2161, 2001.

[8] T. Hagerup. Improved shortest paths on the word ram. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 61–72, London, UK, 2000. Springer-Verlag.

[9] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, New Orleans, LA, January 2007.

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

[11] K. Madduri, D. Bader, J. Berry, and J. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, New Orleans, LA, January 2007.

[12] U. Meyer and P. Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *European Symposium on Algorithms*, pages 393–404, 1998.

[13] S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*. SIAM, 6–8 2002.

[14] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.

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