High-Performance Distributed Memory Graph Computations

Andrew Lumsdaine
Indiana University
lums@osl.iu.edu

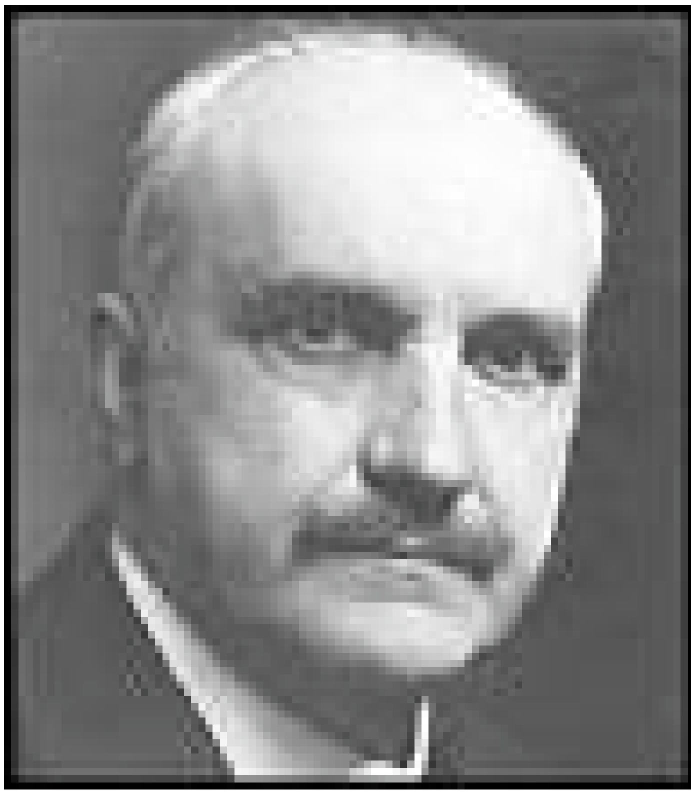
Introduction

- Overview of our high-performance, industrial strength, graph library
  - Comprehensive features
  - Impressive results
- Lessons on software use and reuse

Advancing Scientific 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 Scientific Software



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

Advancing Scientific Software

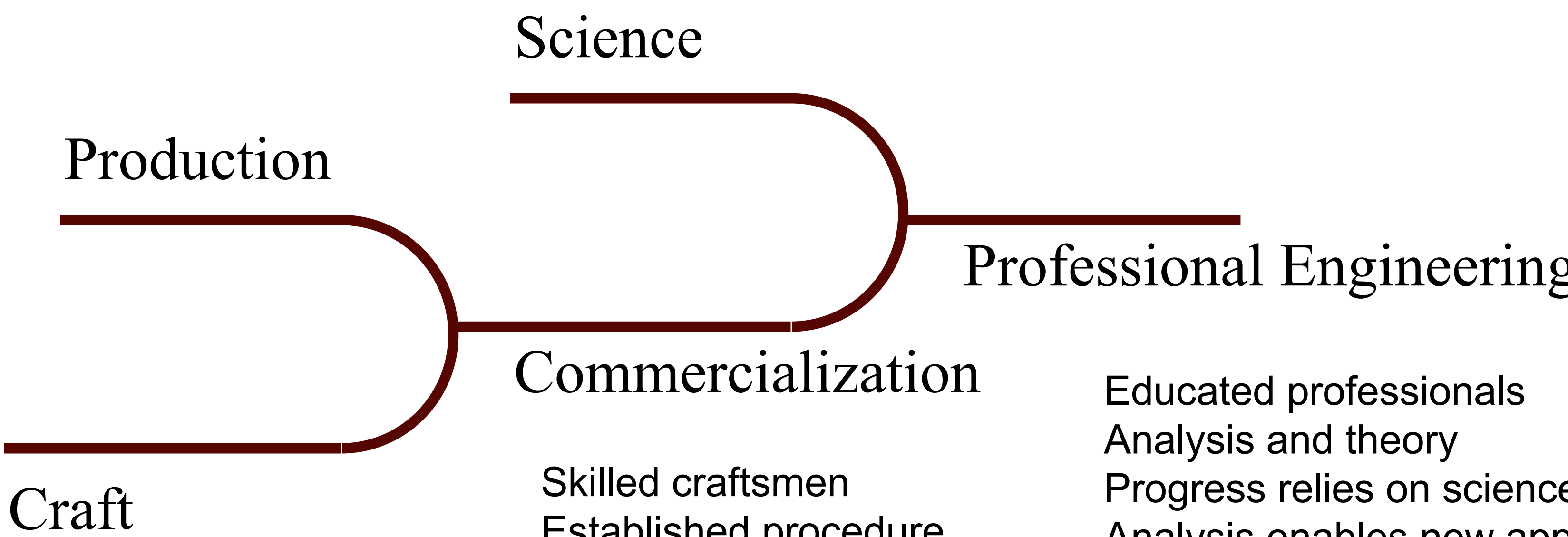
- Name the two most important pieces of scientific software over last 20 years
  - BLAS
  - MPI
- Why are these so important?
- Why did they succeed?

