# SANDIA REPORT

SAND2009-6278
Unlimited Release
Printed September 2009

# LDRD Final Report: Massive Multithreading Applied to National Infrastructure and Informatics

Jonathan W. Berry

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-6278
Unlimited Release
Printed September 2009

# LDRD Final Report: Massive Multithreading Applied to National Infrastructure and Informatics

Jonathan W. Berry (PI)
Scalable Algorithms Department (01416)
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1318
jberry@sandia.gov

Brian W. Barrett (01423)
Bruce Hendrickson (01410)
Randall A. LaViolette (05635)
Vitus J. Leung (01415)
Gregory E. Mackey (6321)
Brad Mancke (formerly 9511)
Richard C. Murphy (1422)
Cynthia A. Phillips (1412)
Ali Pinar (8962)
Kyle Wheeler (Notre Dame/01422)

**Abstract**

Large relational datasets such as national-scale social networks and power grids present different computational challenges than do physical simulations. Sandia's distributed-memory supercomputers are well suited for solving problems concerning the latter, but not the former. The reason is that problems such as pattern recognition and knowledge discovery on

large networks are dominated by memory latency and not by computation. Furthermore, most memory requests in these applications are very small, and when the datasets are large, most requests miss the cache. The result is extremely low utilization.

We are unlikely to be able to grow out of this problem with conventional architectures. As the power density of microprocessors has approached that of a nuclear reactor in the past two years, we have seen a leveling of Moores Law. Building larger and larger microprocessor-based supercomputers is not a solution for informatics and network infrastructure problems since the additional processors are utilized to only a tiny fraction of their capacity.

An alternative solution is to use the paradigm of massive multithreading with a large shared memory. There is only one instance of this paradigm today: the Cray MTA-2. The proposal team has unique experience with and access to this machine. The XMT, which is now being delivered, is a Red Storm machine with up to 8192 multithreaded "Threadstorm" processors and 128 TB of shared memory. For many years, the XMT will be the only way to address very large graph problems efficiently, and future generations of supercomputers will include multithreaded processors. Roughly 10 MTA processor can process a simple short paths problem in the time taken by the Gordon Bell Prize-nominated distributed memory code on 32,000 processors of Blue Gene/Light.

We have developed algorithms and open-source software for the XMT, and have modified that software to run some of these algorithms on other multithreaded platforms such as the Sun Niagara and Opteron multi-core chips.

# Acknowledgment

We wish to acknowledge our frequent collaborators Kamesh Madduri (now of Lawrence Berkeley Laboratories), and David Bader (Georgia Tech.), along with Joseph Crobak, a Sandia summer student intern whose work was converted into a workshop paper using the LDRD.

We also wish to thank Alyson Wilson and Dan Nordman of the Iowa State University statistics department for many helpful conversations. The triangles algorithm currently used in the MTGL is due to Jonathan Cohen of DoD. We wish to thank John Siirola of 01433 for help with setting up the MTGL trac web site, and in general, for helpful advice with software quality assurance tools and processes.

We thank Aaron Clauset (Santa Fe Institute) and Mark Newman (U. Michigan) for helpful conversations regarding the paper "Community Detection via Facility Location."

We also thank Santo Fortunato (ISI), Joseph McCloskey (DoD), Cris Moore (UNM), Tamara Kolda (Sandia), Dan Nordman (Iowa St.), and Alyson Wilson (Iowa St.) for helpful discussions and comments regarding the paper "Tolerating the Community Detection Resolution Limit via Edge Weights."

We acknowledge the great influence of Simon Kahan and Petr Konecny (formerly of Cray, Inc.), who introduced us to the Cray XMT, and to Kristyn Maschhoff of Cray, Inc., who has helped us with many technical issues associated with programming on the Cray XMT.

We thank Rolf Riesen for contributing the LaTeXtemplate for this report.

# Contents

# List of Figures

# List of Tables

# Preface

This LDRD followed from a WFO project in which Bruce Hendrickson, Jonathan Berry, Keith Underwood, Curtis Janssen, and others studied and programmed the Cray MTA-2 massively multithreaded supercomputer, and simulated its successor, the Cray XMT. We became convinced that Sandia should develop software and algorithms for these machines and others that may adopt the massive multithreading paradigm.

These Cray multithreaded machines have processors that are quite different from traditional processors. The latter are designed under the assumption that most codes have good locality, and much chip real estate is dedicated to processor cache. The Cray multithreaded processors ("threadstorms") have *no* processor cache because their strategy is to tolerate latency rather than mitigating it. The chip space is instead devoted to storing 128 hardware thread contexts (sets of registers, program counter, etc.). There is a context switch every clock cycle, and each thread can have at most one instruction in the 21-stage instruction pipeline at any time. The clock rate is quite slow (100's of MHz), so serial computation of any sort cannot be tolerated.

The network features NIC's that support shared memory, and data structures are automatically hashed throughout the entire shared memory of the machine. In this way, spatially locality is explitly and intentionally defeated. The prerequisite for efficient computation is a large degree of concurrency in the application. In order to show an advantage over traditional supercomputers, the latter must typically be characterized by asynchronous computations that make nearly random references throughout memory. Graph and pointer chasing computations sometimes meet these requirements.

Recently, we have envisioned large potential advantages of the large shared memory of these machines, even without limiting computations to pointer chasing. Specifically, the Cray XMT may be an excellent accelerator for MapReduce computations. Sandia is now exploring this possibility in another project.

# Summary

The most significant technical accomplishments of this LDRD are the development of new algorithms for community detection in networks (see Chapters 1 and 2) and of the open-source MultiThreaded Graph Library (see Chapters 7 and 8.

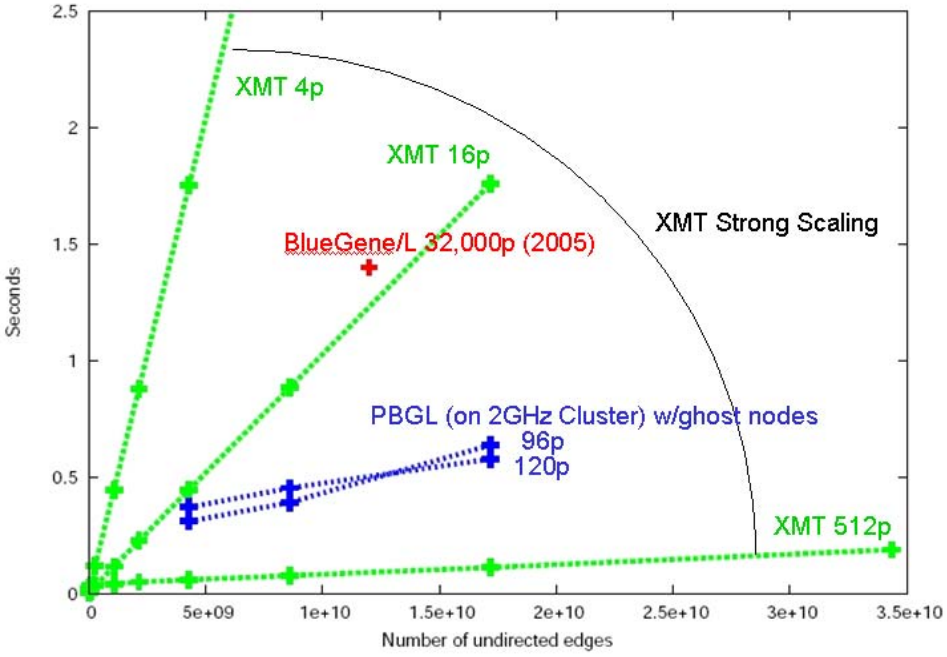



**Figure 1.** S-T connectivity comparison

This final report unites several of our publications on these main themes and also our thoughts on some others. Before this exposition, however, we will describe the current status of a prediction made by Bruce Hendrickson and Jon Berry in 2005 that helped motivate this LDRD.

Borrowing text from Chapter 4 below, s-t connectivity is computed considering the following simple algorithm: given two vertices $s$ and $t$: find the neighbors of $s$, see if any of them is $t$. If not, then find the neighbors of $t$ and see if any of them is one of the vertices in $s$'s expanding frontier. Repeat this process by expanding the smaller of the frontiers of $s$ and $t$ until the two frontiers intersect (see Figure 4.1 on Page 56). Yoo, et al. [80] used this algorithm on Erdös-Rényi graphs of 3.2 billion nodes, and reported results on 32,768 processors of the world's largest distributed memory machine, BlueGene/L. See Chapter 4 for more detail. On an Erdös-Rényi graph, it is straightforward to analyze the expected

number of vertices to be visited to find the shortest path between s and t. For the algorithm sketched above applied to the instances studied by Yoo, et al., the number of vertices visited should be about 177 times larger for the graph on 32,768 nodes than for the graph on 1 node. Extrapolation out to the size of the large instance treated by Yoo, et al., we expect roughly 200,000 vertices to be visited. Given the average time per visit per MTA-2 processor, we conjectured in 2005 that roughly 5-10 MTA-2 processors could perform S-T connectivity in roughly the time taken by 32,000 BlueGene/L processors.

At the time of that conjecture, no massively multithreaded computer was large enough to store such an instance. However, in 2009 Cray completed a 512 processor XMT with a terabyte of RAM. We ran our S-T connectivity code (developed by Kamesh Madduri while visiting Sandia as a summer graduate assistant) on this machine and produced the results in Figure . The red symbol marks the BlueGene/L computation of Yoo, et al., and the rays originating from the origin mark our XMT results. Note that between 4 and 16 XMT processors are required to compute S-T connectivity in roughly the time of Yoo, et al. Considering the many disadvantages that the Cray XMT has relative to the MTA-2 (relatively slower network and slower memory, in particular), these results confirm the 2005 conjecture. Note that we ran larger instances of roughly 32 billion edges with 512 XMT processors and obtained running times of approximately a quarter of a second. The "PBGL" (Parallel Boost Graph Library) results improve upon those of Yoo, et al. by using asynchronous computations and compressed data structures, but don't show good strong scaling (improving run time on a given instance when the number of processors is increased).

We must be cautious with these results, because they were derived from computations on Erdös-Rényi graphs. These are not representative of most real-world graph datasets. The latter are often characterized by power-law degree distributions, meaning intuitively that there are a few vertices of very high degree, but many of very low degree. The study should be carried forward on such datasets. Unfortunately, the code of Yoo, et al. was never tested on these. However, the PBGL code can be run on such instances.

# Nomenclature

**BGL** the Boost Graph Library, a generic C++ framework on which the MultiThreaded Graph Library is based.

**Closeness Centrality** $C_c(v) = \frac{1}{\sum_{t \in V \setminus v} d_G(v,t)}$, where $d_G(v,t)$ is the distance between vertices $v$ and $t$ in $G$.

**connected component** a set of vertices $C$ in a graph such that every pair of vertices in $C$ is connected by a path of vertices in $C$. Algorithms for finding connected components are challenging to scale on HPC.

**Cray MTA-2** a shared-memory, massively multithreaded supercomputer good for informatics problems, but not scalable to large processor counts

**Cray XMT** the follow-on to the Cray MTA-2: good for informatics problems and scalable to large processor counts (development name: Eldorado)

**Degree** the number of edges incident on a vertex of a simple graph (one with no self-loops or multiple edges)

**Erdös-Rényi graph** a random graph generated by flipping a weighted coin for each possible pair of vertices. This model is ubiquitious in graph theory, as it is often amenable to analysis. However, it is a notoriously poor representative for many real graph datasets.

**Graph** $G = (V, E)$ - a set of vertices and a set of edges linking those vertices

**HPC** "high-performance computing," which typically denotes large-memory supercomputers with advanced networks that tightly couple processing elements

**MTGL** the MultiThreaded Graph Library. Sandia is developing this open-source library for algorithms to run on the Cray XMT and multi-core workstations.

**Network Simplex methods** a generic way to solve many flow and path problems by using paths in graphs rather than solving linear programs.

**PBGL** the Parallel Boost Graph Library, which enables graph algorithms to run on distributed memory clusters.

**Qthreads** a thread virtualization software framework. Some prototype MTGL algorithms leverage Qthreads to run on multi-core workstations.

**R-MAT** a synthetic class of graphs good for approximating the properties of many real datasets

**Subgraph Isomorphism** a computationally hard problem: finding instances of a small graph in a large graph

**Trac** a web-based software project management and bug/issue tracking system. The MTGL is now being managed at Trac site `http://software.sandia.gov/mtgl`.

# Chapter 1

# Tolerating the Community Detection Resolution Limit via Edge Weighting

The text in this chapter is the body of the paper by the same name currently under consideration for publication in the journal *Physical Review E*. This paper is also on www.arxiv.org [14].

Communities of vertices within a giant network such as the World-Wide Web are likely to be vastly smaller than the network itself. However, Fortunato and Barthélemy have proved that modularity maximization algorithms for community detection may fail to resolve communities with fewer than $\sqrt{L/2}$ edges, where $L$ is the number of edges in the entire network. This resolution limit leads modularity maximization algorithms to have notoriously poor accuracy on many real networks.

Fortunato and Barthélemy's argument can be extended to networks with weighted edges as well, and we derive this corollary argument. We conclude that weighted modularity algorithms may fail to resolve communities with fewer than $\sqrt{W\epsilon/2}$ total edge weight, where $W$ is the total edge weight in the network and $\epsilon$ is the maximum weight of an inter-community edge. If $\epsilon$ is small, then small communities can be resolved.

Given a weighted or unweighted network, we describe how to derive new edge weights in order to achieve a low $\epsilon$, we modify the "CNM" community detection algorithm to maximize weighted modularity, and show that the resulting algorithm has greatly improved accuracy. In experiments with an emerging community standard benchmark, we find that our simple CNM variant is competitive with the most accurate community detection methods yet proposed.

## Introduction

Maximizing the modularity of a network, as defined by Girvan and Newman [66], is perhaps the most popular and cited paradigm for detecting communities in networks. There are many algorithms for approximately maximizing modularity and its variants, such as [21, 22, 33]. Community assignments of good modularity feature groups of nodes that are more

tightly connected than would be expected. We give the formal definition of modularity below. Recent literature, however, has begun to focus on paradigms other than modularity maximization. This is in part due to Clauset, Newman, and Moore [20], who now advocate a more general notion of "community" than that associated with modularity. The shift away from modularity maximization is also due to Fortunato and Barthèlemy [36], who prove that any community assignment produced by a modularity maximization algorithm will have predictable deficiencies in certain realistic situations. Specifically, they argue that any solution of maximum modularity will suffer from a *resolution limit* that prevents small communities from being detected in large networks. Furthermore, work by Dunbar [30] indicates that true human communities are generally smaller that 150 nodes. This size is far below the resolution limit inherent in many large networks, such as various social networking sites on the World Wide Web.

We agree with Clauset, Newman, and Moore's [20] idea that it is useful to consider more general definitions for "community"; however, we maintain that it is still important to detect traditional, tightly-connected communities of nodes. In this paper, we revisit the negative result of Fortunato and Barthèlemy and analyze it in a different light. We show that positive results are possible without contradicting the resolution limit. The key is to apply carefully computed weights to the edges of the network.

With one exception, previous methods for tolerating this resolution limit require searching over an input parameter. For example, Li, et al. [55] address the resolution limit problem by defining a modularity alternative called *modularity density*. Given a fixed number of communities $k$, solving a k-means problem will maximize modularity density. Li, et al. generalize modularity density so that tuning a parameter $\lambda$ favors either small communities (large $\lambda$) or large communities (small $\lambda$) [55]. Arenas, Fernandez, and Gomez also address the problem of resolution limits [5]. They provide the user with a parameter $r$ that modifies the natural community sizes for modularity maximization algorithms. By tuning $r$, they influence the natural resolution limit. At certain values of $r$, small communities will be natural, and at other values of $r$, large communities will be natural. Our methods apply without specifying any target scale for natural communities, and resolve small and large communities simultaneously.

One solution that resolves communities at multiple scales with no tuning parameter is the HQcut algorithm of Ruan and Zhang [72]. This algorithm alternates between spectral methods and efficient local improvement. It uses a statistical test to determine whether to split each community. Ruan and Zhang argue that a subnetwork with modularity significantly greater than that expected of a random network with the same sequence of vertex degrees is likely to have sub-communities, and therefore should be split. As Fortunato points out in his recent survey [37], though, this stopping criterion is an ad-hoc construction.

Nevertheless, Ruan and Zhang present compelling evidence that the accuracy of HQcut often exceeds that of competitors such as Newman's spectral method followed by Kernighan-Lin local improvement [64] and the simulated annealing method of Guimerà and Amaral [44]. The HQcut solution is not simply the solution of global maximum modularity, so it is not bound by the resolution limit. We obtained the authors' Matlab code for HQcut and we

present comparisons with our approach below.

# Resolution Limits

Fortunato and Barthélemy [36] define a *module* to be a set of vertices with positive modularity:

$$\frac{l_s}{L} - \left(\frac{d_s}{2L}\right)^2 > 0, \tag{1.1}$$

where $l_s$ is the number of undirected edges (links) within the set, $d_s$ is the sum of the degrees of the vertices within the set, and $L$ is the number of undirected links in the entire network. These modules contain more edges than we would expect from a set of vertices with the same degrees, were edges to be assigned randomly (respecting the invariant vertex degrees). iLet us define such modules to be *natural communities* with respect to modularity maximization. We say that a natural community is *minimal* if it contains no other natural communities. We wish to resolve the minimal natural communities.

In order to ensure that such modules are resolved in a global community assigment with maximum modularity, Fortunato and Barthélemy [36] argue that the following must hold:

$$l_s \geq \sqrt{\frac{L}{2}}. \tag{1.2}$$

They back up this mathematical argument with empirical evidence. Even in a pathologically easy situation, in which the modules are cliques, and only one edge links any module to a neighboring module, the individual modules will not be resolved in any solution of maximum modularity. Instead, several cliques will be merged into one module. Experiments show that the numbers of links in the resulting modules closely track the $\sqrt{L/2}$ prediction.

Work by Dunbar [30] indicates that true human communities are generally limited to roughly 150 members, and this is corroborated by the recent work of Leskovec, Lang, Dasgupta, and Mahoney [53]. Such communities will have dramatically fewer than $\sqrt{L/2}$ edges in practice. Based on this argument, it would seem that there is little hope for the solutions of modularity maximizing algorithms to be applied in real situations in which $L \gg l_s$. Indeed, partially due to the resolution limit result, the general direction of research in community detection seems to have shifted away from modularity maximization in favor of machine learning techniques.

In this paper, we revisit the resolution limit in the context of edge weighting and derive more positive results.

## Resolution with edge weights

The definition of a module in equation [1.1] can easily be generalized when edges have weights. Let $w_s$ be the sum of the weights of all undirected edges connecting vertices within Set $s$. Let $d^w(v)$, the weighted degree of vertex $v$, be the sum of the weights of all edges incident on $v$. We define $d_s^w = \sum_{v \in s} d^w(v)$ to be the sum of weighted degrees of the vertices in Set $s$. Then Set $s$ is a module if and only if:

$$\frac{w_s}{W} - \left(\frac{d_s^w}{2W}\right)^2 > 0. \tag{1.3}$$

Following [36] step-by-step, when considering a module, we use $w_s^{\text{out}}$ to denote the sum of the weights of the edges leaving Set $s$, and also note that $w_s^{\text{out}} = \alpha_s w_s$, where $\alpha_s$ is a convenience that enables us to rewrite the definition of a module in a useful way. We now have $d_s^w = 2w_s + w_s^{\text{out}} = (\alpha_s + 2)w_s$, and a new, equivalent, definition of a module:

$$\frac{w_s}{W} - \left(\frac{(\alpha_s + 2)w_s}{2W}\right)^2 > 0. \tag{1.4}$$

Manipulating the inequality, we obtain the relationship:

$$w_s < \frac{4W}{(\alpha_s + 2)^2}. \tag{1.5}$$

Thus, sets representing communities must not have too much weight in order to be modules.

## The Maximum Weighted Modularity

Fortunato and Barthélemy describe the most modular network possible. This yields both computed figures that can be corroborated by experimental evidence, and intuition that the resolution limit in community detection has a natural scale that is related to the total number of links in the network. We will use the same strategy for the weighted case.

First, we imagine a network in which every module is a clique. For a given number of nodes and number of cliques, the modularity will be maximized if each clique has the same size. Weighting does not change the argument of [36] that the modularity approaches 1.0 as the number of cliques goes to infinity. Now, following [36], we consider a slight relaxation of the simple case above: the most modular connected network. This will be our set of $m$

cliques with at least $m-1$ edges to connect them. Without loss of generality, we consider the case of $m$ connecting edges — a ring of cliques, as studied by [28].

Departing for a moment from [36], we now consider an edge weighting for the network. With edge weights in the range $[0,1]$, the optimal weighting would assign 1 to each intra-clique edge and 0 to each connecting edge. The weighted modularity of this weighted network would be equivalent to the unweighted modularity of the $m$ independent cliques described above, and would tend to 1.

Relaxing this idealized condition, now assume that we have a weighting function that assigns $\epsilon$ to each connecting edge, and 1.0 to each intra-clique edge. We now analyze the resulting weighted modularity.

The total edge weight contained within the cliques is

$$\sum_{s=1}^{m} w_s = W - \epsilon m. \tag{1.6}$$

Each clique is a module by (1.3) provided that $\epsilon$ is sufficiently small. Summing the contributions of the modules, we find the weighted modularity of the network when broken into these cliques is:

$$Q = \sum_s \left[ \frac{w_s}{W} - \left( \frac{2w_s + 2\epsilon}{2W} \right)^2 \right]. \tag{1.7}$$

Since all modules contain the same weight, for all $s$:

$$w_s = \frac{W - \epsilon m}{m} = \frac{W}{m} - \epsilon \tag{1.8}$$

The maximum modularity of any solution with $m$ communities is:

$$Q_M(m, W) = m \left[ \frac{W/m - \epsilon}{W} - \left( \frac{W/m}{W} \right)^2 \right] = 1 - \frac{\epsilon m}{W} - \frac{1}{m} \tag{1.9}$$

To quantify this maximum, we take the derivative with respect to $m$:

$$\frac{dQ_M}{dm}(m, W) = \frac{-\epsilon}{W} + \frac{1}{m^2} \tag{1.10}$$

Setting this to zero, we find the number of communities in the optimal solution:

$$m^* = \sqrt{\frac{W}{\epsilon}}. \tag{1.11}$$

Substituting into (1.9), we find the maximum possible weighted modularity:

$$Q_M(W) = 1 - \frac{2}{\sqrt{W/\epsilon}}. \tag{1.12}$$

The unweighted versions of equations [1.11] and [1.9] from [36] are, respectively, $m^* = \sqrt{L}$, and $Q_M(L) = 1 - \frac{2}{\sqrt{L}}$. In this unweighted case, the natural scale is clearly related to $L$. We don't expect to be able to find many more than $\sqrt{L}$ modules in any solution of optimal unweighted modularity.

Our weighted case is similar, but the introduction of $\epsilon$ leads to some intriguing possibilities. If $\epsilon$ can be made small enough, for example, then there is no longer any limit to the number of modules we might expect in any solution of maximum weighted modularity.

# The Weighted Resolution Limit

In [36], Fortunato and Barthélemy prove that any module in which $l < \sqrt{L/2}$ may not be resolved by algorithms that maximize modularity. Their argument characterizes the condition under which two true modules linked to each other by any positive number of edges will contribute more to the global modularity as one unit rather than as two separate units. This result is corroborated by experiment. In a large real-world dataset such as the WWW, modules with $l \ll L$ will almost certainly exist.

Following the arguments of [36] directly, while considering edge weights, we now argue that any module $s$ in which

$$w_s < \sqrt{\frac{W\epsilon}{2}} - \epsilon \tag{1.13}$$

may not be resolved. Consider a scenario in which two small modules are either merged or not. Suppose that the first module has intra-module edges of net weight $w_1$, and the second has intra-module edges of net weight $w_2$. iWe assume that inter-module edges between these two modules have weight $\epsilon$, explicitly write the expressions for weighted modularity in both

cases, and find their difference. The weighted modularity of the solution in which these two modules are resolved exceeds that in which they are merged, provided:

$$w < \frac{2W\epsilon/w}{(\frac{\epsilon}{w} + \frac{\epsilon}{w} + 2)(\frac{\epsilon}{w} + \frac{\epsilon}{w} + 2)} \tag{1.14}$$

where $w$ could be either $w_1$ or $w_2$. Manipulation of this expression gives (1.13).

Two challenges remain: finding a method to set edge weights that achieve a small $\epsilon$, and adapting modularity maximization algorithms to use weights. The second challenge is partially addressed by [65] and [33], but we take a different approach.

# Edge Weighting

There are myriad ways to identify local structure with local computations. Several approaches to community detection, such as [22, 62, 50], are based upon this idea. We use local computations to derive new edge weights. Our approach is to reward an edge for each short cycle connecting its endpoints. These suggest strong interconnections.

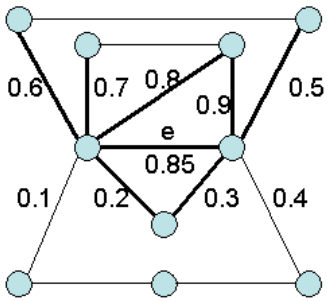



**Figure 1.1.** Edge neighborhood weighting

For a vertex $v$, let $E(v)$ be the set of all undirected edges incident on $v$. We also define the following sets to express triangle and rectangle relationships between pairs of edges.

$$T_e = \{e' : \text{there exists a 3-cycle containing both } e \text{ and } e'\}$$

$$R_e = \{e' : \text{there exists a 4-cycle containing both } e \text{ and } e'\}$$

Note that $e$ can be a member of $T_e$ and $R_e$.

The total weight of edges incident on the endpoints of edge $e = (u, v)$ is

$$W_e = \sum_{e' \in E(u) \cup E(v)} w_{e'}.$$

We consider incident edges that reside on paths of at most three edges connecting the endpoints of $e$ to be "good" with respect to $e$.

$$G_e = \sum_{e' \in E(u) \cup E(v) \cap (T_e \cup R_e)} w_{e'}.$$

Such edges add credence to the proposition that $e$ is an intra-community edge. We define *neighborhood coherence* of $e$ as follows:

$$C(e) = \frac{G_e}{W_e}$$

.

For example, in Figure 1.1, the coherence is computed by summing the weights of the thickened edges and dividing by the total weight of edges incident on the endpoints of $e$: $C(e) = \frac{4.85}{5.35}$. Alternate definitions are possible, of course, but this weighting is intuitive and performs well in practice.

