

Software and Algorithms for Graph Queries on Multithreaded Architectures

Jonathan W. Berry* Bruce Hendrickson† Simon Kahan‡ Petr Konecny§

January 20, 2007

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 isomorphism. 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.

1 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 2, we describe the design goals and primary design pattern of the MTGL. Then, in Section 4,

*Sandia National Laboratories

†Sandia National Laboratories

‡Cray, Inc.

§Cray, Inc.

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.

2 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 [6]. 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 6. 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.

3 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: $G = (V, E)$, where 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 variable (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: $W = ((w_0, w_1), (w_1, w_2), \dots, (w_{l-1}, w_l))$. We say that two walks W and W' are *type-isomorphic* if

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

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 3, 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 3.

In addition to the notation defined in Table 3, 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 $T \stackrel{ts}{\leftarrow} T \cup e$.

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 $e.id$).

4 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 [5].

```
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 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 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 [7] solves the subgraph isomorphism problem, but its computational complexity makes this algorithm 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.

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

Table 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 .

```

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 3: Pseudocode for the PSearch routine, templated to treat the user’s visit test as the only criterion for proceeding.

4.1 Preliminaries

In Figures 1,2, and 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 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 ini-

tial 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 sub-graph isomorphism heuristic introduced in Section 4.5.

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 un-

visited, *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_l(n-k)}
    for v in H: # in serial
        PSearch<OP,Vis>(v)
    for v in L:
        PSearch<OP,Vis>(v)
}

```

Figure 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 4.5 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 4.3.

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 4.

4.2 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) [5], 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 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

```

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  $\stackrel{ts}{\leftarrow}$  T ∪ { (C[v], C[v']) } }
}

```

Figure 5: The visitor object for Kahan’s algorithm, phase 1 . The hash table insertion is made only if $C[v]$ is not equal to $C[v']$.

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 6.

```

V_2 ← {
  C ← V_1.C

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

```

Figure 6: The visitor object for Kahan’s algorithm, phase 3

Kahan’s algorithm in its entirety is given in Figure 7.

```

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: Kahan’s algorithm for connected components

4.3 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 8. The

```

V_3 ← {
  C ← array of |V(G)| ints

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

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

```

Figure 8: The bully algorithm

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 Ka-

han’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 6, the performance of the bully algorithm is good.

4.4 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 filtering*. 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 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 op-

```

V_A ← {
  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[ieid] = 1
    }
  oe(v, v') ← te(v, v')
}

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

```

Figure 9: Compound type filtering. The % symbol denotes modular arithmetic.

eration 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 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 6, 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.

4.5 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 [3]. Furthermore, the problem of testing isomorphisms between a relatively small

```

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 10: A visitor class to help find the edges of G_B

“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 [7] and are not practical for large instances. There is more recent literature on heuristics, such as [4], [2], and others, but we haven’t yet compared our heuristic with this work.

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 neigh-

bors 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.

```

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 if a found, 1
      if f == 0
        # return the first match
        for (v̄, v̂) in (v, v'), ancestors:
          S ← S ∪ (M(v̄), M(v̂))
  }
}

```

Figure 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.

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”). Sup-


```

SubgraphIsomorphism( $G, T_E, W$ ) {
   $B \leftarrow \text{FindBipartiteEdges}(G, T_E, W)$ 
   $V\_B \leftarrow \{(i, j) : \exists k B[i, j, k] = 1\}$ 
   $E\_B \leftarrow \{(i, j), (i+1, k) :$ 
       $(i, j), (i+1, k) \in V\_B \wedge$ 
       $B[i, j, k] = 1\}$ 

   $lv((i, j) \in V\_B) = i$ 
   $s \leftarrow (0, j) \in V\_B : \exists k B[0, j, k] = 1$ 
  define  $V\_6$ 
  PSearch<AND,  $V\_6$ >(s)
  return  $V\_6.S$ 
}

```

Figure 12: Subgraph isomorphism pseudocode

posing that the walk traverses l edges,

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

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, \dots, 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 10 shows MTGL pseudocode that finds the edges of G_B .

4.6 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.

