The MultiThreaded Graph Library

November 17, 2009

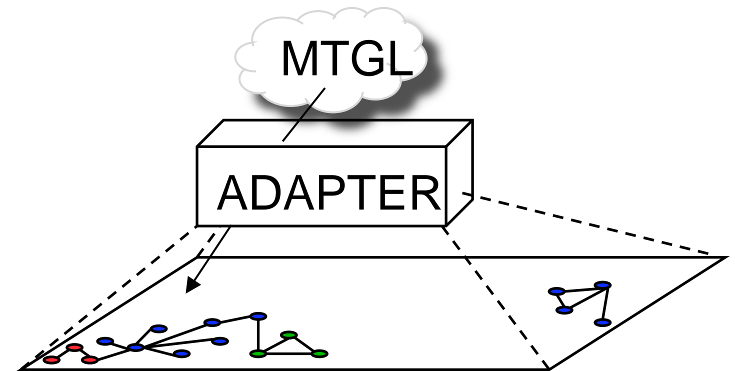
Jon Berry
Greg Mackey

Sandia National Laboratories

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-AC04-94AL85000.

Outline

- **Design goals (why build an MTGL?)**
- **Current status (what does it do now?)**
- **MTGL elements (how do you code?)**
- **Performance of primitives (what's the overhead?)**
- **Future (what's the vision for using it?)**



Sandia National Laboratories

Design Goals

- **Enable a *generic* C++ library on multithreaded platforms**
  - Once an algorithm is benchmarked in C on the Cray XMT
    - *We may want to compose it with other algorithms*
      - Accept the graph data structures they produce
      - Produce output that other algorithms can accept
    - *We may want to allow programmers to customize it*
      - E.g. Run it seamlessly on only blue and red edges
      - E.g. Execute a user analytic upon events like vertex visits
    - *We don't want users to change key multithreaded code*
      - Encapsulate these portions in the library
      - Allow users enough access to tailor without endangering themselves
- **Retain good multithreaded performance on the Cray XMT!**
- **Run/debug on more conventional multicore or even serial workstations**

Current Status

- **Open-Source: http://software.sandia.gov/trac/mtgl**
  - Expanding set of tutorials, documentation
- **Active development associated with several projects**
- **Converging on efficient primitives, API**
  - Not settled; community input welcomed
    - eldorado-graph@sandia.gov
    - jberry@sandia.gov
- **Notable recent research activity**
  - Triangles, rectangles, community detection
    - Berry,Hendrickson,LaViolette,Phillips, 2009 http://arxiv.org/abs/0903.1072
  - “MEGRAPHS” graph database system uses the MTGL
  - Barrett, Berry, Murphy, Wheeler, MTAAP 2009
    - MTGL/Qthreads for XMT/Niagara/Operteron portability

MTGL Elements

- **Each graph type stores its *traits***
  - E.g. vertex descriptor, size_type
  - No hardcoding of types like “int”
    - Algorithm A will run on Joe’s data structure that uses “unsigned long” and Bob’s structure that uses “uint32_t”
    - Important to get this right since auto typecasting can kill XMT performance
  - The algorithms retrieve these traits to determine typing of variables
- **Each graph type exports a common API**
  - How do you get the adjacencies of a vertex?
  - How do you get the id of a vertex? .. etc.
- **The programming associates auxiliary data with vertices and edges via *property maps***
  - E.g. global vertex id
  - E.g. distance, capacity, flow, component number, etc.

MTGL Prerequisites

- C++ experience at the complexity level of the C++ Standard Template Library (STL)
- Basic mta-pe (Cray XMT programming environment)

The MTGL is simpler than the Boost Graph Library, but also less generic

It's fine to start with C and "mtgl-ize" later

Sandia National Laboratories

MTGL Performance Considerations

- **Test case 1: traversing all adjacencies in a graph**
  - A) to do something very simple
  - B) to do something generic that the user provides

- **Test case 2: breadth-first search**
  - A) with the best XMT algorithm for simple data
  - B) with "mtgl-ized" versions of A)
  - C) with an alternative algorithm

"Simple data" : ~2B edge Erdos-Renyi Random Graphs

"Realistic data": ~0.5B edge power-law distributed data (Much tougher than R-MAT)