Arenas, Fernandes, and Gomez, by contrast, add self-loops to vertices according to their $r$ parameter, thereby "weighting" the nodes, and also adding more intra-community edges to each module. Thus, they pack more edges into each module in order to satisfy Inequality [1.2].

We have considered generalizing $C(e)$ to include cycles of length 5 and greater, but this would be a considerable computational expense, and we expect diminishing marginal benefit.

Now we give a simple iterative algorithm for computing edge weights:

1. Set $w_e = 1.0$ for each edge $e$ in the network (or accept $w_e$ as input if the edges are already weighted).
2. Compute $C(e)$ for each $e$, set $w_e = C(e)$.
3. If any $w_e$'s changed within some tolerance, go to Step 2

This process will tend to siphon weight out of the inter-module edges (those with smaller $C(e)$), thus lowering $\epsilon$. We find in practice that it terminates in a small number of iterations. Computing $C(e)$ reduces to finding the triangles and 4-cycles in the graph. This can be done naively in $O(mn \log n)$ time on scale-free graphs. We use Cohen's data structures [23] that

admit more efficient algorithms in practice. For WWW-scale graphs, it may be necessary for efficiency reasons to ignore edges incident on high-degree vertices. This would isolate these vertices. However, since such vertices often have special roles in real networks, they might require individual attention anyway.

We define Algorithm $W(k)$ to be $k$ iterations through the loop in Steps 2–3.

# Weighted Clauset-Newman-Moore

Any modularity maximization algorithm could be made to leverage edge weights such as those computed in the previous section. Newman replaces individual weighted edges with sets of multiple edges, each with integral weight [65]. We modify the agglomerative algorithm of Clauset, Newman, and Moore (CNM) [21] to handle arbitrary weights directly.

The CNM algorithm efficiently computes the change in modularity ($\Delta Q$) associated with all possible mergers of two existing communities. At the beginning, each vertex is in its own singleton community. Unweighted modularity is defined as follows:

$$\begin{aligned} Q &= \frac{1}{2L} \sum_{vw} \left[ A_{vw} - \frac{k_v k_w}{2L} \right] \delta(c_v, c_w) \\ &= \sum_s (e_{ss} - a_s^2). \end{aligned}$$

$A_{vw}$ is the adjacency matrix entry for directed edge $(v, w)$, $k_v$ is the degree of vertex $v$, $e_{rs}$ is the fraction of edges that link vertices in community $r$ to vertices in community $s$, and $a_s = \sum_r e_{rs}$ is the sum of the degrees of all vertices in community $s$ divided by the total degree. The function $\delta(c_v, c_w)$ equals 1 if $v$ and $w$ are in the same community, and 0 otherwise.

Since vertices $i$ and $j$ initially reside in their own singleton communities, $e_{ij}$ is initially simply $\frac{A_{ij}}{2L}$. The first step in CNM is to initialize $\Delta Q$ for all possible mergers:

$$\Delta Q = \begin{cases} 1/(L) - 2k_i k_j/(2L)^2 & \text{if } i,j \text{ are connected} \\ 0 & \text{otherwise.} \end{cases} \tag{1.15}$$

CNM also initializes $a_i = \frac{k_i}{2L}$ for each vertex $i$. Once the initializations are complete, the algorithm repeatedly selects the best merger, then updates the $\Delta Q$ and $a_i$ values, until only one community remains. The solution is the community assigment with the largest value of $Q$ encountered during this process. Clever data structures allow efficient update of the $\Delta Q$ values.

To modify CNM to work on weighted graphs, we need only change the initialization step.

| Algorithm | $\epsilon$ | $m^*$ | $\lvert S\rvert$ | $Q_M$ | $Q$ |
|---|---|---|---|---|---|
| CNM | N.A. | 108 | 108 | 0.980 | 0.980 |
| $\text{wCNM}_1$ | 0.111 | 286 | 263 | 0.9930 | 0.9928 |
| $\text{wCNM}_5$ | $< 0.000001$ | 1000 | 1000 | 0.9999 | 0.9986 |

**Table 1.1.** These results from the ring of 1000 5-cliques illustrate gains made by considering weighting. Predicted ($m^*$) and algorithmically discovered ($|S|$) numbers of communities match well and indicate that careful weighting makes it possible to resolve all 1000 cliques as modules in a solution of maximal weighted modularity. $Q_M$ is defined in (1.12), $m^*$ is defined in (1.11), and $\epsilon$ is the weight of the heaviest edge between two communities.

The update steps are identical. We simply define and compute the weighted degree of each vertex $k_i^w = \sum_j w_{ij}$. The initialization becomes:

$$\Delta Q^w = \begin{cases} w_{ij}/(W) - 2k_i^w k_j^w/(2W)^2 & \text{if } i,j \text{ are connected} \\ 0 \quad \text{otherwise,} \end{cases} \tag{1.16}$$

and $a_i^w = \frac{k_i^w}{2W}$. With these initializations, normal CNM merging greedily maximizes weighted modularity $Q^w$. We refer to this algorithm as wCNM. Note that our definition of $Q^w$ is equivalent to that of [33].

# Results

Given an undirected, weighted or unweighted network, we apply the Algorithm $W(k)$ to set our edge weights, then run wCNM. We use $\text{wCNM}_k$ to denote this two-step process. Note that running $\text{wCNM}_0$ is equivalent to running CNM.

We will consider two different datasets: the ring of cliques example discussed above, and the benchmark of [51], which is a generalization of the 128-node benchmark of Girvan and Newman [40].

## The ring of cliques

Refer to Table 1.1 for the following discussion. Danon, Díaz-Guilera, Duch, and Arenas [28] considered $m$ disconnected cliques as a pathological example of maximum modularity (which approaches 1.0 as the number of cliques increases). Fortunato and Barthèlemy [36] add single connections between cliques to form a ring. Our intuition is that the natural

communities in such a graph are the cliques. However, the resolution limit argument of Fortunato and Barthèlemy indicates that this will not be the solution of maximum modularity if each clique has fewer than $\frac{\sqrt{L}}{2}$ edges. They confirm this via experiment, and we have reproduced their results for an instance with 1000 cliques of size five. Table 1.1 summarizes the performance of CNM and wCNM for this case. The $m^*$ column contains the number of communities expected in a solution of maximum weighted modularity, as defined in 1.11. The first row shows the unweighted case, in which $m^*$ is equivalent to that defined in [36]. CNM achieves this theoretical maximum by finding 108 communities, which is much smaller than the number of cliques.

If we run $\text{wCNM}_1$, which performs one iteration of neighborhood coherence, we obtain the results in Row 2 of Table 1.1. The value of $\epsilon$ we observe is 0.047, leading via (1.11) to an estimate of 286 resolved communities. The $\text{wCNM}_1$ algorithm resolves 263. In a run with five iterations, labeled $\text{wCNM}_5$, we both expect and find 1000 communities, resolving all of the natural communities and simultaneously observing our highest weighted modularity. Iterating further reduces $\epsilon$ without changing the community assignment.

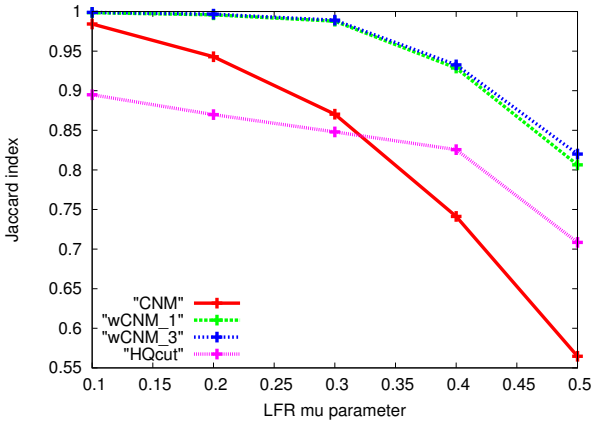
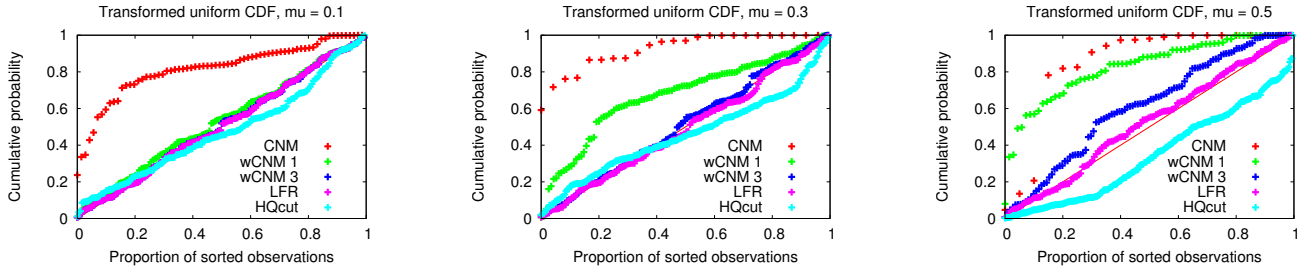


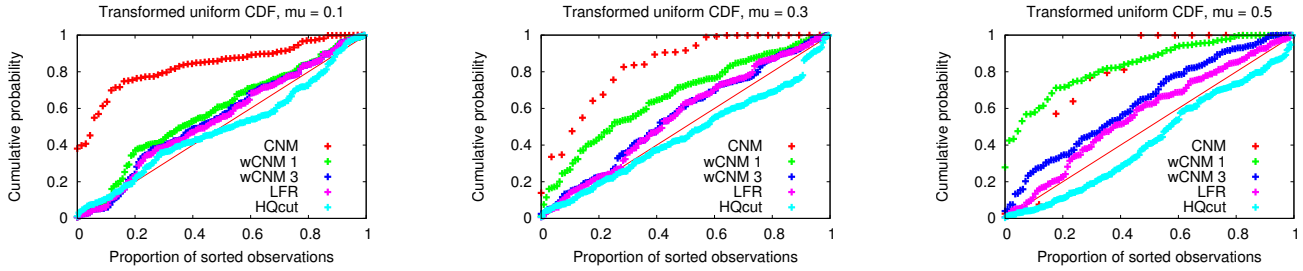

**Figure 1.2.** Mutual information study for the LFR benchmark.

## The LFR Benchmark

Lancichinetti, Fortunato, and Radicchi [51] (LFR) give a generalization of the popular Girvan and Newman benchmark [40] for evaluating community detection algorithms. The latter consists of 128-vertex random graphs, each with 4 natural communities of size 32. The user tunes a parameter to adjust the numbers of intra-community and inter-community edges. Many authors use this benchmark to create plots of "mutual information," or agreement in node classification between algorithm-discovered communities and natural communities. The LFR benchmark is similar in spirit, but considerably more realistic. It allows the user to




(a)




(b)

**Figure 1.3.** The distributions of community sizes (transformed back to uniform) compared to the uniform distribution. The three (a) images depict instances in which the LFR distribution of community sizes passes the Anderson-Darling test. In this case, $\text{wCNM}_3$ passes that test as well for $\mu = 0.1$ and $\mu = 0.3$. The three (b) images show instances in which the LFR distribution does not pass. In these cases, no algorithm solutions pass, but note that wCNM's distribution is relatively close to the LFR ground truth.

specify distributions both for the community sizes and the vertex degrees. Users also specify the average ratio (per vertex) of inter-community adjacencies to total adjacencies, called *mixing parameter* $\mu$. At $\mu = 0.0$, all edges are intra-community.

The LFR benchmark construction process begins by sampling vertex degrees and creating a graph with the selected degree distribution. It then samples community sizes. A vertex of degree $k$ should have about $(1 - \mu)k$ neighbors from the same community. Therefore, it is assigned to a community with at least $(1 - \mu)k + 1$ vertices. LFR assigns vertices to communities via an interated random process enforcing this constraint, then rewires until the average $\mu$ meets the desired value. We have a special interest in the LFR benchmark because it generates graphs with both small and large natural communities.

For several different values of $\mu$, we used the C code from Fortunato's web site (cited

in [51]) to generate 30 instances each of LFR benchmark graphs, each with 5000 vertices and average degree 8. The community sizes were selected from the power-law distribution $f(k) \sim k^{-1.5}$, with $k \in [10, 105]$. The degree distribution was $f(k) \sim k^{-2}$, with $k \in [2, 50]$. We specified an average degree of 8, which is roughly comparable to that of the WWW.

Figure 1.2 contains the mutual information plot for our experiments with LFR. Our metric for comparison is the Jaccard index [45]:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where $A$ is the set of intra-community edges in the LFR ground truth, and $B$ is the set of intra-community edges in an algorithm solution. As predicted by the resolution limit argument, CNM, an unweighted modularity maximization algorithm, is not able to resolve most of the natural communities. However, even with these more realistic data, wCNM achieves greater accuracy than the sophisticated HQcut algorithm. This is notable, considering the reputation for poor accuracy recently associated with agglomerative algorithms such as CNM and its variants [71]. The accuracy of our CNM variant, on the other hand, is competitive.

We observe for these data that iterating the neighborhood coherence weighting provides diminishing marginal returns. However, as we show below, such iteration does add value.

In addition to the mutual information, we wish to compare the distributions of the sizes of communities discovered by CNM and its weighted variants to the original distributions used in LFR generation. There is currently no accepted, canonical method for fitting empirical data to power-law distributions. However, we define a reasonable transformation from community sizes back to uniform variates. We then rigorously compare the distributions of these variates to the uniform distribution using a classical test. We also provide visualizations of these distributions.

LFR uses the following precise sampling process to determine ground truth community sizes:

1. Compute $\frac{1}{k}^{\tau}$, the probability that a community will have size $k$.
2. For all $a \leq k \leq b$, where $a$ and $b$ bound the community sizes, compute the empirical cumulative distribution function for $k$: $p_k = \sum_{k'=a}^{k} (\frac{1}{k'})^{\tau}$.
3. For a uniform random variate $x \in [0, 1]$, find the minimum $k'$ such that $p_{k'} \geq x$.

This process continues until the sum of the community sizes exceeds the number of vertices, and the final community is truncated.

Given an integer community size $k$, we invert this process to generate a value in the interval $[0, 1]$. Specifically, we determine the cumulative probability bounds $[p_k, p_{k+1}]$, then sample uniformly from this range.

Thus, a community assignment $\Gamma$ with $c$ communities and community sizes $\{s_1, \ldots, s_c\}$ is transformed into a set of $[0, 1]$ values $\{x_1, \ldots, x_c\}$. The $x_i$'s are a set of coin tosses that would have generated the observed set of community sizes using LFR sampling. If these $x_i$'s are uniformly distributed, it is an indication that the community sizes are distributed according to the given power law distribution. We sort the $x_i$'s, then apply the classical and discriminating Anderson-Darling test [3] to compare the result with the uniform distribution. The results of these tests are shown in Table 1.2. Note that increasing the $k$ in wCNM$_k$ improves the Anderson-Darling pass rate. The wCNM$_5$ algorithm can pass the Anderson-Darling test roughly 60% of the time as the LFR mixing parameter $\mu$ is increased to 0.4. We find that none of the alternatives can pass this test at $\mu \geq 0.5$.

| | LFR $\mu$ | | | | |
|---|---|---|---|---|---|
| Algorithm | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 |
| CNM | 0/30 | 0/30 | 0/30 | 0/30 | 0/30 |
| wCNM_1 | 2/30 | 0/30 | 0/30 | 0/30 | 0/30 |
| wCNM_3 | 8/30 | 18/30 | 14/30 | 0/30 | 0/30 |
| wCNM_5 | 11/30 | 15/30 | 18/30 | 18/30 | 0/30 |
| LFR ground truth | 9/30 | 17/30 | 18/30 | 17/30 | 14/30 |
| HQcut | 0/29 | 0/30 | 0/30 | 0/30 | 0/29 |

**Table 1.2.** This table shows Anderson-Darling results for experiments with LFR instances (#passes/#instances). The LFR row indicates the proportion of instances in which the ground truth community sizes produced by LFR themselves pass the test.

Figure 1.3 shows empirical cumulative distribution functions (CDF) of the $x_i$'s for algorithm results and LFR ground truth. The three (a) images depict instances in which the LFR distribution of community sizes passes the Anderson-Darling test. In this case, wCNM$_3$ passes that test as well for $\mu = 0.1$ and $\mu = 0.3$. The three (b) images show instances in which the LFR distribution does not pass. In these cases, no algorithm solutions pass. However, note the wCNM is relatively close to the LFR distribution.

We have not included formal running-time comparisons since Ruan and Zhang's publically available HQcut implementation is in Matlab and our implementation of wCNM is in C/C++. For anecdotal purposes, the wCNM runs on our 5000-vertex LFR instances took roughly 10s on a 3Ghz workstation, even with several iterations of weighting. The HQcut instances took 5-10 minutes on the same machine, though there were instances that took many hours. We killed such instances, and that is why we sometimes present fewer than 30 instances of HQcut results per $\mu$.

# Conclusions

We agree with Arenas, Fernandez, and Gomez [5] that it may be premature to dismiss the idea of modularity maximization as a technique for detecting small communities in large networks. Our weighted analogue to Fortunato and Barthèlemy's resolution argument leaves open the possibility for much greater community resolution, given proper weighting. Furthermore, our simple adaptation of the CNM heuristic, when combined with a careful computation of edge weights, is able to resolve communities of varying sizes in test data. Furthermore, we have given empirical evidence that the true ability of such techniques to resolve small, local communities may be greater than that suggested by analysis.

Arguably, the original, unweighted CNM already provides output that could help mitigate the resolution limit. This agglomerative heuristic constructs a dendrogram of hierarchical communities, and therefore does recognize small communities as modules before merging them into larger communities. In this sense, these small communities actually are "resolved" – they are stored in the dendrogram included in the CNM output. A cut through this dendrogram defines the community assignments. The resolution limit leads us to expect that the communities defined by this cut will be unnaturally large. One potential research direction would be to mine this dendrogram for the true communities. In effect, this would mean ignoring the cut provided by CNM, and therefore abandoning the idea of maximizing modularity.

Our wCNM heuristic likewise produces a dendrogram and a cut through that dendrogram defining communities. However, the cut provided by wCNM is much deeper and more uneven. It is analagous to the potential result of mining the CNM dendrogram for natural communities, yet the tie with modularity is maintained since wCNM's solution exhibits a maximal weighted modularity.

The edge weighting we describe is just one of many possible alternatives, and wCNM is just one of many potential weighted modularity algorithms. The main contribution of this paper is to spread awareness that resolution limits may in fact be tolerated while retaining the advantages of modularity maximization and the efficient algorithms for this computation.

# Chapter 2

# Community Detection via Facility Location

The text in this chapter is the body of the paper by the same name currently under revision as a result from feedback to our submission to the journal *Physical Review E*. The idea for this paper came when we made the connection between the concept of "closeness centrality" (see the nomenclature beginning on Page ) and a facility location problem called "p-median." Several LDRD staff had significant experience solving the latter, so we applied that experience to community detection.

Since the submission of this paper, we have significantly revised the formulations. We are in the process of evaluting the effectiveness of the resulting algorithms. The citation for the draft below is [13]

**Abstract**

In this paper we apply theoretical and practical results from facility location theory to the problem of community detection in networks. The result is an algorithm that computes bounds on a minimization variant of local modularity. We also define the concept of an edge support and a new measure of the goodness of community structures with respect to this concept. We present preliminary results and note that our methods are massively parallelizable.

# Introduction

In this paper, we apply results from facility location theory to community detection. Leveraging recent developments in both fields, we compute a weighting of the input graph that represents pertinent information for community detection algorithms. We show how to compute this weighting efficiently using techniques from facility location theory. We can interpret the weights as probabilities and randomly sample over a space of good community assignments. Computing the weights involves solving a linear program [48] with special structure that admits an elegant solution strategy requiring only linear space and near-linear time. Furthermore, this solution strategy is amenable to massive parallelism.

We also give new measures for evaluating the quality of community assignments and show that our algorithms provide a provable lower bound on solution quality with respect to one of these. We demonstrate empirically that another of our measures is complementary to modularity, and that optimizing based on this new measure better resolves small communities in large graphs and better matches common sense community structures in familiar datasets. Thus, we make four contributions in this work: we demonstrate a connection between community detection and facility location; we use that connection to compute lower bounds on solution quality; we show how to compute new measures for the goodness of community structure that contrast with modularity; and we apply massively parallelizable methods to compute these bounds and measures.

# Background

Newman and Girvan's concept of *modularity* [66] is now ubiquitous in the community detection literature. There are several variations on this concept, such as [17, 34, 39, 62, 82], and many heuristics to optimize the original concept and these variations, e.g. [70][21]. In order to compute community structures with good modularity in large network instances, researchers commonly use one of two approaches: greedy heuristics, such as [21] and [78], and metaheuristic approaches, such as the simulated annealing used in [70]. Unpublished work posted in October 2007 applies mathematical programming to the problem of maximizing modularity, resulting in an algorithm to compute upper bounds for that measure [2].

We present an alternative that employs results from the vast facility location literature to community detection. We model a variation of modularity as an *uncapacitated facility location problem* (to be defined below), and employ the simple and powerful *Volume algorithm* [8] to solve the problem. Mulvey and Crowder [63] used similar techniques, applying older subgradient methods, to solve p-median problems that approximately cluster points in $n$-dimensional space.

We first observe that specializing a minimization version of the modularity problem produces an uncapacitated facility location problem. We then discuss its solutions and the interpretation and use of its results.

# Strongly-Local Modularity (SLM)

Girvan and Newman define the *modularity* ($Q$) for a graph $G$ as follows: $Q = \sum_s (e_{ss} - a_s^2)$, where $s$ identifies a community in the domain $\{1 \dots q\}$, $e_{rs}$ is the fraction of $E(G)$ (the edge set of the graph) that connects a node in community $r$ to one in community $s$, and $a_s$ is the fraction of edges that have at least one endpoint in $s$ ($a_s = \sum_r e_{rs}$). Squaring $a_s$ gives the probability that an edge would have both endpoints in community $s$ in a random graph with the same endpoint degree distribution. Modularity is a way to measure the quality of community assignment: it rewards communities that are better connected than would be expected in a random graph reflecting the endpoint degree distribution.

Now consider a simple variation of the modularity concept: $Q^- = \sum_s (1 - (e_{ss} - a_s^2))$. Minimizing $Q^-$ is similar to, though not identical to, maximizing $Q$. Basic algebra shows that a community assignment minimizing $Q^-$ has at most as many communities as one that maximizes $Q$, and this is typically a strict inequality.

It is well-known that community assigments of maximum modularity fail to resolve small communities in large graphs [36]. Reflecting on this work, it would seem that the $Q^-$ measure will compound this problem by resolving even fewer communities. However, we provide a remedy via a further modification described below, and our switching of optimization sense will prove useful.

Muff, Rao, and Caflisch [62] define the *local modularity* to be the same as modularity, except that the denominators in the fractions $e_{rs}$ are the numbers of edges in a cluster's "neighborhood," defined to be itself and all neighboring clusters. We use a metric that also focuses on local structure, but is even more restrictive, requiring no information about the structure of neighboring communities. We define a *strongly local community* to consist of a single representative node and all of its immediate neighbors, i.e., a full community of radius one. Let $Q_s = e_{ss} - a_s^2$. We can compute this measure for any strongly local community without knowing any community assignments other than the vertices in $s$. Ignoring algorithmic details, we need only know the number of triangles in the strongly local community and the degree of each node.

Now we give the key definition that allows us to model the problem using facility location theory. Let

$$\tilde{Q}_s = \begin{cases} Q_s & \text{if } s \text{ is a strongly local community} \\ 0 & \text{otherwise} \end{cases}$$

We define the *Strongly-Local Modularity* (SLM) as follows:

$$\tilde{Q}^- = \sum_s (1 - \tilde{Q}_s).$$

We use SLM in combination with a relaxed notion of community assignment in which community representatives can share common neighbors within their respective communities.

# Modeling SLM as a facility location problem

We transform instances of the community detection problem into instances of the *Uncapacitated Facility Location Problem* (UFLP)[48]. Given a set of potential facility locations $L$, a set of customers $C$, a set of facility opening costs $f_i$, and a set of service costs $c_{ij}$ (the cost to serve customer $j$ using facility $i$), the objective function of UFLP is

$$F(x) = \sum_{i \in L} f_i x_i + \sum_{i \in L, j \in C} c_{ij} y_{ij},$$

where the variables $x_i$ indicate whether or not location $i$ hosts a facility, and the variables $y_{ij}$ indicate whether or not location $i$ serves customer $j$. Solutions to UFLP minimize $F(x)$ subject to the constraints that every customer must be served, and that no customer can be served by a facility that does not exist. UFLP is a well known NP-hard problem [25, 38, 43], yet it has special structure that enables efficient computations of fractional solutions.

We consider all vertices to be potential facility locations, with facility opening costs $f_s = (1 - \tilde{Q}_s)$. Each vertex is a customer that must be served by a facility (and may serve itself if it hosts a facility). The service cost is zero for a node to serve a neighbor in the graph. Nodes cannot serve non-neighbors (cost is effectively infinite). The solution to the UFLP is a minimum-cost facility and service assignment in which every vertex is served.

UFLP is modeled and solved using integer programming (IP), but we need only solve the linear programming relaxation of the IP[48]. This relaxation has special structure that obviates the need for a general linear program solver. We apply Lagrangian relaxation in conjunction with an elegant subgradient method known as the Volume algorithm (VA) [8] in the Lagrangian relaxation framework of [9]. The memory usage of this combined procedure is on the order of the problem input size. The runtime is not precisely understood. VA makes a series of near-linear-time passes over the data. In practice, it is comparable to the $O(n \log^2 n)$ runtime of the most familiar fast modularity heuristic, the *CNM* greedy algorithm [21]. We have observed this experimentally on graphs with up to 100 million edges.

The volume algorithm provides a fractional solution to the UFLP that in turn provides a provable lower bound on $\tilde{Q}^-$ where all communities are strongly local.

Our community-assignment procedure selects a set of facilities to "open." Each open facility represents a leader of a subset of a strongly-local community. That is, every community has at least one node that is adjacent to all other nodes in the community. The set of leaders, therefore, forms a *dominating set*, that is, a set of vertices $D$ such that each vertex in the graph is either in $D$ or adjacent to an element of $D$.

