Graph Software Development and Performance on the MTA-2 and Eldorado

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

May 10, 2006

Abstract

We will discuss our experiences in designing and using a software infrastructure for processing semantic graphs on massively multithreaded computers. We have developed implementations of several algorithms for connected components, subgraph isomorphism, and s-t connectivity. We will discuss their performance on the existing Cray MTA-2, and their predicted performance on the upcoming Cray Eldorado. We will also describe ways in which the underlying architecture and programming model have informed algorithm design and coding paradigms. We conclude with a revealing comparison of s-t connectivity performance on the MTA-2 and BlueGene/L.

1 Introduction

In this paper we describe a new graph software infrastructure for the Cray MTA/Eldorado series of supercomputers. The MTA-2 is currently the fastest machine for processing graph algorithms because it has been designed to tolerate latency rather than to mitigate it. 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, can 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." As the intelligence community shifts its focus from cold war challenges to terrorism challenges, the processing of graph algorithms on large, unstructured semantic graphs has rapidly become very important.

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

We present our infrastructure, which we call the MultiThreaded Graph Library (MTGL), in stages. First, in Section 2, we describe the design goals and primary design pattern of the MTGL. Then, in Section 4, 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.

2 Basic Design Methodology

The MTGL is a small prototype C++ library that is inspired by the Boost Graph Library (BoostGL) of Siek, Lee, and Lumsdaine [5]. However, our library is not an extension of BoostGL. The MTA/Eldorado compilers are not fully compliant with the C++ standard, and BoostGL makes very 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 the MTA/Eldorado 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. Our

*Sandia National Laboratories
†Sandia National Laboratories
‡Cray, Inc.
§Cray, Inc.

software currently runs on the MTA-2 and on standard workstations as well.

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 will show several examples of the use of visitors below. performance results in Section 6. We omit actual code samples in this paper.

3 Notation

In order to describe multithreaded graph algorithms and their implementations in the MTGL, it is convenient to define some notation. Beginning with the familiar definition of a graph: $G = (V, E)$, were $V(G)$ is the vertex set of G $E(G)$ is the edge set of G, $E(v)$ 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 undirected. That is, whenever (v, w) exists, (w, v) will exist as well. The rationale for this seemingly limiting decision is that 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'. If 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 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 $t[w_i, w_{i+1}] = t[w'_i, w'_i + 1]$ for all $0 \leq i \leq l - 1$.

When multiple threads access a piece of shared memory, the MTA's 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 the table.

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 $T \overset{\text{ts}}{\leftarrow} T \cup e$.

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

As a final notational convenience, we encapsulate MTGL logic that determines whether or not it is worthwhile to parallelize a loop. The pseudocode

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

indicates that the MTGL will instruct the MTA to parallelize the loop if $E(v)$ is large enough. Otherwise, the loop will run in serial. This explanation holds unless there is a comment in the pseudocode indicating otherwise.

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. This is an elementary problem in graph theory, and linear-time solutions exist. Furthermore, efficient parallel algorithms exist as well [4].

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 [6] solves the subgraph isomorphism problem, but its computational complexity makes this algorithm unusable for large inputs. We will give a heuristic for the subgraph isomorphism problem that demonstrates the flexibility of the MTGL and scales almost perfectly on the MTA-2.

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

Table 1: Some MTA primitives and their pseudocode and MTGL designations

```
PSearch<AND,Vis>(v)
{
   Vis.d(v)
   for (v,v') in E(v):
      if (not Vis.vt(v,v')): return
      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."

4.1 Preliminaries

Below we will 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 $d(v)$ method, to be called upon the discovery of vertex v,
3. A $vt(v, v')$ (visit-test) method, to be called before traversing edge (v, v').
4. A $te(v, v')$ (tree-edge) method, to be called upon visiting edge (v, v') to first discover v'.
5. An $oe(v, v')$ (other-edge) method, to be called upon visiting edge (v, v') to revisit v'.

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.

For example, if the user wishes to search the subgraph induced by 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,