Sandia National Laboratories

Traversing All Adjacencies in a Graph

- **Algorithm 1 (pure C): Use the compiler's "Manhattan Loop Collapse"**

```
      *        #pragma mta assert nodep
               #pragma mta assert noalias *in_degree
55             for (i = 0; i < order; i++)
               {
                 size_type begin = g->index[i];
                 size_type end = g->index[i + 1];
                 for (j = begin; j < end; j++)
60               {
   I 4 PP:m        mt_incr(in_degree[g->end_points[j]], 1);
   I   *         }
     1 X       }
```

```
Loop 4 in compute_in_degree(mtgl::static_graph<(int)1> *, unsigned long *) at line 61 in loop 3
    parallel section of loop from level 2
    Loop summary: 2 memory operations, 0 floating point operations
        2 instructions, needs 45 streams for full utilization
        pipelined
```

2 memops, 2 instructions!

Seeing this PP:m is good!

Sandia National Laboratories

Traversing All Adjacencies in a Graph

- **Algorithm 2 (generic C): What if the inner loop calls a generic function via function pointer? The compiler can't inline.**

```
I    *        #pragma mta assert noalias *in_degree
              #pragma mta assert parallel
100           for (i = 0; i < order; i++) {
                size_type begin = index[i];
     *
     *          size_type end =   index[i + 1];
                size_type u = end_points[i];
                #pragma mta assert parallel
105             for (j = begin; j < end; j++) {
      18 pp:m
      13 pp:m     size_type v = end_points[j];
I    *
      18 pp:m     if (my_func(u, v)) {
I     18 pp:m
      13 pp:m       mt_incr(in_degree[v], 1);
                  }
110             }
              }
```

my_func(i,j): return (i < j);

Loop summary: 26 instructions, 0 floating point operations
4 loads, 1 stores, 12 reloads, 0 spills, 2 branches, 2 calls

So far so good!

But now we have > 10x memrefs and instructions! The code is unusable on large data.

Sandia National Laboratories

Traversing All Adjacencies in a Graph

- **Algorithm 3 (generic C++): What if the inner loop calls a method of a generic function object ("functor")?**

```
                     #pragma mta assert parallel
100                  for (i = 0; i < order; i++) {
                       size_type begin = index[i];
        *
        *              size_type end =   index[i + 1];
                       size_type u = end_points[i];
                       #pragma mta assert parallel
105                    for (j = begin; j < end; j++) {
      16 pp:m            size_type v = end_points[j];
    I   *
      16 pp:m            if (my_func(u, v)) {
    I 16 pp:m              mt_incr(in_degree[v], 1);
                         }
110                    }
                     }
        *          }
```

Loop summary: 6 instructions, 0 floating point operations
1 loads, 1 stores, 0 reloads, 0 spills, 2 branches, 0 calls

The same code!

but now my_func is an object

This will scale!

Sandia National Laboratories

Traversing All Adjacencies in a Graph

- **Algorithm 4 (partial MTGL): Use generic C++ strategy with loop merge, but use the MTGL API.**

```
175           template <class Graph, class visitor>
    I   *     void visit_adj_partial(Graph& g, visitor f)
        *     {
                typedef typename graph_traits<Graph>::size_type size_type;
                typedef typename graph_traits<Graph>::vertex_descriptor vertex_t;
180 I   *       const size_type *index = g.get_index();
    I           const vertex_t *end_points = g.get_end_points();
    I           const size_type order = g.get_order();
    I           vertex_id_map<Graph> vid_map = get(_vertex_id_map, g);
                size_type i, j;
185             #pragma mta assert parallel
                for (i = 0; i < order; i++)
                {
    I             vertex_t u = g.get_vertex(i);
    I             size_type begin = index[get(vid_map,u)];
190 I             size_type end = index[get(vid_map,u) + 1];
                  #pragma mta assert parallel
                  for (j = begin; j < end; j++)
                  {
    I 19 DD:m$      f(u, end_points[j]);
195               }
                }
              }
```

Extracting information From graph API

The same number of instructions and memory references as C++ alg. 3 in the merged loop.

