Parallel Graphs circa 2009:
Concepts, Hardware Platforms, and Communication Infrastructure (oh my!)

Nick Edmonds
Outline

• Introduction to BGL
• Machines as they were, motivation for the existing architecture
• New architectures, new resources, new insights, new directions
• Future work, a little something for everybody

What we’ve got now, and where we’re going...
Introduction
The Boost Graph Library (BGL)

- Graph library developed with the generic programming paradigm
Generic Programming

- 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.
BGL Genericity

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

Important: Proper abstractions allow for efficient concrete implementations
Sequential Programming

```c
int main() {
    (void) printf("Hello world");
    return 0;
}
```
SPMD Programming
BGL + SPMD = PBGL
The Parallel BGL As It Is
Current Work

- Target architectures
- Anatomy of a distributed graph
- Performance
IBM POWER systems have some additional L3 cache, otherwise fairly normal memory system
June 2004 Top 500

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Earth Simulator Center, Japan</td>
<td>Earth-Simulator, NEC</td>
</tr>
<tr>
<td>2</td>
<td>Lawrence Livermore National Laboratory, USA</td>
<td>Thunder - Intel Itanium2 Tiger4 1.4GHz - Quadrics, California Digital Corporation</td>
</tr>
<tr>
<td>3</td>
<td>Los Alamos National Laboratory, USA</td>
<td>ASCI Q - AlphaServer SC45, 1.25 GHz, Hewlett-Packard</td>
</tr>
<tr>
<td>4</td>
<td>IBM - Rochester, USA</td>
<td>BlueGene/L DD1 Prototype (0.5GHz PowerPC 440 w/Custom), IBM/ LLNL</td>
</tr>
<tr>
<td>5</td>
<td>NCSA, USA</td>
<td>Tungsten - PowerEdge 1750, P4 Xeon 3.06 GHz, Myrinet, Dell</td>
</tr>
<tr>
<td>6</td>
<td>ECMWF, UK</td>
<td>eServer pSeries 690 (1.9 GHz Power4+), IBM</td>
</tr>
<tr>
<td>7</td>
<td>Institute of Physical and Chemical Res. (RIKEN), Japan</td>
<td>RIKEN Super Combined Cluster, Fujitsu</td>
</tr>
<tr>
<td>8</td>
<td>IBM Thomas J. Watson Research Center, USA</td>
<td>BlueGene/L DD2 Prototype (0.7 GHz PowerPC 440), IBM/ LLNL</td>
</tr>
<tr>
<td>9</td>
<td>Pacific Northwest National Laboratory, USA</td>
<td>Mpp2 - Cluster Platform 6000 rx2600 Itanium2 1.5 GHz, Quadrics, Hewlett-Packard</td>
</tr>
<tr>
<td>10</td>
<td>Shanghai Supercomputer Center, China</td>
<td>Dawning 4000A, Opteron 2.2 GHz, Myrinet, Dawning</td>
</tr>
</tbody>
</table>

BlueGene/L - Torus interconnect network
<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DOE/NNSA/LLNL, United States</td>
<td>BlueGene/L - eServer Blue Gene Solution IBM</td>
</tr>
<tr>
<td>2</td>
<td>IBM Thomas J. Watson Research Center, United States</td>
<td>BGW - eServer Blue Gene Solution IBM</td>
</tr>
<tr>
<td>3</td>
<td>NASA/Ames Research Center/NASA, United States</td>
<td>Columbia - SGI Altix 1.5 GHz, Voltaire Infiniband SGI</td>
</tr>
<tr>
<td>4</td>
<td>The Earth Simulator Center, Japan</td>
<td>Earth-Simulator NEC</td>
</tr>
<tr>
<td>5</td>
<td>Barcelona Supercomputer Center, Spain</td>
<td>MareNostrum - JS20 Cluster, PPC 970, 2.2 GHz, Myrinet IBM</td>
</tr>
<tr>
<td>6</td>
<td>ASTRON/University Groningen, Netherlands</td>
<td>Stella - eServer Blue Gene Solution IBM</td>
</tr>
<tr>
<td>7</td>
<td>Lawrence Livermore National Laboratory, United States</td>
<td>Thunder - Intel Itanium2 Tiger4 1.4GHz - Quadrics California Digital Corporation</td>
</tr>
<tr>
<td>8</td>
<td>Computational Biology Research Center, AIST, Japan</td>
<td>Blue Protein - eServer Blue Gene Solution IBM</td>
</tr>
<tr>
<td>9</td>
<td>Ecole Polytechnique Federale de Lausanne, Switzerland</td>
<td>eServer Blue Gene Solution IBM</td>
</tr>
<tr>
<td>10</td>
<td>Sandia National Laboratories, United States</td>
<td>Red Storm, Cray XT3, 2.0 GHz Cray Inc.</td>
</tr>
</tbody>
</table>