In our community-finding procedure, called *SNL*, we set the facility-opening costs as described above and use VA to compute an optimal fractional placement of facilities. We then open each facility with probability equal to its fractional assignment value. If this does not produce a dominating set, then we repair it to make a dominating set. We then assign all the other vertices to a community. There are a number of ways one can do this. In

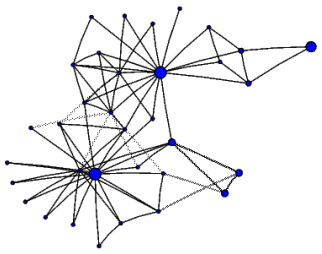


**Figure 2.1.** The support of Zachary's karate club. Solid edges have stronger support than speckled edges and larger vertices are more likely to be leaders. Note the nearly-invisible edges linking portions of the club destined to split.

this paper, we assign each non-selected node to the selected neighbor with highest fractional facility placement.

# The Support

We define the *support* of an edge $(u, v)$ to be a number between 0 and 1 that indicates the level of support/evidence for nodes $u$ and $v$ being in the same community. Given any randomized algorithm $A$ for community detection, such as the metaheuristic approach of [70], we can compute a support with respect to $A$ by sampling. We generate many community assignments using $A$, then compute the fraction of times each edge has endpoints in the same community. In this section, we show how to compute an edge support with respect to *SNL* without any sampling.

Given a fractional solution $x$ to an instance of UFLP, we define the support with respect to *SNL* to be a set of values $z$, where $z_j$ is a probability that in a set of community leaders sampled from $x$, edge $j$ could link two vertices in the same community. Formally,

$$z_{e=(v,w)} = 1 - [(1 - x_v) * (1 - x_w) * \Pi_{u \in N(v) \cap N(w)}(1 - x_u)].$$

An edge $e = (v, w)$ has strong support if it is unlikely that *none* of the vertices capable of serving both $v$ and $w$ will become a server. This includes $v$, $w$, and their mutual neighbors. Figure 2.1 depicts the support of Zachary's karate club dataset [81], an abstraction of a social network that famously split into two. The larger vertices and darker edges have higher $x$ and $z$ values, respectively. Even before community assignments have been specified, the community structure begins to emerge in fractional form. Note that some edges that are destined to become inter-community edges have very low support and are therefore almost invisible.

Given the support of a graph, we define a new measure to evaluate the effectiveness of

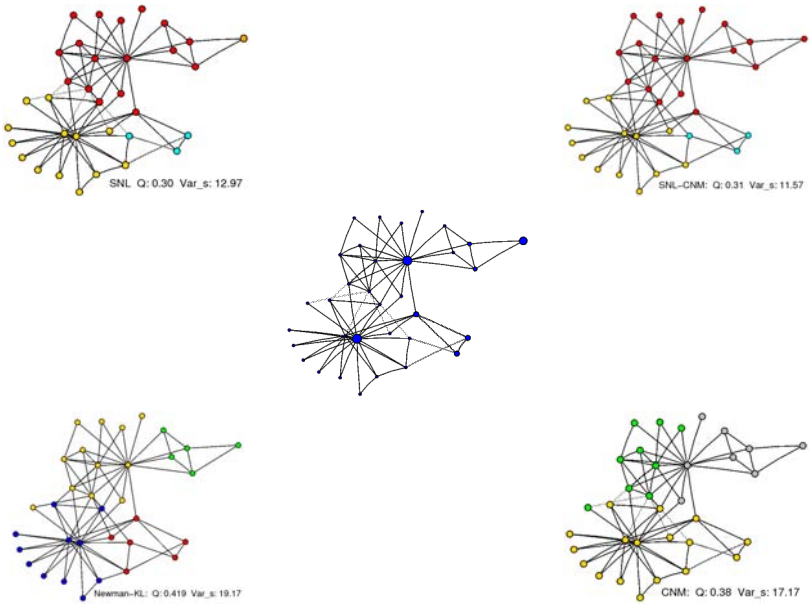



**Figure 2.2.** The support of Zachary's karate club and its relationship to actual solutions of various algorithms. The support variance $\mathrm{Var}_s$ decreases as solutions agree more closely with the support. Note that the community assignments with maximum modularity split edges with strong support within both of the true communities.

community assigments. We define the *support variance* ($\mathrm{Var}_s$) as follows, assuming that $\delta(v, w)$ is an indicator function with value 1 if $v$ and $w$ are in the same community and 0 otherwise.

$$\mathrm{Var}_s = \sum_{(v,w)\in E(G)} (\delta(v, w) - y_{vw})^2.$$

# Preliminary Computational Results

Our focus in this paper is to demonstrate a useful link between facility location theory and community detection. We will present detailed computational studies in future papers. However, we do address several familiar datasets here. Figure 2.2 shows the support of Zachary's karate club The colored images in Figure 2.2 depict the solutions of four algorithms: our facility location-based rounding heuristic (*SNL*); the *CNM* greedy algorithm; a combination of these two (*SNL-CNM*), in which *SNL* is used to compute strongly-local communities, then *CNM* is allowed to merge these; and the eigenvector-based approach of Newman, augmented with a Kernighan-Lin-like postprocessing step (*Newman-KL*) [64]. *Newman-KL* gives one of the best known values for modularity.

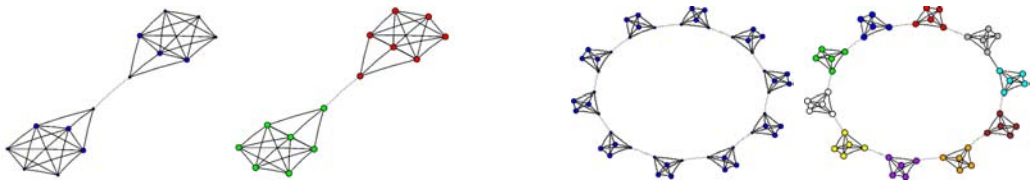


**Figure 2.3.** Maximizing modularity on these instances is known to produce non-intuitive answers. However, each instance has a support that agrees with common sense and leads to intuitive rounded solutions. The left hand instance is from [70], and the right instance is the ring-of-cliques example from[36]. As more cliques are added to the ring, modularity optimization will merge cliques, increasing the support variance. The facility location-based solution is not sensitive to the number of cliques.

In this case, intuition and history favor the facility location-based community assigments with low support variance over those with high modularity. For example, the latter split the topmost community despite reasonably strong support for the edges holding it together.

Figure 2.3 shows two instances that have been demonstrated in recent literature to present inherent problems for modularity algorithms. The modularity of the left hand instance, from [70], tricks greedy algorithms into merging the endpoints of the edge that has the least support in their first step. The right hand instance, from [36], has been used to show that modularity optimization fails to resolve small communities in large graphs. The example shown is a ring of ten 5-cliques, and grouping the 5-cliques individually both minimizes support variance and maximizes modularity. However, as the number of 5 cliques increases, the common sense solution continues to minimize support variance, but is discarded by modularity optimization methods in favor of larger communities.

# Conclusions

We have applied models and algorithms from facility location theory to the problem of community detection, yielding an algorithm to compute a provable lower bound on a minimization variant of local modularity, a support measure that can be computed without sampling, and a randomized rounding heuristic that can be generalized into a class of heuristics. We have also introduced a new measure for evaluating the quality of community structures. The effectiveness of our heuristics for computing quality results on large graphs remains open, but the solution techniques themselves are scalable and based upon simple traversals of the network that are massively parallelizable in a more natural way than the priority queue-based methods previously published. We will explore the scalability of our methods on supercomputers in work to come.

# Chapter 3

# Maximum Flow and Maximum Density Subgraph

We featured the "Network Simplex" algorithmic paradigm in our proposal since it has features that are well-suited to massively multithreaded computing architectures. Originally, we envisioned spending significant effort to develop a generic Network Simplex solver for many different problems. Brad Mancke developed a kernel solver and showed how to parallelize the fundamental operation: finding edge-disjoint paths between pairs of vertices. This is not quite a network simplex kernel, but we believe that it does capture the main computational difficulty. Brad's code has been incorporated into the MTGL, and it supports basic maximum flow problems. Although our work in community detection came to dominate the research direction of the out-years, Greg Mackey did apply this network simplex kernel to another problem: finding the maximum density subgraph. The following writeup by Greg Mackey, not yet a publication, describes this work.

## Introduction

The density of a graph is defined as the number of edges divided by the number of vertices. Consider the undirected graph $G = (V, E)$ with $n$ vertices and $m$ edges. It has a density of $m/n$. The maximum density subgraph problem is to find the subgraph of $G$ that has the maximum density. In [41] Goldberg describes an algorithm for finding the maximum density subgraph by performing $O(\log n)$ minimum cut computations. The algorithm constructs a new graph $N = (V_N, E_N)$ with $n+2$ vertices and $2(n+m)$ edges and runs the minimum cut computation on $N$. Let $M(v, e)$ be the time required to find a minimum cut on a graph with $v$ vertices and $e$ edges. The running time of Goldberg's algorithm is $O\left(M(n, n+m) \log n\right)$.

## Implementation on the XMT

The parallelization available in Goldberg's algorithm mostly exists in implementing a parallel minimum cut algorithm as the binary search around the minimum cut operation must be performed in serial. The Multithreaded Graph Library (MTGL) has a parallel

implementation of the maximum flow algorithm that was coded by Brad Mancke. It is essentially a parallel implementation of the Edmond's Karp algorithm. The parallelization comes from using a parallel breadth first search and looking for multiple augmenting paths simultaneously. The maximum flow code is located in the MTGL library at "mtgl/disjoint_paths_max_flow.hpp."

I modified Brad's maximum flow code to also return a minimum cut, and I use it to find the minimum cut in my implementation of Goldberg's algorithm. There is also a little more parallelism available in the constructing and updating $N$, which I take advantage of. The maximum density subgraph code is located in the MTGL library at "mtgl/maximum_density_subgraph.hpp."

Currently, the maximum density subgraph code is suffering numerically during the minimum cut computation when running on large rmat graphs. However, the maximum flow code does scale well. A run of the maximum density subgraph on an undirected R-MAT graph of size 11 will perform a couple of iterations of the minimum cut computation before running into numerical issues. I timed the first iteration of the minimum cut computation running with different numbers of processors to determine the speedup. The results are shown in Figure 3.1. We have superlinear speedup up to 16 processors, and then it slows down a bit. Notice that the speedup for 32 processors is just under 30. The speedup declines for 64 processors.

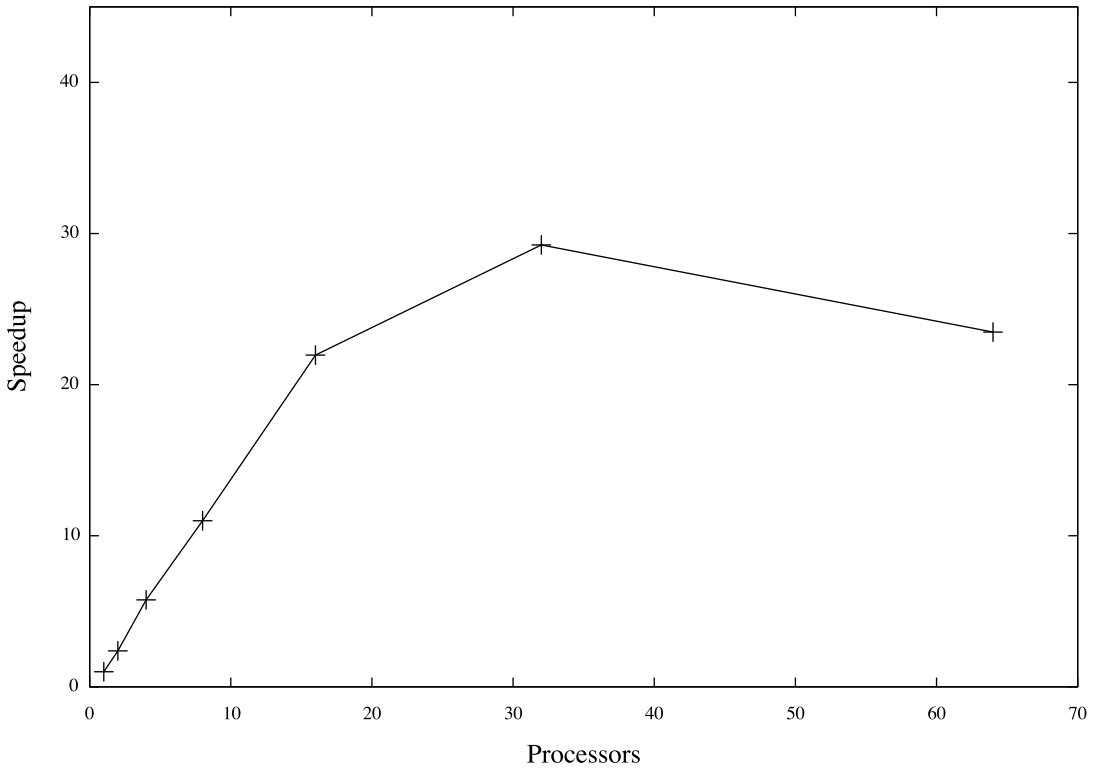



**Figure 3.1.** Speedup for maxflow on R-MAT 11 graph

# Chapter 4

# Graph Analysis with High-Performance Computing

The following is a published article by Bruce Hendrickson and Jonathan Berry that was supported under this LDRD. It appeared in the IEEE's *Computers in Science & Engineering* journal/magazine, which has a circulation of roughly 2000 [47].

## Introduction

Graphs are among the most widely used combinatorial tools in computing. In science and engineering, they are used to describe the structure of sparse matrices, to facilitate load balancing in parallel computations, to study the structure of molecules, and to help with meshing of complex geometries. Graphs are also used to model distribution networks, economies and epidemics, to study social systems, and to describe sets of protein interactions.

Graphs are applicable to such diverse settings because they are an abstract way of describing interactions between entities. A graph consists of a set of entities known as *vertices*, and a set of pairwise relationships between entities known as *edges*. Many refinements and augmentations of this basic model are possible in which vertices and edges have additional properties.

There is a vast literature of graph theory, algorithms, and applications. A typical step in a graph algorithm involves visiting a vertex $v$ and then visiting $v$'s neighbors – the set of vertices connected to $v$ by an edge. For some graphs, e.g. one describing the nonzero structure of a finite difference matrix, the set of neighbors of $v$ can have a regular and predictable structure. This structure can be exploited in the design of data structures to improve cache performance when accessing $v$'s neighbors. However, in many emerging applications like social and economic modeling, the graph has very little exploitable structure. In these settings, the neighbors of $v$ can be widely dispersed in global memory. This leads to data access patterns that make very poor use of memory hierarchies, which can result in idle processors most of the time. As the access patterns are data dependent (i.e., they are a function of the edge structure in the graph) standard prefetching techniques are often ineffective. In addition, graph algorithms typically have very little work to do when visiting

a vertex, so there is little computation for each memory access. For all these reasons, graph computations often achieve a very low percentage of theoretical peak performance on traditional processors. It is worth noting that similar challenges plague many other combinatorial applications.

As the applications of graphs continue to grow in breadth and in size, there is a need for effective parallelizations of graph algorithms. Parallelism presents yet another set of challenges for graph algorithms. Although the literature on theoretical PRAM algorithms is expansive, there are comparatively few success stories of practical parallel graph implementations. In this paper, we will argue that this is a reflection of the mismatch between the demands of graph algorithms and the capabilities of mainstream parallel computer architectures. Graphs in scientific computing often reflect the geometry of a physical object, and so can be partitioned among the processors of a parallel machine in such a way that few edges cross between processors. This is not true of more abstract graphs arising in some emerging applications, and this hinders effective parallelization. In addition, parallelism in graph algorithms tends to be fine-grained and the degree of parallelism varies during the course of an algorithm. This style of parallelism is not well supported by traditional parallel architectures and programming models.

To overcome these challenges, we have recently been developing graph algorithms on a non-traditional, *massively multithreaded* supercomputer. For reasons discussed in §4, this architecture has some attractive attributes for graph algorithms and other latency-dominated computations. In this section, we also review the landscape of parallel computers and programming models through the lens of graph algorithms. In § 4 we talk about graph software. Then in §4 we discuss experiments involving parallelizations of several graph operations on a massively multithreaded machine, the Cray MTA. In §4 we conclude and suggest directions for further research.

# The High Performance Computing Landscape

By far the most popular class of parallel machines are *distributed memory computers*. These machines consist of a set of commodity processors connected by a network. These machines are relatively inexpensive, and they are very effective on many scientific problems.

Distributed memory machines are generally programmed by explicit message passing via MPI. With MPI, the user needs to divide the data among the processors and to determine which processor performs which tasks. Data are exchanged between processors by user-controlled messages. Although high performance is achievable for many applications, the detailed control of data partitioning and communication can be tedious and error prone.

Typically, MPI programs are written in a Bulk-Synchronous style, in which processors alternate between working independently on local data, and participating in collective communication operations. By grouping data exchanges into large, collective operations, the overall latency cost is substantially reduced. However, this comes at the expense of algo-

rithmic flexibility. Data cannot be transmitted on demand, but only at the pauses between computational steps. This makes it difficult to exploit fine-grained parallelism in an application, which is problematic for many graph algorithms.

## Partitioned global address space computing

MPI is not the only way to program distributed memory parallel computers. An important alternative, that is better suited to fine-grained parallelism is to use a *partitioned global address space* language, epitomized by UPC [31]. In a UPC program, the programmer is still responsible for distinguishing between local and global data. But the language supports operations on remote memory locations with simple syntax. This support for a global address space facilitates writing programs with complex data access patterns. UPC sits on top of a communication layer that allows for more fine-grained communication than MPI, and so can sometimes achieve higher performance. However, as with MPI, in a UPC program the number of threads of control is constant, generally equal to the number of processors or cores. As we argue below, the lack of dynamic threads is a significant impediment to the development of high performing graph software.

## Shared-memory computers

UPC provides a software illusion of globally addressable memory on distributed memory hardware. Support for a global address space can also be provided in hardware. Such *shared memory* computers can be categorized in various ways. Here we consider cache-coherent machines and massively multithreaded machines.

### Cache-coherent parallel computers

In symmetric multiprocessors (SMPs), global memory is universally accessible by each processor. The most common ways to program these machines are OpenMP [27], or a threading approach like POSIX threads [69]. The key feature of an SMP is that it provides hardware support for access to addresses in global memory, so any address in the machine can be retrieved quickly. This allows for higher performance on highly unstructured problems than is possible on distributed memory machines. The latency challenge is addressed by faster hardware for accessing memory. However, SMPs have some inherent performance limitations. In a multiprocessor machine with multiple caches, the cache-coherence problem is a significant challenge. This adds overhead which degrades performance, even for problems in which reads are much more common than writes.

A second performance challenge in SMPs is the protocol for thread synchronization and scheduling. If several threads try to access the same region of memory, the system must apply some protocol to ensure correct program execution. Some threads may be blocked

for a period of time. Current versions of OpenMP require the number of threads to equal the number of processors, so a blocked thread corresponds to an idle processor. Although a more dynamic threading model may appear in future versions of OpenMP, currently this problem can cause significant performance challenges for graph algorithms.

### Massively multithreaded architectures

Massively multithreaded machines, such as the Cray MTA-2 [4] and its successor the XMT, address the latency challenge in a very different manner than other architectures. Instead of trying to reduce the latency of single memory access, the MTA-2 tries to *tolerate* latency by ensuring that a processor has other work to do while waiting for a memory request to be satisfied. Each processor can have a large number of outstanding memory requests. The processor has hardware support for many concurrent threads, and switches between them in a single clock cycle. Thus, when a memory request is issued, the processor immediately switches its attention to another thread that is ready to execute. In this way, the processor tolerates latency and is not stalled waiting for memory.

This execution model depends upon the availability of a large number of fine grained, hardware supported, threads to keep the processor occupied. Many graph algorithms can be written in a thread-rich style; however, with a large number of threads, the likelihood of access contention increases. The MTA-2 addresses this problem by supporting word-level synchronization primitives. Each word of memory can be locked independently. Thus, locks have a minimal impact on the execution of other threads.

Another unusual feature of the MTA-2 is its support for fast and dynamic thread creation and destruction. The programmer needn't limit the program to a fixed degree of parallelism, but can instead let the data determine the number of threads. The MTA-2 supports a virtualization of threads, which it then maps onto physical processors. This facilitates adaptive parallelism and dynamic load balancing.

However, massively multithreaded machines also have significant drawbacks. Because the processors are custom and not commodity, they are more expensive and have a much slower clock than mainstream microprocessors. For instance, MTA-2 processors have a clock rate of only 220 MHz, well more than an order of magnitude slower than state-of-the-art microprocessors. Furthermore, the programming model of the MTA-2, while simple and elegant, is not portable to other parallel architectures.

## Software

The different architectures discussed above all have their own programming models. Explicit message passing with MPI is the most portable and widely used paradigm. OpenMP is restricted to shared memory machines, but has some portability among this class. The

MTA-2 programming model is unique to Cray's line of massively multithreaded machines. This raises significant impediments to cross-architectural comparisons. One mechanism to alleviate these problems is to use generic programming libraries that hide machine-specific details.

*Generic programming* underlies the C++ Standard Template Library [75], the Boost C++ Libraries, and in particular the Boost Graph Library (BGL) [74]. This programming paradigm features the implementation of concepts such as iterators using language features such as templates. Generic programming libraries are not only expressive, but efficient as well. BGL algorithms implement the *visitor pattern*, a software methodology that allows programmers to provide custom routines that are executed at predetermined times during execution. For example, graph search algorithms *visit* vertices via edges, and each visitation event presents an opportunity for custom computation. With the visitor pattern, BGL algorithms can be implemented without worrying about low-level details. For example, graphs may be represented with adjacency matrices, adjacency lists, or some other data structure, yet the same algorithm code will run on any of these.

The generic nature of the BGL makes it extensible into an HPC context. BGL algorithms can run on any graph representation that exports a certain interface, and therefore they can run on *distributed* data structures that exploit cluster architectures and export this interface. The *Parallel Boost Graph Library* (PBGL) [42] provides a suite of such data structures. In its purest sense, the PBGL provides a way to run serial graph algorithms on very large problem instances that require the distributed memory of large clusters for storage. However, inherently parallel algorithms have also been implemented in the PBGL. For reasons explained in Sections 4 and 4, there are barriers to consistently achieving *strong scaling* of running time (running faster on the same problem instance when more processors are used) for graph algorithms on distributed memory architectures.

In order to leverage the massively multithreaded architectures described in Section 4, we have extended a small subset of the BGL to become the MultiThreaded Graph Library (MTGL) [11]. This library retains the look and feel of the BGL, yet encapsulates the use of non-standard features such as compiler directives for parallelization and word-level synchronization operators. The visitor pattern is still used to provide algorithm programmers with entry points for custom computation. Although much of the architecture-specific detail is encapsulated within software abstractions, the custom routines provided must still be thread safe, and therefore demand a higher level of programmer expertise.

# Algorithmic results

In this section we describe some recent work comparing graph algorithm implementations on different platforms. More details can be found in some of the citations, e.g. [56]. We will consider two fundamental graph algorithms: $s-t$ *connectivity* and *single-source shortest paths*. In $s-t$ connectivity, the goal is to find a path from vertex $s$ to vertex $t$ that traverses

the fewest possible number of edges. In single-source shortest paths, each edge has a *length* and the goal is to find the shortest length path from a specific vertex to all other vertices in the graph.

## Data

As discussed in §8, graphs associated with physical simulations often have structure induced from the physical geometry. Edges tend to connect vertices that are geometrically near each other. However, the growing field of *informatics* is characterized by datasets of relationships deduced by analyzing information rather than by modeling physical objects. A canonical example of an informatics dataset is the network of relationships between people in some population (a *social network*). The *small world experiment* of Stanley Milgram [60], which led to the "six degrees of separation" principle (popularized in a Hollywood context by actors' distance from Kevin Bacon), found that by following a small number of edges in a social network, one might end up anywhere. Informatics datasets with this small world property lack spacial locality, and therefore are more challenging to map to distributed memory parallel computers.

Another common characteristic of informatics graphs is an *inverse power law* degree distribution. In other words, the vast majority of entities in these networks tend to be connected to just a few other entities, while a few "high-degree" entities are connected to an enormous number of other entities.

In our experiments, we will discuss two different, purely synthetic classes of graphs: Erdös-Rényi [32] random graphs, and a class of inverse power law graphs known as RMAT [18]. Erdös-Rényi graphs are constructed by assigning a uniform edge probability to each possible edge, then using a random number generator to determine which edges exist. RMAT graph construction involves recursively partitioning an adjacency matrix, and assigning neighbor relationships in an uneven manner. Unlike Erdös-Rényi graphs, RMAT graphs have an inverse power law degree distribution.

However, among the machines we discuss below, only the MTA-2 has a programming model and architecture sufficiently robust to easily test instances of inverse power law graphs with close to a billion edges. The work of Yoo, et al. [80] (described below) was limited to Erdös-Rényi graphs, and the current RMAT generator of the PBGL does not scale to large instances. We know of no distributed memory results for giga-scale inverse power law graphs. Given this limitation, we describe results for Erdös-Rényi graphs only and note that the MTA-2 performance on like-sized RMAT graphs is almost identical.

High degree nodes are a challenge for distributed memory machines for several reasons. A standard practice in scientific computing on distributed memory platforms is to store "ghost nodes" on each processor that represent the neighbors of all graph vertices owned by that processor. With ghost nodes, vertices can traverse all of their neighbors, know which of them are stored remotely, and avoid some remote communication. However, this simple

strategy does not scale to large instances of graphs with inverse power law distributions since a single processor cannot be expected to store ghost nodes for the neighbors of high-degree nodes. As an alternative, Yoo, et al. avoided the use ghost nodes, but with their approach high degree vertices result in the need for very large message buffers. A fundamental tension exists between runtime scalability, which ghost nodes help, and memory scalability, which ghost nodes limit.

## S-T Connectivity

For s-t connectivity, we consider the following simple algorithm: given two vertices $s$ and $t$: find the neighbors of $s$, see if any of them is $t$. If not, then find the neighbors of $t$ and see if any of them is one of the vertices in $s$'s expanding frontier. Repeat this process by expanding the smaller of the frontiers of $s$ and $t$ until the two frontiers intersect (see Figure 4.1). Yoo, et al. [80] used this algorithm on Erdös-Rényi graphs of 3.2 billion nodes, and reported results on 32,768 processors of the world's largest distributed memory machine, BlueGene/L. Notably, this implementation was memory-efficient since it did not use ghost nodes. However, as will be discussed below, this came at the cost of significantly reduced performance. For a fixed sized problem, Yoo, et al. report a speedup of about 65 on 450 processors. Yoo, et al. also report runtimes, for a series of scaled problems in which the size of the graph grows with the number of processors. Since the amount of work in s-t connectivity grows less quickly than the size of the graph, however, the assessment of scalability requires some care. On an Erdös-Rényi graph, it is straightforward to analyze the expected number of vertices to be visited to find the shortest path between s and t. For the algorithm sketched above applied to the instances studied by Yoo, et al., the number of vertices visited should be about 177 times larger for the graph on 32,768 nodes than for the graph on 1 node. With the runtime growing by a factor of three, this suggests an overall speedup of around 60. Unfortunately, the code of Yoo, et al. was never tested on RMAT instances, in part due to the concerns about message buffer sizes mentioned above.