MPI is the Worst Way to Program

Except for all the others!

pervasivetechnologylabs
AT INDIANA UNIVERSITY

Evolution of a Discipline



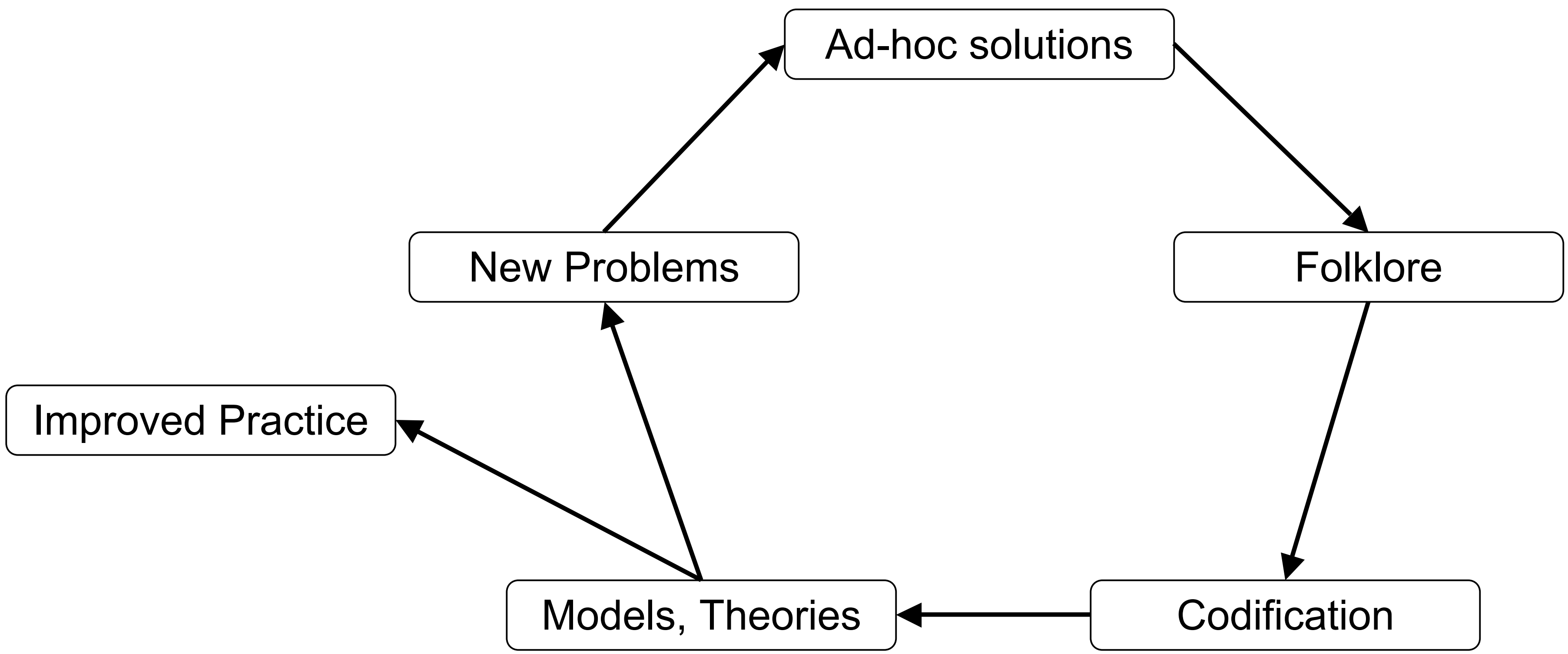
Virtuosos, talented amateurs
Extravagant use of materials
Design by intuition, brute force
Knowledge transmitted slowly, casually
Manufacture for use rather than sale

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

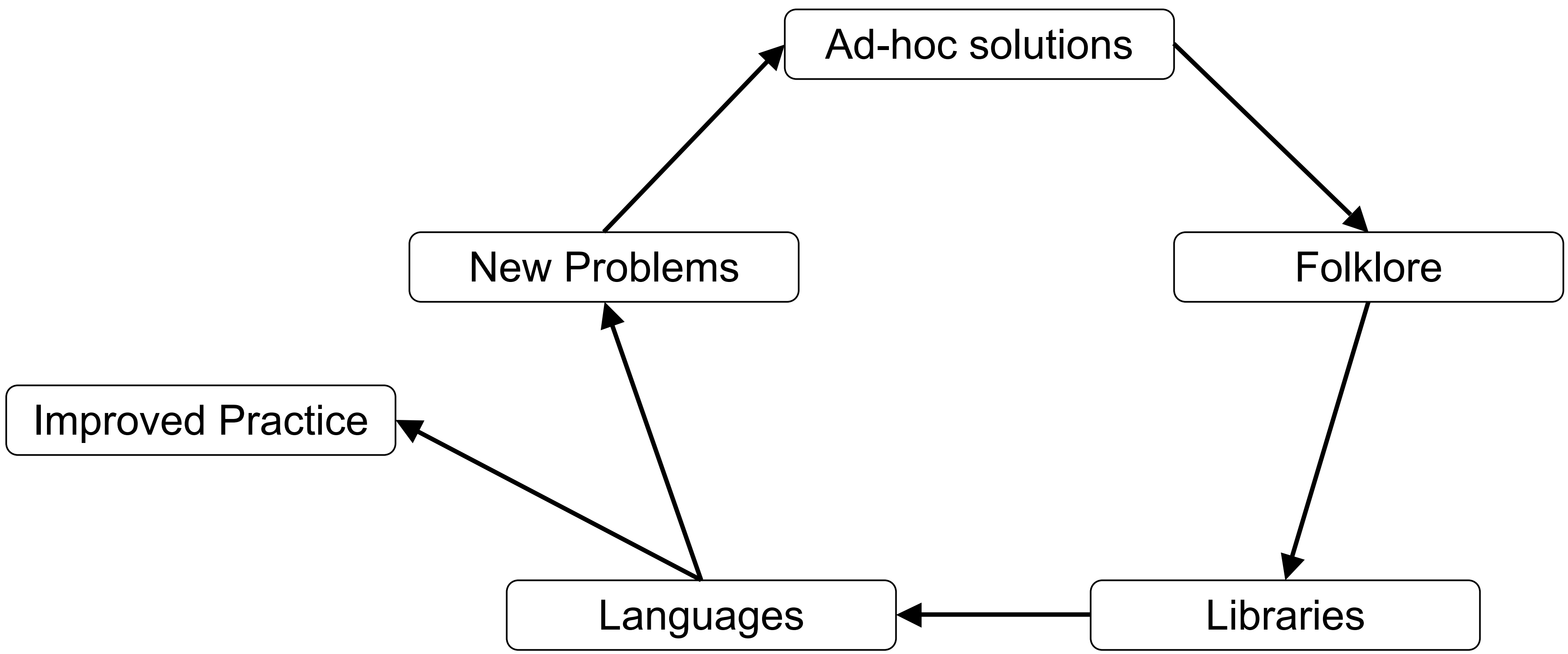
Educated professionals
Analysis and theory
Progress relies on science
Analysis enables new apps
Market segmented by
product variety