Columbia -> Cluster of Altixes, beginning of interesting architecture trend
HPC circa ~2005

- Clusters of workstations
- Lots of FLOPs
- Commodity memory subsystem
- Single core, possibly a few sockets
Concepts circa 2005

- **ProcessGroup** - coordinating group of communicating processes
- **GlobalDescriptor, DistributedGraph + BGL concepts**
- **Data handling similar to DSM with weak consistency**

Immediate is exactly what it sounds like, lots of small messages, handed to MPI layer as soon as they are created.

Linear Process Group – requires that process numbers are in the range [0, num_processes)

DistributedGraph – graph with vertices and edges that model GlobalDescriptor

GlobalDescriptor – a descriptor that identifies a data object (local descriptor) and it’s owner (processor id)
Data Distribution

- Row-wise distribution of adjacency matrix
- Owner-computes model
- Cache non-local data
- Various consistency models for cached data

Owner computes -> either move data to work, or work to data, we do the latter (moving data is expensive)

Every data element has a single owner, at the end of a superstep that process has the authoritative value for any properties associated with that data.

Structural information only available to processes that own vertices or (one) endpoint of an edge.

Consistency models for cached data never proved very useful.
Performance Characteristics

- Dense Numerical Codes
  - Good locality
  - FLOPS
  - Memory bandwidth
- Graphs
  - Little to no locality
  - Memory latency
Interface Compatible w/ BGL

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

Parallelism effected by supplying appropriate types

- Distributed graph
- Distributed queue
- Distributed property map
Algorithms

Strongly connected components
Crauser et al. shortest paths
Eager Dijkstra shortest paths
Delta-Stepping shortest paths
Biconnected components
Boman et al. graph coloring
Connected Components
PageRank
Fruchterman-Reingold
Betweenness Centrality
Depth-first search
s-t connectivity
Minimum Spanning tree
Breadth-first search

*Other experimental algorithm implementations exist, this is the list from the latest release.

Multiple variants of MST, Betweenness Centrality, and CC
These are the algorithms in the release, more experimental algorithms.
Performance

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

Performance of three SSSP algorithms on fixed-size graphs with ~24M vertices and ~58M edges.
Weak scalability of three SSSP algorithms using graphs with an average of 1M vertices and 10M edges per processor.
Performance Issues

- Good scaling (generally)
- Able to solve very large problems
- *Somewhat* faster than sequential algorithms for small numbers of processors
Performance Issues

- Distributing data gets problems in core so we can work on them
- Move the work to the data
- Low ratio of computation to communication
- High ratio of network latency to CPU resources

Low CPU utilization
Performance: Latency

Network/Memory latency dominates performance

- Hide latency
- Perform additional work, greedy algorithms
- Cache aggressively
- Exploit known communication patterns

Hiding latency is easier said than done, but large numbers of slow(er) processors performing asynchronous work is a good start
Conceptual Issues

- Locking process group requires buffering which isn’t strictly necessary
- Immediate ProcessGroup only suitable for some algorithms, sends lots of small messages
- BSP ProcessGroup doesn’t effectively overlap communication and computation
- DSM doesn’t work

Process groups define communication style
User must be aware of PG implementation to get good performance
PG implementation should be more dependent on hardware
Conceptual Issues

• Locking process group requires buffering which isn’t strictly necessary

• Immediate ProcessGroup only suitable for some algorithms, sends lots of small messages

• BSP ProcessGroup doesn’t effectively overlap communication and computation

• DSM doesn’t work

This is BSP
Communication phase length determined by work imbalance, forces alignment rather than letting algorithm recover on it’s own
Conceptual Issues

• Locking process group requires buffering which isn’t strictly necessary
• Immediate ProcessGroup only suitable for some algorithms, sends lots of small messages
• BSP ProcessGroup doesn’t effectively overlap communication and computation
• DSM doesn’t work

Statistically random data distribution will load balance... eventually (This does NOT apply on small timescales)
Conceptual Issues

- Locking process group requires buffering which isn’t strictly necessary.
- Immediate ProcessGroup only suitable for some algorithms, sends lots of small messages.
- BSP ProcessGroup doesn’t effectively overlap communication and computation.
- DSM doesn’t work.