Madduri, et al. [7] implemented the same s-t connectivity algorithm on the MTA-2 and achieved a speedup factor of about 28 on 40 processors, for both Erdös-Rényi and RMAT instances. A simple counting argument based on the number of vertices touched during the s-t connectivity algorithm suggests that the computation done by 32,768 processors of BlueGene/L could be done by 5-10 processors of an MTA-2 with sufficient memory.

The same algorithm was implemented in the PBGL [56], and achieved excellent single processor performance. Lumsdaine, et al. used compact data structures to improve cache utilization resulting in single processor performance comparable to that obtained by Yoo, et al. on 30,768 processors of BlueGene/L. However, Lumsdaine, et al. were not able to get any reduction in runtime as processors were added, even with the use of ghost nodes. Fundamentally, there isn't much work to do in an s-t connectivity algorithm. Even if the graph is larger, only a small subset of vertices need to be visited for Erdös-Rényi graphs. Thus, it is difficult to outperform a fast serial algorithm.

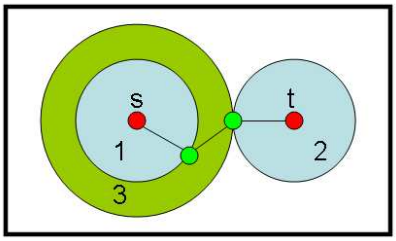



**Figure 4.1.** A simple s-t connectivity algorithm: 1) find the neighbors of $s$ and see if $t$ is one of them, 2) find the neighbors of $t$ and see if $s$ or one of its neighbors is one of them, 3) alternate back to $s$ and expand its frontier one more level (etc.)

## Single-Source Shortest Paths

A fundamental problem in graph theory is that of finding *single-source shortest paths* (sssp). Given a single starting vertex, sssp algorithms compute a shortest path to each vertex in the graph, as illustrated in Fig. 4.2. A classical algorithm by Dijkstra solves this problem by sequentially finding and "settling" the closest unsettled vertex to the source [29]. This elegant algorithm is inherently serial, but several variations of it attempt to find and exploit parallelism. They do this by trying to find *many* vertices that may be settled at the same time. Such algorithms are highly sensitive to the type of graph processed, and some graph types, such as road networks, do not offer enough parallelism for these schemes to work well. However, in the case of Erdös-Rényi random graphs and RMAT graphs, some positive results have been obtained. Perhaps most notable of these is that of Madduri, et al. [57], who used an implementation of Meyer and Sanders' *delta stepping* algorithm [59], to find sssp on an RMAT graph of roughly one billion edges in about 10 seconds on a 40 processor MTA-2. The single MTA-2 processor time for the same instance was 371 seconds, yielding a parallel speedup factor of about 30.

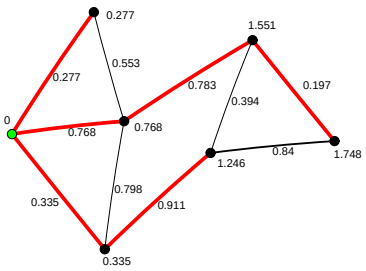



**Figure 4.2.** The result of calling an algorithm for sssp. The vertices are labeled with their distance from the single source, and the edges are labeled with their lengths. The red edges form a single-source shortest paths tree.

A PBGL version of the same algorithm was developed by Lumsdaine, et al. [56]. On an Opteron cluster, they report performance that is about an order of magnitude slower than the MTA-2 performance. The Opterons in that experiment have 2.0 GHz clocks, while the MTA-2 processors at clocked at 220 MHz. This suggests that the MTA-2 is about two orders more efficient than the Opterons for this problem. It is also worth noting that the PBGL code used ghost nodes, making it less memory-efficient than the MTA-2 software.

However, unlike in the s-t connectivity study, the PBGL implementation of delta stepping displayed excellent scalability. This scalability speaks well for the generic programming software model of PBGL. The implementation of delta stepping required only about a day of programmer effort. Pre-PBGL distributed graph algorithm implementations of similar complexity (e.g. the Yoo, et al. code discussed above) often required orders of magnitude more development effort.

# Conclusions

As combinatorial algorithms become increasingly important in science, engineering and other applications, their distinctive computational requirements will grow in significance. In this paper we have focused on the challenges of graph algorithms, but we believe that many combinatorial (and other) algorithms confront similar challenges. Unlike most scientific computing kernels, graph algorithms exhibit complex memory access patterns and limited amounts of actual processing. As a consequence, their performance is determined by the ability of a computer to access memory, and not by the speed of the processor itself. Complex data dependencies and dynamic, fine-grained parallelism result in poor parallel performance on traditional machines.

Although graphs may be an extreme case, we believe that there is a broad trend in the scientific computing community towards increasingly complex and memory-limited simulations. Unstructured grids involve much more complex memory access patterns than structured grids. Adaptive grids are even more challenging, and lead to dynamic parallelism. Multiphase and multiphysics simulations lead to an additional degree of dynamism in a computation. These complex calculations generally achieve a very low percentage of peak performance on a single processor and exhibit poor parallel scalability.

We believe that our work with massively multithreaded machines suggests an alternative with the potential to significantly improve the performance of challenging computations. Conveniently, thanks to the the continued march of Moore's Law, there is silicon to spare on current microprocessors. We believe that this space could be used to support massive multithreading, resulting in processors and parallel machines that are applicable to a much broader range of applications than current offerings.

# Chapter 5

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

The following was published in the "ALENEX" (Algorithm engineering and experimentation), a workshop of the ACM/SIAM Symposium on Discrete Algorithms (SODA) conference. ALENEX is a prestigious workshop with an acceptance rate of %30 or below. The LDRD supported the latter stages of this work.

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

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

**Abstract**

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

## 1 Introduction

We present an experimental study of the $\Delta$-stepping parallel algorithm [29] for solving the single source shortest path problem on large-scale graph instances. In addition to applications in combinatorial optimization problems, shortest path algorithms are finding increasing relevance in the domain of complex network analysis. Popular graph theoretic analysis metrics such as betweenness centrality [19, 9, 39, 41, 32] are based on shortest path algorithms. Our parallel implementation targets graph families that are representative of real-world, large-scale networks [6, 22, 12, 30, 50]. Real-world graphs are typically characterized by a low diameter, heavy-tailed degree distributions modeled by power laws, and self-similarity. They are often very large, with the number of vertices and edges ranging from several hundreds of thousands to billions. On current workstations, it is not possible to do exact in-core computations on these graphs due to the limited physical memory. In such cases, parallel computing techniques can be applied to obtain exact solutions for memory and compute-intensive graph problems quickly. For instance, recent experimental studies on Breadth-First Search for large-scale graphs show that a parallel in-core implementation is two orders of magnitude faster than an optimized external memory implementation [4, 2]. In this paper, we present an efficient parallel implementation for the single source shortest paths problem that can handle scale-free instances in the order of billions of edges. In addition, we conduct an experimental study of performance on several other graph families, also used in the 9th DIMACS Implementation Challenge [14] on Shortest Paths. Please refer to our technical report [25] for additional performance details.

Sequential algorithms for the single source shortest path problem with non-negative edge weights (NSSP) are studied extensively, both theoretically [16, 18, 18, 24, 32, 34, 33, 21, 46] and experimentally [15, 28, 27, 13, 59, 20]. 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 [16] algorithm. Using Fibonacci heaps [18], Dijkstra's algorithm can be implemented in $O(m + n \log n)$ time. Thorup [34] presents an $O(m + n)$ RAM algorithm for undirected graphs that differs significantly different from Dijkstra's approach. Instead of visiting vertices in the order of increasing distance, it traverses a *component tree*. Meyer [47] and Goldberg [20] propose simple algorithms with linear average time for uniformly distributed edge weights.

Parallel algorithms for solving NSSP are reviewed in detail by Meyer and Sanders [46, 29]. There are no known PRAM algorithms that run in sub-linear time and $O(m + n \log n)$ work. Parallel priority queues [17, 10] 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 [22, 26], proposed for the parallel All-Pairs Shortest Paths (APSP), involve running time and efficiency trade-offs. Parallel approximate NSSP algorithms [23, 16, 33] based on the randomized Breadth-First search algorithm of Ullman and Yannakakis [36]

*Georgia Institute of Technology
†Sandia National Laboratories
‡Rutgers University

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

Meyer and Sanders give the $\Delta$-stepping [29] NSSP algorithm that divides Dijkstra's algorithm into a number of *phases*, each of which can be executed in parallel. For random graphs with uniformly distributed edge weights, this algorithm runs in sub-linear time with linear average case work. Several theoretical improvements [28, 26, 27] are given for $\Delta$-stepping (for instance, finding shortcut edges, adaptive bucket-splitting), but it is unlikely that they would be faster than the simple $\Delta$-stepping algorithm in practice, as the improvements involve sophisticated data structures that are hard to implement efficiently. On a random d-regular graph instance ($2^{19}$ vertices and $d = 3$), Meyer and Sanders report a speedup of 9.2 on 16 processors of an Intel Paragon machine, for a distributed memory implementation of the simple $\Delta$-stepping algorithm. For the same graph family, we are able to solve problems three orders of magnitude larger with near-linear speedup on the Cray MTA-2. For instance, we achieve a speedup of 14.82 on 16 processors and 29.75 on 40 processors for a random d-regular graph of size $2^{29}$ vertices and $d$ set to 3.

The literature contains few experimental studies on parallel NSSP algorithms [35, 31, 37, 35]. Prior implementation results on distributed memory machines resorted to graph partitioning [12, 1, 31], and running a sequential NSSP algorithm on the sub-graph. Heuristics are used for load balancing and termination detection [36, 38]. The implementations perform well for certain graph families and problem sizes, but in the worst case, there is no speedup.

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

We present our shortest path implementation results on the Cray MTA-2, a massively multithreaded parallel machine. The MTA-2 is a high-end shared memory system offering two unique features that aid considerably in the design of irregular algorithms: fine-grained parallelism and low-overhead word-level synchronization. The MTA-2 has no data cache; rather than using a memory hierarchy to reduce latency, the MTA-2 processors use hardware multithreading to tolerate the latency. The word-level synchronization support complements multithreading and makes performance primarily a function of parallelism. Since graph algorithms have an abundance of parallelism, yet often are not amenable to partitioning, the MTA-2 architectural features lead to superior performance and scalability. Our recent results highlight the exceptional performance of the MTA-2 for implementations of key combinatorial optimization and graph theoretic problems such as list ranking [3], connected components [3, 7], subgraph isomorphism [7], Breadth-First Search and *st*-connectivity [4].

The main contributions of this paper are as follows:

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

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

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

## 2 Review of the $\Delta$-stepping Algorithm

Let $G = (V, E)$ be a graph with $n$ vertices and $m$ edges. Let $s \in V$ denote the source vertex. Each edge $e \in E$ is assigned a non-negative real weight by the length function $l : E \rightarrow \mathbb{R}$. 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 $\delta(s) = 0$.

Most shortest path algorithms maintain a *tentative distance* value for each vertex, which are updated by *edge relaxations*. Let $d(v)$ denote the tentative distance of a vertex $v$. $d(v)$ is initially set to $\infty$, 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. $\Delta$-stepping belongs to the label-correcting type of shortest path algorithms.

The $\Delta$-stepping algorithm (see Alg. 1) is an "approximate bucket implementation of Dijkstra's algorithm" [29]. It maintains an array of buckets $B$ such that $B[i]$ stores the set of vertices $\{v \in V : v$ is queued and $d(v) \in [i\Delta, (i+1)\Delta)\}$. $\Delta$ is a positive real number that denotes the "bucket width".

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

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

**Algorithm 1:** $\Delta$-stepping algorithm

**Input**: $G(V, E)$, source vertex $s$, length function $l : E \to \mathbb{R}$
**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         B[i] ⟵ φ;
13         foreach (v,x) ∈ Req do
14             relax(v, x);
15     Req ⟵ {(w, d(v) + l(v,w)) : v ∈ S ∧
       ⟨v,w⟩ ∈ heavy(v)};
16     foreach (v,x) ∈ Req do
17         relax(v, x);
18     i ⟵ i + 1;
19 foreach v ∈ V do
20     δ(v) ⟵ d(v);
```

**Algorithm 2:** The *relax* routine in the $\Delta$-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;
```

similar to the Bellman-Ford algorithm. It has a high degree of parallelism, but is inefficient compared to Dijkstra's algorithm. $\Delta$-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 [29]. We summarize the salient results.

Let $d_c$ denote the maximum shortest path weight, and $P_\Delta$ denote the set of paths with weight at most $\Delta$. Define a parameter $l_{max}$, an upper bound on the maximum number of edges in any path in $P_\Delta$. 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 [29] theoretically prove that $\Delta = \theta(1/d)$ is a good compromise between work efficiency and parallelism. The sequential algorithm performs $O(dn)$ expected work divided between $O(\frac{d_c}{\Delta} \cdot \frac{\log n}{\log \log n})$ phases *with high probability*. In practice, in case of graph families for which $d_c$ is $O(\log n)$ or $O(1)$, the parallel implementation of $\Delta$-stepping yields sufficient parallelism for our parallel system.

## 3 Parallel Implementation of $\Delta$-stepping

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

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

We also support fast parallel insertions into the request set $R$. $R$ stores $\langle v, x \rangle$ pairs, where $v \in V$ and $x$ is the requested tentative distance for $v$. We add a vertex $v$ to $R$ only if it satisfies the condition $x < d(v)$. We do not store duplicates in $R$. We use a sparse set representation similar to one used by Briggs and Torczon [9] for storing vertices in $R$. This sparse data structure uses two arrays of size $n$: a *dense* array that contiguously stores the elements of the set, and a *sparse* array that indicates whether the vertex is a member of the set. Thus, it is easy to iterate over the request set, and membership queries and insertions are constant time. Unlike other Dijkstra-based algorithms, we do not relax edges in one step. Instead, we inspect adjacencies (light edges) in each phase, construct a request set of vertices, and then relax *vertices* in the relax step.

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

To saturate the MTA-2 processors with work and to obtain high system utilization, we need to minimize the number of phases and non-empty buckets, and maximize the request set sizes. Entering and exiting a parallel phase involves a negligible running time overhead in practice. However, if the number of phases is $O(n)$, this overhead dominates the actual running time of the implementation. Also, we enter the relax routine once every phase. The number of implicit barrier synchronizations in the algorithm is proportional to the number of phases. Our implementation reduces the number of barriers. Our source code for the $\Delta$-stepping implementation, along with the MTA-2 graph generator ports, is freely available online [24].

## 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 $\Delta$-stepping code is written in C with MTA-2 specific pragmas and directives for parallelization. We com-

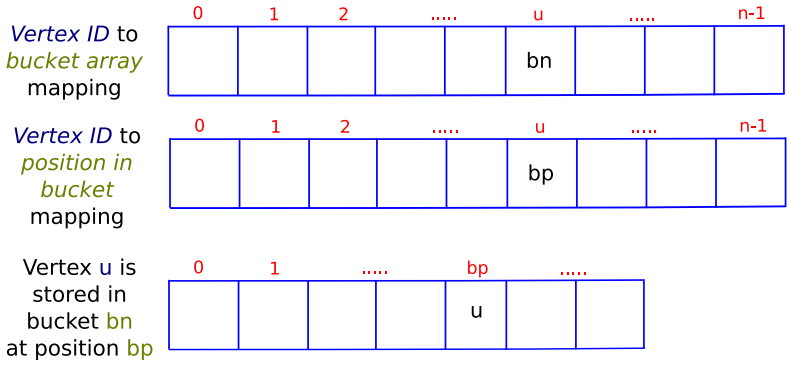



Figure 1: Bucket array and auxiliary data structures

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

The MTA-2 code also compiles and runs on sequential processors without any modifications. Our test platform for the sequential performance results is one processor of a dual-core 3.2 GHz 64-bit Intel Xeon machine with 6GB memory, 1MB cache and running RedHat Enterprise Linux 4 (linux kernel 2.6.9). We compare the sequential performance of our implementation with the DIMACS reference solver [14]. Both the codes are compiled with the Intel C compiler (icc) Version 9.0, with the flags `-O3`.

**4.2 Problem Instances** We evaluate sequential and parallel performance on several graph families. Some of the generators and graph instances are part of the DIMACS Shortest Path Implementation Challenge benchmark package [14]:

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

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

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

In addition, we also study the following families:

- *Scale-free graphs*: We use the R-MAT graph model [11] for real-world networks to generate scale-free graphs. We define the family *ScaleFree4-n* similar to random graphs.

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

**4.3 Methodology** For sequential runs, we report the execution time of the reference DIMACS NSSP solver (an efficient implementation of Goldberg's algorithm [21], which has expected-case linear time for some inputs) and the baseline Breadth-First Search (BFS) on every graph family. The BFS running time is a natural lower bound for NSSP codes and is a good indicator of how optimized the shortest path implementations are. It is reasonable to directly compare the execution times of the reference code and our implementation: both use a similar adjacency array representation for the graph, are written in C, and compiled and run in identical experimental settings. Note that our implementation is optimized for the MTA-2 and we make no modifications to the code before running on a sequential machine. The time taken for semi-sorting and mechanisms to reduce memory contention on the MTA-2 both constitute overhead on a sequential processor. Also, our implementation assumes real-weighted edges, and we cannot use fast bitwise operations. By default, we set the value of $\Delta$ to $\frac{n}{m}$ for all graph instances. We will show that this choice of $\Delta$ may not be optimal for all graph classes and weight distributions.

On a sequential processor, we execute the BFS and

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

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

On the MTA-2, we compare our implementation running time with the execution time of a multithreaded level-synchronized breadth-first search [5], optimized for low-diameter graphs. The multithreaded BFS scales as well as $\Delta$-stepping for all the graph instances considered, and the execution time serves as a lower bound for the shortest path running time.

The first run on the MTA-2 is usually slower than subsequent ones (by about 10% for a typical $\Delta$-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 [25], and report execution time in seconds.

## 5 Results and Analysis

**5.1 Sequential Performance** First we present the performance results of our implementation on the reference sequential platform, experimenting with various graph families. Fig. 2 compares the execution time across graph instances of the same size, but from different families. The DIMACS reference code is about 1.5 to 2 times faster than our implementation for large problem instances in each family. The running time on the *Random4-n* is slightly higher than the rest of the families. For additional details such as performance as we vary the problem size for BFS, $\Delta$-stepping, and the DIMACS implementation execution times, please refer to Section B.1 of [25]. Our key observation is that the ratio of the $\Delta$-stepping execution time to the BFS time varies between 3 and 10 across different problem instances.

**5.2 $\Delta$-stepping analysis** To better understand the algorithm performance across graph families, we use

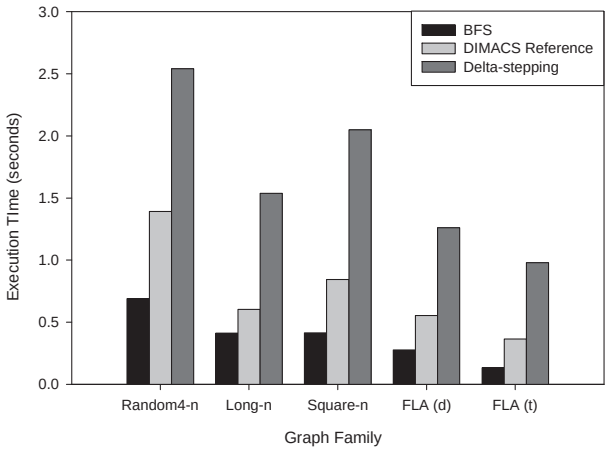
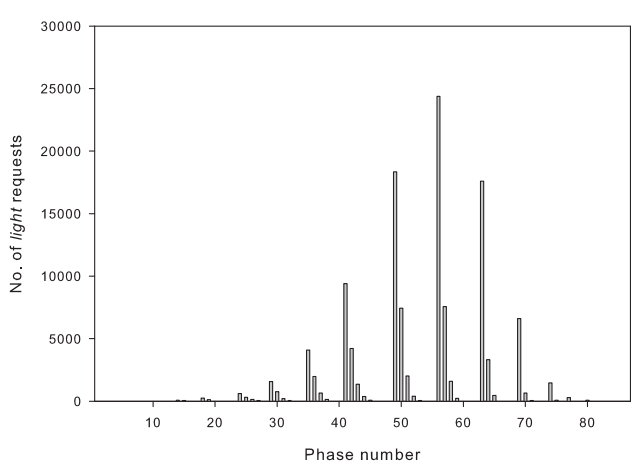


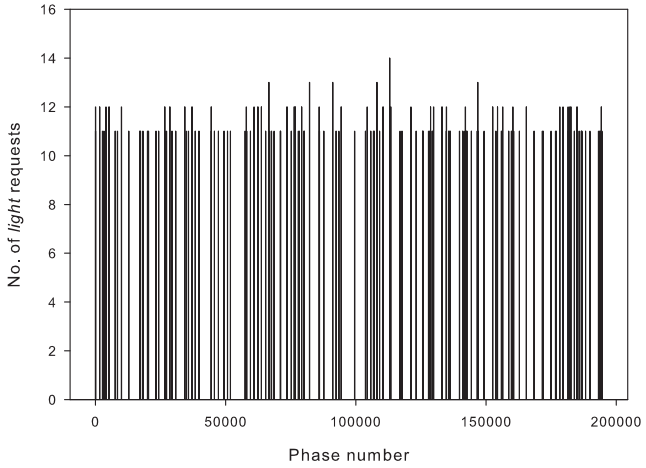

Figure 2: Sequential performance of our $\Delta$-stepping implementation on the core graph families. All the synthetic graphs are directed, with $2^{20}$ vertices and $\frac{m}{n} \approx 4$. FLA(d) and FLA(t) are road networks corresponding to Florida, with 1070376 vertices and 2712768 edges

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

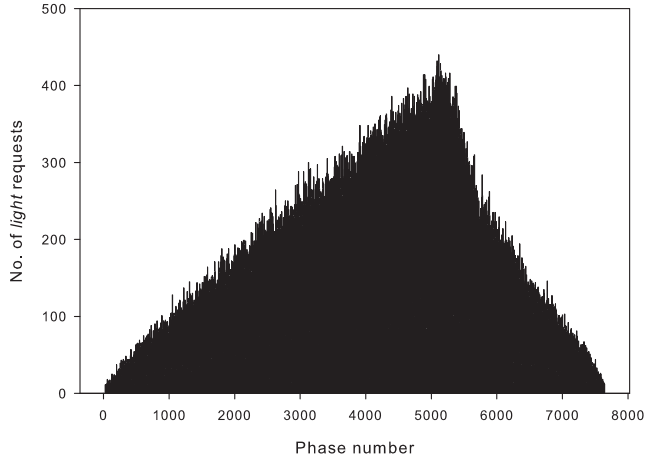
Fig. 4 plots several key $\Delta$-stepping operation counts for various graph classes. All synthetic graphs are roughly of the same size. Fig. 4(a) plots the average



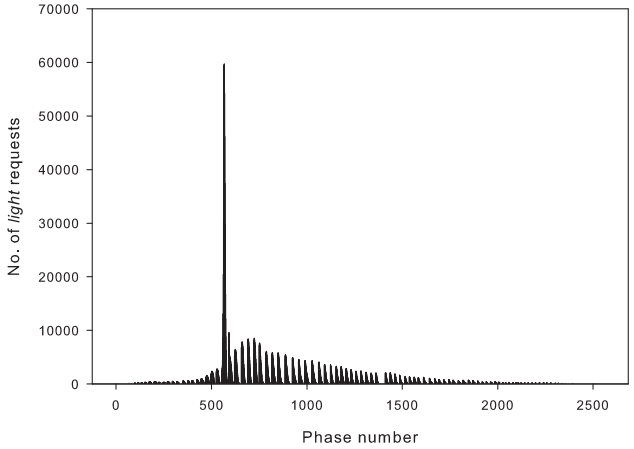

(a) Random4-n family, $n = 2^{20}$.




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




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




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

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

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

We next evaluate the performance of the algorithm as $\Delta$ 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. $\Delta$ 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 $\Delta$ is varied (See Section B.2 in [25]). 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 $\Delta$ 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 $\Delta$ 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 $\Delta$ (and the execution time on the MTA-2) is also dependent on the number of processors. For a particular $\Delta$, 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 $\Delta$.

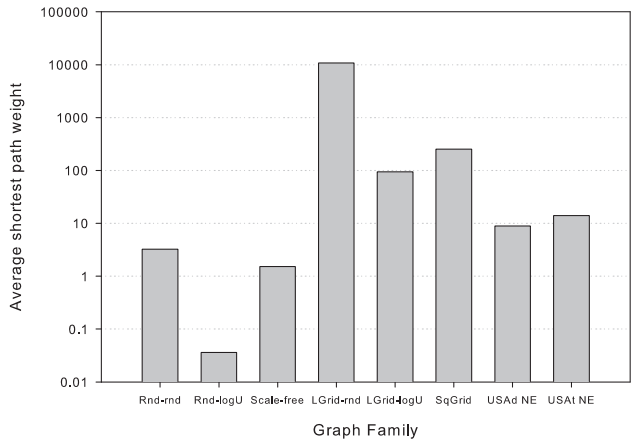
**5.3 Parallel Performance** We present the parallel scaling of the $\Delta$-stepping algorithm in detail. We ran $\Delta$-stepping and the level-synchronous parallel BFS on all graph instances described in Section 4.2 (see [25] for complete tabular results from all experiments). We define the speedup on $p$ processors of the MTA-2 as the ratio of the execution time on 1 processor to the execution time on $p$ processors. In all graph classes except long grids, there is sufficient parallelism to saturate a single processor of the MTA-2 for reasonably large problem instances.