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

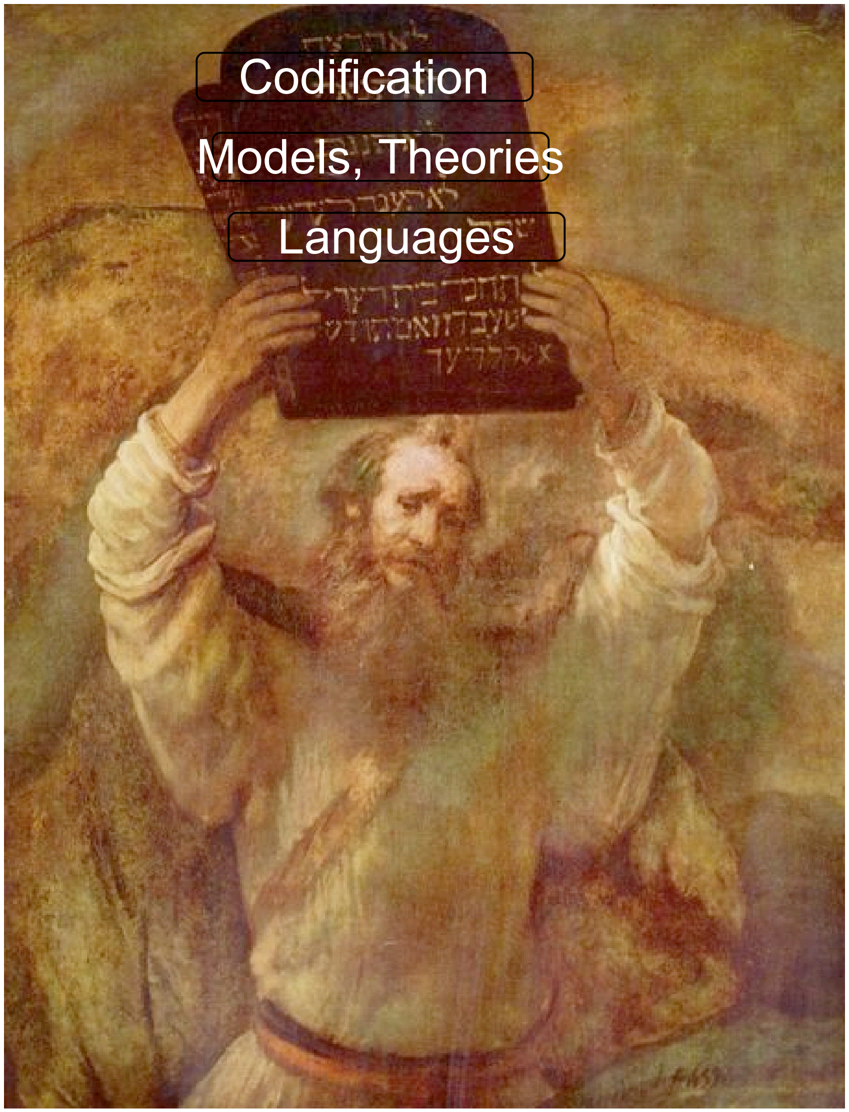
Evolution of Software Practice



Evolution of Software Language



What Doesn't Work

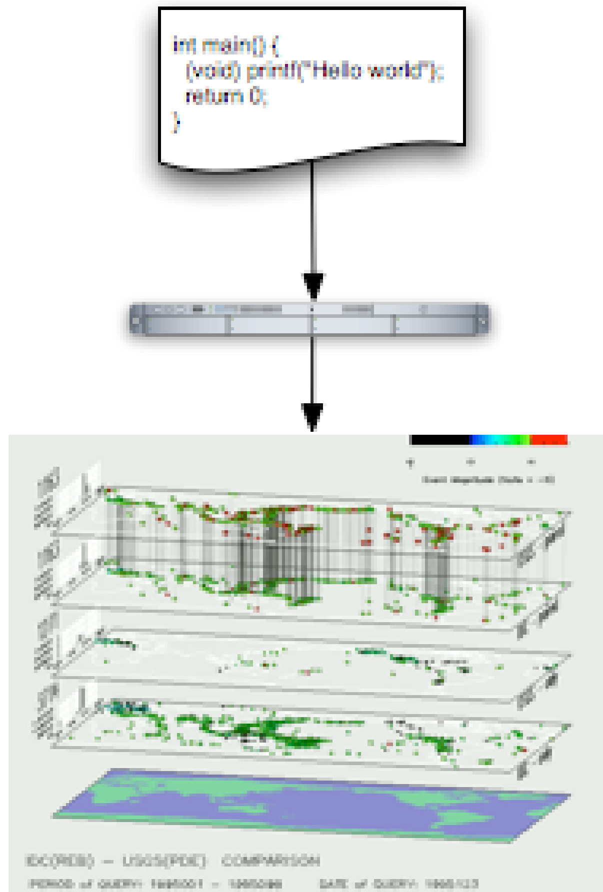


→ Improved Practice

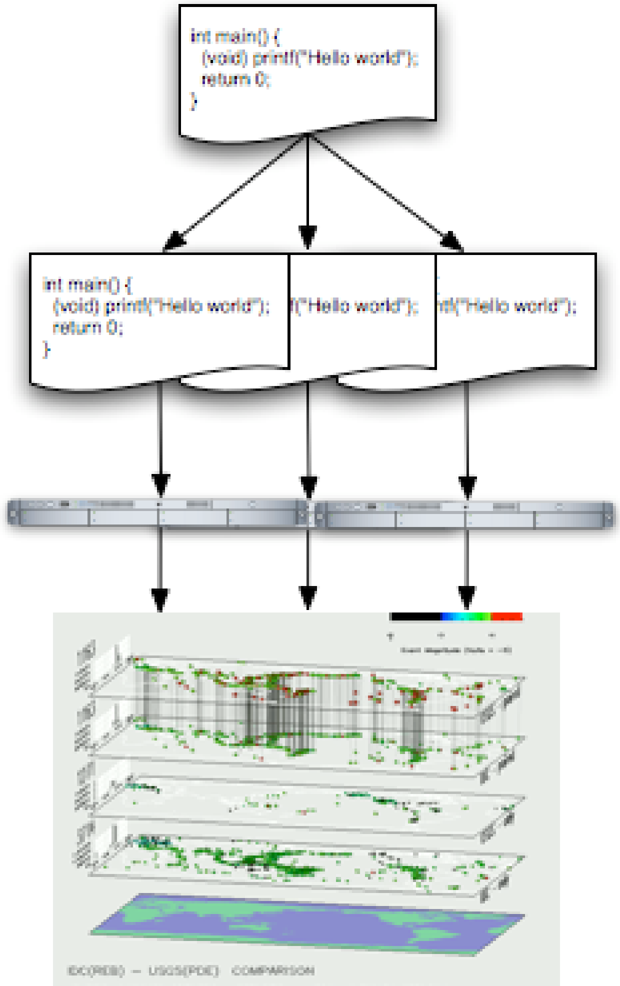
The Parallel Boost Graph Library

- ***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. Reuse sequential BGL software base.
- ***Result***: Parallel BGL. Saved years of effort.