Loop summary: 6 instructions, 0 floating point operations
1 loads, 1 stores, 0 reloads, 0 spills, 2 branches, 0 calls

The key work

Sandia National Laboratories

Traversing All Adjacencies in a Graph

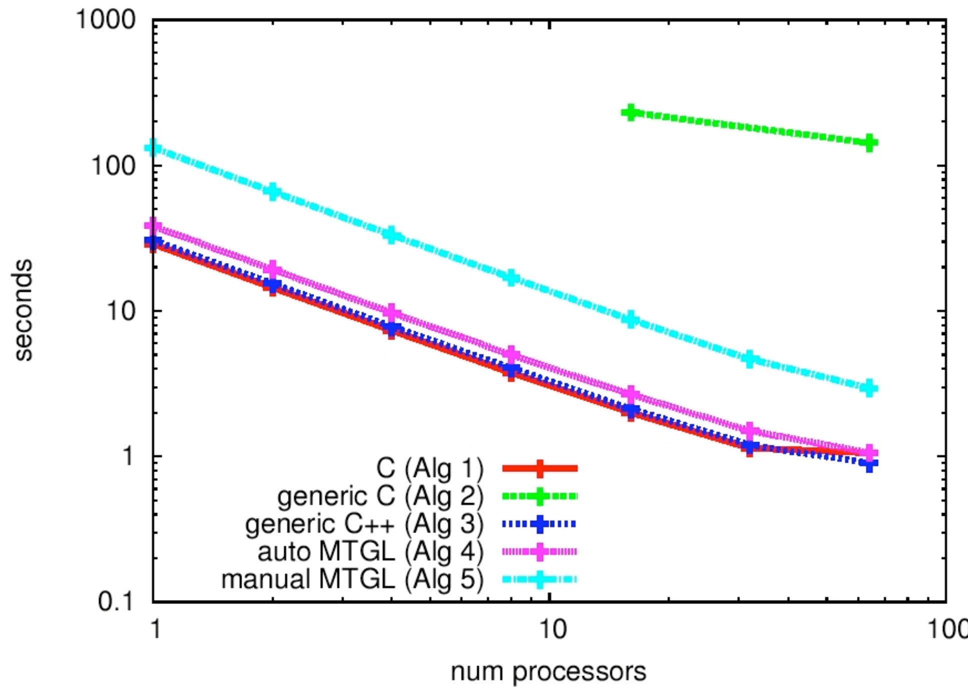
- **Algorithm 5 ("visit_adj" in the MTGL): Manually load balance among adjacencies – fully generic**

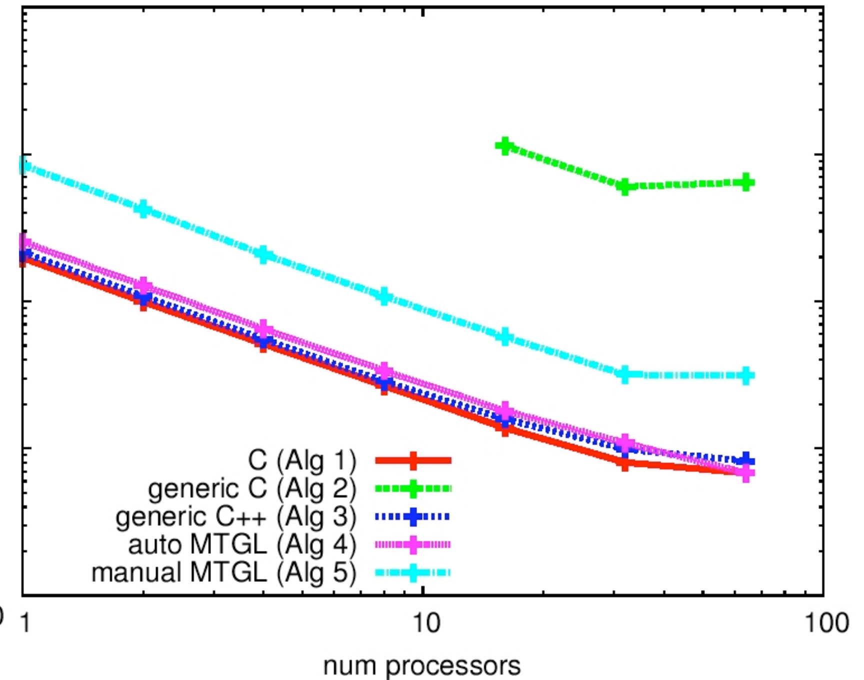
Why do this? Stay tuned for BFS.

thread 0 thread 1 thread 2

Sandia National Laboratories

XMT Results: Adjacency List Traversal





Simple Data
(~2B edge Erdos-Renyi)