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

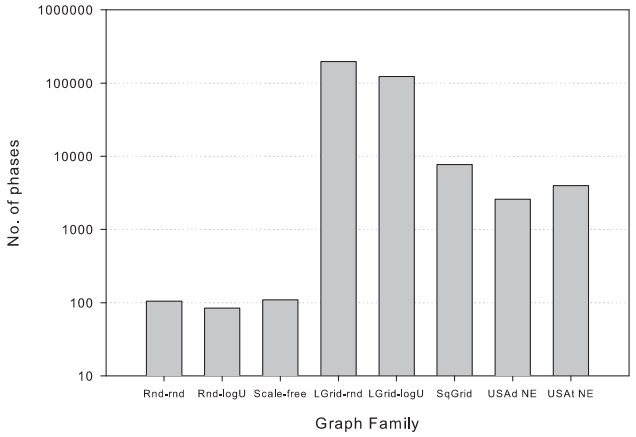
In case of all the graph families, the relative speedup increases as the problem size is increased (for e.g., on 40 processors, the speedup for a *Random4-n* instance with $n = 2^{21}$ is just 3.96, whereas it is 31.04 for $2^{28}$ vertices). This is because there is insufficient parallelism in a problem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

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

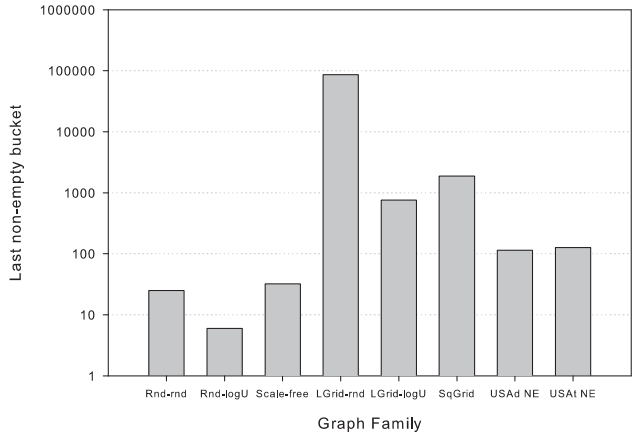
In case of *Long-n* graphs and $\Delta$ set to $\frac{n}{m}$, there is insufficient parallelism to fully utilize even a single processor of the MTA-2. The execution time of the level-synchronous BFS also does not scale with the number of processors. In fact, the running time goes up in case of multiprocessor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



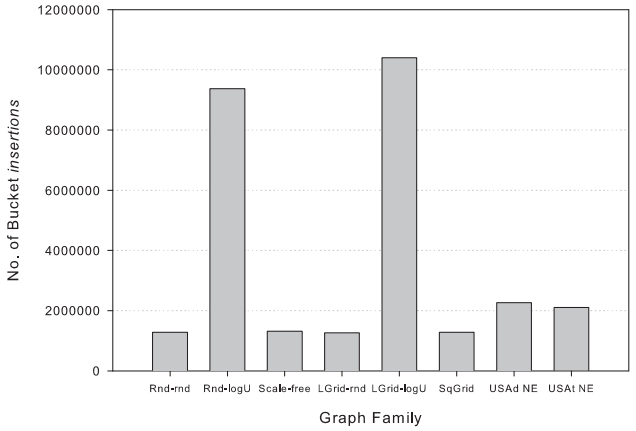

(a) Average shortest path weight ($\frac{1}{n} * \sum_{v \in V} \delta(v)$)




(b) No. of phases

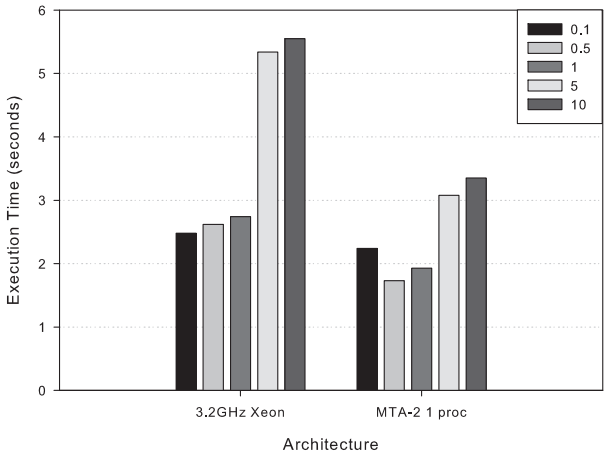



(c) Last non-empty bucket

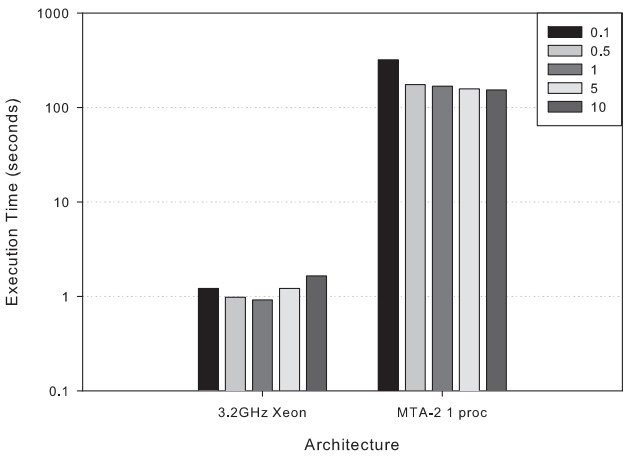



(d) Number of relax requests

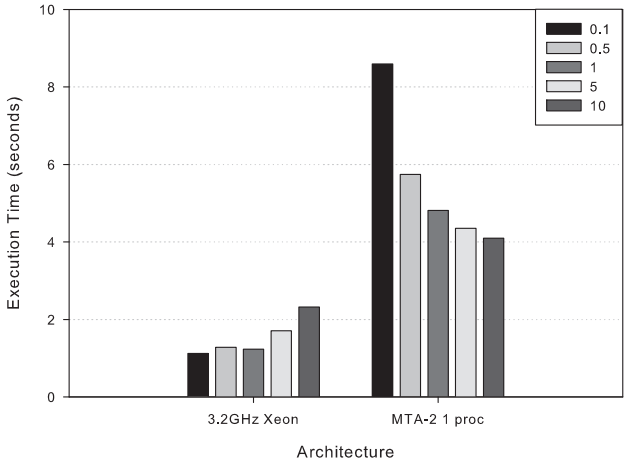
Figure 4: $\Delta$-stepping algorithm performance statistics for various graph classes. All synthetic graph instances have $n$ set to $2^{20}$ and $m \approx 4n$. Rand-rnd: Random graph with random edge weights, Rand-logU: Random graphs with log-uniform edge weights, Scale-free: Scale-free graph with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



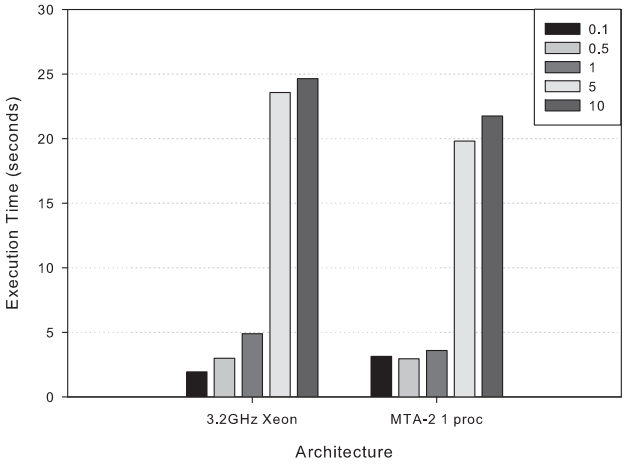

(a) Random4-n family. $2^{20}$ vertices




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




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




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

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

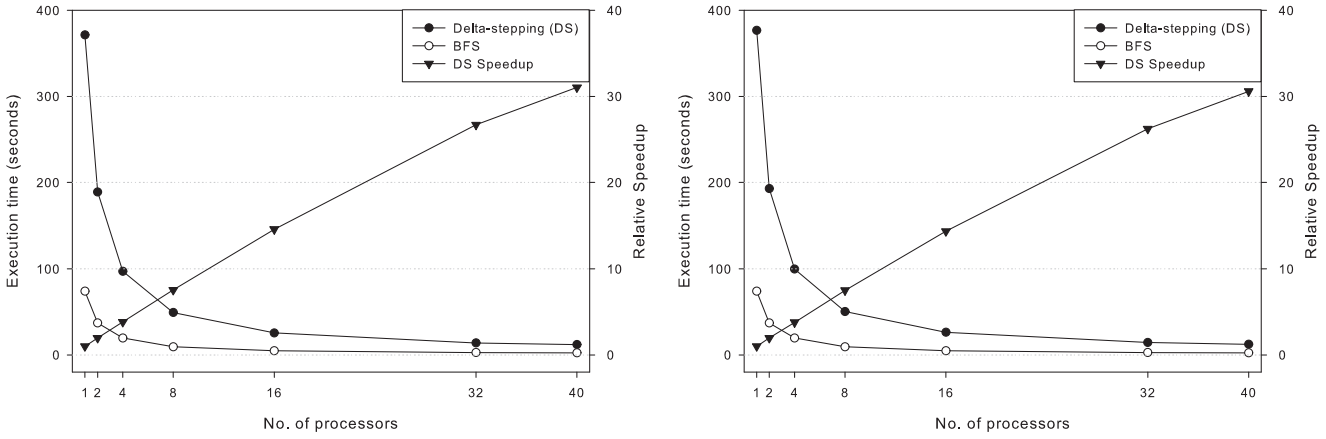



Figure 6: Δ-stepping execution time and relative speedup on the MTA-2 for Random4-n (left) and ScaleFree4-n (right) graph instances (directed graph, $n$=$2^{28}$ vertices and $m = 4n$ edges, random edge weights).

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

## 6 Conclusions and Future Work

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

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


### Acknowledgments

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

## References


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## References

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

[2] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory BFS algorithms. In

# Chapter 6

# A multithreaded algorithm for finding triangles

The following is unpublished work describing a multithreaded algorithm for finding triangles (3-cycles) in graphs. It has been superceded in terms of Cray XMT performance by an algorithm of Jonathan Cohen [23]. However, our ideas below are novel and might have application in some context.

## Background

A simple algorithm for finding triangles (3-cycles) in an undirected graph is to compute the intersection of the neighborhoods of the endpoints of each edge. Implementations of this approach involve defining and searching data structures for each vertex. When input instances are social networks with inverse power law degree distributions, for example, we might wish to store the adjacencies of high-degree nodes in hash tables or treaps to cut down on search time, or to leverage efficient set intersection operations.

The strategy above is conceptually simple, but incurs large overheads in actual implementation. We propose to replace this conceptually simple strategy with a more subtle one that obviates the need for extra memory and search logic. Our algorithm repeatedly finds maximal independent sets, and performs a three-phase process on each set in order to to identify all of the triangles.

We give the algorithm below, without any correctness proof yet.

## The Algorithm

The basic idea of our algorithm is to find triangles using a mark and sweep process rather than computing set intersections. Let $G = (V, E)$ be an undirected graph and $S \subset V$ be an initially empty set of *settled* vertices. A vertex is consided to be settled if all triangles containing that vertex have been identified.

Define a function $f : V \to \mathcal{R}$. For example, we might define $f(v) = \delta(v)$ (the degree of $v$). This will be used to arbitrate between processing elements in the algorithm below. Also, define $m(v) \in V$ to be the *mark* of $v$. Vertices will mark each other in the algorithm. Define $C_v$ to be an initially empty set of *confounded* vertices. The meaning of "confounded" will be given below.

Let $N(v)$ be the set of vertices adjacent to $v$, and let $E(v)$ be the set of edges incident on $v$. Let $E_f$ be an initially empty set of edges.

## Processing a Maximal Independent Set

Find a maximal independent set $S' \in (V - S, E - E_f)$ using, for example, Luby's algorithm. Set $C_v = \emptyset$, and set $m(v) = v$ for all $v \in V - S$. Process $S'$ using the three phases described below.

### Mark Phase

For each $v \in S'$, do the following. For every $w \in N(v)$, attempt to set $m(w) = v$ in a thread safe manner. If it is determined that $m(w)$ was set previously in this phase, then let $C_v = C_v \cup \{w\}$.

if $f(v) > f(w)$, then set $m(w) = f(v)$. In other words, if several vertices in $S'$ are adjacent to $w$, then $w$ is confounded, and the dominant $S'$ vertex in $N(w)$ with respect to $f$ will mark $w$.

### Sweep Phase

Iterate through all edges $E - E_f$. Each such edge with identically marked endpoints identifies a single triangle. Each $v \in S'$ that successfully marked all of its neighbors is now settled; let $S = S \cup v$ and $E_f = E_f \cup E(v)$ for all such vertices.

### Conflict Resolution Phase

Some vertices of $S'$ may not be settled after the mark and sweep phases. This is exactly the set of vertices

$$U = \{w \in S' : \exists_{u \in N(w)} m(u) \neq w\}.$$

It is convenient to classify the neighbors of $w \in S'$:

$$N_c(w) = \{u \in N(w) : m(u) \neq w\}, \text{ and } N_g(w) = \{u \in N(w) : m(u) = w\}.$$

We will not be able to settle $w \in U$ in this phase. However, we can find all triangles involving the *non-confounding* edges $(w, u) : u \in N_g(w)$.

Conceptually, this is done by identifying all *confounding edges* $(w, u) : u \in N_c(w)$. Note that a settled vertex is not incident to any confounding edges. Given a confounding edge $(w, u)$, we could search $N(u)$. Each vertex $v \in N(u)$ such that $m(v) = w$ identifies a triangle. Once all confounding edges have been processed, all non-confounding edges incident on vertices of $S'$ can be added to $E_f$, since all triangles involving those edges have been identified.

However, this conceptual strategy is best implemented without processing the confounding edges explicitly. The adjacencies of confounding vertices of high degree will become hot spots, as many incident confounding edges may trigger adjacency list traversals.

Instead of processing the confounding edges, we process the confounding vertices. For each $c_v \in C_v$, the conflict resolution phase traverses $N(c_v)$ exactly once. Define the set of mark value classes of vertices adjacent to a confounded vertex $c_v$ as follows:

$$M_{c_v} = \{m_i : \exists_{u \in N(c_v)} m_i = m(u)\}.$$

We will consider each class of mark value $m_i$ independently. Let

$$N_i(c_v) = N(c_v) \cap \{u : m(u) = m_i\}.$$

If $N_i(c_v) \cap S' \neq \emptyset$, then we have identified the $|N_i(c_v)| - 1$ triangles that include both $c_v$ and a single independent set vertex in $N_i(c_v) \cap S'$. On the other hand, if this intersection is empty, then the mark class $N_i(c_v)$ contains no vertex in $U$ and hence, no triangles incident on any vertex of $U$ touch $N_i(c_v)$.

When all $m_i \in M_{c_v}$ have been processed for all $c_v \in C_v$, then we have identified all triangles including non-confounding edges incident on vertices of $S'$. These edges can safely be added to $E_f$.

## Termination

The algorithm repeatedly finds maximal independent sets in $(V - S, E - E_f)$, then executes the mark, sweep, and conflict resolution phases until all vertices in $V$ have been settled. In practice, a cleanup phase follows the conflict resolution phase. This settles degree one vertices and their indicent edges.

# Chapter 7

# Software and algorithms for graph queries on massively multithreaded architectures

The following was published as part of the 1st IEEE workshop on *MultiThreaded Architectures & Applications* (MTAAP). This LDRD supported the writing and publication of this paper, though much of the work was accomplished prior to the beginning of the LDRD. It is the inital MTGL paper. Note that significant LDRD effort went into further developing the MTGL after this publication. The proper citation is [12].


**Abstract**

Search-based graph queries, such as finding short paths and isomorphic subgraphs, are dominated by memory latency. If input graphs can be partitioned appropriately, large cluster-based computing platforms can run these queries. However, the lack of compute-bound processing at each vertex of the input graph and the constant need to retrieve neighbors implies low processor utilization. Furthermore, graph classes such as scale-free social networks lack the locality to make partitioning clearly effective.

Massive multithreading is an alternative architectural paradigm, in which a large shared memory is combined with processors that have extra hardware to support many thread contexts. The processor speed is typically slower than normal, and there is no data cache. Rather than mitigating memory latency, multithreaded machines tolerate it. This paradigm is well aligned with the problem of graph search, as the high ratio of memory requests to computation can be tolerated via multithreading.

In this paper, we introduce the MultiThreaded Graph Library (MTGL), generic graph query software for processing semantic graphs on multithreaded computers. This library currently runs on serial machines and the Cray MTA-2, but Sandia is developing a run-time system that will make it possible to run MTGL-based code on Symmetric MultiProcessors. We also introduce a multithreaded algorithm for connected components and a new heuristic for inexact subgraph isomophism. We explore the performance of these and other basic graph

algorithms on large scale-free graphs. We conclude with a performance comparison between the Cray MTA-2 and Blue Gene/Light for s-t connectivity.

# Introduction

Typical microprocessors combine several layers of cache into a memory hierarchy, then rely on the spacial and temporal locality inherent in many applications. Graph algorithms, however, might have neither. This is especially true when they are applied to unstructured graphs such as social networks.

A *semantic* graph (or *attributed relational graph*) is a graph with types on the vertices and/or edges. Vertices are typically "nouns" and edges are typically "verbs." Social networks, for example, are semantic graphs. The world focus on counter-terrorism as a primary challenge has made the processing of large, unstructured semantic graphs an important research area.

The shared-memory programming model of the massively multithreaded Cray MTA and Eldorado machines offer the mixed blessing of a higher level of abstraction than message passing/MPI models, but relatively more subtle concurrency and performance issues. The MTGL is designed to encapsulate many of these subtleties for standard graph kernel algorithms.

We present the MTGL in stages. In Section 7, we describe the design goals and primary design pattern of the MTGL. Then, in Section 8, we give high-level pseudocode descriptions of the MTGL implementations of three kernel algorithms: connected components, subgraph isomorphism, and s-t connectivity. These descriptions will highlight the generic nature of the graph search primitives within the MTGL, as they are reused several times.

Note in advance that there is no graph or matrix partitioning in the MTGL kernel algorithms we describe. The MTA programmer does not explicitly make assignments of tasks or data to specific processors. This is handled by the runtime system of the MTA. In fact, the memory of the MTA-2 is hashed at the word level in order to intentionally destroy locality.

Synchronization in the MTA-2 is handled with a full/empty bit associated with every word. The architecture does support concurrent reads and writes, but the programmer must be wary of hot spots.

# MTGL Design Methodology

The MTGL is a small prototype C++ library that is inspired by the Boost Graph Library (BoostGL) of Siek, Lee, and Lumsdaine [74]. However, our library is not an extension of BoostGL. The MTA and Eldorado compilers are not fully compliant with the C++ standard, and BoostGL makes aggressive use of the language in order to maximize its flexibility. The MTGL is not designed to be as generic as BoostGL. Rather, our primary design goals are to maximally expose the performance of multithreaded machines and to maximally encapsulate the threats to successful development of applications: race conditions and hotspotting. Whereas the BoostGL has numerous graph representations, data structures, and algorithms, our prototype MTGL has but a few.

The primary design pattern of the MTGL is the *visitor pattern*. Algorithms are defined by library programmers as objects, and are customized by user-defined "visitor" classes. We show several examples of the use of visitors below. Performance results are then presented in Section 7. We omit actual code samples in this paper. At the time of this writing, we are in the process of obtaining an open-source license for the MTGL.

# Notation

In order to describe multithreaded graph algorithms and their implementations in the MTGL, it is convenient to define some notation. We begin with the familiar definition of a graph: $G = (V, E)$, were $V$ is the vertex set of $G$ (also denoted $V(G)$), $E$ is the edge set of $G$ ($E(G)$), and $E(v)$ is the set of edges incident on vertex $v$. we define a type function $t$ such that $t(v)$ is the type of $v \in V$, and $t(v, w)$ is the type of edge $(v, w)$. In this paper and in the prototype MTGL, all graphs are assumed to be directed. Undirected graphs are constructed by enforcing the property that whenever $(v, w)$ exists, $(w, v)$ will exist as well. In social networks, reciprocal relationships almost always exist. For example, if $v$ is the father of $w$, then $w$ is the son of $v$. In the rare cases in which there is a relationship between $v$ and $w$, but no relationship between $w$ and $v$, we define the edge type $t(w, v)$ to be `null`. Furthermore, we allow multiple edges between two vertices $v$ and $v'$, and so the notation for an edge varible $(v, v')$ allows for multiple instances of edges between $v$ and $v'$. It will not be important to name these instances in this paper.

We often refer to the quadruple of types associated with an undirected edge between two vertices $v$ and $w$. We use the shorthand notation $t[v, w]$ to denote the quadruple ($t(v)$, $t(v, w)$, $t(w, v)$, $t(w)$). Since the semantic graphs that motivated the MTGL may be multigraphs, and hence any pair of vertices $v$ and $v'$ may have many edges of different types between them, it is convenient for us to speak of walks in terms of edges rather than vertices. We define a walk of length $l$ to be a sequence of $l$ edges: $W = ((w_0, w_1), (w_1, w_2), \ldots, (w_{l-1}, w_l))$. We say that two walks $W$ and $W'$ are *type-isomorphic* if

$$t[w_i, w_{i+1}] = t[w'_i, w'_{i+1}]$$

| MTA primitive | meaning | notation | MTGL |
|---|---|---|---|
| b = int_fetch_add(a,i) | atomic read, then increment of $a$ | $b \overset{\text{ifa}}{\leftarrow} a, i$ | mt_incr |
| b = readfe(a) | wait for $a$ to be "full," read $a$ leave it "empty" | $b \overset{\text{fe}}{\leftarrow} a$ | mt_readfe |
| b = readff(a) | wait for $a$ to be "full," read $a$, leave it "full" | $b \overset{\text{ff}}{\leftarrow} a$ | mt_readfe |
| writeef(a,v) | wait for $a$ to be "empty," write $a$, leave it "full" | $a \overset{\text{ef}}{\leftarrow} v$ | mt_writeef |

**Table 7.1.** Some MTA primitives and their pseudocode and MTGL designations. The *int_fetch_add* intrinsic is an atomic read and increment instruction. In this example, $b$ gets the old value of $a$, then $a$ is incremented by $i$.

for all $0 \leq i \leq l - 1$.

When multiple threads access a piece of shared memory, the MTA's word-level concurrency mechanisms, listed in Table 7, are used by the MTGL infrastructure, and sometimes by user programs. When we need to specify a concurrent access in our pseudocode, we use the associated notation shown in Table 7.

In addition to the notation defined in Table 7, when we wish to specify that some high level series of operations, such as an insertion of element $e$ into a hash table $T$, is done in a thread safe manner, we use the notation $T \overset{\text{ts}}{\leftarrow} T \cup e$.

Visitor objects in MTGL algorithms have fields (*member data* in C++ lingo), and we use the standard C/C++ notation $I.f$ to denote field $f$ of object instance $I$. Visitor objects will also have associated *methods*, and these are defined using a generic pseudocode format.

We encapsulate MTGL logic that determines whether or not to parallelize a loop. The pseudocode

```
for (v,v') in E(v):
```

indicates that the MTGL will instruct the underlying machine to parallelize the loop if parallelization is supported and $E(v)$ is large enough. Otherwise, the loop will run in serial. One hundred iterations is the default threshold in the MTGL. This explanation holds unless there is a comment in the pseudocode indicating otherwise.

In the pseudocode below, we assume that all vertices and edges have id's. However, in our notation a vertex's name as its id ($v$, as opposed to $v$.id), while an edge's id is called out explicitly ($(v, v')$.id or $e.id$).

# Algorithmic Kernels of the MTGL

Using pseudocode and the notation defined above, we will now give descriptions of three algorithmic kernels of many graph queries that might be submitted to a semantic graph algorithm server. These kernels are *connected components*, *subgraph isomorphism*, and *s-t connectivity*. A connected component $C$ is defined as a maximal subset of the vertex set $V$ such that any $v$ and $w$ in $C$ are connected by a path. Finding connected components is an elementary problem in graph theory, and linear-time solutions exist. Efficient parallel algorithms exist as well [73].

```
PSearch<AND,Vis>(v)
{
   Vis.d(v)
   for (v,v') in E(v):
      if Vis.vt(v,v'):
         if (v,v') unvisited:
            Vis.te(v,v')
            PSearch<Vis>(v')
         else:
            Vis.oe(v,v')
}
```

**Figure 7.1.** Pseudocode for the PSearch routine, templated to treat the user's visit test as a logical "and."

```
PSearch<OR,Vis>(v)
{
   Vis.d(v)
   for (v,v') in E(v):
      if (v,v') unvisited OR Vis.vt(v,v'):
         Vis.te(v,v')
         PSearch<Vis>(v')
      else:
         Vis.oe(v,v')
}
```

**Figure 7.2.** Pseudocode for the PSearch routine, templated to treat the user's visit test as a logical "or."

Subgraph isomorphism, however, is an NP-complete problem, and hence computationally intractable barring an epochal theoretical development. Given a graph $G$ and a smaller graph $H$, is there a subgraph of $G$ isomorphic to $H$? A classical algorithm by Ullman [76] solves the subgraph isomorphism problem, but its computational complexity makes this algorithm

```
PSearch<REPLACE,Vis>(v)
{
   Vis.d(v)
   for (v,v') in E(v):
      if Vis.vt(v,v'):
         Vis.te(v,v')
         PSearch<Vis>(v')
      else:
         Vis.oe(v,v')
}
```

**Figure 7.3.** Pseudocode for the PSearch routine, templated to treat the user's visit test as the only criterion for proceeding.

unusable for large inputs. We will give a new heuristic for the subgraph isomorphism problem on semantic graphs that demonstrates the flexibility of the MTGL and scales almost perfectly on the MTA-2.

## Preliminaries

In Figures 7.1,7.2, and 7.3, we give pseudocode for a basic MTGL primitive: parallel graph search *PSearch*. We do not specify "depth-first" or "breadth-first" search since the primitive has elements of both. A single instance of *PSearch(v)* will initiate a single search from vertex $v$, and each time the neighbors of a vertex are explored, a decision is made whether to parallelize the loop of recursive PSearch'es from the neighbors of $v$. As no queue is used to enforce breadth-first visitation of vertices, PSearch reduces to depth-first search when MTGL code is run on a serial machine.

