PBGL: A High-Performance Distributed-Memory Parallel Graph Library

Andrew Lumsdaine
Indiana University
lums@osl.iu.edu

My Goal in Life

- Performance with elegance

Introduction

- Overview of our high-performance, industrial strength, graph library
  - Comprehensive features
  - Impressive results
  - Separation of concerns
- Lessons on software use and reuse
- Thoughts on advancing high-performance (parallel) software

Advancing HPC Software

- Why is writing high performance software so hard?
- Because writing software is hard!
- High performance software is software
- All the old lessons apply
- No silver bullets
  - Not a language
  - Not a library
  - Not a paradigm
- Things do get better
  - but slowly

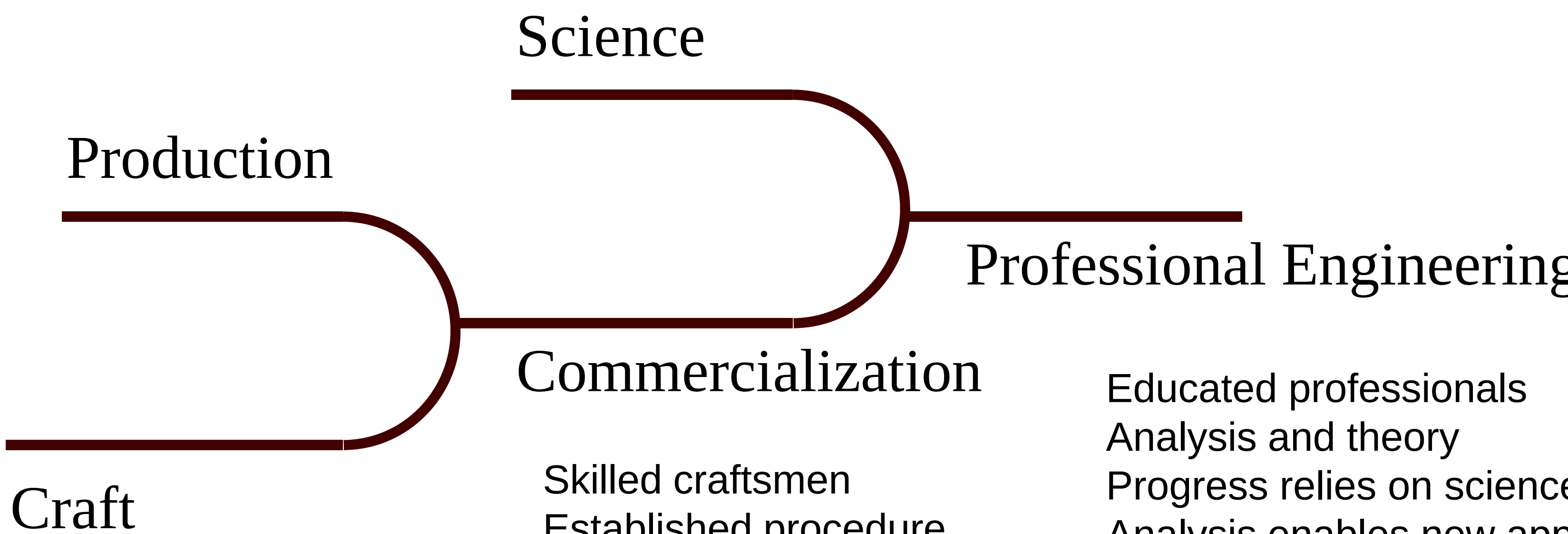
Advancing HPC Software

Progress, far from consisting in change, depends on retentiveness. Those who cannot remember the past are condemned to repeat it.

Advancing HPC Software

- Name the two most important pieces of HPC software over last 20 years
  - BLAS
  - MPI
- Why are these so important?
- Why did they succeed?

Evolution of a Discipline



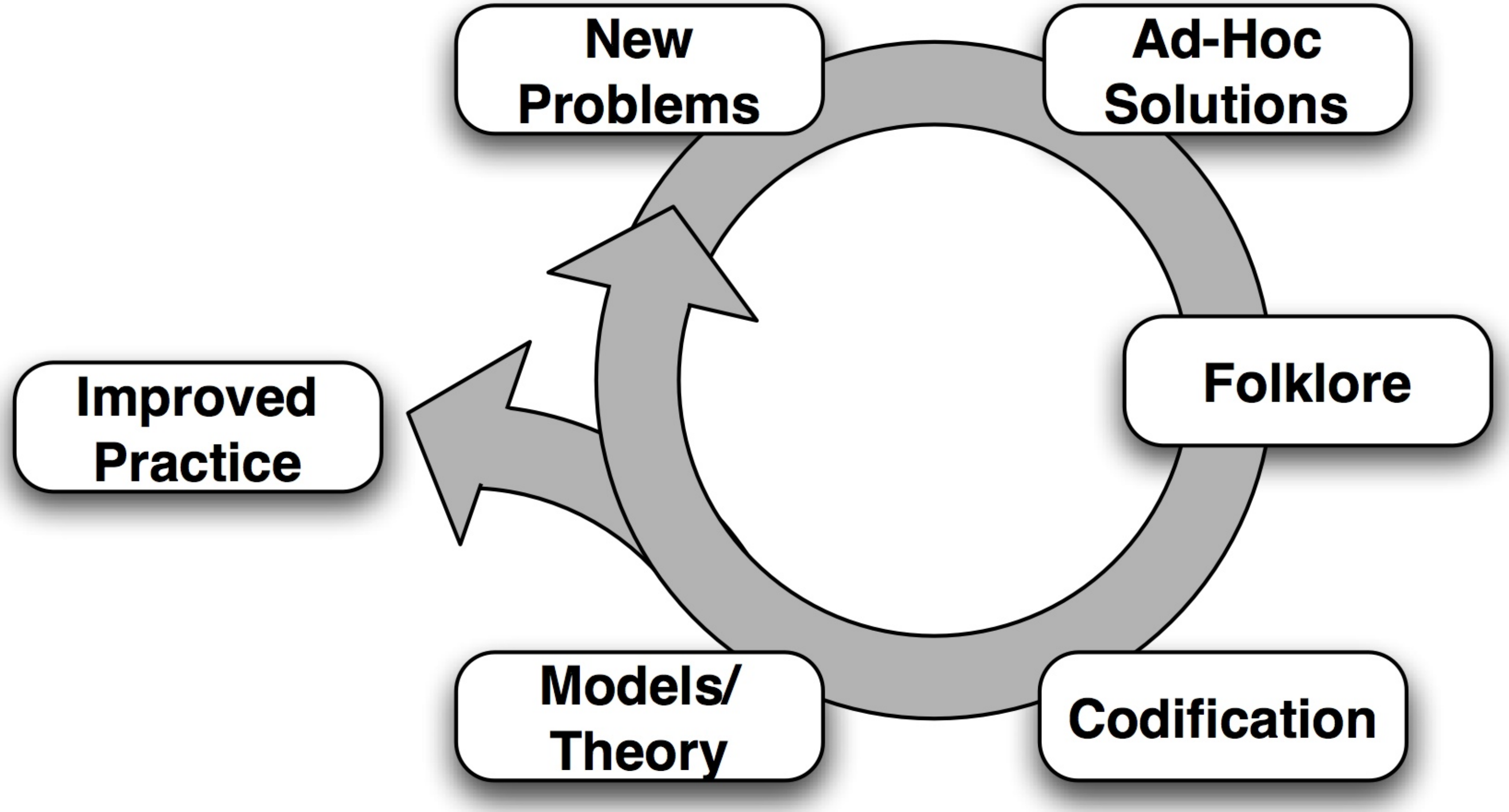
Skilled craftsmen
Established procedure
Training in mechanics
Concern for cost
Manufacture for sale

Educated professionals
Analysis and theory
Progress relies on science
Analysis enables new apps
Market segmented by
 product variety

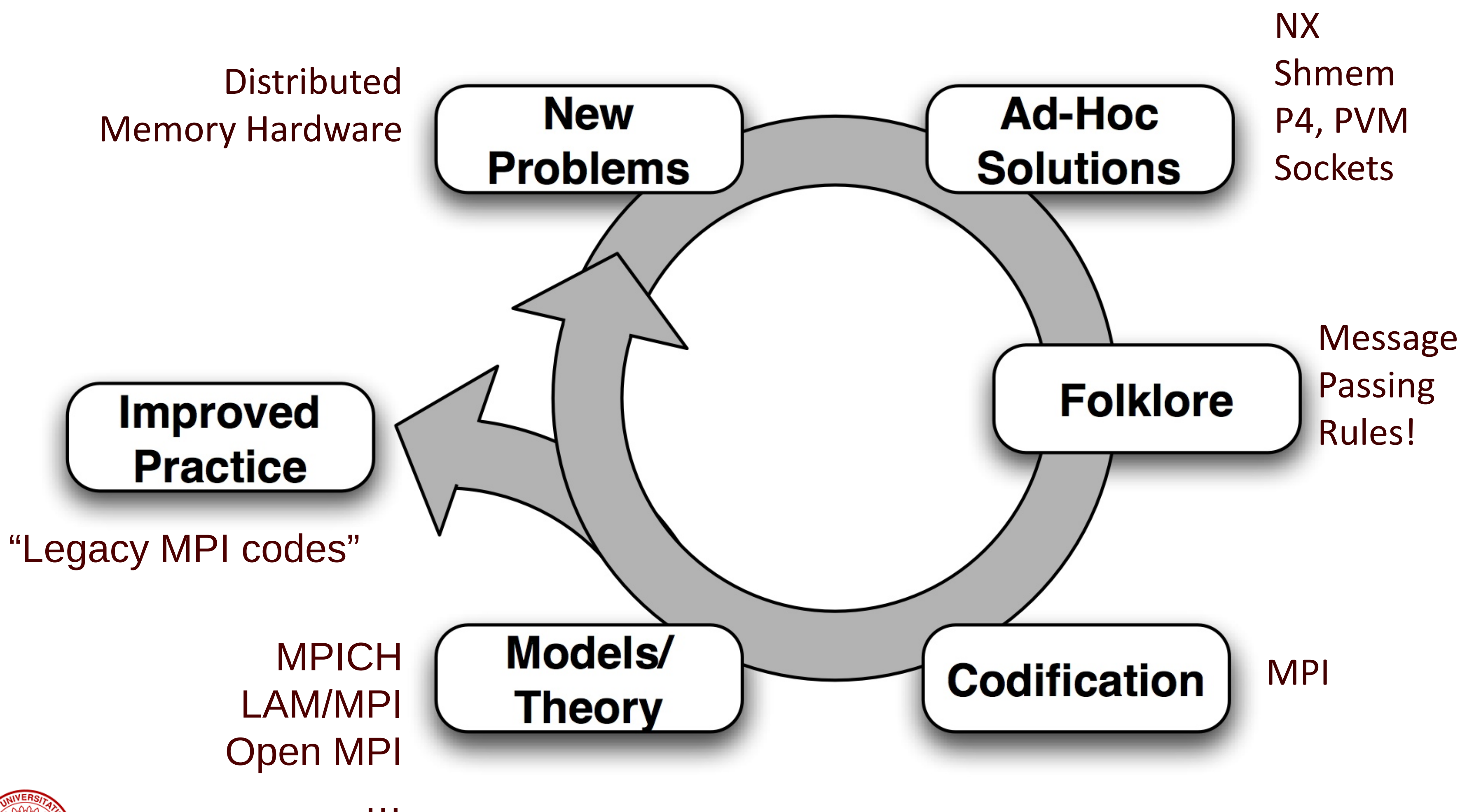
Virtuosos, talented amateurs
Extravagant use of materials
Design by intuition, brute force
Knowledge transmitted slowly, casually
Manufacture for use rather than sale

Cf. Shaw, Prospects for an engineering discipline of software, 1990.