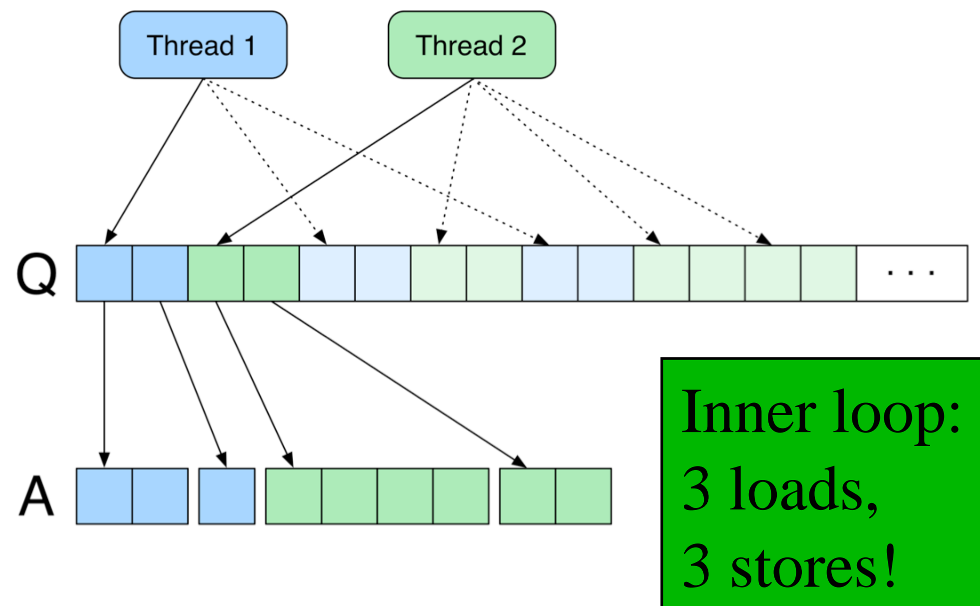
Realistic Data
(~0.5B edge power law)

- "auto MTGL" code semi-generic at no efficiency cost
- "manual MTGL" code fully-generic at 2-3X

Sandia National Laboratories

Petr Konecny's BFS Algorithm (2007)

- **Q is a circular queue that contains the search vertices.**
- **A is the virtual adjacency list for the vertices in Q.**
- **For each level of the search:**
  - Divide current level vertices in Q into equal sized chunks.
  - Each thread grabs the next unprocessed vertex chunk and the next output chunk in Q.
  - Each thread visits the adjacencies of every vertex in its input chunk writing the next level of vertices to its output chunk. New output chunks are grabbed as needed. Unused portion of output chunk filled with marker to indicate no vertex.



Inner loop: 3 loads, 3 stores!