Would rather defer synchronization as long as possible, sending data along the way does this. Overlap communication and computation, helps to hide latency.
Release History

- **0.5.0 - October 2005**
  - Adjacency List, BSP & Immediate ProcessGroups, handful of algorithms

- **0.6.0 - ~January 2007**
  - Updated process group, more efficient data structures and communication, more algorithms

- **0.7.0 - March 2009**
  - More algorithms, more tests, more docs, in Boost!

New process group provides better overlap of communication and computation (something like Nagle’s algorithm), also provides out-of-band (immediate) interface and allows for blocking communication
The Parallel BGL As It Will Be?
New Directions

• New architectures
• New types of parallelism
• New concepts
• New infrastructure
### November 2008 Top 500

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>Computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DOE/NNSA/LANL United States</td>
<td>Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband IBM</td>
</tr>
<tr>
<td>2</td>
<td>Oak Ridge National Laboratory United States</td>
<td>Jaguar - Cray XT5 QC 2.3 GHz Cray Inc.</td>
</tr>
<tr>
<td>3</td>
<td>NASA/Ames Research Center/NAS United States</td>
<td>Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz SGI</td>
</tr>
<tr>
<td>4</td>
<td>DOE/NNSA/LLNL United States</td>
<td>BlueGene/L - eServer Blue Gene Solution IBM</td>
</tr>
<tr>
<td>5</td>
<td>Argonne National Laboratory United States</td>
<td>Blue Gene/P Solution IBM</td>
</tr>
<tr>
<td>6</td>
<td>Texas Advanced Computing Center/Univ. of Texas United States</td>
<td>Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband Sun Microsystems</td>
</tr>
<tr>
<td>7</td>
<td>NERSC/LBNL United States</td>
<td>Franklin - Cray XT4 QuadCore 2.3 GHz Cray Inc.</td>
</tr>
<tr>
<td>8</td>
<td>Oak Ridge National Laboratory United States</td>
<td>Jaguar - Cray XT4 QuadCore 2.1 GHz Cray Inc.</td>
</tr>
<tr>
<td>9</td>
<td>NNSA/Sandia National Laboratories United States</td>
<td>Red Storm - Sandia/ Cray Red Storm, XT3/4, 2.4/2.2 GHz dual/quad core Cray Inc.</td>
</tr>
<tr>
<td>10</td>
<td>Shanghai Supercomputer Center China</td>
<td>Dawning 5000A - Dawning 5000A, QC Opteron 1.9 Ghz, Infiniband, Windows HPC 2008 Dawning</td>
</tr>
</tbody>
</table>

**XT4/5** – Multicore, custom interconnect network
Roadrunner includes Mercury blades
Red Storm – 8 cores/node
## Performance Measures

### System Information

<table>
<thead>
<tr>
<th>System Information</th>
<th>G-HPL</th>
<th>G-Random Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cray Inc. XT5 AMD Opteron</td>
<td>IBM Blue Gene/P PowerPC 450</td>
</tr>
<tr>
<td>System</td>
<td>2.3GHz</td>
<td>0.85GHz</td>
</tr>
<tr>
<td>Processor</td>
<td>74529</td>
<td>32768</td>
</tr>
<tr>
<td>Speed</td>
<td>274529</td>
<td>1131072</td>
</tr>
<tr>
<td>Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Results

- **Cray Inc. XT5 AMD Opteron**
  - G-HPL: 901,999,0000
  - G-Random Access: 16,611,5000
- **IBM Blue Gene/P PowerPC 450**
  - G-HPL: 191,325,0000
  - G-Random Access: 6,793,0500
- **SiCortex SC5832 Ice9**
  - G-HPL: 4,729,8700
  - G-Random Access: 3,997,6000
New Architectures

- Commodity multi-core, multi-socket nodes
- Many-core nodes (Niagra, SiCortex, etc.)
- Heterogeneous systems (Roadrunner)
- Massively multithreaded architectures (Tera MTA, Cray XMT)
- SIMD, GPUs

These architectures actually exist now!

SiCortex is only 6 cores/node, but there are a lot of cores physically close together (and the network should reflect this logically)
MPI SM interface?

- Further subdivides data structures
- Load balancing issues
- $N$ nodes, $P$ processes/nodes = $NP$ message buffers
- Contention for network resources

Reasonable evidence that adding more MPI processes doesn’t work for unstructured problems.
Another Level of Parallelism?