Evolution of Software Practice



Why MPI Worked

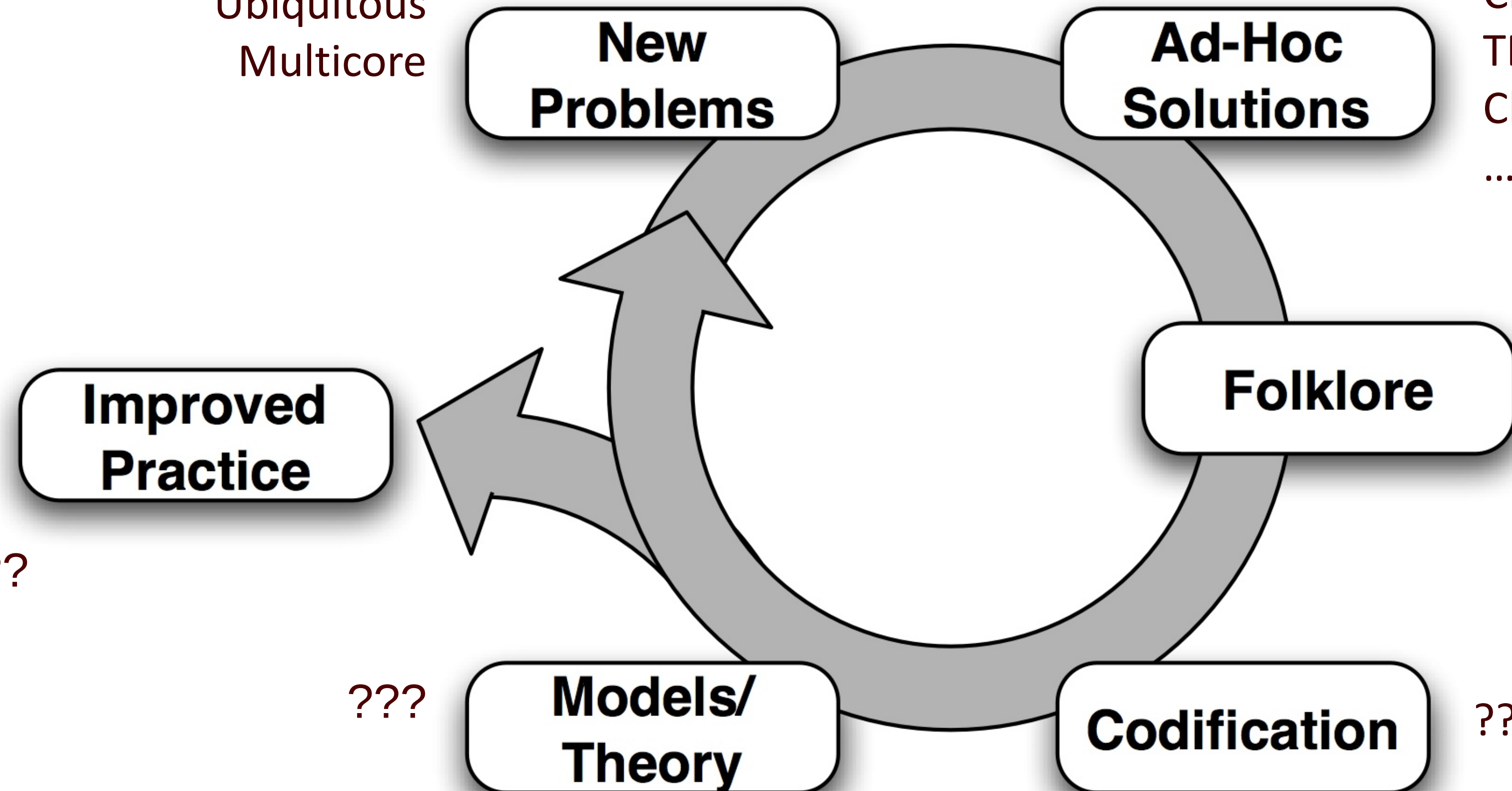


Today

Ubiquitous
Multicore

Pthreads
Cilk
TBB
Charm++
...

Tasks,
not threads

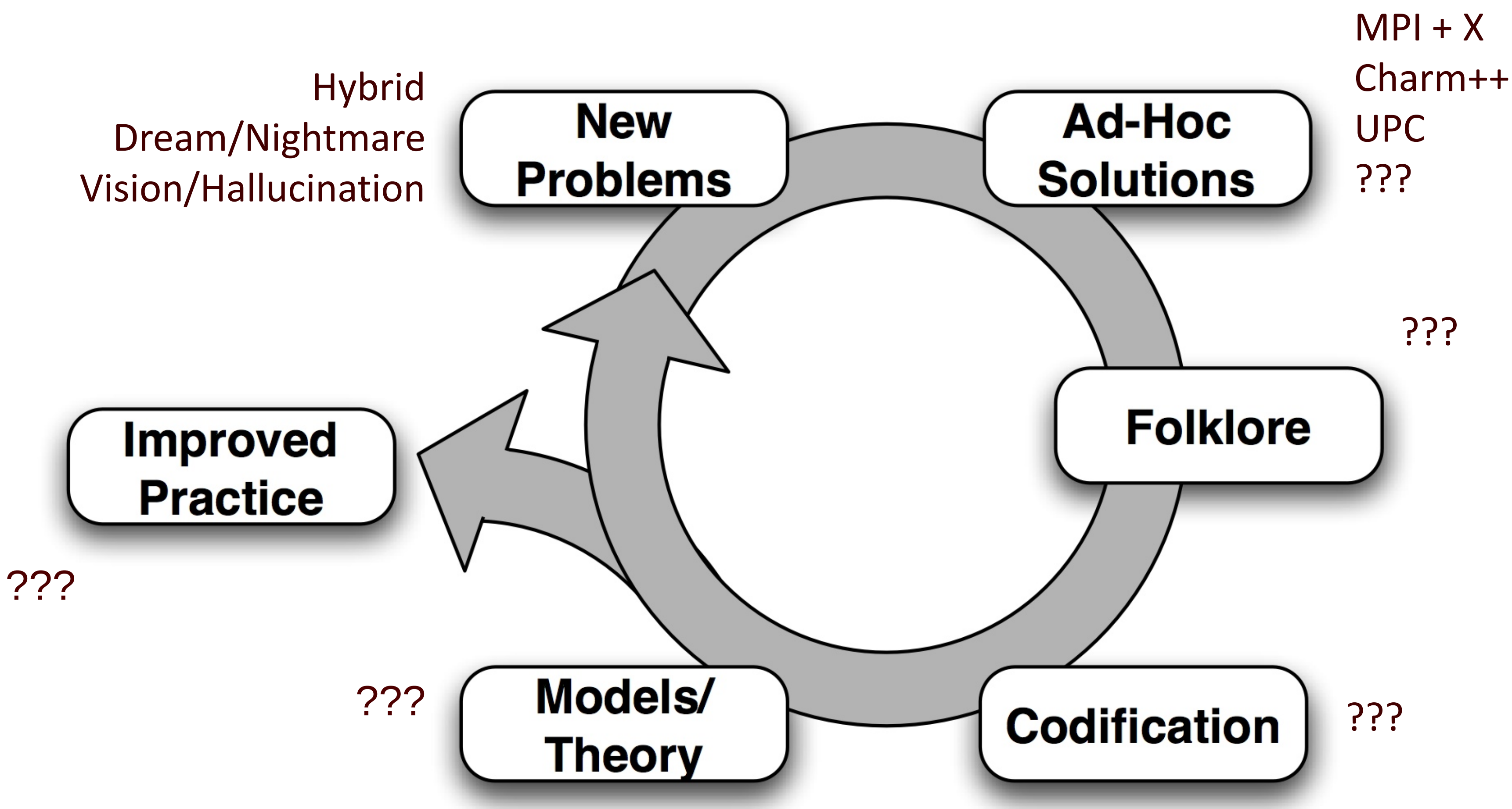


???

???

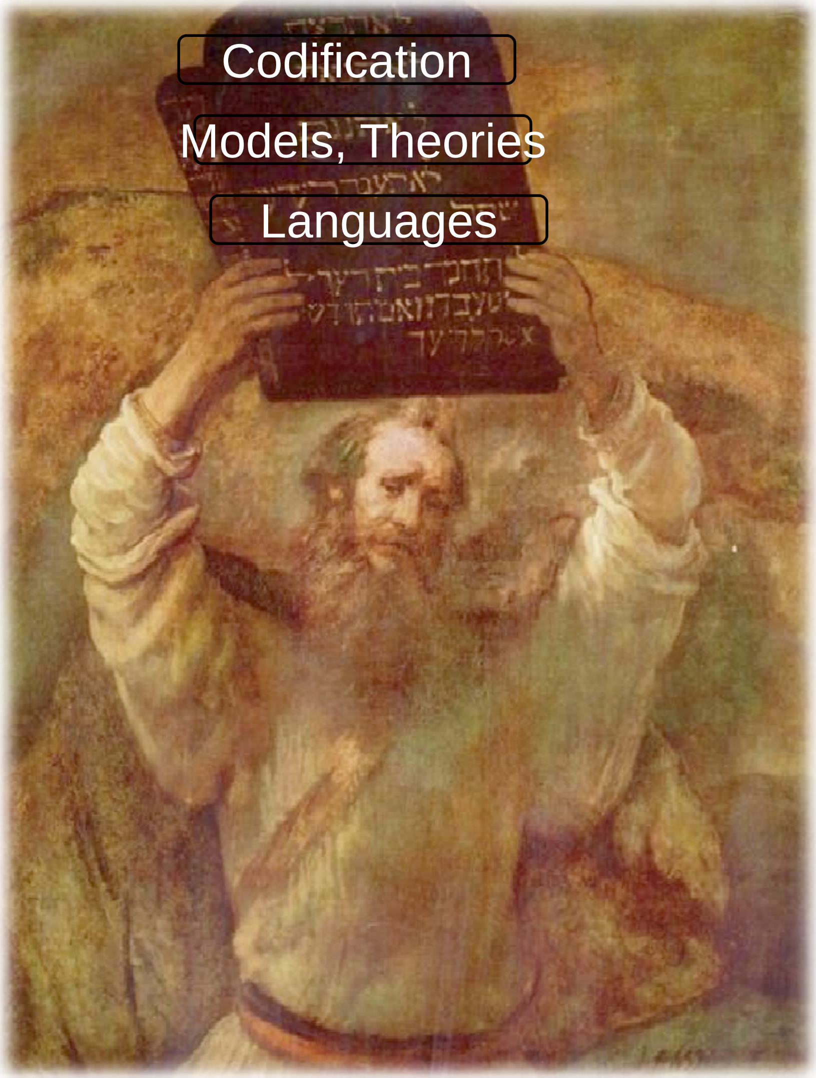
???

Tomorrow



Hybrid
Dream/Nightmare
Vision/Hallucination
New Problems
Ad-Hoc Solutions
MPI + X
Charm++
UPC
???
???
Folklore
Improved Practice
???
???
Models/ Theory
Codification
???