Following the visitor pattern, PSearch is an object, and it is customized by two template parameters. One of these is a a visitor object that will provide PSearch with five things:

1. User-defined fields, such a data structures to hold results,
2. A $sr(v)$ method, to be called upon the initial discovery of vertex $v$ as a search tree root (called once per psearch),
3. A $d(v)$ method, to be called upon the discovery of vertex $v$ during search,
4. A $vt(v, v')$ (visit-test) method, to be called before traversing edge $(v, v')$.
5. A $te(v, v')$ (tree-edge) method, to be called upon visiting edge $(v, v')$ to first discover $v'$.
6. An $oe(v, v')$ (other-edge) method, to be called upon visiting edge $(v, v')$ to revisit $v'$.

7. An optional *copy(vis)* method, to be called in order to copy visitor objects. This is used to create linked lists of visitors corresponding to nodes in the search tree. Such lists are constructed and used in the subgraph isomorphism heuristic introduced in Section7.

In particular, the visit-test (*vt*) method gives *psearch* significant flexibility. The MTGL programmer can use this method, for example, to specify forward, backward, or undirected searches, or to continue or halt searches based on customized criteria.

The other template parameter is an *operation type* that will tell the search primitive how to interpret the visitor's *vt* (visit-test) method. Acceptable operation types are:

- logical `OR`, which indicates that the search should proceed via edge $(v, v')$ if $v'$ is unvisited, *or* if the user's visit-test returns true;
- logical `AND`, which indicates that the search should terminate if the user's visit-test returns false, regardless of whether $v'$ has been visited;
- the symbol `REPLACE`, which indicates that whether or not $v'$ has been visited is irrelevant. The user's visit-test alone will determine whether to continue the search.

```
SearchHighLow<OP, Vis>(G)
{
      # high-degree vertices
       H ← {v_h1, v_h2, ..., v_hk}
      # low-degree vertices
       L ← {v_l1, v_l2, ..., v_h(n − k)}
      for v in  H:     # in serial
          PSearch<OP,Vis>(v)
      for v in  L:
          PSearch<OP,Vis>(v)
}
```

**Figure 7.4.** Pseudocode for the SearchHighLow routine . H and L are found in parallel on a multithreaded platform. Although the loop over H is in serial, each iteration launches a parallel PSearch.

For example, if the user wishes to search the subgraph induced by type "green" edges only, the `AND` operation would be used. Another example of an `AND` visitor is given in Section 7 below. If, on the other hand, the user wishes to take a random walk through the graph while disregarding repeat visits, the `REPLACE` operation would be used. An example of a meaningful use of the `OR` operation is given in Section 7.

The nested parallelism in the *psearch* pseudocode can be handled well by the MTA-2 if the proper compiler directives are used. The MTGL encapsulates the choice of these compiler directives, as well as several concurrency issues.

A common operation in multithreaded graph algorithms is to run a large number of PSearch instances concurrently in the same graph. In order to avoid repetition of this operation, we define and reuse a function that implements a heuristic variety of this operation due to Jace Mogill. Assuming that there are $k$ vertices of "high degree," where the latter can be defined by the MTGL programmer, initiate PSearches from those, *using a serial loop*. Attempting to initiate these searches in parallel overwhelms even the MTA with threads. After searching from the high-degree vertices, we initiate searches from all remaining vertices in parallel. Note that many of these searches will terminate immediately, as they encounter previously visited vertices. Mogill's heuristic, and Kahan's C implementation of it, recursively segregates high-degree neighbors from low-degree neighbors during the search. However, our MTGL implementation uses the simpler logic given in Figure 7.4.

## Kahan's Algorithm for Connected Components

Kahan's algorithm labels the connected components of $G$ in a three-phase process:

1. *SearchHighLow* is called to cover the graph with concurrent searches. The result is a partial labelling of connected components and a hash table containing pairs of components that must be merged into one.
2. A standard concurrent-read, concurrent-write parallel algorithm (Shiloach-Vishkin) [73], is used to find the connected components of the graph induced by the component pairs in the hash table.
3. A set of PSearches is initiated from each component leader identified by phase 2. Each PSearch labels all vertices in a single component.

The MTGL implementation of Kahan's algorithm illustrates the flexibility of the visitor pattern. In order to implement phase 1, we define a visitor object that will customize the SearchHighLow operation. The pseudocode is shown in Figure 7.5.

Phase 2 of Kahan's algorithm is a call to the Shiloach-Vishkin algorithm to find the connected components of the graph induced when we treat each pair in $T$ as an edge. We omit the MTGL pseudocode for this phase, and simply describe phase 2 with the following code:

$$L \leftarrow \text{ShiloachVishkin}(V_1.T),$$

where $L$ is the set of component leaders determined by the algorithm.

To implement phase 3, we define another visitor class to customize another call to a search primitive. This simpler visitor is shown in Figure 7.6.

Kahan's algorithm in its entirety is given in Figure 7.7.

```
V_1 ← {
   C ← array of |V(G)| ints
   T ← hash table of (int, int) pairs

   sr(v) { C[v] ← v }
   vt(v,v') { }
   te(v,v') { C[v'] ← C[v] }
   oe(v,v') { T ts← T ∪ { (C[v], C[v']) } }
}
```

**Figure 7.5.** The visitor object for Kahan's algorithm, phase 1 . The hash table insertion is made only if $C[v]$ is not equal to $C[v']$.

```
V_2 ← {
   C ← V_1.C

   d(v) { }
   vt(v,v') { }
   te(v,v') ← V_1.te(v,v')
   oe(v) { }
}
```

**Figure 7.6.** The visitor object for Kahan's algorithm, phase 3

## The bully algorithm for connected components

The running time of Kahan's algorithm is dominated by the construction of the hash table $T$ in phase 1. If we exploit multithreading and the MTGL, we can remove the hash table entirely. Rather than remembering which two concurrent searches encounter one another, we arbitrate between them. Only one of the searches is allowed to continue, and it overwrites the component numbers written by the other search. In this way, the algorithm completes in one phase without building a data structure. The continuing search is the "bully."

The bully algorithm requires only one visitor class. This is defined in Figure 7.8. The non-empty visit-test method enables the bully searches to continue even though their destination vertices were previously discovered. When a "bullying" operation is occurring, we use full-empty synchronization logic to ensure that the marking of vertices is correct.

The bully algorithm is less general than Kahan's three-phase algorithm since we expect no speedup in the pathological cases in which the entire graph a single chain or ladder. However, for the power-law semantic graphs that we explore in Section 7, the performance of the bully algorithm is good.

```
Kahan(G) {
   define V_1
   SearchHighLow<OR,V_1>(G)
   L ← ShiloachVishkin(V_1.T),
   define V_2
   for v in L:
     PSearch<OR,V_2>(v)
   return V_2.C
}
```

**Figure 7.7.** Kahan's algorithm for connected components

## Compound type filtering

The MTGL is designed to process semantic graphs, and our next example illustrates what we anticipate to become a common operation: filtering the edges of $G$ by the quadruples of types associated with a small set of edges $T_E$. We call this operation *compound type filterning*. Recall that for any $(v, v') \in T_E$, we have defined

$$t[v, v'] = (t(v), t(v, v'), t(v', v), t(v')).$$

Suppose that we wish to find in $G$ an isomorphic or nearly-isomorphic instance of a smaller graph. Some authors call the small graph a *pattern* graph and the large graph a *target* graph. However, we adopt the convention that both of these terms apply only to the small graph (and the large graph is simply "the graph").

Letting $T_E$ denote the set of edges in a target graph, we start by finding the size of the edge-induced subgraph $S$ of $G$ such that for every undirected edge $(v, w) \in S$, there exists an undirected edge $(v', w') \in T_E$ with $t[v, w] = t[v', w']$. If subgraph $S$ is found to have sufficiently few edges, we may extract $S$ and apply a subgraph isomorphism heuristic to it.

The MTGL pseudocode to identify the edges of $S$ is shown in Figure 7.9. This is our fourth example of a visitor class customizing the search primitives.

Note that the intuitive way of accomplishing this compound filtering operation would be simply to loop through an array of all of the edges in the large graph, checking the types of each one against each edge in the target graph. This is logically correct, but a very poor alternative in a multithreaded environment since, for example, all edges of a given vertex would be trying to retrieve its type at the same time. We use the search primitives to accomplish the logical operation of examining each edge and to mitigate the hot spots inherent in the naive approach.

Note also that the `for` loop in the *te(v,v')* method is written so that different threads will examine the edge set $T_E$ in different orders. This would become unnecessary if the

```
V_3 ← {
   C ←  array of |V(G)| ints

   sr(v) { C[v] ts← v}
   vt(v,v') {
      if (C[v] < C[v']):
         return true
      else:
         return false
   }
   te(v,v') {
      c fe← C[v']
      if ((v' unvisited) or (c > C[v])):
       C[v'] ef← C[v]
      else:
       C[v'] ef← c
   }
   oe(v) { }
}

Bully(G) {
     define V_3
     SearchHighLow<OR,V_3>(G)
     return V_3.C
}
```

**Figure 7.8.** The bully algorithm

programmer had the ability to allocate local memory. In the latter case, s/he would allocate one copy of the target graph for each processor.

As we will show in Section 7, the routine *CorrectlyTypedEdges* has memory reference properties that make it the best candidate of our graph kernels for near-perfect scaling as multithreaded machines increase in size.

## Subgraph isomorphism for semantic graphs

A fundamental problem in graph algorithms is topological pattern matching. The famous graph isomorphism problem still defies classification, though some heuristic solutions work very well in practice [58]. Furthermore, the problem of testing isomorphisms between a relatively small "target" graph and all equivalently-sized subgraphs of a larger graph, i.e., subgraph isomorphism, is known to be NP-complete. Early attempts at subgraph isomorphism heuristics included branch and bound processes that exploit matrix operations [76] and are not practical for large instances. There is more recent literature on heuristics, such as [67], [54], and others, but we haven't yet compared our heuristic with this work.

```
V_4 ← {
   T_E ← the k edges of a target graph
   s ← 0 # s used to store a vertex type
   M ← an empty bitmap of size |E(G)|

   #upon discovery, access t(v) only once
   d(v) { s ← t(v) }

   #called for each v' ∈ E(v); avoid t(v)
   te(v,v') {
     i ← (v, v').id
     for e in (i%k, (i+1)%k, ..., (i+k−1)%k):
       (w, w') ← T_E[e]
       if ((s, t(v, v'), t(v', v), t(v'))=t[w,w']):
          M[eid] = 1
   }
   oe(v,v') ← te(v, v')
}

CorrectlyTypedEdges(G, T_E) {
     define V_4
     SearchHighLow<OR,V_4>(G)
     return V_4.M
}
```

**Figure 7.9.** Compound type filtering. The % symbol denotes modular arithmetic.

We assume for this discussion that whenever edge $(v, w)$ exists, $(w, v)$ will exist as well. If $v$ is adjacent to $w$ via some type of relationship, then $w$ is adjacent to $v$ via the inverse of that relationship. In semantic graphs, vertex and edge types make the otherwise intractable subgraph isomorphism problem more approachable. A simple heuristic would start many concurrent searches at appropriately typed nodes, then employ branch & bound to explore the space of matching choices between the neighbors of a vertex in the large graph and those of its analogue in the small graph. We considered such an approach, but abandoned it in favor of the method we describe next.

In undirected semantic graphs, we are assured that there will be an *Euler tour* through the target graph. Such a tour traverses each edge exactly once, and ends up at its starting point. Euler tours exist in undirected semantic graphs as we have described them since each undirected edge is really a pair of directed edges, and a basic theorem states that Euler tours must exist if, for each vertex, the in-degree equals the out-degree.

Let us name our small, target graph $T_G$. Our subgraph isomorphism heuristic begins by finding an Euler tour through $T_G$, and constructing a sequence of edges $W$ (for "walk").

```
V_5 ← {
   B ← sparse collection of triples
   W ← a walk through the target graph
   i ← the current stage

   d(v) {}

   te(v,v') {
      if (i = 0 or ∃_v̄ B[i − 1, v̄, v] = 1) and
         (t[v, v'] = t[w_i, w_(i + 1)])
        B[i,v,v'] = 1
   }
   oe(v,v') ← te(v,v')
}

AdvanceOneStage<V_5>(i) {
   SearchHighLow<OR,V_5>(G)
   return V_5.B
}

FindBipartiteEdges(G, T_E, W) {
   B ← null
   define V_5
   for i = 0 to l(W):
      V_5.B ← AdvanceOneStage<V_5>(i)
   return V_5.B
}
```

**Figure 7.10.** A visitor class to help find the edges of $G_B$

Supposing that the walk traverses $l$ edges,

$$W = ((w_0, w_1), (w_1, w_2) \ldots, (w_{l-1}, w_l)).$$

We also denote the edge set $E(T_G)$ by $T_E$. Our heuristic will perform $l$ *SearchHighLow* operations on the large graph $G$ in order to construct a data structure encapsulating all possible subgraphs of $G$ that have a walk type-isomorphic to $W$. If there is an exact topological match, it will be among these possibilities. Furthermore, any metric for comparing closeness of matches could be used to inform a branch & bound search through all possibilities.

The data structure we construct is a bipartite graph $G_B$. The vertices of $G_B$ are arranged into rows $r_0, r_1, \ldots, r_l$, and all vertices in $r_i$ correspond to vertices in $G$ that are *active* after traversing the first $i-1$ edges of $W$. A vertex $v \in G$ is defined to be active at stage $i$ if the first $i-1$ edges of $W$ are type-isomorphic to at least one walk in $G$ that ends with $v$.

The edges of $G_B$ connect active vertices at stage $i$ with active vertices at stage $i+1$, thus documenting all ways that a given vertex can become active. Figure 7.10 shows MTGL pseudocode that finds the edges of $G_B$.

```
V_6 ← {
   lv ← levels of V(G_B)
   M ← map: V(G_B) → V(G)
   S ← an empty subgraph
   found ← 0, next ← null

   # visitor objects are copied during the
   # search; keep linked list of ancestors
   copy(V_6 parent) {
      next ← parent
   }

   d(v) {}

   vt(v,v') {
      if lv(v') = lv(v) + 1:
         return true
      else:
         return false
   }
   te(v,v') {
      if lv(v') == l:
         f ←ifa found, 1
         if f == 0
            # return the first match
            for (v̄, v̂) in (v, v'), ancestors:
              S ← S ∪ (M(v̄), M(v̂))
   }
}
```

**Figure 7.11.** Subgraph extraction visitor pseudocode. This code returns only the first match, but a full branch and bound search could be made, given a suitable metric.

## S-T Connectivity

Given a graph and two of its vertices, $s$ and $t$, a simple problem is to find a path of minimum length connecting $s$ to $t$. With unit-length edges, this path can be found via breadth-first search. This could be done by searching from $s$ until $t$ is encountered, but a more efficient approach is to search from both ends in phases. In one phase, we determine which of the two searches has discovered fewer vertices, then expand one level of that search.

When one search encounters a vertex discovered by the other search, a shortest s-t path has been found. This approach was used in the Gordon Bell-finalist paper [80] to explore s-t connectivity on BlueGene/Light. A distributed-memory code applicable only to Erdös-Renyí random graphs was run on an instance of order 4 billion vertices and 20 billion edges. The s-t search completed in about 1.5 seconds. In Section 7, we will discuss the performance

```
SubgraphIsomorphism(G, T_E, W) {
   B ← FindBipartiteEdges(G, T_E, W)
   V_B ← {(i, j) : ∃_k B[i, j, k] = 1}
   E_B ← {[(i, j), (i + 1, k)] :
                          (i, j), (i                    +
1, k) ∈ V_B ∧
                          B[i, j, k] = 1}

   lv((i, j) ∈ V_B) = i
   s ← (0, j) ∈ V_B : ∃_k B[0, j, k] = 1
   define V_6
   PSearch<AND,V_6>(s)
   return V_6.S
}
```

**Figure 7.12.** Subgraph isomorphism pseudocode

comparisons we were able to make.

# Experiments with MTGL Kernels

In order to evalute the performance of our MTGL graph kernels, we compiled an MTGL application with a power-law, semantic graph generator. The latter was written and tuned by Cray for benchmarking purposes.

In order to generate a graph, the programmer specifies $k$ *levels*, each of which determines the number of vertices that will have a certain degree. That is, level i specifies that $n_i$ vertices will share the tails of $m_i$ directed edges, where assignments are made randomly. The heads of the $m_i$ edges are selected at random from $V(G)$. Our MTGL wrapper for this graph generator has a parameter to generate the reciprocal edges in order to make the graph undirected.

## Data

For our experiments, the types of vertices and edges are selected randomly from $\{0, 1, \ldots, 99\}$, with the constraint that if an edge $(v, w)$ has type $k$, then its reciprocal $(w, v)$ will have type $99 - k$. The Cray graph generator allows multiple edges and self-loops, but these occur sparingly.

We experimented with graphs of sizes ranging from 3 million edges to 500 million edges. Our set of types, and the uniformly random distribution of these types may not reflect the reality of current social networks. However, it is plausible that some type ontologies would

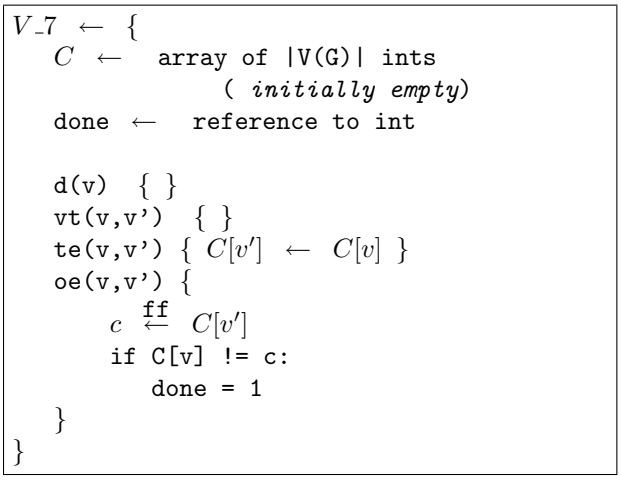



**Figure 7.13.** The visitor object shared by two breadth-first searches in the S-T connectivity algorithm

have sufficient robustness that no large majority of vertices or edges would have the same type.

For this paper, we report results on one graph only. Therefore, we do not claim this to be a thorough experimental study. Rather, this paper serves as a case study for the applicability of massive multithreading to unstructured graph problems, and as an introduction to the MTGL.

Our instance of concern is a power-law graph with 32 million vertices and 234 million edges. The degree distribution is approximately:

- $2^5$ vertices of degree $2^{20}$
- $2^{15}$ vertices of degree $2^{10}$
- $2^{25}$ vertices of degree 5

## Experimental Setup

We explored the performance of connected components and subgraph isomorphism MTGL kernels. The reason we limited ourselves to few graph instances is that our analyses of the results involved time-consuming efforts to profile and simulate each run in order to predict its performance on Eldorado. We will report in detail on this process in another paper. Here, we will present only MTA-2 performance results and abstract Eldorado predictions.

MTGL implementations of Kahan's and the bully algorithm for connected components were compared to Kahan's original C implementation of his algorithm on the MTA-2. The

```
BFS<OR,Vis>(v)  # examine v's out-edges
                # queue v's neighbors
{
   Vis.d(v)
   for (v,v') in E(v):
      if v' unvisited OR Vis.vt(v,v'):
         Vis.te(v,v')
         Q.push(v')
      else:
         Vis.oe(v,v')
}
```

**Figure 7.14.** Pseudocode for the BFS routine, which is a breadth-first analogue to *PSearch*. However, a call to BFS expands one level, as opposed to doing a complete search.

canonical representation for an adjacency list in the MTGL is a dynamic array. Kahan's C code, on the other hand, uses k-ary trees to represent these lists. That choice of data structure was imposed by other benchmarking pressures, and Kahan conjectures that his C version can be made to run roughly three times faster, given a dynamic array representation.

The prototype MTGL has no Euler tour routine at the moment. In order to implement our subgraph isomorphism heuristic in the face of this deficiency, we generated random walks through the target graph via another customizing visitor to the *PSearch* MTGL primitive. In general, the heuristic described in Section 7 can be given any walk. For example, many different Euler tours may be concatenated in order to increase the likelihood of an exact topological match. We approximated this input by taking long random walks. We report results for walks of length 120.

In order to generate our target graphs, we defined another visitor to customize *PSearch*. This one starts a single search and cuts it off when enough edges have been gathered. The power law nature of our large graph implies that the resulting target graphs were usually star graphs (see Figure 7.17). For our experiments, we generated target graphs of size 20.

To ground the absolute performance in terms of modern workstations, we also ran our experiments on a 3Ghz, 68GB linux workstation.

## Graph kernel performance

All of our experiments with the connected components and subgraph isomorphism heuristic demonstrate near-perfect scaling on the MTA-2. The single processor performance was in the same order of magnitude as that obtained on the 3Ghz workstation.

```
STConnectivity(G,s,t)
{
   define V_7
   bfs1 ← BFS<OR,V_7>(s)
   bfs2 ← BFS<OR,V_7>(t)
   while not V_7.done:
      if bfs1.nvisited < bfs2.nvisited:
         for v' in bfs1.topshell:
             bfs1(v')
      else
         for v' in bfs2.topshell:
             bfs2(v')
}
```

**Figure 7.15.** Pseudocode for the S-T connectivity. Two concurrent breadth-first searches converge, and each seach level of each search is explored in parallel. The *nvisited* variables store the number of nodes visited by each search, and the "topshell" notation indicates all vertices discovered by the previous call to the search.

## MTA-2 performance

Figure 7.16 shows the results of our MTA-2 performace test on the connected components algorithms. Without considering the issue of differing edge set representations, our MTGL implementation of Kahan's connected components algorithm is competitive with the original C implementation, scales almost perfectly, and achieves 70+% utilization of the MTA-2. The bully algorithm, with its lack of a requirement to build a type-safe hash table, is roughly twice as fast as the MTGL Kahan implementation, and achieves 95+% utilization of the MTA-2.

Perhaps most interesting are the performance results for the s-t connectivity kernel. The pseudocode in Figure 7.15 is imperfect since each breadth-first search relies on a global queue. The tail of this queue becomes a hot spot when the number of MTA-2 processors exceeds 10. This problem can be addressed via a distributed queue, but we have not yet implemented this fix. However, 10 MTA-2 processors is enough to bring the average running time for s-t connectivity on a 32 million vertex Erdös-Renyí graph with average degree 8 down to 0.09s. In this computation, roughly 23,000 vertices (combined) were visited by the $s$ and $t$ searches.

The 4 billion vertex Erdös-Renyí graph that was processed in 1.5 seconds using 32,000 processors of BlueGene/L in [80] had average degree 10. The expected shortest path length for this graph is between 9 and 10, so each breadth-first search will expand roughly 5 levels on average before the searches meet. After expanding shell $k$, each of the two searches will have discovered roughly $10^k$ vertices. Thus, about 200,000 vertices must be discovered in this large instance. This is fewer than ten times as many vertices as our 32 million vertex instance

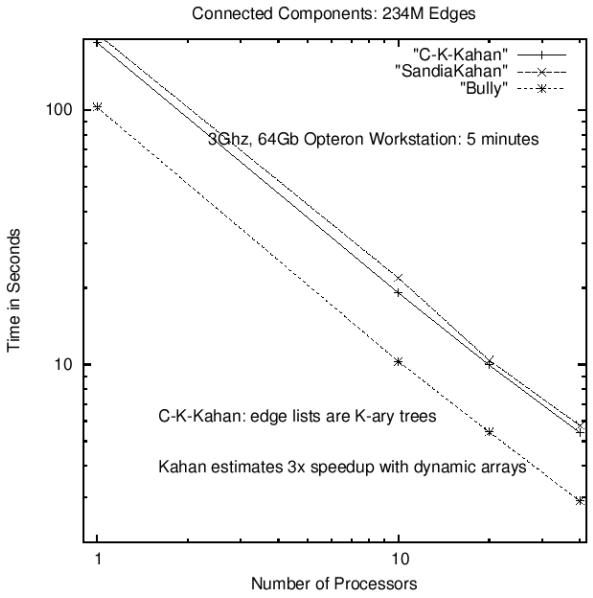



**Figure 7.16.** Connected components kernel performance

had to process. Thus, 10 MTA processors should be able to proceess 200,000 vertices in well under a second. So for this problem, a single digit number of MTA-2 processors is faster than a 32,000 BlueGene/L machine.

Exploring further, in Figure 7.19, we note the performance trends of 10 processor MTA runs of MTGL and C versions of the s-t connectivity algorithm corroborate our counting argument. The two lines in the figure show the scaling trajectories of the respective codes as graph size increases, holding average degree constant. The MTGL trajectory is slightly worse than the C implementation, but we have not yet explored the reason why.

An MTA-2 with enough memory to verify this performance prediction will never exist. However, Eldorado machines of sufficient size will. Eldorados will not scale as well as MTA-2's would have scaled, but as discussed below, we expect them to perform very well.

## Eldorado performance

The Cray Eldorado system is being developed as a follow-on to the MTA-2. It can be thought of as a larger MTA with faster processors and a slower network. In this paper, we intend only to give an idea of the expected performance of our codes on a 512 processor Eldorado. For a detailed architectural description of Eldorado, see [35], and for details of our performance predictions, see [77]. Working with Keith Underwood of Sandia National Laboratories, Megan Vance of Notre Dame, and Wayne Wong of Cray, Inc. we went through the following process for each graph kernel:

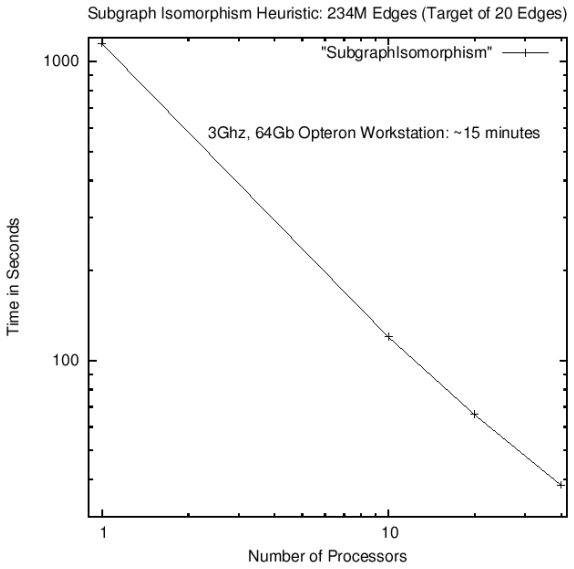



**Figure 7.17.** Subgraph isomorphism kernel performance

1. We used MTA hardware counters to find the memory reference rate of each kernel.

2. We used Cray's *zebra* MTA simulator to generate the actual memory address trace. This information was used to distinguish stack references from non-stack references. The former will be local references on Eldorado.