- MPI is hard
- Threads are harder
- MPI + Threads is too hard

**Conclusion**: Limit abstractions to a single level of parallelism

e.g. **Processes** own data, **Threads** perform *tasks*

Processes and threads, *at the same time* are too hard
Tasks are the unit of work, Processes determine data ownership
Process and thread are arbitrary names
Exposing Parallelism

Decomposing work into tasks exposes parallelism

- Data ownership determines where tasks are executed (Coarse-grained parallelism)
- *How* tasks are executed depends on the architecture (Fine-grained parallelism)

Is the ‘owner-computes’ model still appropriate?
Does data need an ‘owner’
Tasks are Active Messages

- Route to data and run
- Eliminate most of the data caching from current DSM-style model by chaining AMs

```plaintext
PropertyMap p1, p2;
GlobalID x;
Closure f;
request(p1, x);
synchronize(p1);
put(p2, x, f(get(p1, x)));
```

```plaintext
PropertyMap p1, p2, owner;
GlobalID x;
Closure f;
send_AM(get(owner, x),
  \( \lambda () \rightarrow \text{put}(p2, x, f(\text{get}(p1, x))) \));
quiesce()
```

Lightweight first-class functions over the wire (static binding)
The request()ing process no longer has to cache get(p1, x)
Caching may still be useful, but it will eliminate caching when it’s unnecessary, and managing caches is hard.
Quiesce() potentially gives us a longer phase-time, and more opportunities for load balancing
Where AMs Run is Irrelevant

AMs are routed to address space determined by data distribution.

AM handling depends on architecture, can be routed to specialized resources based on work.
Massive Multi-Threading

- Cray XMT provides massive numbers of hardware threads (8K processors, 128 threads/proc)
- Single-cycle thread switching
- Loop-parallelism (similar OpenMP)
- Extended memory semantics
- Operates like a QRQW PRAM

Single cycle thread switching hides memory latency
Extended semantics → Futures, Full/Empty bits, sync variables
Different balance than multicore, similar abstractions
Contention is the major performance issue (operates like a QRQW PRAM)
Massive Multi-Threading

- Loop iterations map 1:1 to Active Messages
- Static parallelism avoids contention
- Dynamic parallelism provides load balancing
- (Good) Parallelizing compilers are... difficult
- Code generation is one option to divorce genericity from the tool chain...
New Concepts

TBD

• Data distribution/ownership
• Work decomposition
• Hybrid collectives?
• Locality

Toss out locality info within an address space, can we recover it?
New Infrastructure

- Higher level communication abstractions
- Lower level interface
  - Interface directly with network hardware
  - Leverage MPI_THREAD_FUNNELED for now
- Collectives
Future Directions
Concepts

- New conceptual framework for parallel communication
- Identify high-level communication abstractions
- ‘Fixed’ or ‘known’ communication schedules
- Concepts for 2D data distribution
- Who ‘owns’ vertices and what does it mean
- Relax ‘owner computes’ requirements?
- Semantic Graphs

Mention fixed-point option
Theoretical Models

- Map theoretical performance of algorithms amongst different types of machines
- LogGP, PRAM useful but difficult to unify
- Incorporate data locality
- Account for network topology

In papers we normally end up providing both LogGP and PRAM analysis
Algorithms

- Subgraph isomorphism
- Kernighan-Lin partitioning
- Multi-level partitioning
- Dynamic/Incremental algorithms
- ...and algorithms you find interesting
Hypergraphs

- Hypergraph: A graph where edges can be incident to >2 vertices
- Data structure
- Traversal algorithms
- Partitioning
Infrastructure

- Map high level abstractions to low level (network hardware) implementations
- MPI_THREAD_FUNNELED vs. MPI_THREAD_MULTIPLE
- 2D data distribution - distribute blocks instead of rows of adjacency matrix
- Push data for ‘known’ communication patterns
- CPU time/memory usage accounting
- AM buffering, coalescing, and demultiplexing

Pushing data is a one-RTT operation, pulling takes 2 RTTs.
Visualization

- VTK/Titan
- Streaming Visualizations (Stencil?)
- Cluster back end to Viz front end
- Dimensionality reduction !?!
Questions?

http://www.osl.iu.edu/research/pbgl
http://www.boost.org
ngedmond@cs.indiana.edu

We always need good students for PBGL
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.

Performance

Crauser et al.
Eager Dijkstra