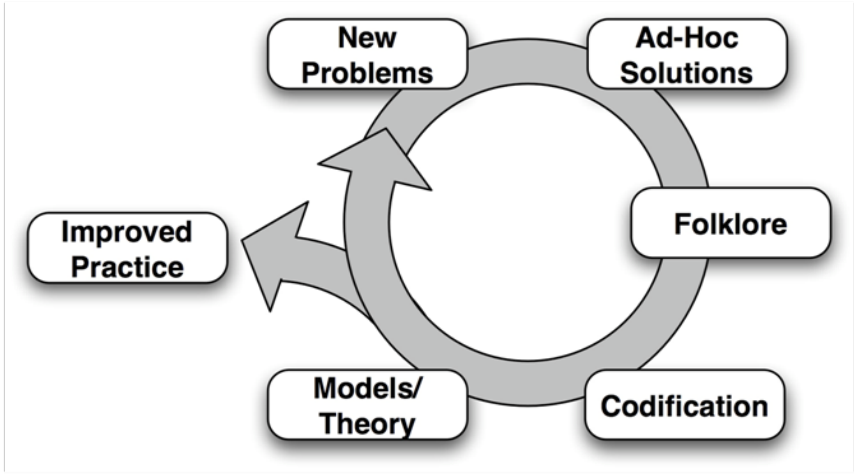
What Doesn't Work



Improved Practice

Performance with Elegance

- Construct high-performance (and elegant!) software that can evolve in robust fashion
- Must be an explicit goal



The Parallel Boost Graph Library

- ***Goal***: To build a generic library of efficient, scalable, distributed-memory parallel graph algorithms.

- ***Approach***: Apply advanced software paradigm (Generic Programming) to categorize and describe the domain of parallel graph algorithms. Separate concerns. Reuse sequential BGL software base.

- ***Result***: Parallel BGL. Saved years of effort.

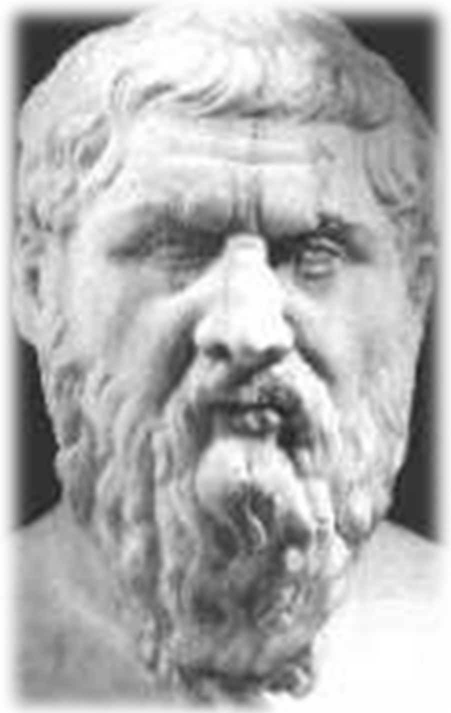
Graph Computations

- Irregular and unbalanced
- Non-local
- Data driven
- High data to computation ratio

- Intuition from solving PDEs may not apply

Generic Programming

- A methodology for the construction of reusable, efficient software libraries.
  - Dual focus on *abstraction* and *efficiency*.
  - Used in the C++ Standard Template Library
- *Platonic Idealism* applied to software
  - Algorithms are naturally abstract, generic (the “higher truth”)
  - Concrete implementations are just reflections (“concrete forms”)



Generic Programming Methodology

- Study the concrete implementations of an algorithm
- Lift away unnecessary requirements to produce a more abstract algorithm
  - Catalog these requirements.
  - Bundle requirements into concepts.
- Repeat the lifting process until we have obtained a generic algorithm that:
  - Instantiates to efficient concrete implementations.
  - Captures the essence of the "higher truth" of that algorithm.

Lifting Summation

```
int sum(int* array, int n) {
  int s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}
```

sum of an array of integers

Lifting Summation

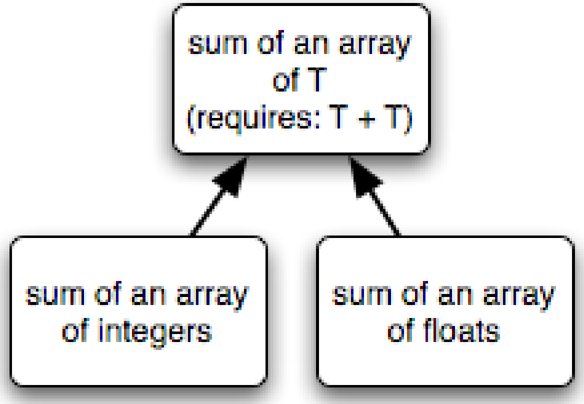
```
float sum(float* array, int n) {
  float s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}
```

sum of an array of integers

sum of an array of floats

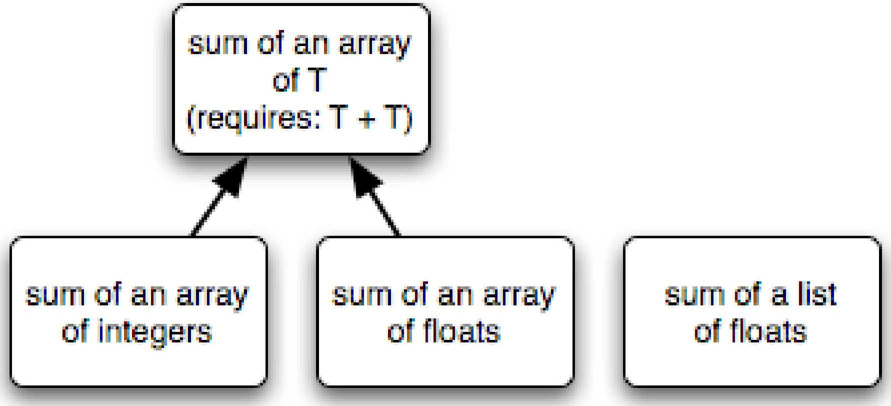
Lifting Summation

```
template<typename T>
T sum(T* array, int n) {
  T s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}
```



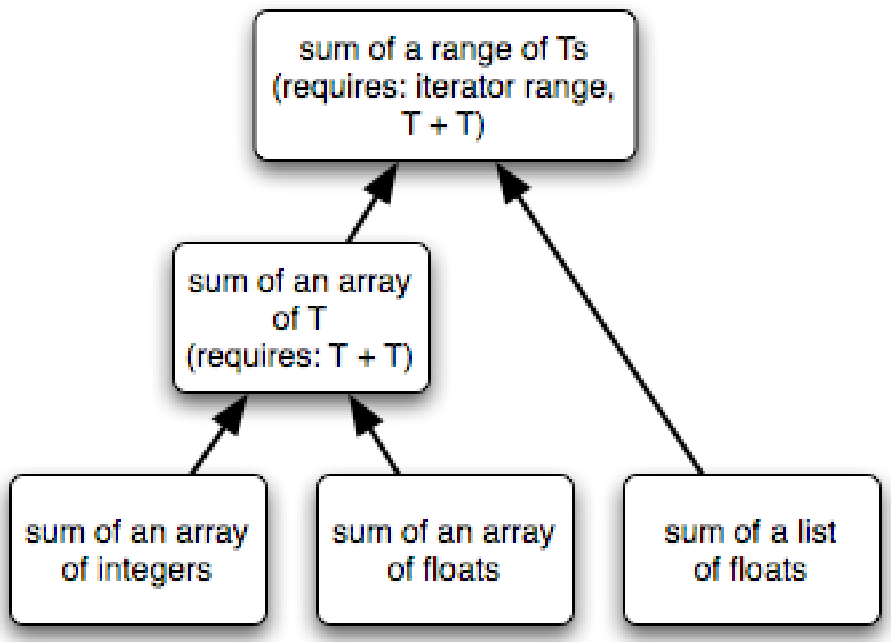
Lifting Summation

```
double sum(list_node* first, list_node* last) {
  double s = 0;
  while (first != last) {
    s = s + first->data;     first = first->next; }
  return s;
}
```

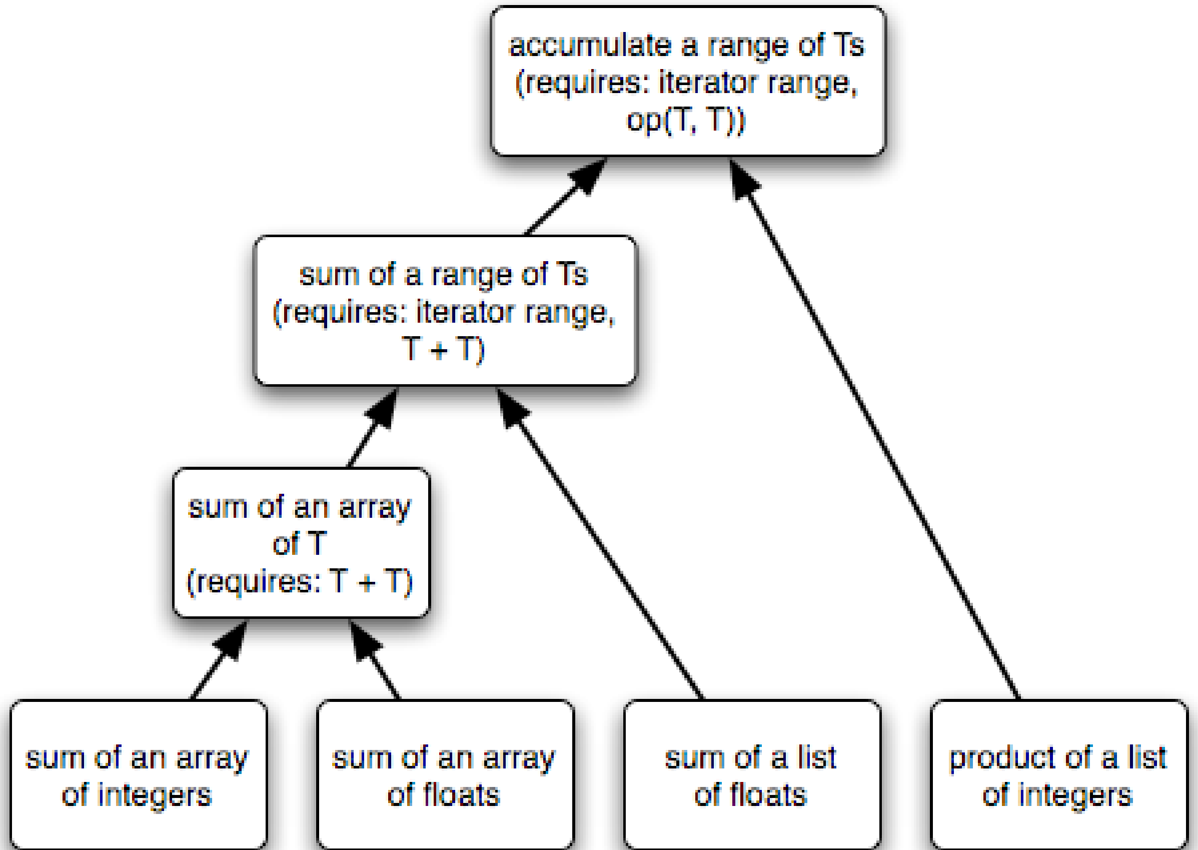


Lifting Summation

```
template <InputIterator Iter>
value_type sum(Iter first, Iter last) {
  value_type s = 0;
  while (first != last)
```



Lifting Summation



Generic Accumulate

```
template <InputIterator Iter, typename T,
                  typename Op>
T accumulate(Iter first, Iter last, T s, Op op) {
  while (first != last)
    s = op(s, *first++);
  return s;
}
```

- Generic form captures all accumulation:
  - Any kind of data (int, float, string)
  - Any kind of sequence (array, list, file, network)
  - Any operation (add, multiply, concatenate)
- Interface defined by ***concepts***
- Instantiates to efficient, concrete implementations