```
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.

```
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 4: Pseudocode for the SearchHighLow routine

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.

We are now ready to define PSearch using our pseudocode notation. The three varieties are shown in Figures 1, 2, and 3.

The nested parallelism in this 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 first, *in 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 [4], 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 implmentation 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.

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

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

Figure 5: The visitor object for Kahan's algorithm, phase 1

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 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 non-empty visit-

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

   d(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 8: The bully algorithm

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 generate than Kahan's three-phase algorithm since we expect no speedup in the pathological cases in which the entire graph a 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 will 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 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 is also a step taken to mitigate hot spots.

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 MTA/Eldorado machines increase in size.

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

4.5 Subgraph isomorphism

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 [2]. 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 [6] and are not practical for large instances. There is more recent literature on heuristics, such as [3], [1], and others, but we haven't made a thorough review of this work. Therefore, the heuristic we will present for subgraph isomorphism on semantic graphs has not yet been compared with the state of the art. However, its performance on the MTA-2 is good, and we plan to explore this question in future work.

The vertex and edge types of a semantic graph make the otherwise intractable subgraph isomorphism

```
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

```
V_6 ← {
   lv ← levels of V(G_B)
   M ← map: V(G_B).id → V(G).id
   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 11: Subgraph extraction visitor pseudocode

problem much 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 algorithm we describe next.

With our model of 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 because 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"). 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 from which we may explore all possible subgraphs of G which have a walk that is 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

```
SubgraphIsomorphism(G, T_E, W) {
   B ← FindBipartiteEdges(G, T_E, W
   V_B ← {(i, j) : ∃_j, k B[i, j, k] = 1}
   E_B ← {[(i, j), (i + 1, k)] : 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 12: Subgraph isomorphism pseudocode

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.

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 [7] 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 6, we will discuss the interesting performance comparisons we were able to make.

```
V_7 ← {
   C ←  array of |V(G)| ints
            (initially empty)
   done ←  reference to int

   d(v)  { }
   vt(v,v')  { }
   te(v,v') { C[v'] ← C[v] }
   oe(v,v') {
       c ff← C[v']
       if C[v] != c:
          done = 1
   }
}
```

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

```
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.

5 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

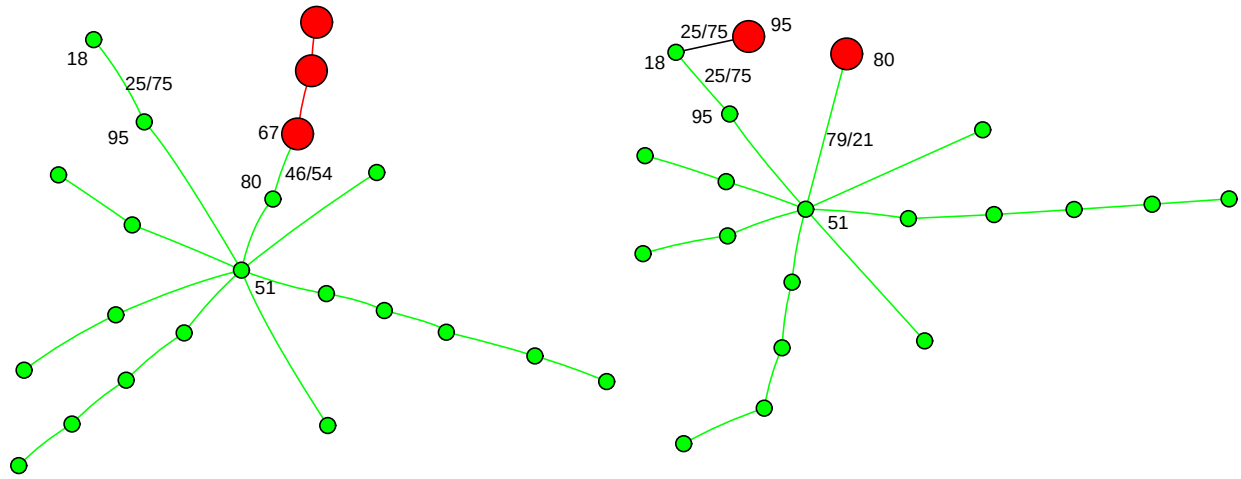

Figure 18: Subgraph isomorphism results. The target graph is on the left, and the subgraph found by the heuristic is on the right. Some vertex and edge types are shown for context. The large vertices represent places where the type-isomorphic walks did not produce topological isomorphism.

```
STConnectivity(G,s,t)
{
   define V_7
   bfs1 ← BFS<OR,V_2>(s)
   bfs2 ← BFS<OR,V_2>(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 seach level of each search is explored in parallel. The "topshell" notation indicates all vertices discovered by the previous call to the search.