Sandia National Laboratories

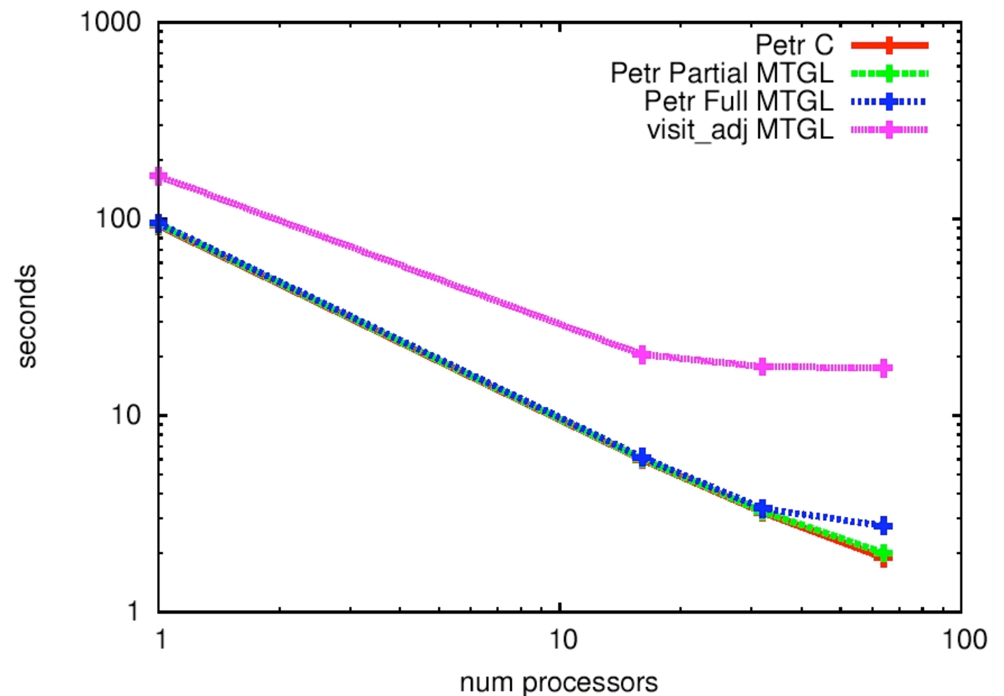
MTGL-ized Versions of Petr's C code

- **Partial**
  - Generic for compressed sparse row (CSR structures)
  - Inner loop does the same number of instructions and memrefs as the pure C code
  - Thanks to Mike Ringenburg & Kristi Maschhoff of Cray for helping find a troublesome auto typecast problem (which had added 2 memrefs/adjacency and prevented scaling past 32p)
- **Full**
  - Fully generic for any MTGL graph adapter
  - Inner loop does the same number of memrefs, 2 more instructions, and one more register spill
  - Haven't yet worked with Cray to see if this can be improved

BFS Results for "Fake" Data

- **Petr C: original C code**
- **Petr Partial MTGL**
  - Perfomance almost identical (same #instructions, memrefs)
- **Petr Fully MTGL**
  - The extra 2 instructions and 1 spill currently slows scaling past 32p
- **Visit_adj MTGL uses Alg 2**
  - Looks hopeless, but wait..

~2 Billion edge Erdos-Renyi

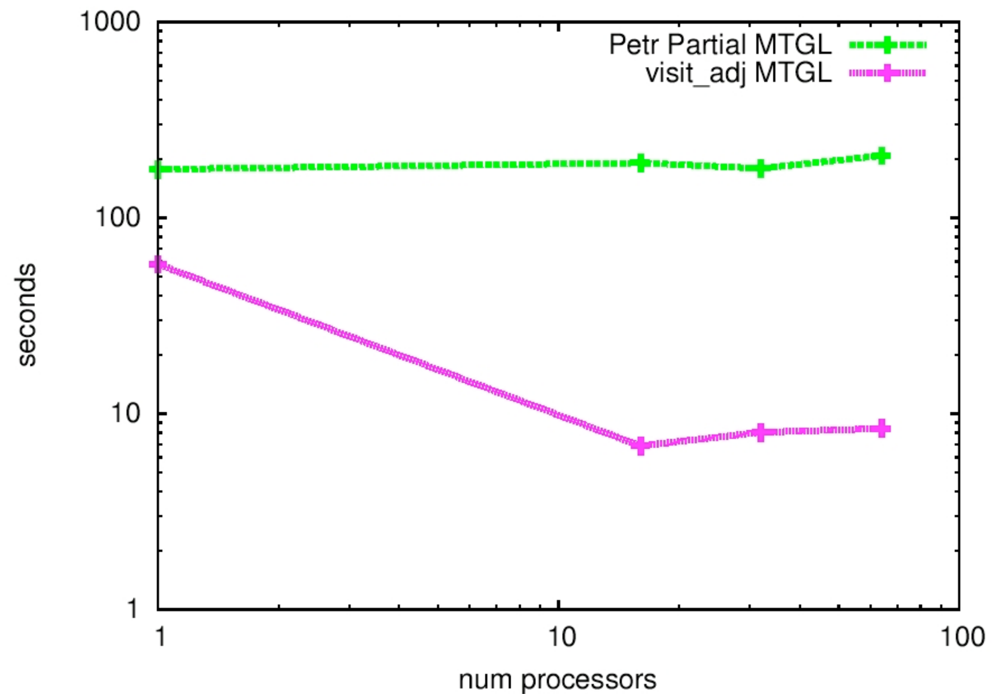


Sandia National Laboratories

BFS Results for Realistic Data

- **Petr Partial MTGL**
  - Algorithmic issue: high-degree vertex early in search means serialization
- **Visit_adj MTGL uses Alg 2**
  - Chunks over adjacencies, not the bfs queue

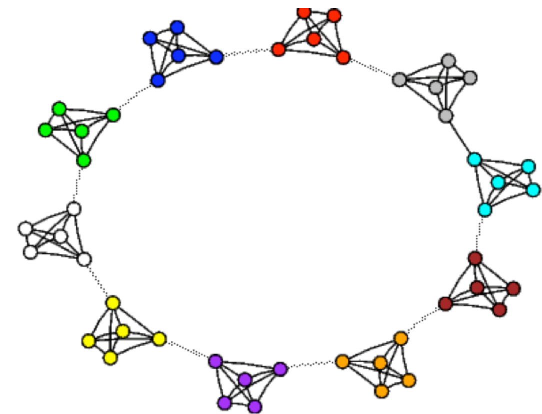
~0.5 Billion edge power-law



We know of no efficient algorithm to scale past 16p on these data!