3. We simulated the memory system of Eldorado and predicted the hit rate in the DRAM buffer accounting for network traffic.

4. Using these numbers, we predicted the expected slow down in the graph kernels on a 512 processor Eldorado system.

The high-level results were that the expected slow down when scaling the connected components kernels to 512 processors is 2-3. Since Eldorado processors are more than twice as fast as MTA-2 processors, we thus expect our connected components kernels to run on a 512 processor Eldorado as if it were a 512 processor MTA-2. The results for subgraph isomorphism were even more optimistic since the memory reference pattern of the *CompoundTypeFilter* routine, which dominates the running time, is much less demanding of the network than that of the connected components kernels.

We also simulated the network to explore the implications of hot spots. We found these to be of much greater consequence on Eldorado than they are on the MTA-2. However, with the exception of the end-of-queue hotspot in our current breadth-first search implementation, our kernels do not exhibit hot spotting on the MTA-2.

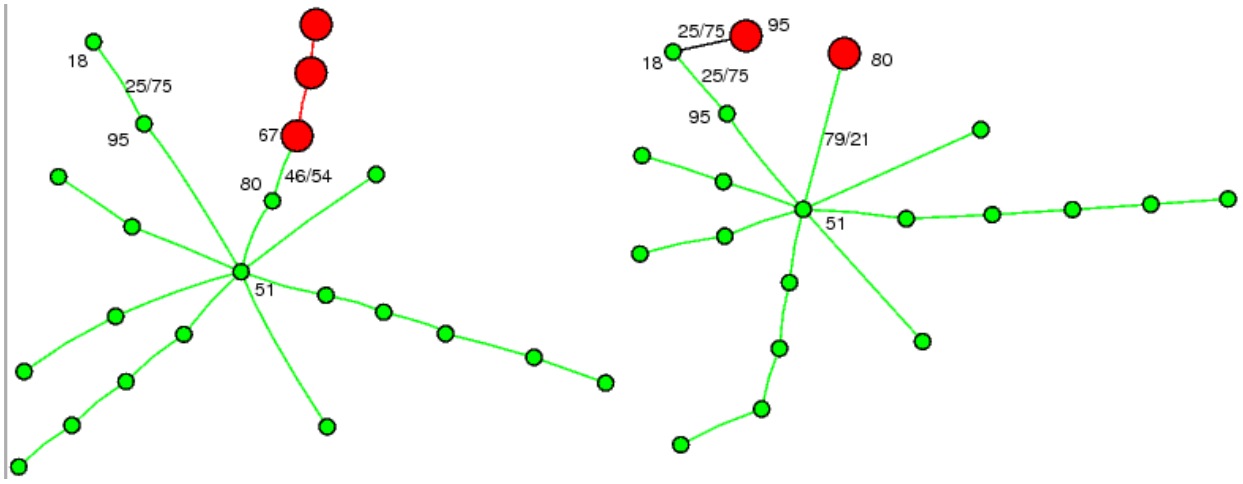



**Figure 7.18.** Subgraph isomorphism results. The target graph is on the left, and the subgraph found by the heuristic is on the right. Some vertex and edge types are shown for context. The large vertices represent places where the type-isomorphic walks did not produce topological isomorphism.

# Conclusions

Growing awareness of the applicability of massive multithreading to unstructured graph problems has encouraged a number of researchers to take an interest in the multithreaded machines. Our main contribution is a demonstration that this excellent performance can be preserved when programs are written using a generic software framework that abstracts away potentially troublesome details. A common criticism of shared memory programming, as opposed to message passing, is that correctness is more problematic. The shared-memory programmer has less explicit control and must better appreciate concurrency subtleties. Further, MTA programming is delicate since hot spots must be avoided. The prototype MTGL that we have introduced via pseudocode handles many of these correctness and concurrency issues for the application programmer.

We have also introduced two new multithreaded algorithms that leverage the flexibility of the MTGL: the bully algorithm for connected components and a heuristic for subgraph isomorphism on semantic graphs. We anticipate that as multithreaded programming matures, more algorithms will be developed that use similar techniques.

Our prototype MTGL is under active development, and we plan to release the software in an open-source form in the coming year. Current repository versions of the software are available by contacting `jberry@sandia.gov`.

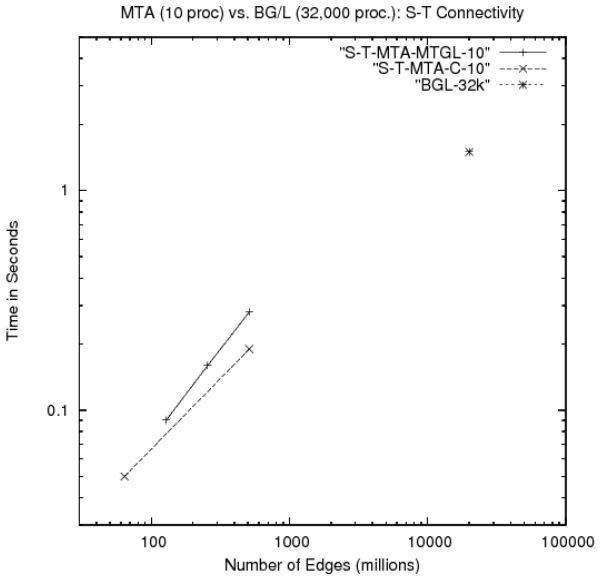



**Figure 7.19.** S-T connectivity comparison with BlueGene/L

# Chapter 8

# Implementing a Portable Multi-threaded Graph Library: the MTGL on Qthreads

The following describes our adaptation of the MTGL to work with Sandia's "qthreads" thread virtualization library. The result is a software framework that supports graph algorithms on multithreading machines ranging from the Cray XMT to the Sun Niagara to multi-core Opteron chips. The proper citation is [10].

## Introduction

Graph-based Informatics applications challenge traditional high-performance computing (HPC) environments due to their unstructured communications and poor load-balancing. As a result, such applications have typically been relegated to either poor efficiency or specialized platforms, such as the Cray MTA/XMT series. The multi-threaded nature of the Cray MTA architecture[1] presents an ideal platform for graph-based informatics applications. As commodity processors adopt features to enable greater levels of multi-threaded programming and higher memory densities, the ability to run these multi-threaded algorithms on less expensive, more available hardware becomes attractive.

The Cray MTA architecture provides both an auto-threading compiler and a number of architectural features to assist the programmer in developing multi-threaded applications. Unfortunately, commodity processors have increased the amount of concurrency available by adding an ever-growing number of processor cores on a single socket, but have not added the fine-grained synchronization available on the Cray MTA architecture. Further, while auto-threading compilers are being discussed, none provide the feature set of the Cray offerings.

Although massively multi-threaded architectures have shown tremendous potential for graph algorithms, development poses unique challenges. Algorithms typically use light-weight synchronization primitives (Full/Empty bits, discussed in Section 8) for synchroniza-

[1]Throughout this paper, we will use the phrase MTA architecture to refer to Cray's multi-threaded architecture, including both the Cray MTA-2 and Cray XMT platforms.

tion. Parallelism is not expressed explicitly, but instead compiler hints and careful code construction allow the compiler to parallelize a given code. Unlike the Parallel Boost Graph Library (PBGL) [42], which runs on the developers laptop as well as the largest supercomputers, applications developed for the MTA architecture only run on the MTA architecture. Experiments with the programming paradigm require access to the platform, which is obviously a constrained resource.

In this paper, we explore the possibility of using the Qthreads user-level threading library to increase the portability of scalable multi-threaded algorithms. The Multi-Threaded Graph Library (MTGL) [15], which provides generic programming access to the XMT, is our testbed for this work. We show the use of important algorithms from the MTGL on on emerging commodity multi-core and multi-threaded platforms, with only minor changes to the code base. Although performance is not at the same level as the same algorithm on a Cray XMT, the performance motivates our technique as a workable solution for developing multi-threaded codes for a variety of architectures.

# Background

Recent work [46, 53] has described the challenges in HPC graph processing. These challenges are fundamentally related to locality (both spatial and temporal), and the lack thereof when graph algorithms are applied to highly unstructured datasets. The PBGL [42] attempts to meet these challenges through storage techniques that reduce communication. These techniques have been shown to work well in certain contexts, though they introduce other challenges such as memory scalability. Even when they achieve run-time scalability, the processor utilization on commodity CPUs is considerably lower than that found in the MTA architecture.

## Cray XMT

The Cray XMT is the successor to the Cray MTA-2 highly multi-threaded architecture. Unlike the MTA-2, in which all memory was equidistant from any processor on the network, the XMT uses a more traditional model in which memory is closer to a single processor than all others. The Cray XMT utilizes similar processors to the MTA-2, including the ability to sustain 128 simultaneous hardware threads, but with an improved 500 MHz clock rate. Rather than the custom network found on the MTA-2, the XMT utilizes the SeaStar based network found on the Cray XT massively parallel processor distributed memory platform.

### Multi-core Architectures

As processor vendors have begun offering quad-core processors, as well as more commodity multi-threaded processors such as the Sun Niagara processors, it has become possible to write multi-threaded applications on more traditional platforms. Given the high cost of even a small XMT platform, the ability of modern workstations to support a growing number of threads makes them attractive for algorithm development and experimentation.

The Sun Niagara platform opens even greater multi-threaded opportunities, supporting 8 threads per core and 8 cores per socket, for a total of 64 threads per socket. Current generation Niagara processors support single, dual, and quad socket installations. Unlike the Cray XMT, the Sun Niagara uses a more traditional memory system, including L1 and shared L2 cache structures, and an unhashed memory system. The machines are also capable of running unmodified UltraSPARC executables.

### Multi-threaded Programming

Our approach is to take algorithm codes that have already been carefully designed to perform on the MTA architecture, and run them without altering the core algorithm on commodity multi-core machines by simulating the threading hardware. In contrast, codes written using frameworks specifically designed for multi-core commodity machines (e.g. SWARM [61]) won't run on the MTA architecture.

Standard multi-core software designs, such as Intel's Thread Building Blocks [49], OpenMP [27], and Cilk [16], target current multicore systems, and their architecture reflects this. For example, they lack a means of associating threads with a locale. This becomes a significant issue as machines get larger and memory access becomes more non-uniform.

Another important consideration is the granularity and overhead of synchronization. Existing large scale multithreaded hardware, such as the XMT, implement full/empty bits. This provides for blocking synchronization in a locality-efficient way. Existing multi-threaded software systems tend to use lock-based techniques, such as mutexes and spinlocks, or require tight control over memory layout. These methods are logically equivalent, but are not as efficient to implement. FEB's are memory efficient when implemented in hardware, and thus allow tight memory structures that can be safely operated upon without requiring locking structures to be inserted into them.

## Qthreads

The Qthread API [79] is a library-based API for accessing lightweight threading and synchronization primitives similar to those provided on the MTA architecture. The API was designed to support large-scale lightweight threading and synchronization in a cross-platform

library that can be readily implemented on both conventional and massively parallel architectures. On architectures where there is no hardware support for the features it provides, or where native threads are heavyweight, these features are emulated. There are several existing threading models that support lightweight threading and lightweight synchronization, but none that sufficiently closely emulate the MTA architecture semantics.

Equivalents for basic thread control, FEB-based read and write functions, as well as basic threaded loops (analogs for many of the pragma-defined compiler loop optimizations available on the MTA architecture) are all provided by the API. Even though the operations that do not have hardware support, such as FEB-based operations, are emulated, they retain usefulness as a means of intra-thread communication.

The API establishes convenient management of the basic memory requirements of threads as they are created. When insufficient resources are available, either thread creation fails or it waits for the resources to become available, depending on how the API is used.

Relatively speaking, locality of reference is not an important consideration to the MTA architecture, as the address space is hashed and divided among all processors at word boundaries. This is an unusual environment, and locality is an important consideration in most other large parallel machines. To address this, the Qthread API provides a generalized notion of locality, called a "shepherd", which identifies the location of a thread. A machine may be described to the library as a set of shepherds, which can refer to memory boundaries, CPUs, nodes, or whatever is a useful division. Threads are assigned to specific shepherds when they are created.

## Implementation of MTA Intrinsics

The MTA architecture has several features that are intrinsic to the architecture, which the Qthread library emulates. These features include full/empty bits (FEBs), fast atomic increments, and conditionally created threads.

On the MTA architecture, a full/empty bit (FEB) is an extra hardware flag associated with every word in memory, marking that word either full or empty. Qthreads uses a centralized collection data structure to achieve the same effect: if an address is present in the collection, it is considered "empty", and if not, it is considered "full". Thus, all memory addresses are considered full until they are operated upon by one of the commands that will alter the memory word's contents and full/empty status. The synchronization protecting each word is pushed into the centralized data structure. Not all of the semantics of the MTA architecture can be fully emulated, however. For example, on the MTA architecture, all writes to memory implicitly mark the corresponding memory words as full. However, when pieces of memory are being used for synchronization purposes, even implicit operations are done purposefully by the programmer, and replacing implicit writes with explicit calls is trivial.

The MTA architecture also provides a hardware atomic increment intrinsic. Atomic

increment functions have often been considered useful, even on commodity architectures, and so hardware-based techniques for doing atomic increments are common. The Qthread API provides an atomic increment function that uses a hardware-based implementation on supported architectures, but which falls back to using emulated locks to achieve the same behavior on architectures without explicit hardware support in the library. This is an example of opportunistically using hardware features while providing a standardized interface; a key feature of the Qthread API.

### Qthreads implementation of thread virtualization

Conditionally created threads are called "futures" in MTA architecture terminology, and are used to indicate that threads need not be created now, but merely whenever there are resources available for them. This can be crucial on the MTA, as each processor can handle at most 128 threads, and extremely parallel algorithms may generate significantly more. The Qthread API provides an analogous feature by providing alternate thread creation semantics that allow the programmer to specify the permissible number of threads that may exist concurrently, and which will stall thread creation until the number of threads is less than the number of permissible threads.

A key application of this is in loops. While a given loop may have a large number of entirely independent iterations, it is typically unwise to spawn all of the iterations as threads, because each thread has a context and eventually the machine will run out of memory to hold all the thread contexts. Limiting the number of concurrently extant threads limits the amount of overhead that will be used by the threads. In a loop, the option to stall the thread creation while the maximum number of threads still exist provides the ability to specify a threaded loop without the risk of using an excessive amount memory for thread contexts. The limit on the number of threads is a per-shepherd limit, which helps with load balancing.

## The Multi-Threaded Graph Library

The Multi-Threaded Graph Library is a graph library designed in the spirit of the Boost Graph Library (BGL) and Parallel Boost Graph Library. The library utilizes the generic component features of the C++ language to allow flexibility in graph structures, without changes to a given algorithm. Unlike the distributed memory, message passing based PBGL, the MTGL was designed specifically for the shared-memory multi-threaded MTA architecture. The MTGL includes a number of common graph algorithms, including the breadth-first search, connected components, and PageRank algorithms discussed in this paper.

To facilitate writing new algorithms, the MTGL provides a small number of basic intrinsics upon which graph algorithms can be implemented. The intrinsics hide much of the complexity of multi-threaded race conditions and load-balancing from algorithm developers and users. Parallel Search (PSearch), a recursive parallel variant of depth-first search (which

is not truly depth-first in order to achieve parallelism), combined with an extensive vertex and edge visitor interface, provides powerful parallelism for a number of algorithms.

MTA architecture-specific features used by the MTGL are either compiler hints specified via the `#pragma` mechanism or are encapsulated into a limited number of templated functions, which are easily re-implementable for a new architecture. An example is the `mt_readfe` call, which translates to `readfe` on the MTA architecture, a simple read for serial builds on commodity architectures, and `qthread_readfe` on commodity architectures using the Qthreads library.

The combination of an internal interface for explicit parallelism and the set of core intrinsics upon which much of the MTGL is based provides an ideal platform for extension to new platforms. While auto-threading compilers like those found on the MTA architecture are not available for other platforms, the small number of intrinsics can be hand-parallelized with a reasonable amount of effort.

# Qthreads and the MTGL

Making the MTGL into a cross-platform library required overcoming significant development challenges. The MTA architecture programming environment has a large number of intrinsic semantics, and its cacheless hashed memory architecture has unusual performance characteristics. The MTA compiler also recognizes common programming patterns, such as reductions, and optimizes them transparently. For these reasons, the MTA developer is encouraged to develop "close to the compiler".

The size of stack necessary, for example, presents a challenge. Some MTGL routines are highly recursive, and the MTA transparently handles expanding the stack for each thread as-needed. The Qthread library, however, has a fixed stack size. Iterative solutions, combined with using larger stacks was required to address the issue.

Both the MTA architecture and commodity processors are susceptible to the problem of hot spotting, performance degradation due to repeated access to the same memory location. The MTA architecture suffers from both read and write hot spotting, due to constraints in traffic across the platform's network. Commodity processors, however, provide cache structures to improve performance and benefit from read hot spotting. Commodity architectures also have a larger granularity of memory sharing: a cache line, which can be as large as 64 bytes. Concurrent writes within a cache line create a hot spot, even if the writes affect independent addresses. The cache was a consideration for atomic operations as well, as they typically cause a cache flush to memory. Avoiding atomic operations where possible, such as in reductions, is important for performance.

# Multi-platform Graph Algorithms

We consider three graph kernel algorithms: a search, a component finding algorithm, and an algebraic algorithm. There are myriad other graph algorithms, but we use these three as primitive representatives on which other algorithms can be built.

## BFS

Breadth-first search (BFS) is, perhaps, the most fundamental of graph algorithms. Given a vertex $v$, find the neighbors of $v$, then the neighbors of those neighbors, etc. Furthermore BFS is well-suited for parallelization. Pseudocode for BFS from [24] is included in Figure 8.1.

```
BFS(G,s)
1  for each vertex u ∈ V[G] − {s}
2     do color[u] ← WHITE
3        d[u] ← inf
4  color[s] ← GRAY
5  d[s] ← 0
6  Q ← ∅
7  while Q ≠ 0
8     do u ← DEQUEUE(Q)
9        for each vertex v ∈ Adj[u]
10         do if color[v] ← WHITE
11              then color[v] ← GRAY
12                   d[v] ← d[u] + 1
13                   ENQUEUE(Q, v)
14       color[v] ← BLACK
```

**Figure 8.1.** The basic BFS algorithm

There are two inherent problems with using this basic algorithm in a multithreaded environment. The first is that a parallel version of the *for* loop beginning on Line 9 will make many synchronized writes to the *color* array. This is a problem on machines like the Niagara regardless of the data characteristics. It is also a problem on the XMT if there is a vertex $v$ of high in-degree (since many vertices $u$ would test $v$'s color simultaneously, making it a hot spot).

The second problem is even more basic: the ENQUEUE operation of Line 13 typically involves incrementing a tail pointer. As all threads will increment this same location, it is an obvious hot spot.

We avoid these problems by chunking and sorting: suppose that the next BFS level contains $k$ vertices, whose adjacency lists have combined length $l$. We divide the work of processing these adjacencies into $\lceil l/C \rceil$ chunks, each of size $C$ (except for the last one). Then $\lceil l/C \rceil$ threads process the chunks individually, saving newly discovered vertices to local stores. Each thread can then increment the $Q$ tail pointer only once, mitigating that hot spot. However, in order to handle the *color* hot spot, we do not write the local stores directly into the $Q$. Rather, we concatenate them into a buffer, sort that buffer with a thread-safe sorting routine (qsort in Qthreads, or a counting sort on the XMT), then have a single thread put the unique elements of this array into the $Q$. This thread does linear work in serial, but the "hot spot" is now used to advantage in cache-based multicore architectures.

A better BFS algorithm is known for the XMT. Although we currently do not have an implementation of this algorithm, it would be a straightforward exercise to incorporate it into the MTGL so that the same program could run efficiently on either type of platform.

## Connected Components

A connected component of a graph $G$ is a set $S$ of vertices with the property that any pair of vertices $u, v \in S$ are connected by a path. Finding connected components is a prerequisite for dividing many graph problems into smaller parts. The canonical algorithm for finding connected components in parallel is the Shiloach-Vishkin algorithm (SV) [73], and the MTGL has an implementation of this algorithm that roughly follows [6].

Unfortunately, a key property of many real-world datasets will limit the performance of SV in practice. Specifically, it is known both theoretically [32] (for random graphs), and in practice (for interaction networks such, the World-Wide Web, and many social networks) that the majority of the vertices tend to be grouped into one "giant component" (GCC). Algorithms like SV work by assigning a representative to each vertex. Toward the end of these algorithms, all vertices in the GCC are pointing at the same representative, making it a severe hot spot.

We adopt a simple alternative to SV, which we call GCC-SV. It is overwhelmingly likely (though we do not not provide any formal analysis here) that the vertex of highest degree is in the GCC. Given this assumption, we BFS from that vertex using the method of Section 8 (or psearch on the XMT), then collect all *orphaned edges* that do not link vertices discovered during this search. Running SV on the subgraph induced by the orphaned edges we find the remaining components. This subproblem is likely to be small enough so that even if the largest component of the induced subgraph is a GCC of that graph (which is likely), the running time is dwarfed by that of the original BFS. If there is no GCC in the original graph, then the original SV would perform well.

```
#pragma mta assert nodep
for (int i=0; i<n; i++) {
    double total=0.0;
    int begin = g[i];
    int end = g[i+1];
    for (int j=begin; j<end; j++) {
        int src = rev_end_points[j];
        double r = rinfo[src].rank;
        double incr = (r/rinfo[src].degree);
        total += incr;
    }
    rinfo[i].acc = total;
}
```

**Figure 8.2.** The MTGL code for PageRank's inner loop on the XMT

## PageRank

PageRank, the algorithm made famous by Google for ranking web pages [68], is a linear algebraic technique for modeling the propagation of *votes* through a directed graph, where each page contributes a fraction of its vote to each of its out-neighbors. Ranks continue propagating until convergence. A thorough mathematical explanation of PageRank is beyond the scope of this paper. However, at an abstract level PageRank is a sequence of matrix-vector multiplications, each followed by a normalization step. In graph terms, the most computationally expensive portion of the algorithm is simply traversing all of the adjacencies in the graph in order to accumulate votes.

Figure 8.2 shows the vote accumulation loops of PageRank used by the MTGL on the XMT. The structure of these loops enables the XMT compiler to merge them into one, and to remove the reduction of votes into the variable `total` from the final line of the inner loop. The result is excellent performance. We simulate this in a Qthread-enabled version of this code in the MTGL in order to achieve good scaling on multi-core machines.

## R-MAT graphs

R-MAT [19] is a parameterized generator of graphs that can mimic real-world datasets. The term stands for "Recursive-MATrix," derived from the generation procedure, which is a simulation of repeated *Kronecker products* [52] of the adjacency matrix by itself. Intuitively, the R-MAT procedure can be thought of as repeatedly dropping marbles through a series of plastic trays. The topmost one typically is divided into 4 quadrants, the second one into 16, etc. The bottom tray is the adjacency matrix. At each level, a marble will pass through

one of 4 holes with probability given by 4 input parameters; $a, b, c, d$. Multiple edges are not allowed, so if a marble ends up on top of another marble in the adjacency matrix, it is discarded and we try again.

Varying the parameters $a, b, c, d$ determines much about the structure of the resulting graph. For example, using $a = 0.25, b = 0.25, c = 0.25, d = 0.25$ would generate an Erdös-Rényi random graph. Putting more weight on one of the quadrants tends to generate an inverse power-law degree distribution, which is found in many real datasets.

In our experiments we generate two different classes of R-MAT graphs:

- *nice* graphs have $a = 0.45, b = 0.15, c = 0.15, d = 0.25$. These graphs feature two natural communities at each of many levels of recursion (quadrants $a$ and $d$). However, even in graphs a quarter of a billion edges, the maximum vertex degree is only roughly a thousand.
- *nasty* graphs have $a = 0.57, b = 0.19, c = 0.19, d = 0.05$. These feature a much steeper degree distribution, with a maxmimum degree of roughly 200,000 in our quarter-billion edge example. Load balancing would naturally be more challenging in this case.

Furthermore, we label our graphs with the exponent of the number of vertices and hold the average degree at a constant 16, since this is relatively close to (though an over-estimate of) the average degree of a page in the WWW. For example, graph "R-MAT 21 Nasty" has $2^{21}$ vertices, $2^{24}$ undirected edges, and R-MAT parameters as given above.

# Multiplatform Experiments

We compare performance of the three graph kernel algorithms described in Section 8—breadth-first search, connected components, and PageRank—on three platforms capable of executing multiple threads simultaneously: the Cray XMT, the Sun Niagara T2, and a traditional multi-socket, multi-core platform.

The Cray XMT used in testing contains 64 500 MHz ThreadStorm processors, each capable of sustaining 128 simultaneous hardware threads and 500 GB of shared memory. The SeaStar based network is a 3-d torus in a $8x4x2$ configuration. The system was running version 6.2.1 of the XMT operating system.

A Sun SPARC Enterprise T5240 server, with two 1.2 GHz UltraSPARC T2 processors, each capable of sustaining 64 simultaneous hardware threads, was also used in testing. The system contains 128 GB of memory and was running Sun Solaris 10, 5/08 Release. The Sun CoolThreads version of GCC was used to compile all tests.

Finally, a quad-socket, quad-core Opteron system, clocked at 2.2 GHz, provides a traditional multi-core environment. The system provides 32 GB of memory and is running Red

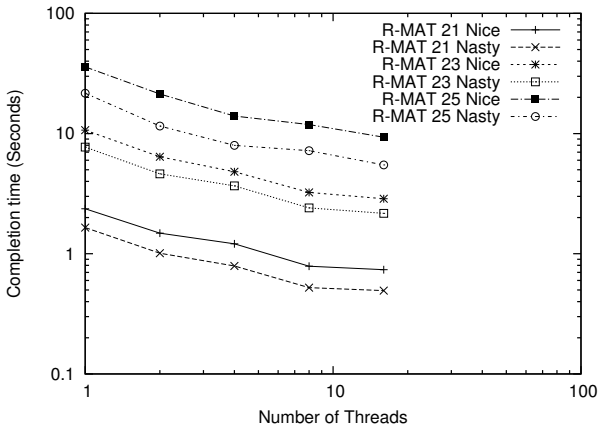



**Figure 8.3.** Opteron Breadth-First Search

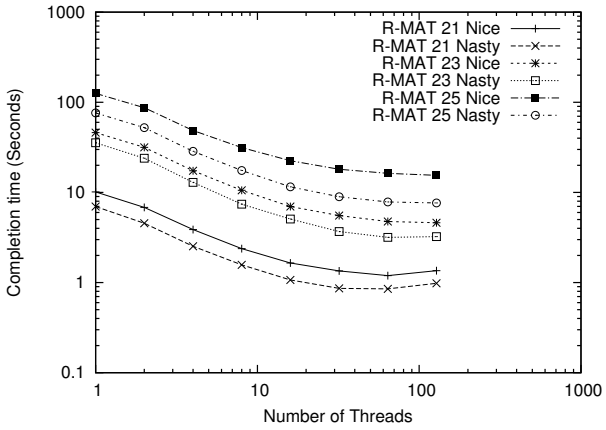



**Figure 8.4.** Niagara T2 Breadth-First Search

Hat EL 5.1. GCC 4.1.2 was used to compile all tests.