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 $\{0, 1, \ldots, 99\}$, 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 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 an experimental paper. Rather, this paper serves as an introduction to the MTGL with a few accompanying experiments. 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

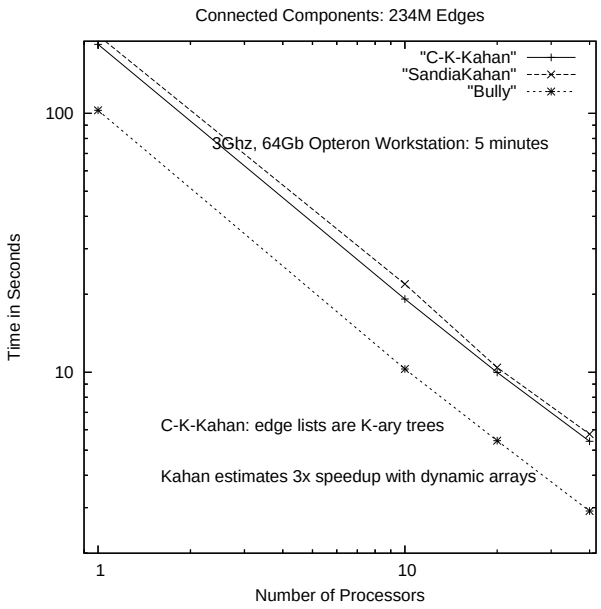

Figure 16: Connected components kernel performance

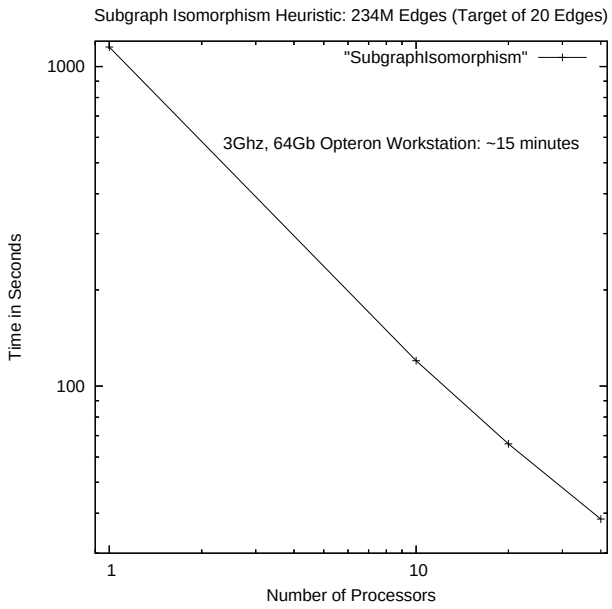

Figure 17: Subgraph isomorphism kernel performance

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 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 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).

To ground the absolute performance in terms of modern workstations, we also ran our experiments on a 3Ghz, 64Gb 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 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

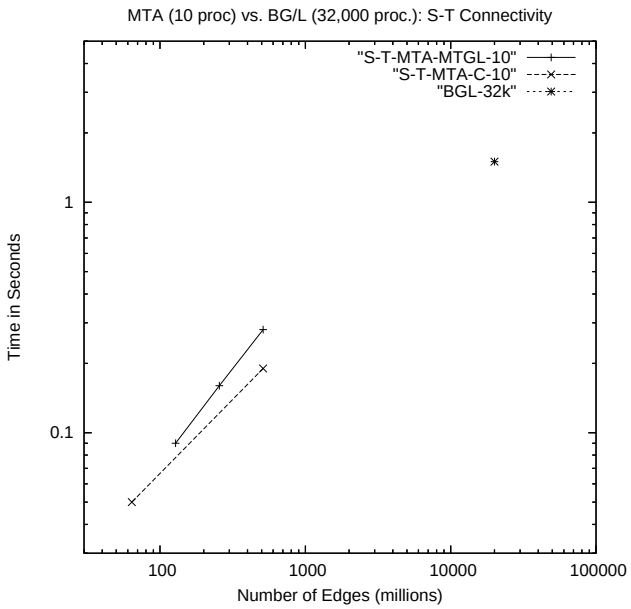

Figure 19: S-T connectivity comparison with BlueGene/L

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 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 [7] 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 expending 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 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 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

Another paper will describe in detail the process we describe in abstract below. Here, we do not even describe the Eldorado system other than to note that 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. 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 memory 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 ker-

nels to run on a 512 processor Eldorado as if it were an O(500) 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 MTA/Eldorado 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. 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`.

References

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

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

[3] 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.

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

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

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

[7] 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.