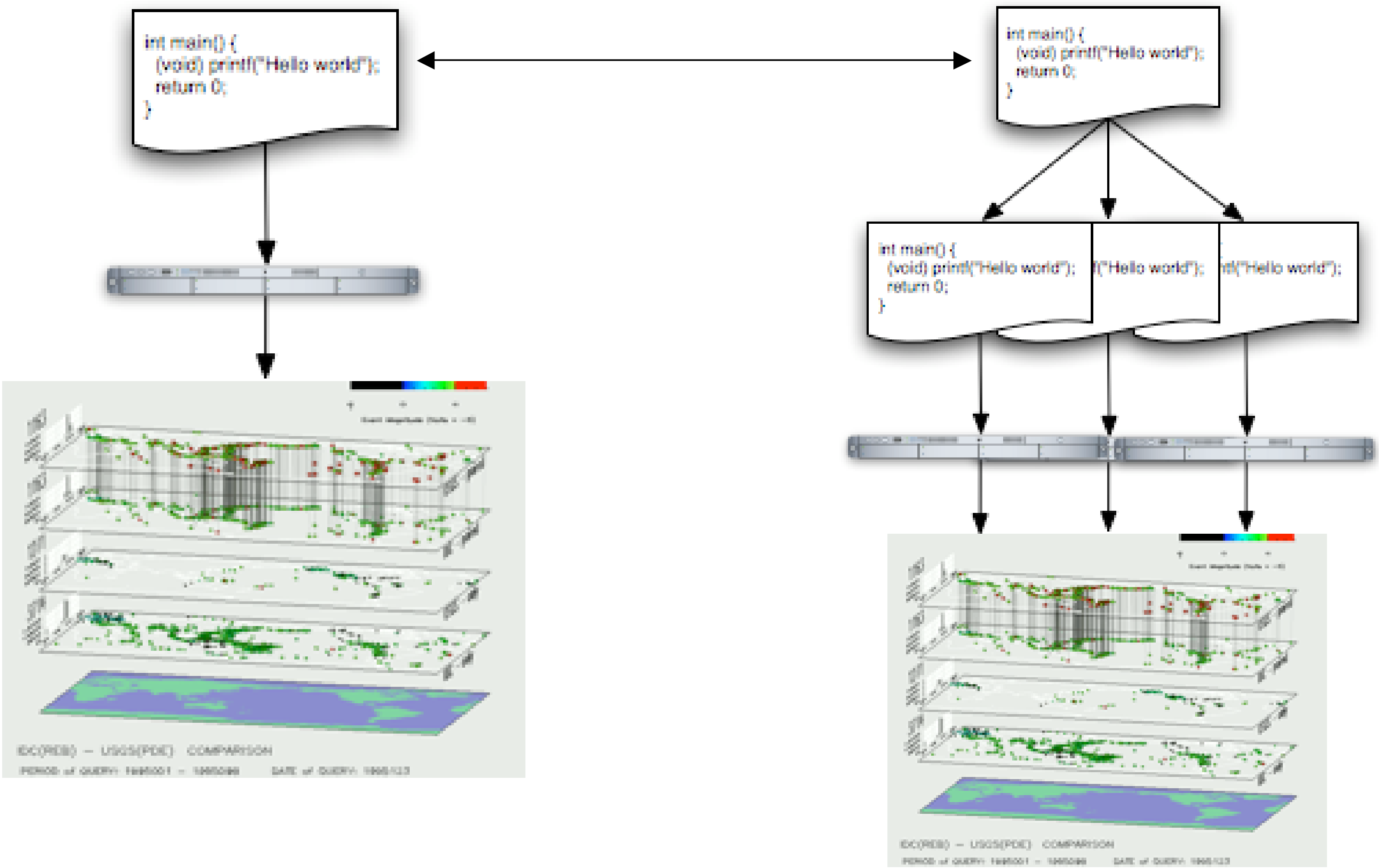
Sequential Programming



SPMD Programming



Reuse



pervasivetechnologylabs
AT INDIANA UNIVERSITY

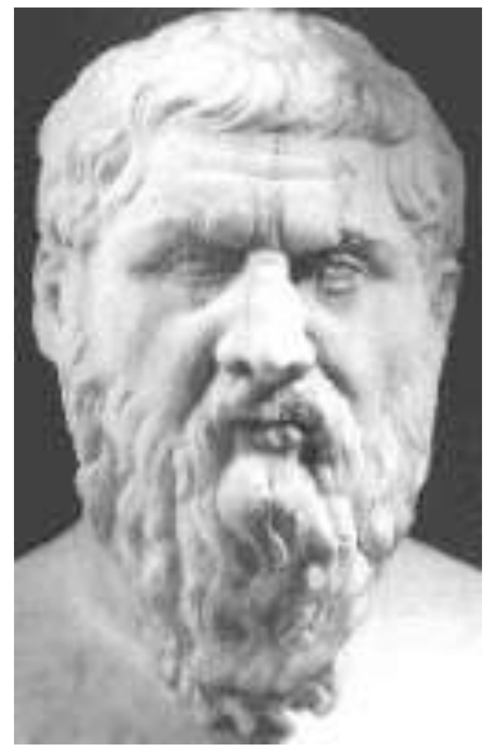
Graph Computations

- 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

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

The Boost Graph Library (BGL)

- A graph library developed with the generic programming paradigm

- Algorithms lift away requirements on:
  - Specific graph structure
  - How properties are associated with vertices and edges
  - Algorithm-specific data structures (queues, etc.)

The Sequential BGL

- The largest and most mature BGL
  - ~7 years of research and development
  - Many users, contributors outside of the OSL
  - Steadily evolving

- Written in C++
  - Generic
  - Highly customizable
  - Efficient (both storage and execution)

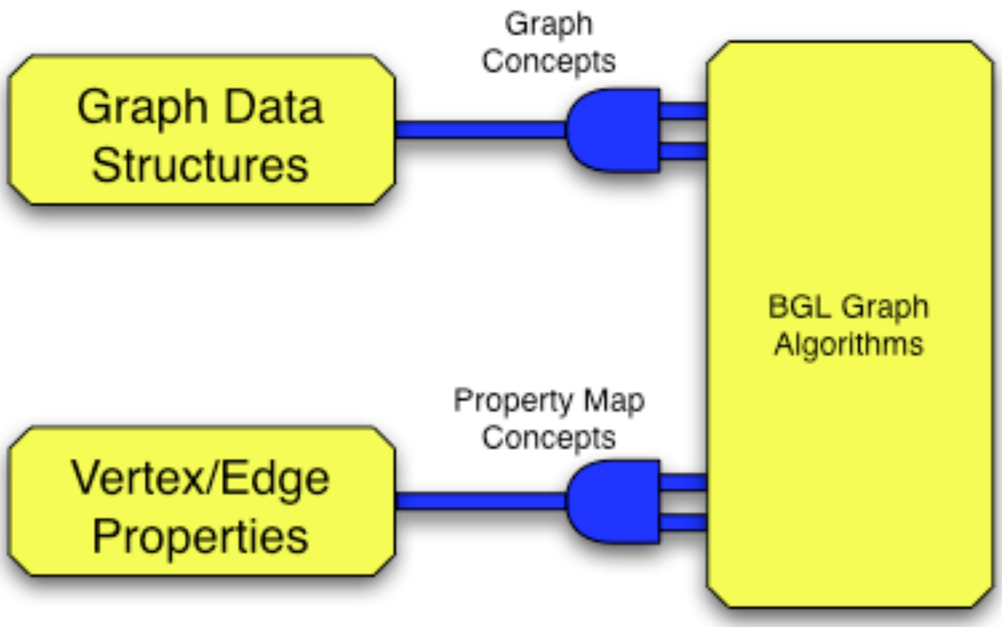
BGL: Algorithms

- 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.