## Breadth-First Search

We find that our method of avoiding hot spots in BFS enables scaling beyond what would be achievable by a naive algorithm. At the time of this writing, our implementation runs on the XMT, but does not perform as well as native XMT BFS implementations have done in the past. However, our method does leverage the multi-core and Niagara platforms effectively. As implied before, MTGL programmers will run BFS by associating a visitor object with the kernel algorithm, then running the latter. Underlying differences in the kernel implementation, such as that likely in the XMT implementation of BFS, will be

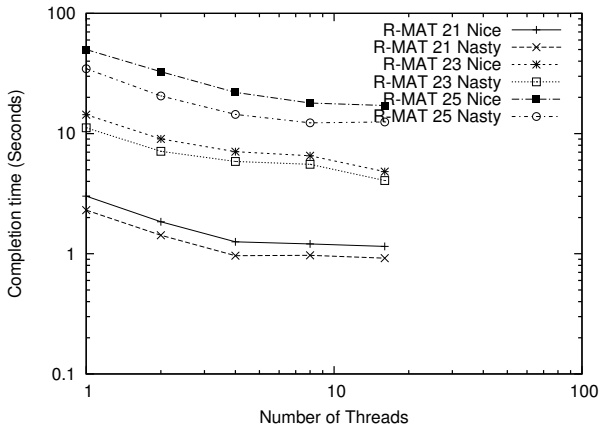



**Figure 8.5.** Opteron Connected Components GCC-SV

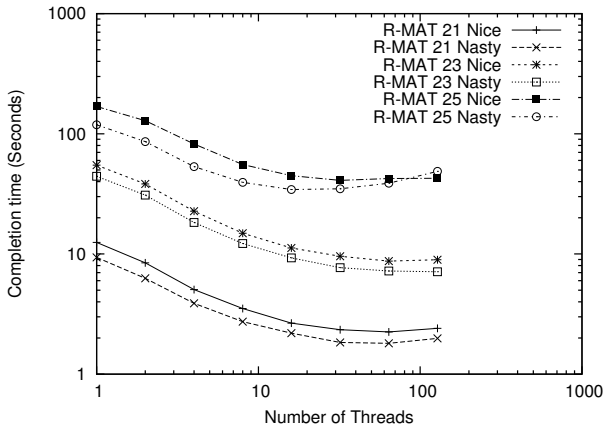



**Figure 8.6.** Niagara T2 Connected Components GCC-SV

hidden from the programmer.

## Connected Components

Our connected components codes demonstrate strong scaling on multi-core and Niagara, as the GCC-SV algorithm is dominated by a single run of BFS on the realistic datasets we address. Furthermore, we are able to demonstrate strong scaling on the XMT as well by replacing the BFS by the recursive psearch. Note the effect of data on algorithm performance in Figure 8.7. Ironically, the "nasty" datasets are most friendly to the algorithm, as the vast majority of all vertices fall into the GCC in this case. As we consider the "nice" datasets, this GCC membership becomes less pathological (and less realistic). Therefore, the inherently

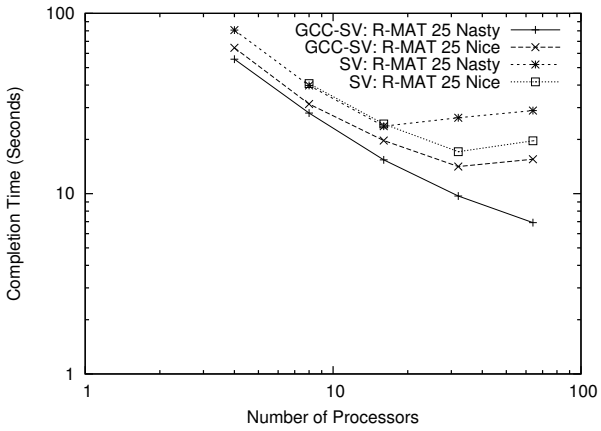



**Figure 8.7.** Cray XMT Connected Components - GCC-SV and SV

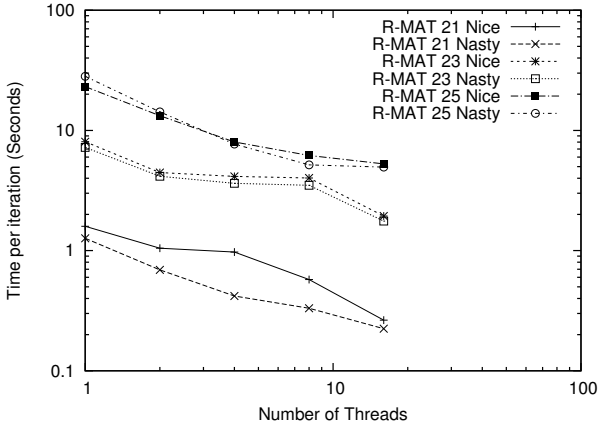



**Figure 8.8.** Opteron PageRank

hot spotting SV algorithm has more work to do once the GCC has been processed.

## PageRank

As we saw in Figure 8.2, PageRank can be written to leverage the auto-parallelizing compiler of the XMT quite effectively. We cannot match the XMT's performance in emulation without work to reconstruct the compiler's optimization. However, a straightforward parallelization of the outer loop using qthreads still provides significant benefit, as we see in Figures 8.8 and 8.9.

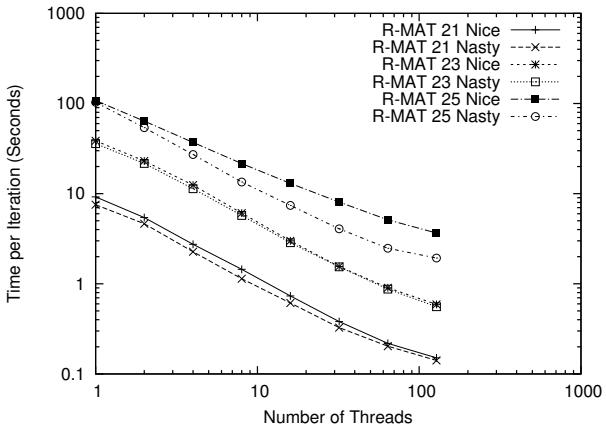



**Figure 8.9.** Niagara T2 PageRank

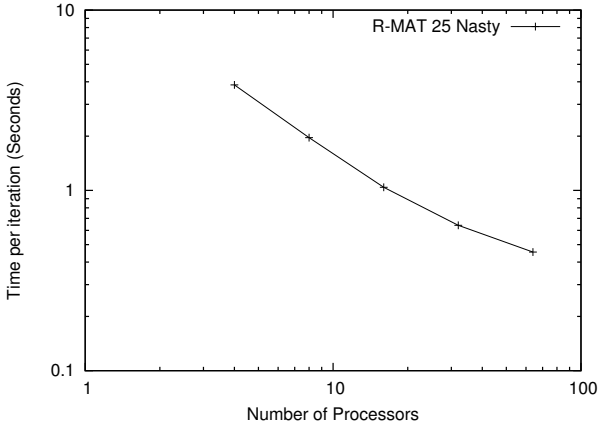



**Figure 8.10.** Cray XMT PageRank

# Conclusions and future work

Developing multi-threaded graph algorithms, even when using the MTGL infrastructure, provides a number of challenges, including discovering appropriate levels of parallelism, preventing memory hot spotting, and eliminating accidental synchronization. In this paper, we have demonstrated that using the combination of Qthreads and MTGL with commodity processors enables the development and testing of algorithms without the expense and complexity of a Cray XMT. While achievable performance is lower for both the Opteron and Niagara platform, performance issues are similar.

While we believe it is possible to port Qthreads to the Cray XMT, this work is still on-going. Therefore, porting work still must be done to move algorithm implementations

between commodity processors and the XMT. Although it is likely that the Qthreads-version of an algorithm will not be as optimized as a natively implemented version of the algorithm, such a performance impact may be an acceptable trade-off for ease of implementation.

# Chapter 9

# Advanced Shortest Paths Algorithms on a Massively-Multithreaded Architecture

In the following paper we demonstrate the Cray XMT's ability to allow many queries to share a pre-computed data structure to accelerate the associated computations. We describe Thorup's algorithm for single-source shortest paths, a complicated procedure with a good worst-case running time for integer-weighted problems. Much of the work of the paper predated this LDRD, but LDRD resources were used in the writing and experimental tuning. The proper citation is [26].

# 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*($w$,$i$).

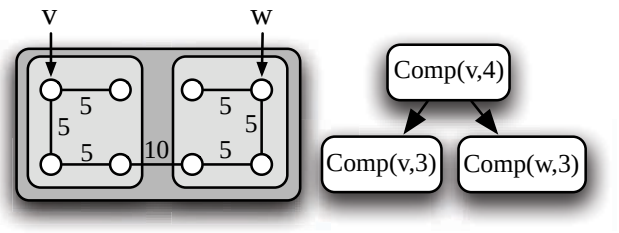



**Figure 1. An example component hierarchy. *Component(v,4)*, the root of this hierarchy, represents the entire graph.**

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 $\ell : E \rightarrow \mathbb{Z}^+$, 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 $\Delta$. 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).*

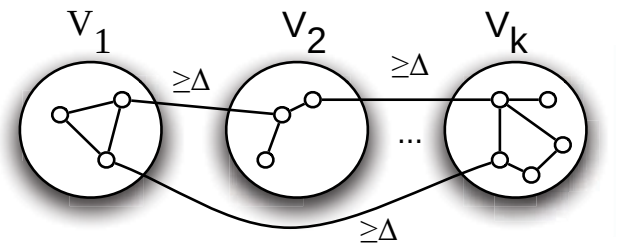



**Figure 2. The vertex set $V$ divided into $k$ subsets.**

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 $\texttt{min}\{d(x)|x \in V_i \backslash S\} \gg \alpha$. Note that $(\texttt{min}\{d(x)|x \in V_i \backslash S\} \gg \alpha) \leq (\texttt{min}\{d(x)|x \in V \backslash S\} \gg \alpha)$ implies that $(\texttt{min}\{d(x)|x \in V_i \backslash S\}) \leq (\texttt{min}\{d(x)|x \in V \backslash S\} + \Delta)$. Consider bucket $B[j]$ such that $j$ is the smallest index of a non-empty bucket. If $V_i \in B[j]$ then $\texttt{min}\{d(x)|x \in V_i \backslash S\} \gg \alpha = \texttt{min}\{d(x)|x \in V \backslash S\} \gg \alpha$. This implies that $\texttt{min}\{d(x)|x \in V_i \backslash S\} \leq \texttt{min}\{d(x)|x \in V \backslash S\} + \Delta$. 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 $\texttt{min}\{d(x)|x \in V_j \backslash S\}$. 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*$(c) = \texttt{min}(d(x)|x \in c \backslash S)$. 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 Multi-Threaded 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 $\texttt{min}(d(x)|x \in c \backslash S)$. 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 more efficient strategy for building the *toVisit* set.

```
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.**

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. 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.

| 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.**

## 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.

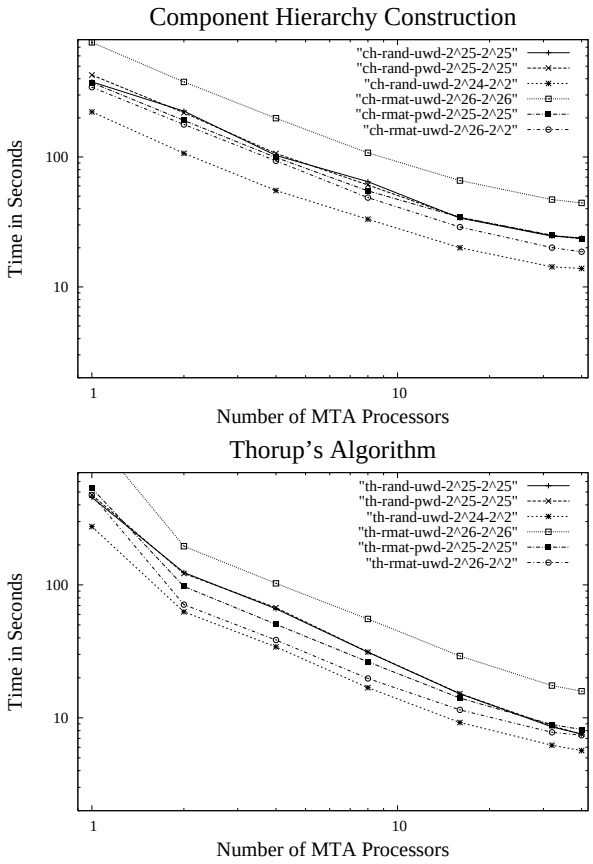



**Figure 4. Scaling of Thorup's algorithm on the MTA-2.**

## 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 $\Delta$-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 | $\Delta$-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.

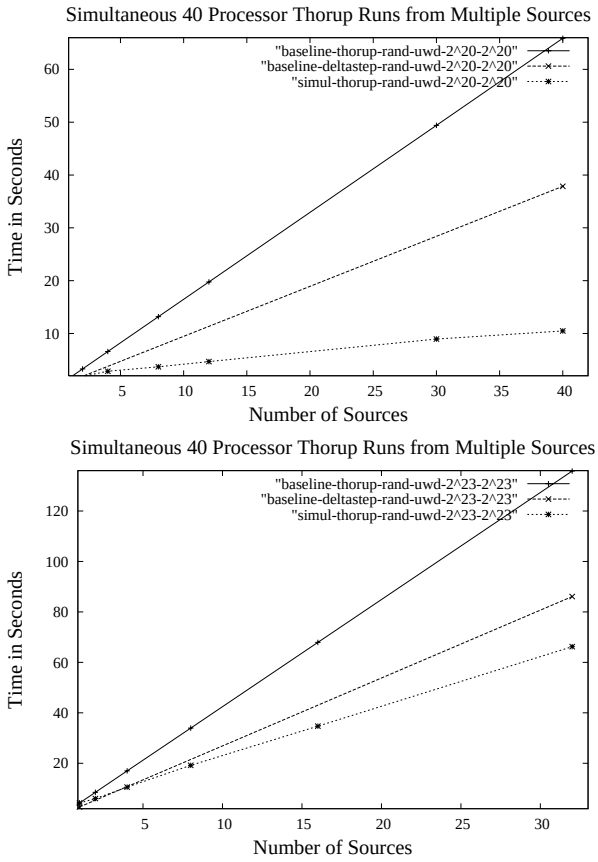



**Figure 5. Simultaneous Thorup SSSP runs from multiple sources using a shared CH.**

## 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.

# References


[1] *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*. IEEE, 2007.

[2] G. Agarwal and D. Kempe. arXiv:0710.2533v1, 2007.

[3] T.W. Anderson and D.A. Darling. Asymptotic theory of certain goodness-of-fit criteria based on stochastic processes. *Annals of Mathematical Statistics*, 23:193–212, 1952.

[4] W. Anderson, P. Briggs, C. S. Hellberg, D. W. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg. Early experiences with scientific programs on the Cray MTA-2. In *Proc. SC'03*, 2003.

[5] A. Arenas, A. Fernandez, and S. Gomez. Multiple resolution of the modular structure of complex networks. *New Journal of Physics*, 10(05039), 2008.

[6] D.A. Bader, G. Cong, and J. Feo. On the architectural requirements of efficient execution of graph algorithms. In *The 33rd International Conference on Parallel Processing (ICPP)*, pages 547–556, 2005.

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

[8] F. Barahona and R. Anbil. *Mathematical Programming*, 87(0025-5610), 2000.

[9] F. Barahona and F. Chudak. *Discrete Optimization*, 2(1), 2005.

[10] Brian W. Barrett, Jonathan W. Berry, Richard C. Murphy, and Kyle B. Wheeler. Portability and performance through emulation of hardware features: The mtgl on qthreads. In *IPDPS*, pages 1–13, 2009.

[11] Jonathan W. Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, March 2007.

[12] Jonathan W. Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IPDPS* [1], pages 1–14.

[13] Jonathan W. Berry, Bruce Hendrickson, Randall A. LaViolette, Vitus J. Leung, and Cynthia A. Phillips. Community detection via facility location, 2007.

[14] Jonathan W. Berry, Bruce Hendrickson, Randall A. LaViolette, and Cynthia A. Phillips. Tolerating the community detection resolution limit with edge weighting, 2009.

[15] Jonathan W. Berry, Bruce A. Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Proceedings of the International Parallel & Distributed Processing Symposium*. IEEE, 2007.

[16] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[17] A. Capocci, V.D.P. Servedio, G. Caldarelli, and F. Colaiori. Detecting communities in large networks. *Physica A*, 352(2-4):669–76, 2005.

[18] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. *SIAM Data Mining*, 2004.

[19] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *In SDM*, 2004.

[20] A. Clauset, C. Moore, and M.E.J. Newman. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453:98–101, 2008.

[21] A. Clauset, M.E.J. Newman, and C. Moore. *Phys. Rev. E*, 70(066111), 2004.

[22] Aaron Clauset. Finding local community structure in networks. *Phys. Rev. E*, 72(2):026132, Aug 2005.

[23] J. Cohen. Graph twiddling in a mapreduce world. *Computers in Science & Engineering*, 11(4):29–41, 2009.

[24] T.H. Corman, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[25] G. Cornuejols, G. L. Nemhauser, and L. A. Wolsey. The uncapacitated facility location problem. In P. Mirchandani and R. Francis, editors, *Discrete Location Theory*, pages 119–171. John Wiley and Sons, New York, 1990.

[26] Joseph R. Crobak, Jonathan W. Berry, Kamesh Madduri, and David A. Bader. Advanced shortest paths algorithms on a massively-multithreaded architecture. In *IPDPS* [1], pages 1–8.

[27] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 05(1):46–55, 1998.

[28] L. Danon, A. Díaz-Guilera, J. Duch, and A. Arenas. Comparing community structure identification. *J. Stat. Mech*, P09008, 2005.

[29] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, (S.):269–271, 1959.

[30] Robin Dunbar. *Grooming, Gossip, and the Evolution of Language*. Harvard Univ Press, October 1998.

[31] Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper. *UPC Language Specification*, 1.1 edition, May 2003. `http://upc.lbl.gov/docs/user/upc_spec_1.1.1.pdf`.

[32] P. Erdös and A. Rényi. On random graphs I. *Publicationes Mathematicae*, (6):290–297, 1959.

[33] Y. Fan, M. Li, P. Zhang, and Z. Di. The effect of weight on community structure of networks. *Physica A*, 12(021), 2006.

[34] Y. Fan, M. Li, P. Zhang, J. Wu, and Z. Di. The role of weight on community structure of networks. arXiv:physics/0609218, 2006.

[35] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 28–34, 2005.

[36] S. Fortunato and M. Barthélemy. *PNAS*, 104(1):36–41, 2007.

[37] Santo Fortunato. Community detection in graphs, 2009.

[38] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[39] D. Gfeller, J.C. Chappelier, and P. De Los Rios. Finding instabilities in the community structure of complex networks. *Physical Review E*, 72(5):056135, 2005.

[40] M. Girvan and M.E.J. Newman. Community structure in social and biological networks. *PNAS*, 99:7821–7826, 2002.

[41] A. V. Goldberg. Finding a maximum density subgraph. Technical Report UCB/CSD-84-171, EECS Department, University of California, Berkeley, 1984.

[42] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[43] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. *Journal of Algorithms*, pages 228–248, 1999.

[44] R. Guimera and L.A. Nunes Amaral. Modularity and community structure in networks. *Nature (London)*, 433:895–900, 2005.

[45] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kauffmann, San Francisco, second edition, 2006.

[46] B. Hendrickson and J.W. Berry. Graph analysis with high-performance computing. *Computers in Science and Engineering*, 10(2):14–19, 2008.

[47] Bruce Hendrickson and Jonathan W. Berry. Graph analysis with high-performance computing. 10(2):14–19, March/April 2008.

[48] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, New York, NY, 1995.

[49] Intel Corporation. *Intel® Thread Building Blocks*, 1.6 edition, 2007.

[50] A. Lancichinetti, S. Fortunato, and J. Kertesz. Detecting the overlapping and hierarchical community structure of complex networks. *New J. Physics*, 11:033015, 2009.

[51] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, 78:046110, 2008.

[52] J. Leskovec and C. Faloutsos. Scalable modeling of real graphs using kronecker multiplication. In *In Proceedings of the 24th International Conference on Machine Learning*, 2007.

[53] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW*, pages 695–704, 2008.

[54] R. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers and Mathematics with Applications*, 23:573–600, 1992.

[55] Z. Li, S. Zhang, R-S Wang, X-S Zhang, and L Chen. Quantitative function for community detection. *Phys. Rev. E*, 77:036109, 2008.

[56] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*. To appear.

[57] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and J.R. Crobak. Parallel shortest path algorithms for solving large-scale instances. In *9th DIMACS Implementation Challenge: Shortest Paths*, November 2006.

[58] Brendan McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1980.

[59] Ulrich Meyer and Peter Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA '98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 393–404, London, UK, 1998. Springer-Verlag.

[60] Stanley Milgram. The small world phenomenon. *Psychology Today*, May 1967.

[61] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. Working Paper 96-06-042, Santa Fe Institute, 1996.

[62] S. Muff, F. Rao, and A. Caflisch. *Phys. Rev. E*, 72(056107), 2005.

[63] J. M. Mulvey and H. P. Crowder. *Management Science*, 25(4):329–340, 1979.

[64] M.E.J. Newman. *PNAS*, 103(23):8577–8582, 2006.

[65] M.E.J. Newman. Analysis of weighted networks. *Phys. Rev. E*, 70:056131, 2006.

[66] M.E.J. Newman and M. Girvan. *Phys. Rev. Lett.*, 69(026113), 2004.

[67] V. Nicholson, C.-C. Tsai, M. Johnson, and M. Naim. A subgraph isomorphism theorem for molecular graphs. In *Graph Theory and Topology in Chemistry*, number 51 in Stud. Phys. Theoret. Chem., pages 226–230. Elsevier, 1987.

[68] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[69] *System Application Program Interface (API) [C Language].* Information technology—Portable Operating System Interface (POSIX). IEEE Press, Piscataway, NJ, 1990.

[70] J. Reinhardt and S. Bornholdt. *Phys. Rev. Lett.*, 93(218701), 2004.

[71] J. Ruan and W. Zhang. An efficient spectral algorithm for network community discovery and its applications to biological and social networks. In *Seventh IEEE International Conference on Data Mining*, pages 643–648, 2007.

[72] J. Ruan and W. Zhang. Identifying network communities with a high resolution. *Physical Review E*, 77:016104, 2008.

[73] Y. Shiloach and U. Vishkin. An o(n log n) parallel connectivity algorithm. *J. Algorithms*, 3(7):57–67, 1982.

[74] J. Siek, L-Q. Lee, and A. Lumsdaine. *The Boost Graph Library.* Addison-Wesley, 2002.

[75] A. Stepanov and M. Lee. The standard template library. Technical Report 95-11 (R.1), HP Laboratories Technical Report, 1995.

[76] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. Assoc. Comput. Mach.*, 23:31–42, 1976.

[77] K. Underwood, M. Vance, J. Berry, and B. Hendrickson. Analyzing the scalability of graph algorithms on eldorado. In *21st IEEE International Parallel & Distributed Processing Symposium, submitted*, 2007.

[78] Ken Wakita and Toshiyuki Tsurumi. Finding community structure in mega-scale social networks: [extended abstract]. In *WWW '07*. ACM Press, 2007.

[79] Kyle Wheeler, Richard Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium*, pages 1–8. MTAAP '08, IEEE Computer Society Press, April 2008.

[80] A. Yoo, E. Chow, K. Henderson, W. McLendon III, B. Hendrickson, and U. Çatalyürek. A scalable distributed parallel breadth–first search algorithm on BlueGene/L. In *Proc. SC'05*, November 2005. Finalist for the Gordon Bell Prize.

[81] W. W. Zachary. *J. Anthropological Res.*, 33:452–473, 1977.

[82] P. Zakharov. Diffusion approach for community discovering within the complex networks: Livejournal study. *Physica A*, 378(2):550–560, 2007.

# Appendix A

# External Impact

## A.1 Invited Presentations

1. (to occur) Invited Plenary on HPC Graph Computations: SIAM Parallel Processing for Scientific Computing, Seattle, WA, February 24–26, 2010 – Jonathan Berry

2. Keynote at 2008 IPDPS "MultiThreaded Architectures and APplications" Workshop (MTAAP) – Jonathan Berry

3. *Computing Maximum Flow on Massively-Multithreaded Supercomputers*: 20th International Symposium on Mathematical Programming (ISMP), Chicago, August 23-28, 2009 – Cynthia Phillips

4. *The MultiThreaded Graph Library (MTGL)*: SIAM Parallel Processing for Scientific Computing, Atlanta, GA, March 12–14, 2008 – Jonathan Berry

## A.2 Service to Professional Societies

1. Jonathan Berry – program committee member, 2008 IEEE MultiThreaded Architectures & Applications

2. Jonathan Berry – program committee member, 2009 IEEE MultiThreaded Architectures & Applications

## A.3 New Ideas for R&D

We submitted a proposal to the DOE ASCR program "Mathematics for the Analysis of Petascale Data" with academic collaborators Joel Bader (Johns Hopkins University), Aaron Clauset (Santa Fe Institute), Nathan Eagle (Santa Fe Institute). We propose to prepare for emerging petascale datasets in biology and social networks by improving current network analysis methods and developing new ones. The proposal was not funded by ASCR, and we are looking for other sources for this work.

DISTRIBUTION:

| | | |
|---|---|---|
| | MS | , |
| | | , |
| 1 | MS 0899 | Technical Library, 9536 (electronic) |

Sandia National Laboratories