Specialization

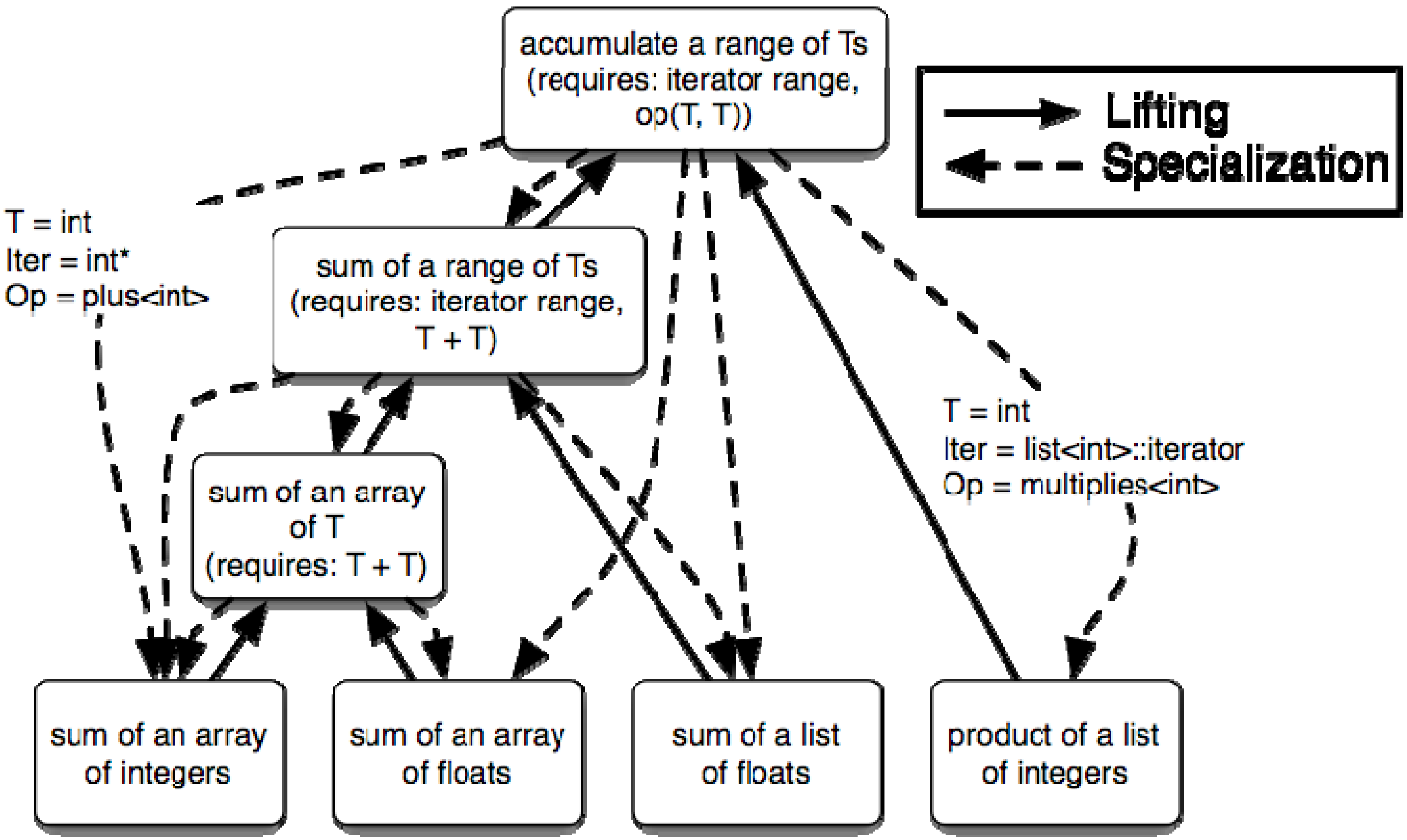
- Synthesizes efficient code for a particular **use** of a generic algorithm:

```
int array[20];
accumulate(array, array + 20, 0,
           std::plus<int>());
```

  - … generates the same code as our initial `sum` function for integer arrays.
- Specialization works by breaking down abstractions
  - Typically, replace type parameters with concrete types.
  - Lifting can only use abstractions that compiler optimizers can eliminate.

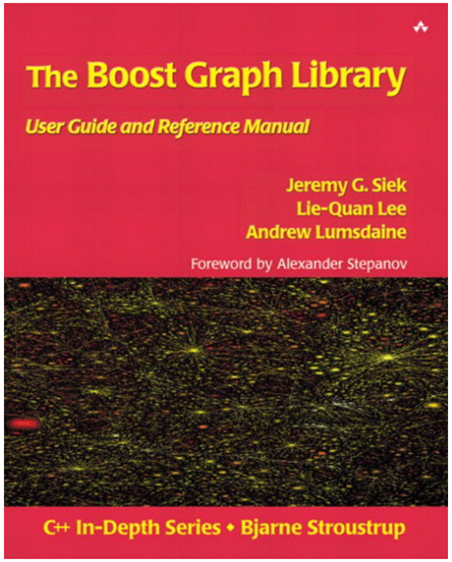
Lifting and Specialization

- Specialization is dual to lifting



The Boost Graph Library (BGL)

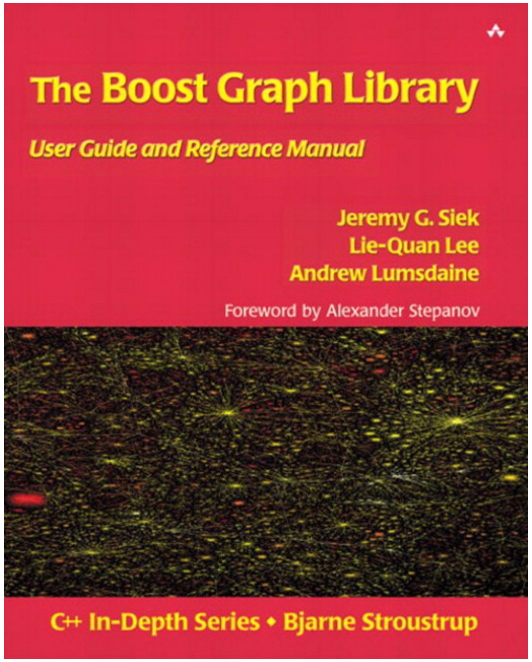
- A graph library developed with the generic programming paradigm



- Lift requirements on:
  - Specific graph structure
  - Edge and vertex types
  - Edge and vertex properties
  - Associating properties with vertices and edges
  - Algorithm-specific data structures (queues, etc.)

The Boost Graph Library (BGL)

- Comprehensive and mature
  - ~10 years of research and development
  - Many users, contributors outside of the OSL
  - Steadily evolving
- Written in C++
  - Generic
  - Highly customizable
  - Highly efficient
    - Storage and execution



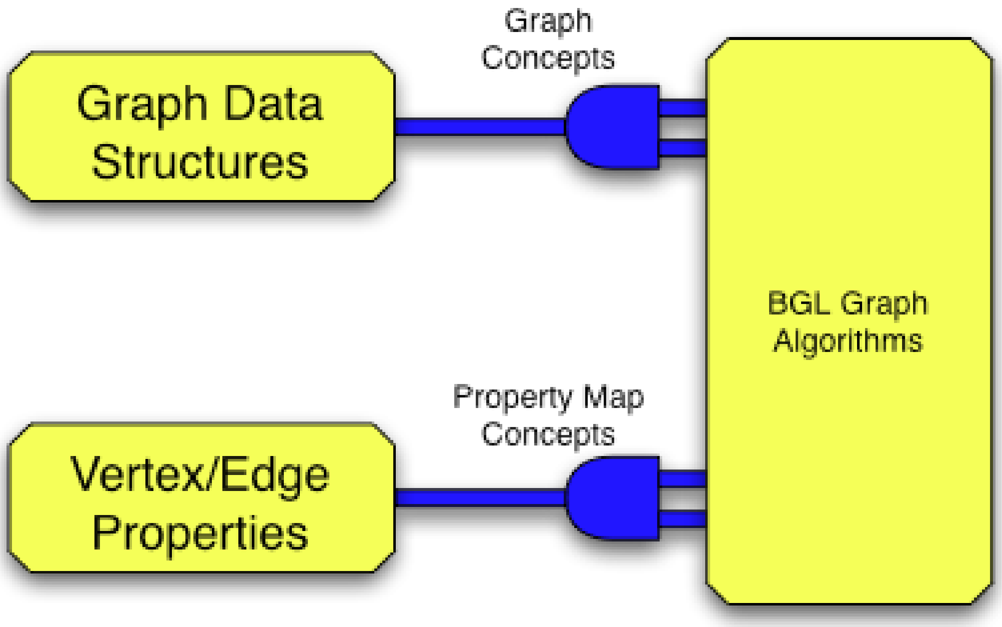
BGL: Algorithms (partial list)

- Searches (breadth-first, depth-first, A*)
- Single-source shortest paths (Dijkstra, Bellman-Ford, DAG)
- All-pairs shortest paths (Johnson, Floyd-Warshall)
- Minimum spanning tree (Kruskal, Prim)
- Components (connected, strongly connected, biconnected)
- Maximum cardinality matching
- Max-flow (Edmonds-Karp, push-relabel)
- Sparse matrix ordering (Cuthill-McKee, King, Sloan, minimum degree)
- Layout (Kamada-Kawai, Fruchterman-Reingold, Gursoy-Atun)
- Betweenness centrality
- PageRank
- Isomorphism
- Vertex coloring
- Transitive closure
- Dominator tree

BGL: Graph Data Structures

- Graphs:
  - `adjacency_list`: highly configurable with user-specified containers for vertices and edges
  - `adjacency_matrix`
  - `compressed_sparse_row`
- Adaptors:
  - subgraphs, filtered graphs, reverse graphs
  - LEDA and Stanford GraphBase
- Or, use your own…

BGL Architecture



Parallelizing the BGL

- Starting with the sequential BGL…

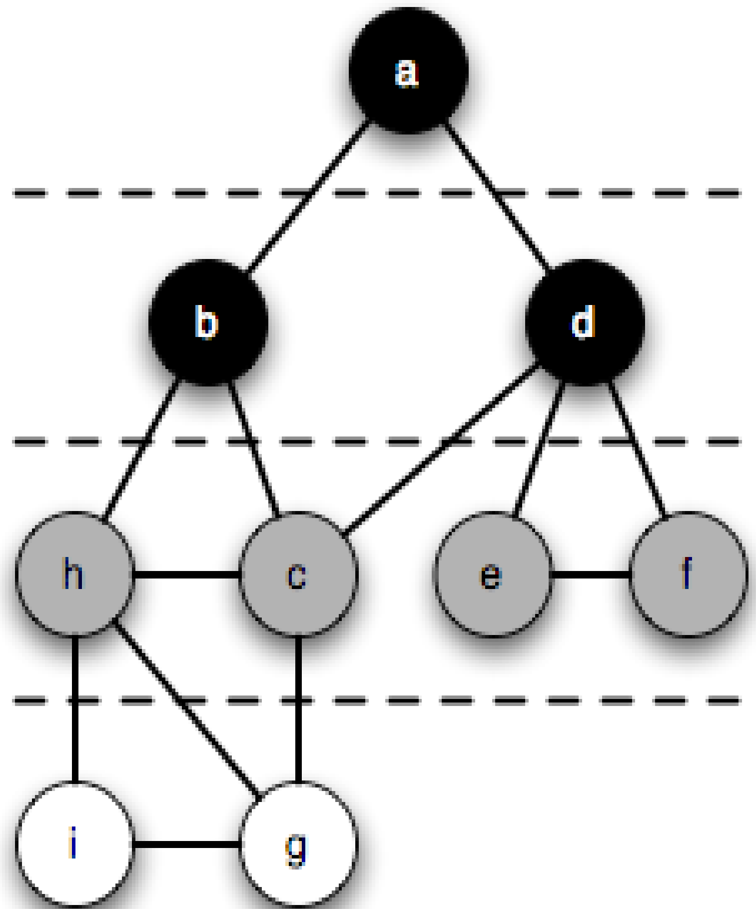
- Three ways to build new algorithms or data structures
  1. *Lift* away restrictions that make the component sequential (unifying parallel and sequential)
  2. *Wrap* the sequential component in a distribution-aware manner.
  3. *Implement* any entirely new, parallel component.