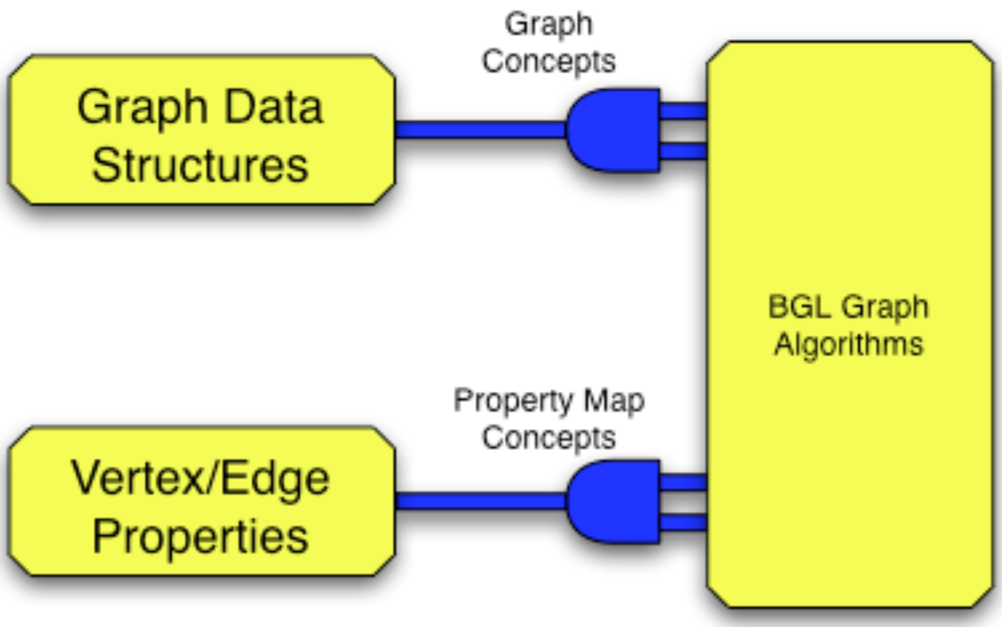
Lifting Breadth-First Search

- Generic interface from the Boost Graph Library

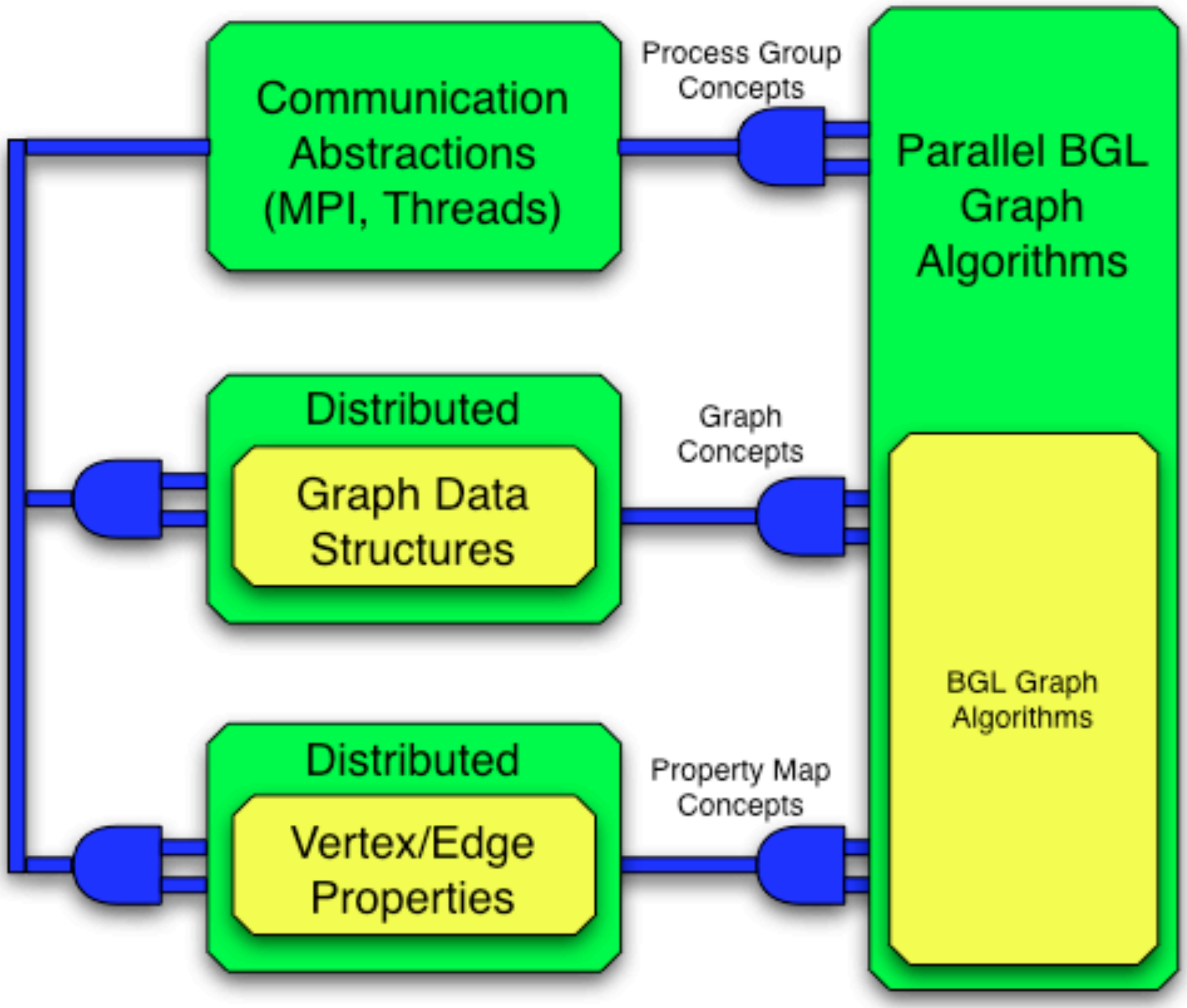
```
template<class IncidenceGraph, class Queue, class BFSVisitor,
         class ColorMap>
void breadth_first_search(const IncidenceGraph& g,
                          vertex_descriptor s, Queue& Q,
                          BFSVisitor vis, ColorMap color);
```

- Effect parallelism by using appropriate types:
  - Distributed graph
  - Distributed queue
  - Distributed property map
- Our sequential implementation is also parallel!

