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!
Evolution of a Discipline

Craft

Production

Science

Commercialization

Professional Engineering

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

Evolution of Software Practice

- Ad-hoc solutions
- New Problems
- Improved Practice
- Folklore
- Models, Theories
- Codification

New Problems → Ad-hoc solutions
Ad-hoc solutions → Folklore
Folklore → Models, Theories
Models, Theories → Improved Practice
Improved Practice → New Problems
Evolution of Software Language

- Ad-hoc solutions
- New Problems
- Folklore
- Libraries
- Improved Practice

New Problems → Ad-hoc solutions → Folklore → Libraries → Improved Practice → Languages → New Problems
What Doesn’t Work

Codification
Models, Theories
Languages

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
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

- Graph Data Structures
- BGL Graph Algorithms
- Vertex/Edge Properties
- Graph Concepts
- Property Map Concepts
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

Graph Data Structures

BGL Graph Algorithms

Vertex/Edge Properties

Graph Concepts

Property Map Concepts
Parallel BGL Architecture

- Communication Abstractions (MPI, Threads)
- Distributed Graph Data Structures
- Distributed Vertex/Edge Properties
- Process Group Concepts
- Graph Concepts
- Property Map Concepts
- Parallel BGL Graph Algorithms
- BGL Graph Algorithms
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

- Accumulate a range of Ts (requires: iterator range, op(T, T))
- Sum of a range of Ts (requires: iterator range, T + T)
- Sum of an array of T (requires: T + T)
- Sum of an array of integers
- Sum of an array of floats
- Sum of a list of floats
- Product of a list of integers

Lifting → Specialization

T = int
Iter = int*
Op = plus<int>

T = int
Iter = list<int>::iterator
Op = multiplies<int>
DIMACS SSSP Results

Parallel BGL Scaling

wall clock time vs. processors
DIMACS SSSP Results

Parallel BGL Weak Scaling

- Crauser: Random Graph
- Crauser-RMAT
- Crauser-SSCA

wall clock time vs. problem size
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](http://www.boost.org/libs/graph/doc)

- Parallel Boost Graph Library
  [http://www.osl.iu.edu/research/pbgl](http://www.osl.iu.edu/research/pbgl)

- Python Bindings for (Parallel) BGL
  [http://www.osl.iu.edu/~dgregor/bgl-python](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?