Vision: Compose Kernels

- **MTGL Example: Hierarchical community detection**
  - Weight edges using a mathematical programming optimization
  - Run a filtered connected components that respects heavy edges
  - Derive a contracted graph by appealing to the result
  - Recurse, maintaining mappings between levels



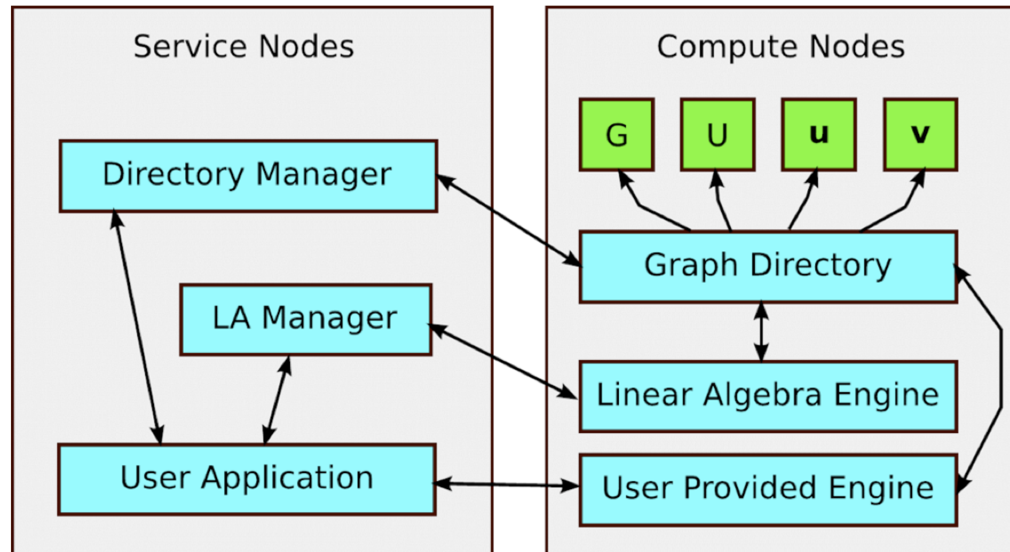
Vision: Compose Kernels

- **MTGL Example: Subgraph Isomorphism**
  - Filter out edges that couldn’t match (returns an edge-induced subgraph)
  - Take an Euler tour in the pattern graph
    - “Duplicate adapter” translates directionality
  - Build a bipartite graph representing potential matches
  - Backwards search, then find connected components
  - Run more exact algorithm on each component

Sandia National Laboratories

MEGRAPHS

(Modular Environment for Graph Research and Analysis with Persistent Hierarchical Storage)



Simplifies graph application implementation on Cray XMT

- Maintains persistent copies of graphs/vectors
- Allows user processes to attach to these objects
- Provides a suite of commonly used primitives
- Uses MTGL as the underlying graph library

Contact: Curt Janssen cljanss@sandia.gov

Sandia National Laboratories

Future

- Finalize basic API
- More tutorials at http://software.sandia.gov/trac/mtgl
- Expand set of MTGL algorithms
- Supply MEGRAPHS with user-defined engines encapsulating MTGL (and other algorithms)
- ? Merge with Boost Graph Library (Boost MultiThreaded Graph Library?)
- Explore synergy with GraphCT, PNNL applications

Acknowledgements

Generic Software Background

Nick Edmonds (Indiana U.)
Douglas Gregor (Apple, formerly Indiana U.)
Andrew Lumsdaine (Indiana U.)
Jeremiah Willcock (Indiana U.)

MultiThreading Background

Simon Kahan (formerly Cray)
Petr Konecny (Google, formerly Cray)
Kristyn Maschhoff (Cray)
David Mizell (Cray)
Mike Ringenburg (Cray)

MTGL Algorithm Design and Development

Brian Barrett (Sandia)
Vitus Leung (Sandia)
Kamesh Madduri (Lawrence Berkeley Labs)
Brad Mancke (BBN, formerly Sandia)
William McLendon (Sandia)
Cynthia Phillips (Sandia)
Kyle Wheeler (Sandia)

Sandia National Laboratories