```

 $V\_7 \leftarrow \{$ 
   $C \leftarrow \text{array of } |V(G)| \text{ ints}$ 
  (initially empty)
   $\text{done} \leftarrow \text{reference to int}$ 

   $d(v) \{ \}$ 
   $vt(v, v') \{ \}$ 
   $te(v, v') \{ C[v'] \leftarrow C[v] \}$ 
   $oe(v, v') \{$ 
     $c \stackrel{ff}{\leftarrow} C[v']$ 
    if  $C[v] \neq c$ :
       $\text{done} = 1$ 
  }
}

```

Figure 13: The visitor object shared by two breadth-first searches in the S-T connectivity algorithm

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 [9] to explore s-t connectivity on BlueGene/Light. A distributed-memory code applicable only to Erdős-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 6, we will discuss the performance comparisons we were able to make.

5 Experiments with MTGL Kernels

In order to evaluate 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

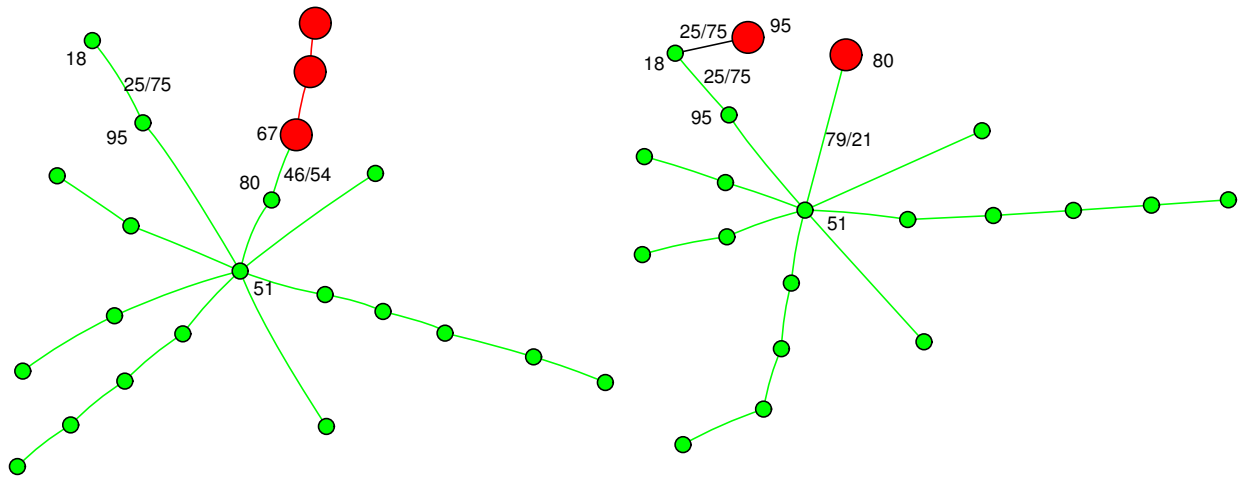


Figure 18: Subgraph isomorphism results. The target graph is on the left, and the subgraph found by the heuristic is on the right. Some vertex and edge types are shown for context. The large vertices represent places where the type-isomorphic walks did not produce topological isomorphism.