Lifting for Parallelism

- Remove assumptions made by most sequential algorithms:
  - A single, shared address space.
  - A single “thread” of execution.
- Platonic ideal: unify parallel and sequential algorithms
- Our goal: Build the Parallel BGL by lifting the sequential BGL.

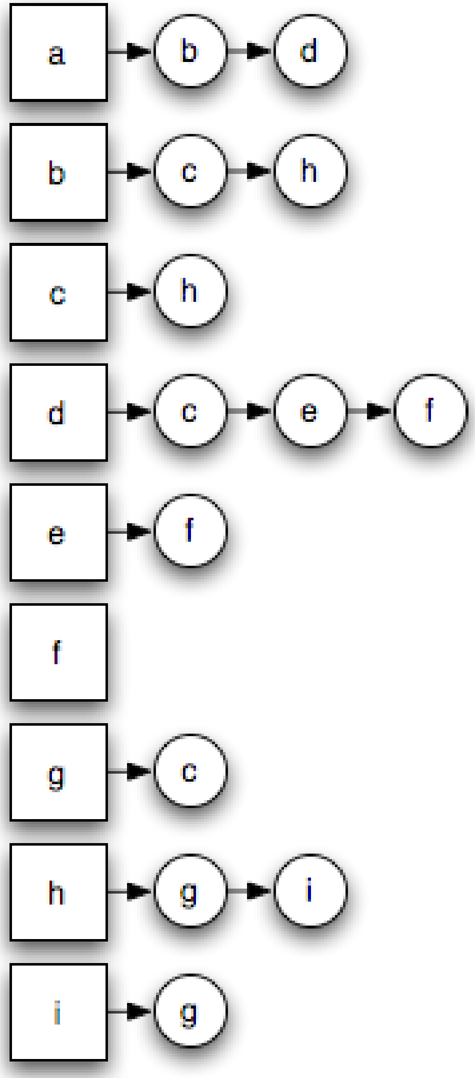
Breadth-First Search

```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
  Vertex u = Q.top(); Q.pop();
  for (tie(ei, ei_end) = out_edges(u, g);
    Vertex v = target(*ei, g);
    ColorValue v_color = get(color, v);
    if (v_color == Color::white()) {
      put(color, v, Color::gray());
      Q.push(v);
    } else {
      if (v_color == Color::gray())
      else
    }
  }
  put(color, u, Color::black());
}
```



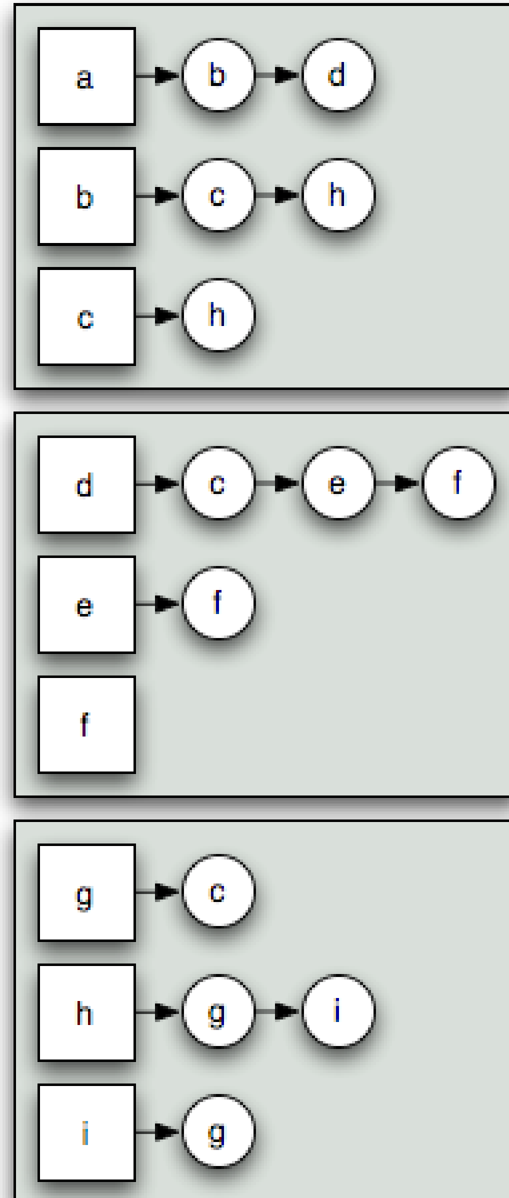
Parallellizing BFS?

```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
  Vertex u = Q.top(); Q.pop();
  for (tie(ei, ei_end) = out_edges(u, g); ei != ei_er
    Vertex v = target(*ei, g);
    ColorValue v_color = get(color, v);
    if (v_color == Color::white()) {
      put(color, v, Color::gray());
      Q.push(v);
    } else {
      if (v_color == Color::gray())
      else
    }
  }
  put(color, u, Color::black());
}
```



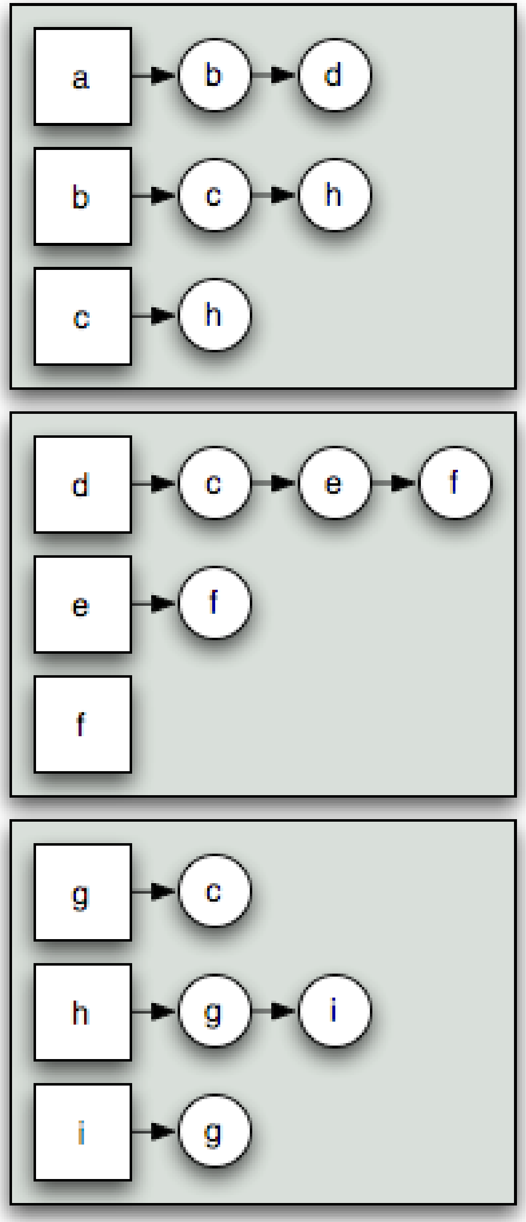
Parallellizing BFS?

```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
  Vertex u = Q.top(); Q.pop();
  for (tie(ei, ei_end) = out_edges(u, g); ei != ei_e
    Vertex v = target(*ei, g);
    ColorValue v_color = get(color, v);
    if (v_color == Color::white()) {
      put(color, v, Color::gray());
      Q.push(v);
    } else {
      if (v_color == Color::gray())
      else
    }
  }
  put(color, u, Color::black());
}
```



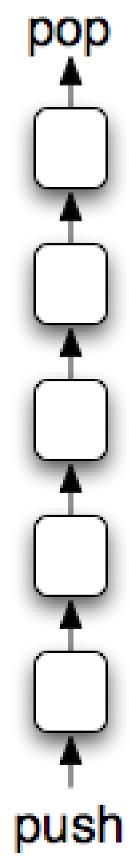
Distributed Graph

- One fundamental operation:
  - Enumerate out-edges of a given vertex
- Distributed adjacency list:
  - Distribute vertices
  - Out-edges stored with the vertices



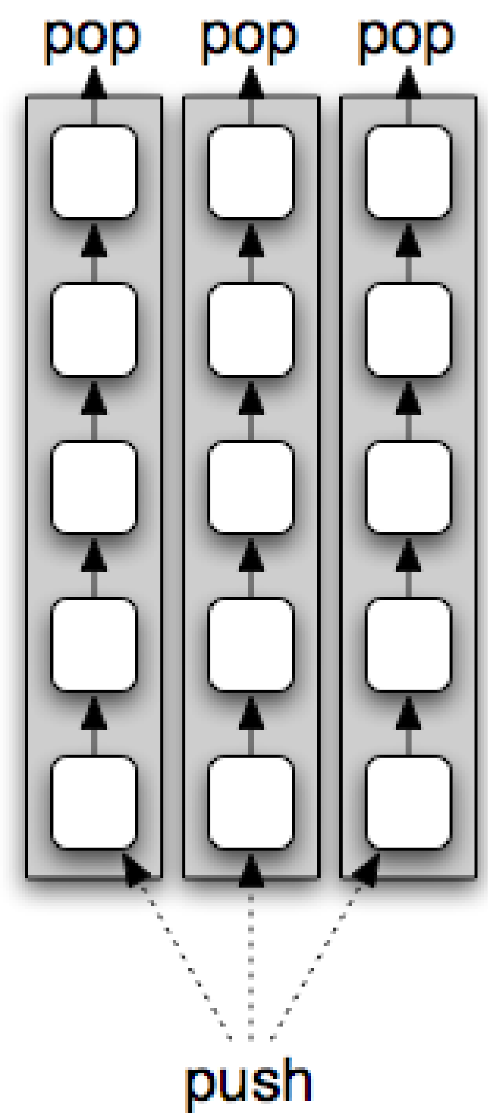
Parallellizing BFS?

```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
  Vertex u = Q.top(); Q.pop();
  for (tie(ei, ei_end) = out_edges(u, g); ei != ei_end; ++ei) {
    Vertex v = target(*ei, g);
    ColorValue v_color = get(color, v);
    if (v_color == Color::white()) {
      put(color, v, Color::gray());
      Q.push(v);
    } else {
      if (v_color == Color::gray())
      else
    }
  }
  put(color, u, Color::black());
}
```