BGL Architecture



Parallel BGL Architecture



Algorithms in the Parallel BGL

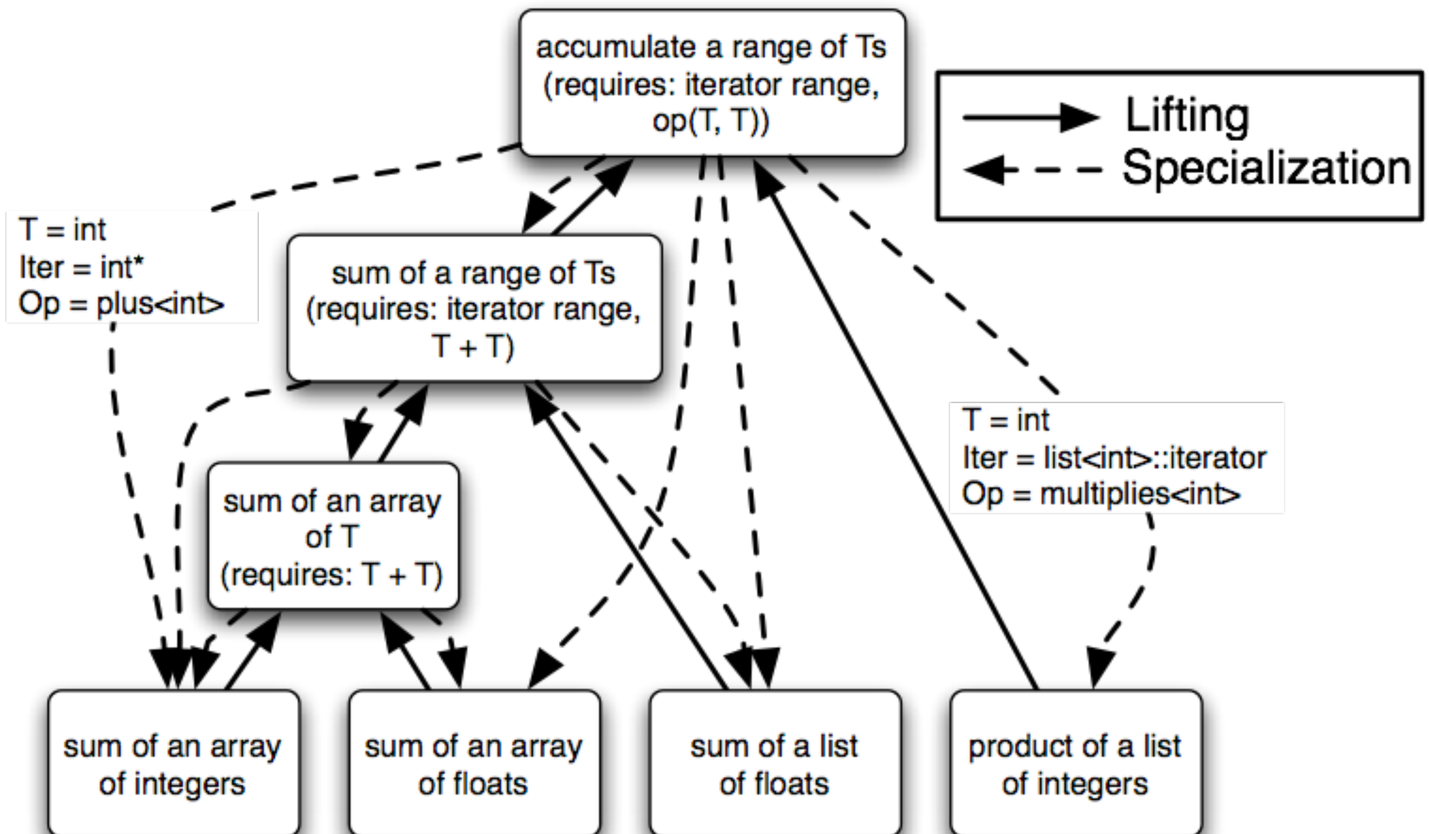
- Breadth-first search*
- Eager Dijkstra's single-source shortest paths*
- Crauser et al. single-source shortest paths*
- Depth-first search
- Minimum spanning tree (Boruvka*, Dehne & Götz‡)
- Connected components‡
- Strongly connected components†
- Biconnected components
- PageRank*
- Graph coloring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