```

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 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.

```

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 15: Pseudocode for the S-T connectivity. Two concurrent breadth-first searches converge, and each search 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.

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.

5.1 Data

For our experiments, the types of vertices and edges are selected randomly from $\{0, 1, \dots, 99\}$, 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 mul-

iple 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 have

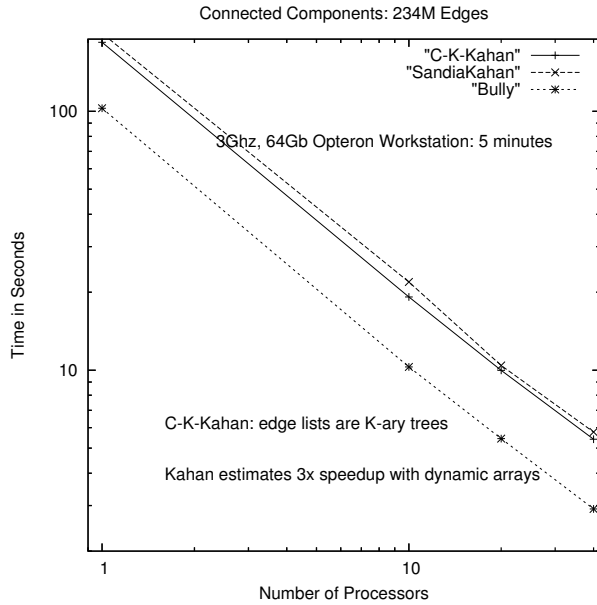


Figure 16: Connected components kernel performance

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

5.2 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

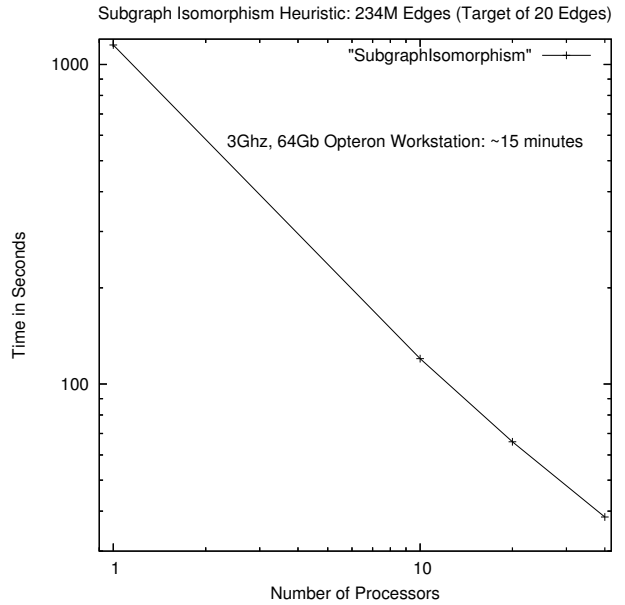


Figure 17: Subgraph isomorphism kernel performance

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 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 4.5 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

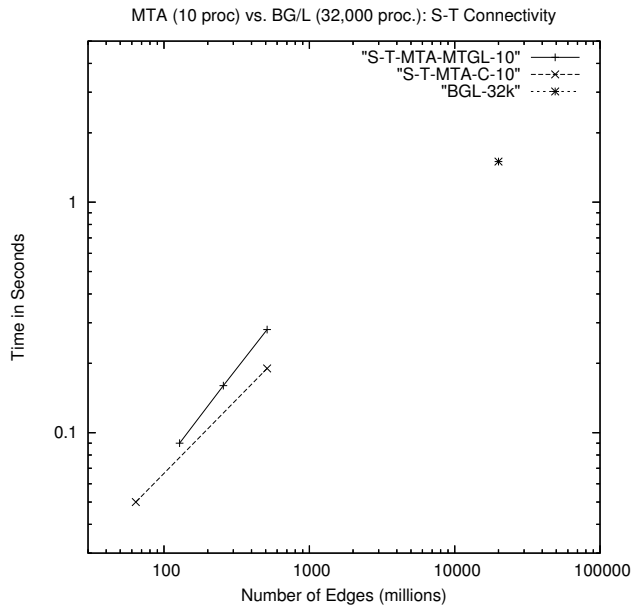


Figure 19: S-T connectivity comparison with BlueGene/L

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 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.

6 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.

6.1 MTA-2 performance

Figure 16 shows the results of our MTA-2 performance test on the connected components algorithms. Without considering the issue of dif-

fering 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 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 Erdős-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 Erdős-Renyí graph that was processed in 1.5 seconds using 32,000 processors of BlueGene/L in [9] 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 had to process. Thus, 10 MTA processors should be able to process 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 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.

6.2 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 [1], and for details of our performance predictions, see [8]. 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:

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.

7 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. Cur-

rent repository versions of the software are available by contacting jberry@sandia.gov.

8 Acknowledgements

We thank Cynthia Phillips for helpful conversations and for a thorough proof-reading. 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] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 28–34, 2005.
- [2] R. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers and Mathematics with Applications*, 23:573–600, 1992.
- [3] Brendan McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1980.
- [4] 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.
- [5] Y. Shiloach and U. Vishkin. An $o(n \log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(7):57–67, 1982.
- [6] J. Siek, L-Q. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.
- [7] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. Assoc. Comput. Mach.*, 23:31–42, 1976.
- [8] 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.

- [9] 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.