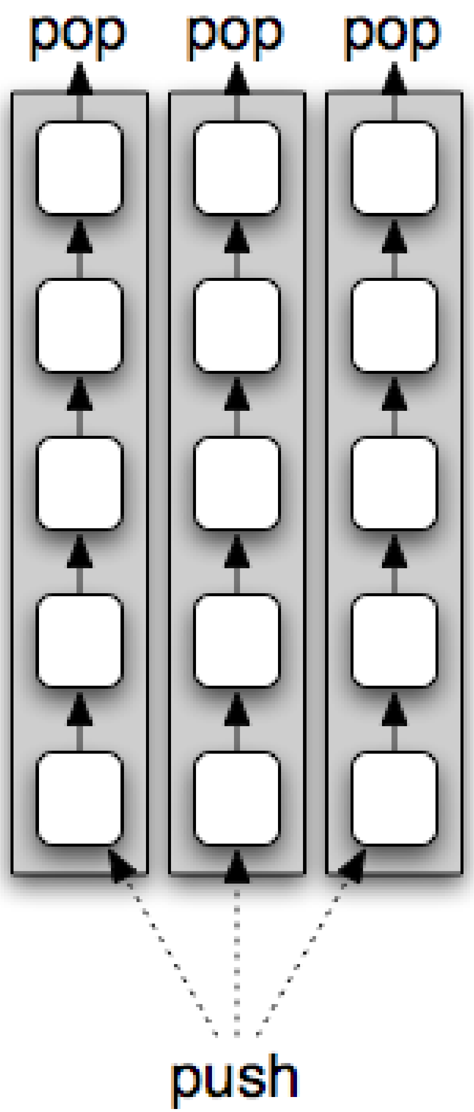
Parallellizing BFS?

```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
  Vertex u = Q.top(); Q.pop();
  for (tie(ei, ei_end) = out_edges(u, g); ei != ei_end;
    Vertex v = target(*ei, g);
    ColorValue v_color = get(color, v);
    if (v_color == Color::white()) {
      put(color, v, Color::gray());
      Q.push(v);
    } else {
      if (v_color == Color::gray())
      else
    }
  }
  put(color, u, Color::black());
}
```



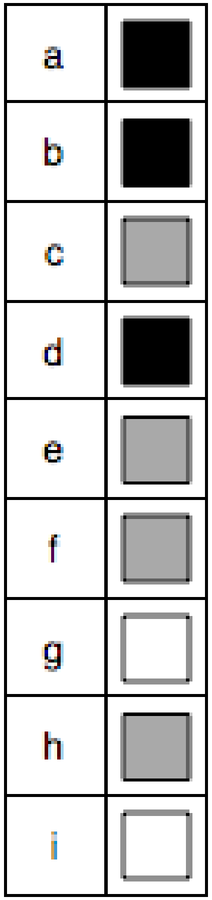
Distributed Queue

- Three fundamental operations:
  - ***top/pop*** retrieves from queue
  - ***push*** operation adds to queue
  - ***empty*** operation signals termination
- Distributed queue:
  - Separate, local queues
  - ***top/pop*** from local queue
  - ***push*** sends to a remote queue
  - ***empty*** waits for remote sends



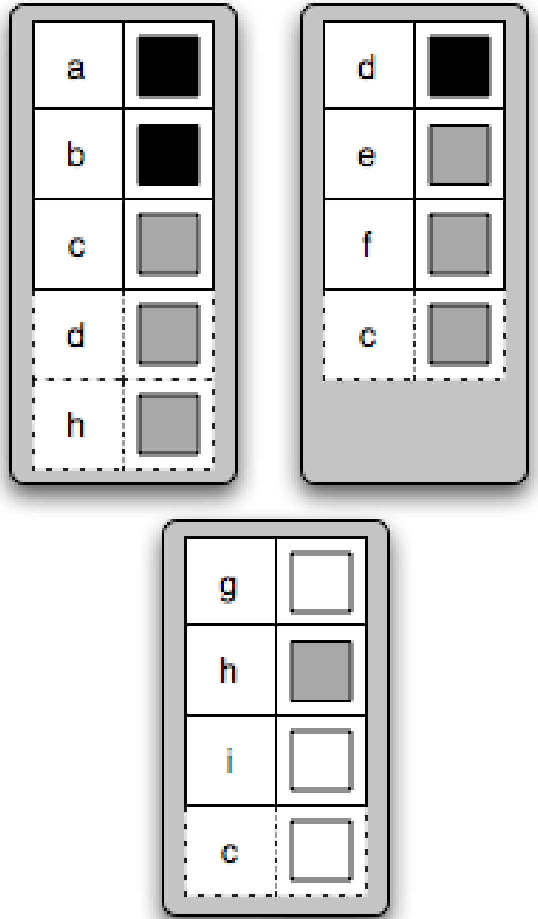
Parallellizing BFS?

```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
   Vertex u = Q.top(); Q.pop();
   for (tie(ei, ei_end) = out_edges(u, g); ei != ei_end; ++ei) {
      Vertex v = target(*ei, g);
      ColorValue v_color = get(color, v);
      if (v_color == Color::white()) {
         put(color, v, Color::gray());
         Q.push(v);
      } else {
         if (v_color == Color::gray())
         else
      }
   }
   put(color, u, Color::black());
}
```



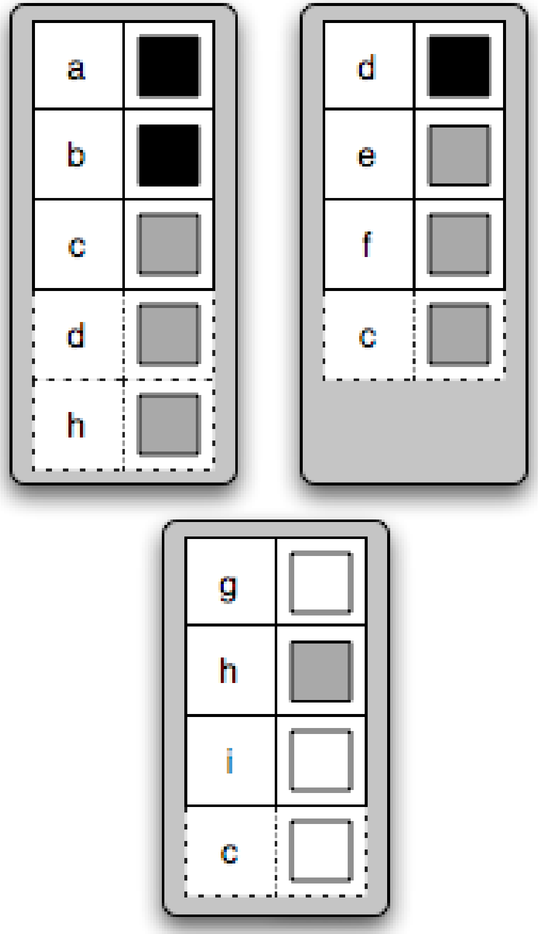
Parallellizing BFS?

```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
  Vertex u = Q.top(); Q.pop();
  for (tie(ei, ei_end) = out_edges(u, g); ei != ei_er
    Vertex v = target(*ei, g);
    ColorValue v_color = get(color, v);
    if (v_color == Color::white()) {
      put(color, v, Color::gray());
      Q.push(v);
    } else {
      if (v_color == Color::gray())
      else
    }
  }
  put(color, u, Color::black());
}
```



Distributed Property Maps

- Two fundamental operations:
  - ***put*** sets the value for a vertex/edge
  - ***get*** retrieves the value
- Distributed property map:
  - Store data on same processor as vertex or edge
  - ***put/get*** send messages
  - Ghost cells cache remote values
  - Resolver combines **puts**



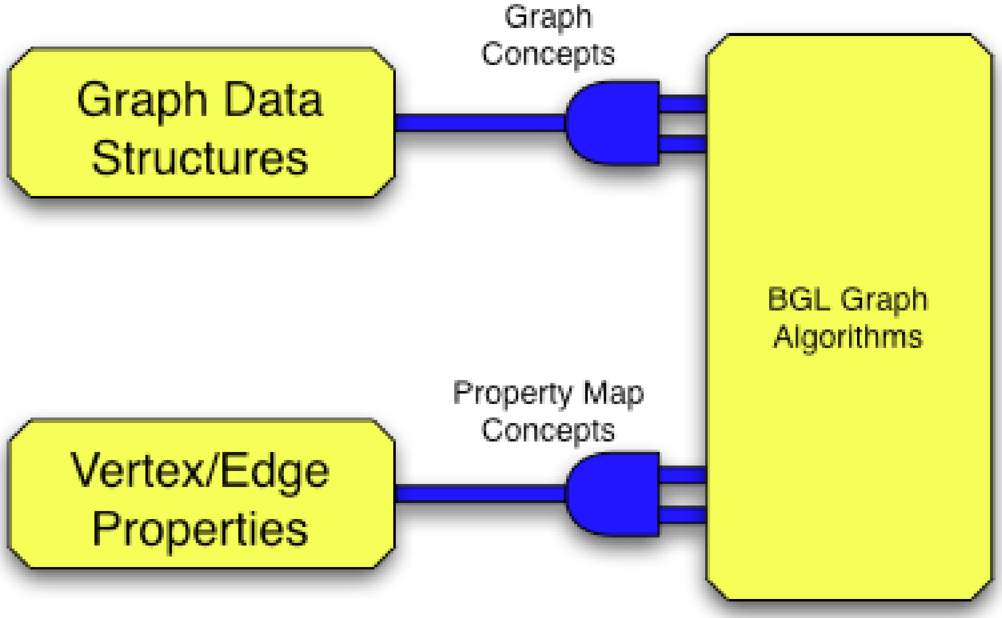
“Implementing” Parallel BFS

- Generic interface from the Boost Graph Library

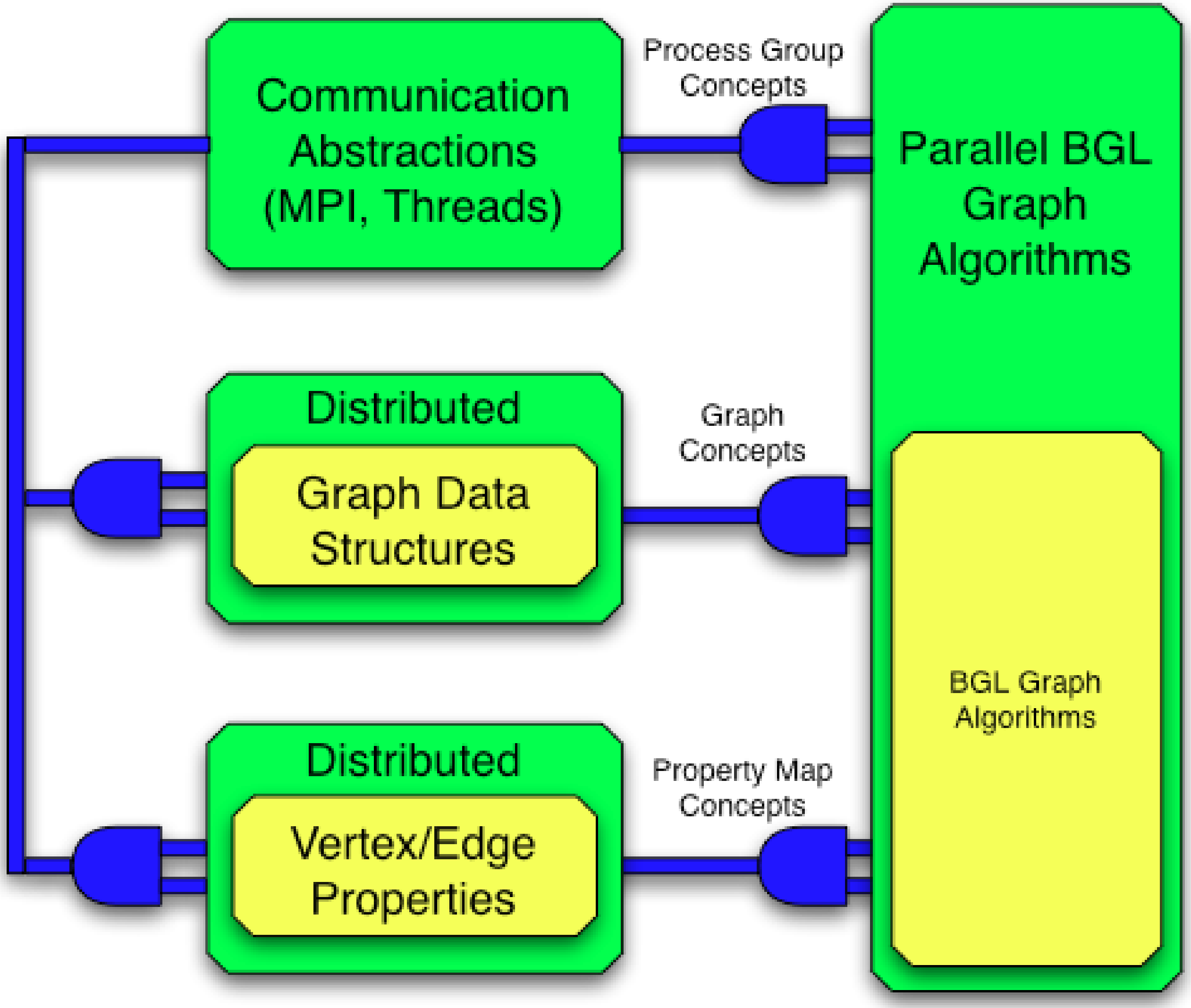
```
template<class IncidenceGraph, class Queue, class BFSVisitor,
         class ColorMap>
void breadth_first_search(const IncidenceGraph& g,
                          vertex_descriptor s, Queue& Q,
                          BFSVisitor vis, ColorMap color);
```