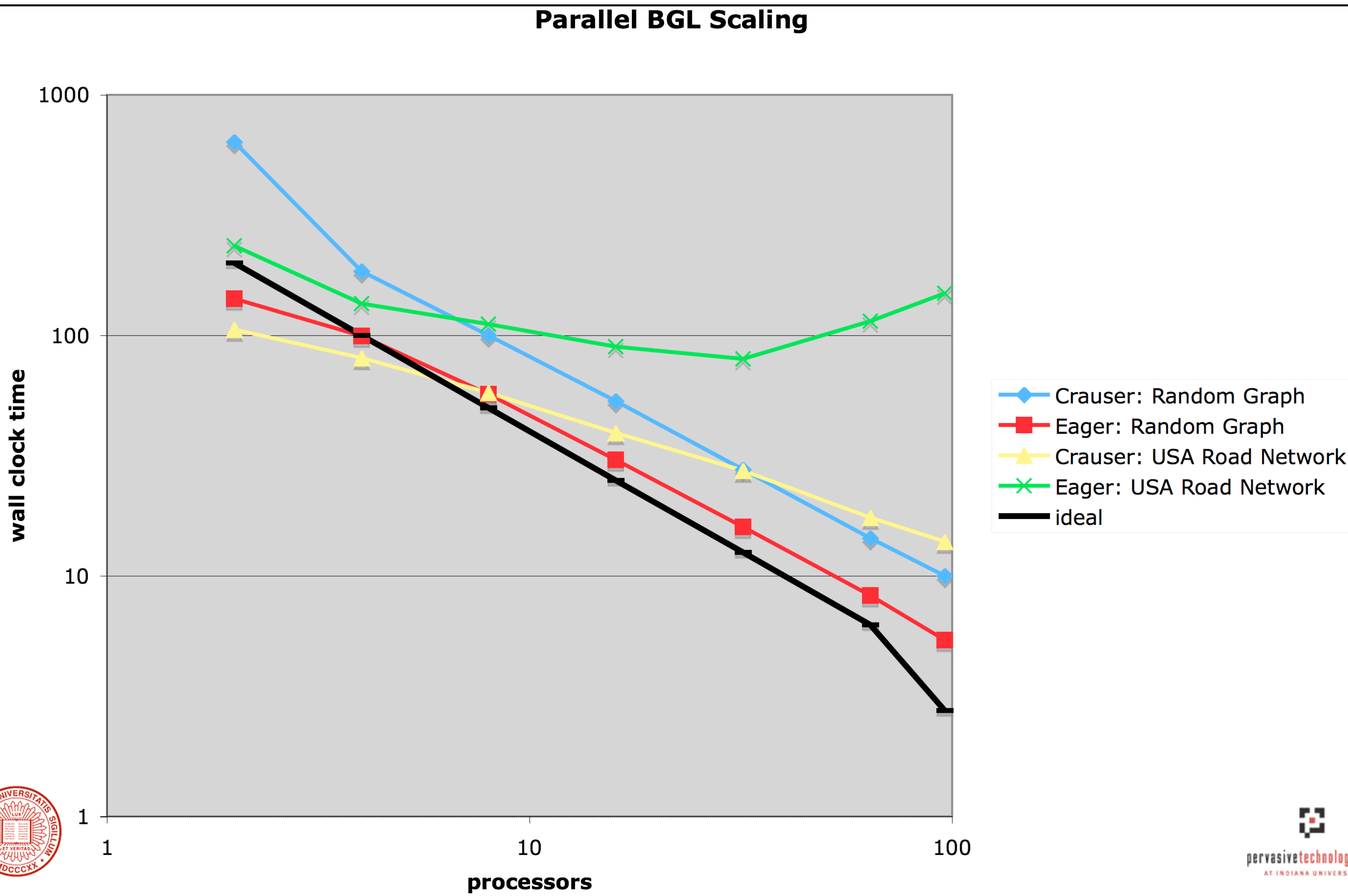
Abstraction and Performance

- ***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.

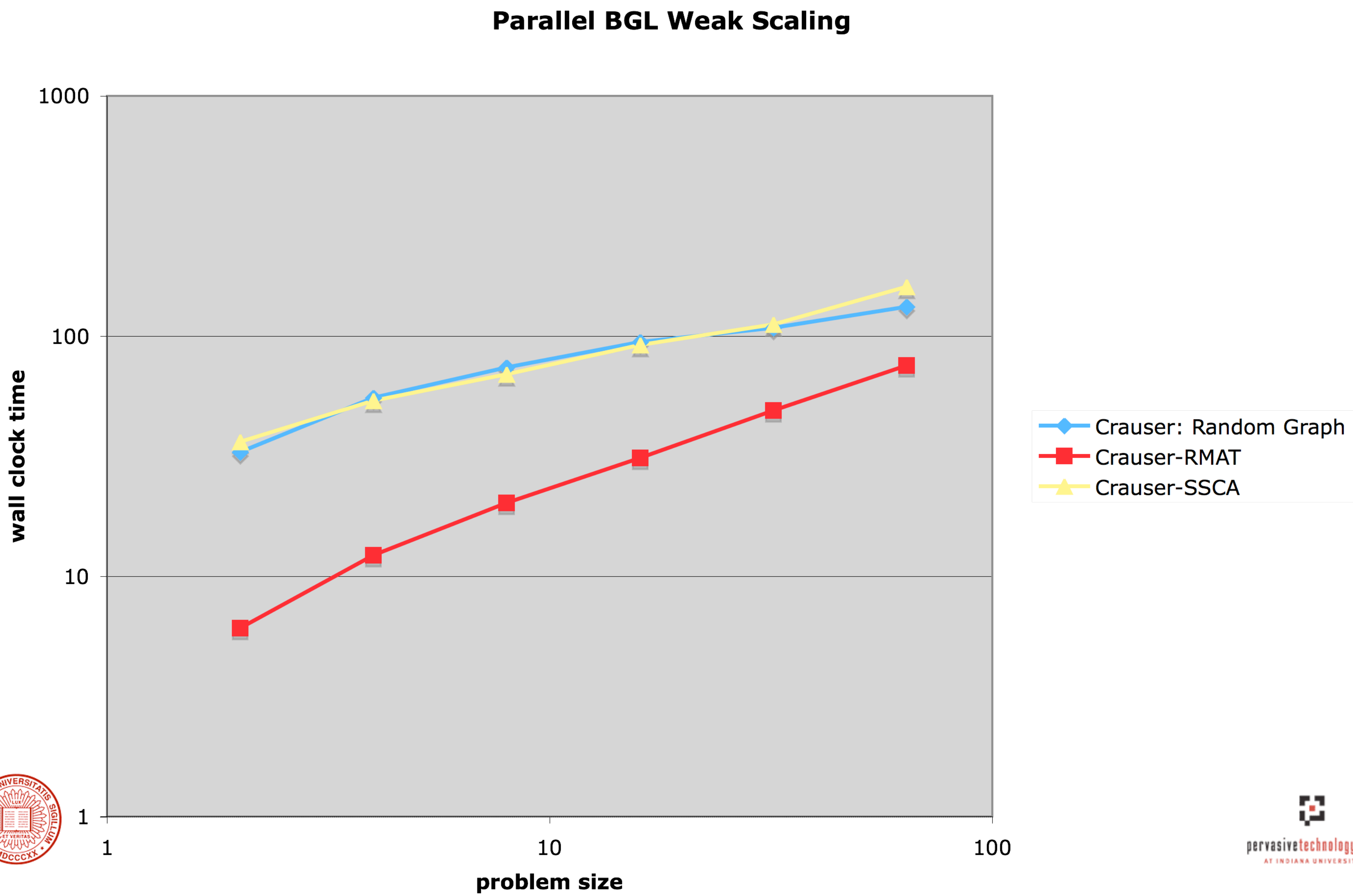
Lifting and Specialization



DIMACS SSSP Results



DIMACS SSSP Results

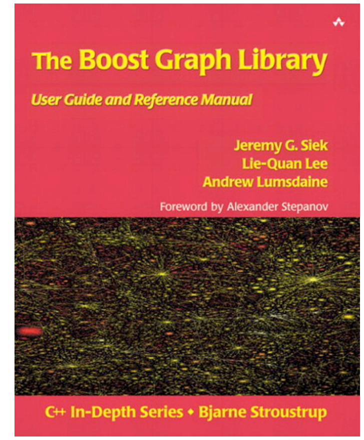


The BGL Family

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

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>
  - Douglas Gregor <dgregor@osl.iu.edu>

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?