- Effect parallelism by using appropriate types:
  - Distributed graph
  - Distributed queue
  - Distributed property map
- Our sequential implementation is also parallel!
  - Parallel BGL can just “wrap up” sequential BFS

BGL Architecture



Parallel BGL Architecture



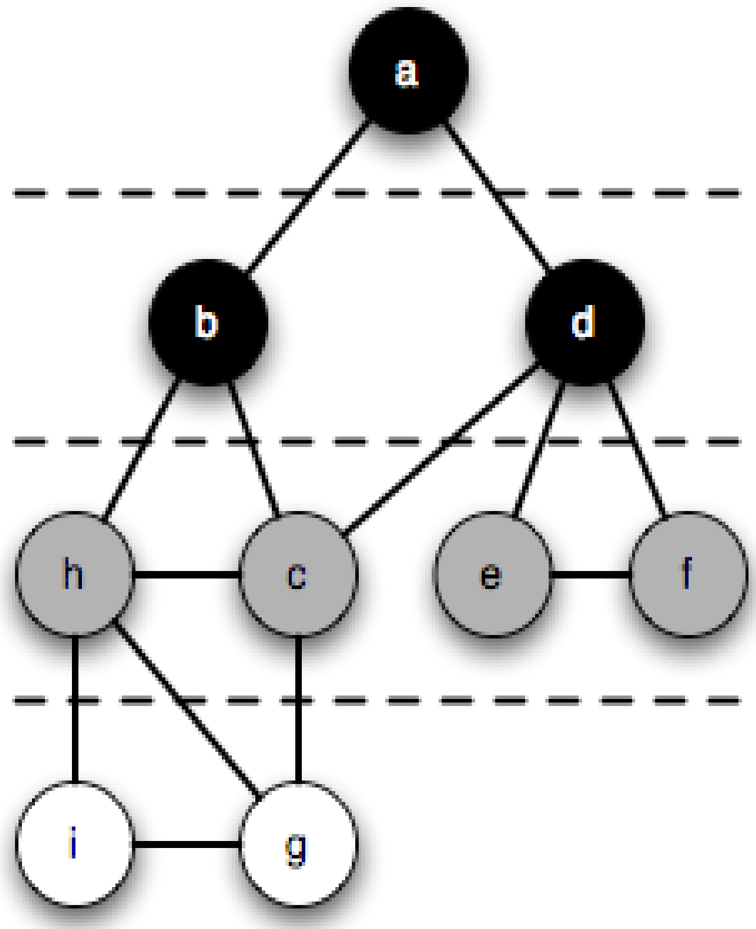
Algorithms in the Parallel BGL

- Breadth-first search*
- Eager Dijkstra's single-source shortest paths*
- Crauser et al. single-source shortest paths*
- Depth-first search
- Minimum spanning tree (Boruvka*, Dehne & Götz‡)
- Connected components‡
- Strongly connected components†
- Biconnected components
- PageRank*
- Graph coloring
- Fruchterman-Reingold layout*
- Max-flow†

* Algorithms that have been lifted from a sequential implementation
† Algorithms built on top of parallel BFS
‡ Algorithms built on top of their sequential counterparts

Lifting for Hybrid Programming?

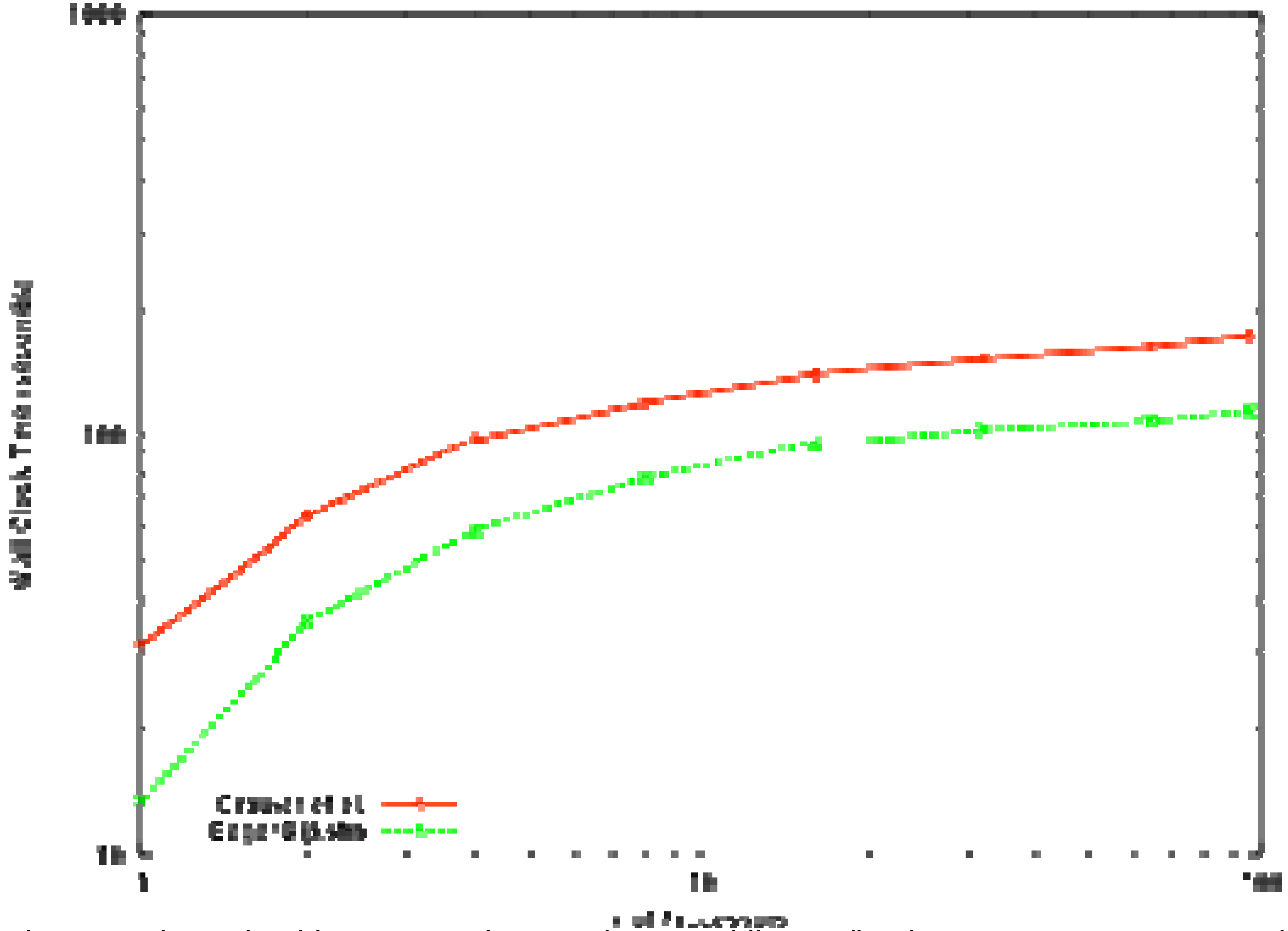
```
put(color, s, Color::gray());
Q.push(s);
while (! Q.empty()) {
   Vertex u = Q.top(); Q.pop();
   for (tie(ei, ei_end) = out_edges(u, g);
      Vertex v = target(*ei, g);
      ColorValue v_color = get(color, v);
      if (v_color == Color::white()) {
         put(color, v, Color::gray());
         Q.push(v);
      } else {
         if (v_color == Color::gray())
         else
      }
   }
   put(color, u, Color::black());
}
```



Abstraction and Performance

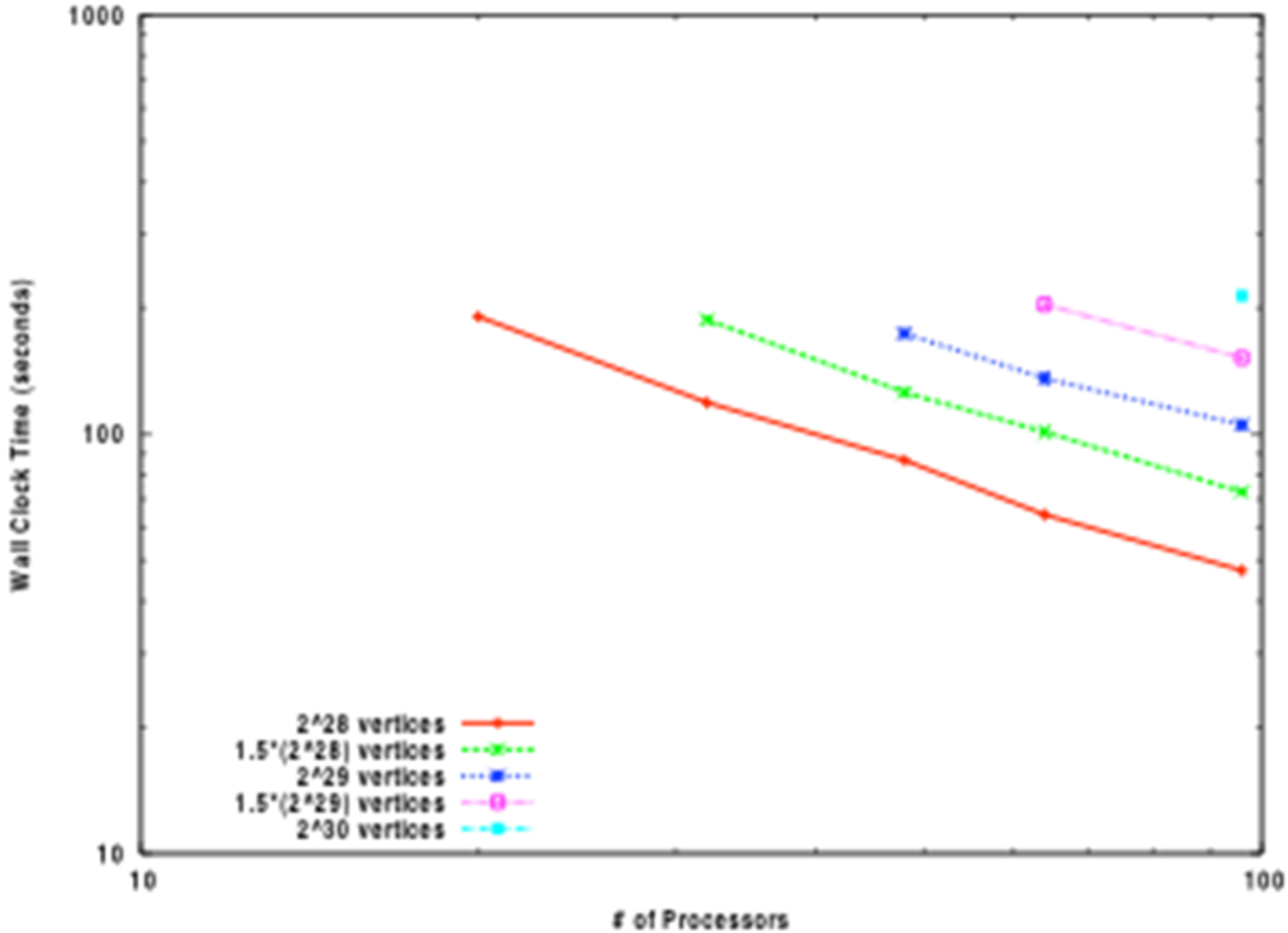
- ***Myth***: Abstraction is the enemy of performance.
- The BGL sparse-matrix ordering routines perform on par with hand-tuned Fortran codes.
  - Other generic C++ libraries have had similar successes (MTL, Blitz++, POOMA)
- ***Reality***: Poor use of abstraction can result in poor performance.
  - Use abstractions the compiler can eliminate.

Weak Scaling Dijkstra SSSP



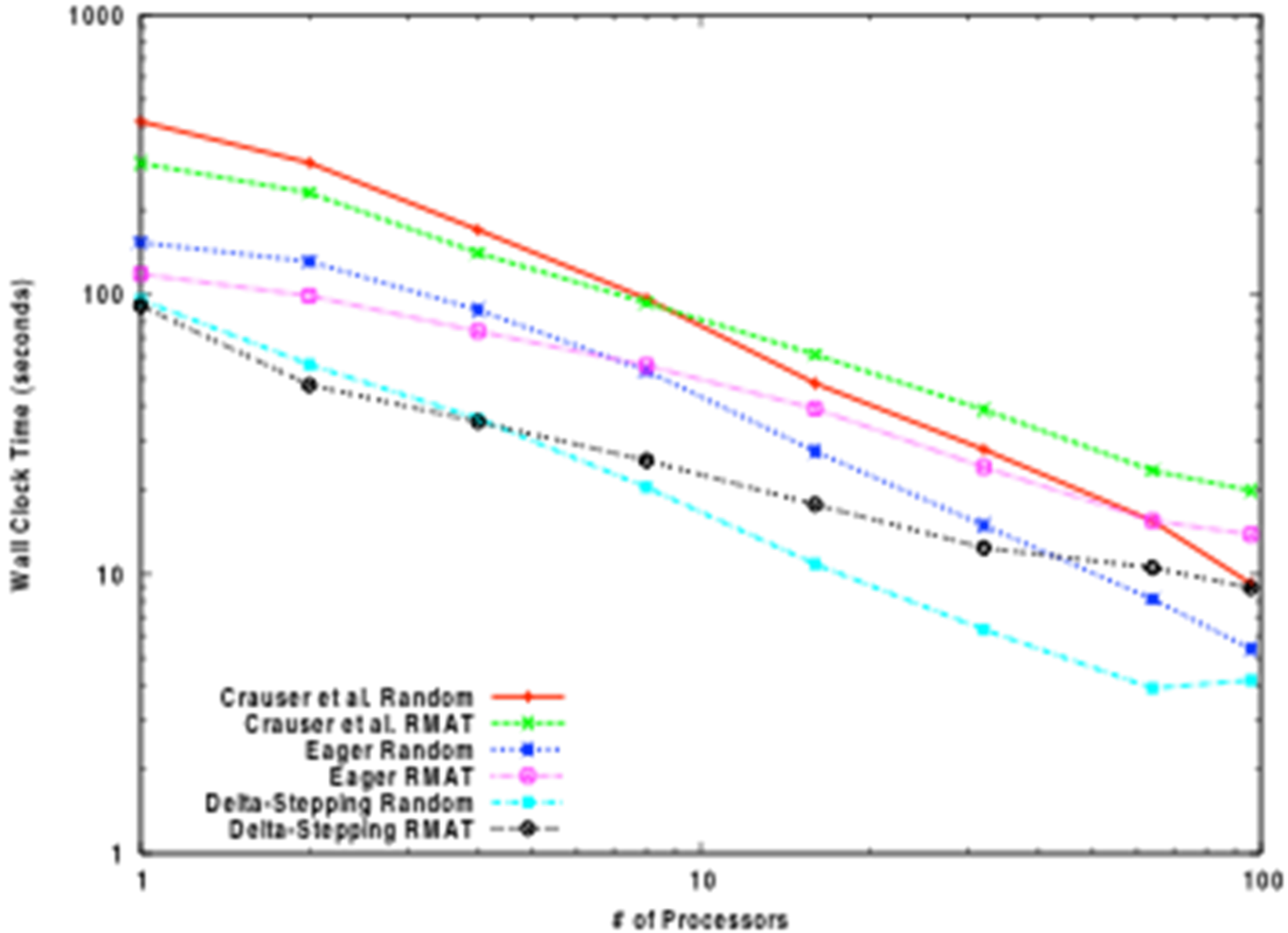
Erdos-Renyi graph with 2.5M vertices and 12.5M (directed) edges per processor. Maximum graph size is 240M vertices and 1.2B edges on 96 processors.

Strong Scaling Delta Stepping



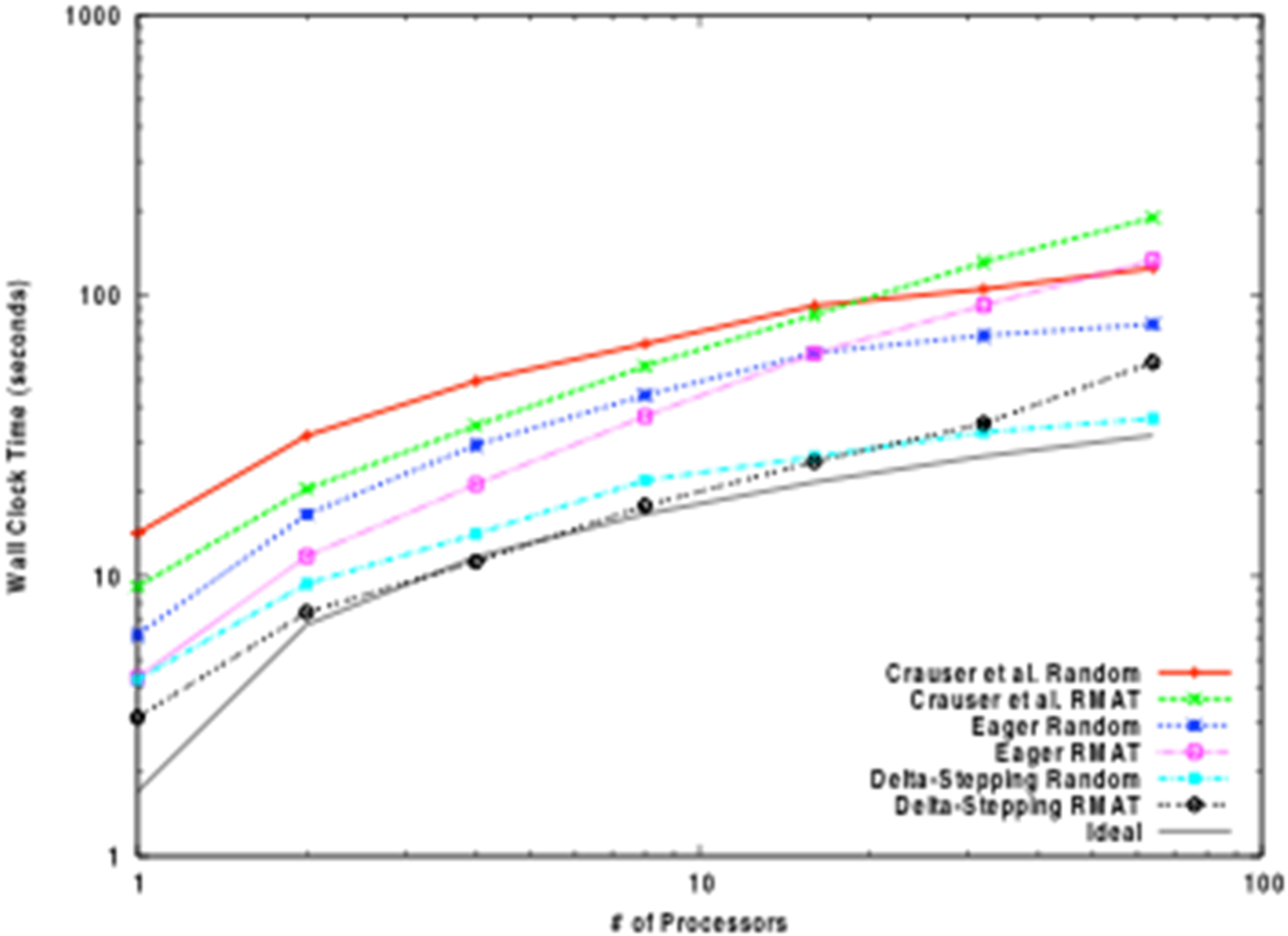
Delta-Stepping on an Erdos-Renyi graph with average degree 4. The largest problem solved is 1B vertices and 4B edges using 96 processors.

Strong Scaling



Performance of three SSSP algorithms on fixed-sized graphs with ~24M vertices and ~58M edges

Weak Scaling



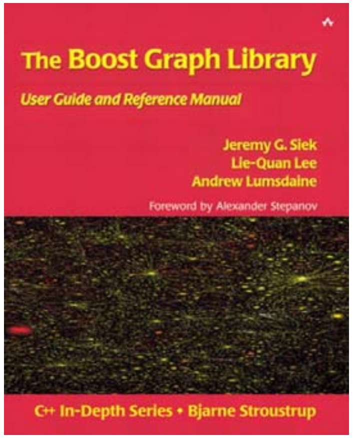
Weak scalability of three SSSP algorithms using graphs with an average of 1M vertices and 10M edges per processor.

The BGL Family

- The Original (sequential) BGL
- BGL-Python
- The Parallel BGL
- Parallel BGL-Python
- (Parallel) BGL-VTK

For More Information…

- (Sequential) Boost Graph Library
  http://www.boost.org/libs/graph/doc
- Parallel Boost Graph Library
  http://www.osl.iu.edu/research/pbgl
- Python Bindings for (Parallel) BGL
  http://www.osl.iu.edu/~dgregor/bgl-python
- Contacts:
  - Andrew Lumsdaine lums@osl.iu.edu
  - Jeremiah Willcock jewillco@osl.iu.edu
  - Nick Edmonds ngedmonds@osl.iu.edu



pervasivetechnologylabs
AT INDIANA UNIVERSITY

Summary

- Effective software practices evolve from effective software practices
  - Explicitly study this in context of HPC
- Parallel BGL
  - Generic parallel graph algorithms for distributed-memory parallel computers
  - Reusable for different applications, graph structures, communication layers, etc
  - Efficient, scalable

Questions?

Disclaimer

- Some images in this talk were cut and pasted from web sites found with Google Image Search and are used without permission. I claim their inclusion in this talk is permissible as fair use.
- Please do not redistribute this talk.