Graphs, Informatics, Architectures and Languages

Bruce Hendrickson
Jon Berry

Discrete Algorithms and Math Dept.
Sandia National Laboratories
Graph-Based Informatics

Attributed Relational Graph

Legend:
- Red: Workplace
- Blue: Town
- Purple: Works at
- Green: Friends with
- Orange: Located in
- Black: Person
- Light Blue: Lives in
Parallelizing Graph-Based Informatics

- Graphs can be huge

- Edge and vertex types make queries more tractable

- Graphs are highly unstructured
  - High variance in number of neighbors
  - No exploitable global structure
  - Little or no locality – Not partitionable
  - Lessons from scientific computing are of limited utility
Performance Challenges

- When it exists, parallelism is often very fine-grained

- Runtime is dominated by latency
  - Random accesses to global address space
  - Perhaps many at once

- Essentially no computation to hide memory costs

- Access pattern is data dependent
  - Prefetching unlikely to help
  - Usually only want small part of cache line

- Potentially abysmal locality at all levels of memory hierarchy
Desirable Architectural Features

• Low latency / high bandwidth
  – For small messages!
• Latency tolerant
• Light-weight, fine-grained synchronization mechanisms
• Global address space
  – No graph partitioning required
  – Avoid memory-consuming profusion of ghost-nodes
  – No local/global numbering conversions

• Cray’s massively multithreaded machines have these properties
How Does the MTA Work?

• Latency tolerance via massive multi-threading
  – Context switch in a single tick
  – Global address space, hashed to reduce hot-spots
  – No cache or local memory. Context switch on memory request.
  – Multiple outstanding loads
• Remote memory request doesn’t stall processor
  – Other streams work while your request gets fulfilled
• Light-weight, word-level synchronization
  – Minimizes access conflicts
• Flexibly supports dynamic load balancing
• Notes:
  – MTA-2 is 5 years old
  – Largest machine is 40 processors (at NRL)
Case Study: MTA-2 vs. BlueGene/L

• With LLNL, implemented S-T shortest paths in MPI
• Ran on IBM/LLNL BlueGene/L, world’s fastest computer
• Finalist for 2005 Gordon Bell Prize
  – 4B vertex, 20B edge, Erdös-Renyi random graph
  – Analysis: touches about 200K vertices
  – Time: 1.5 seconds on 32K processors

• Ran similar problem on MTA-2
  – 32 million vertices, 128 million edges
  – Measured: touches about 23K vertices
  – Time: .7 seconds on one processor, .09 seconds on 10 processors

• Conclusion: 4 MTA-2 processors = 32K BlueGene/L processors
But Speed Isn’t Everything

• MPI code is 3 times larger than MTGL code
  – Took considerably longer to develop

• MTA easily supports multiple, simultaneous users
  – Important for flexible usage model

• But … MPI code runs everywhere
  – MTGL doesn’t. But MTGL does run on SMPs
Multithreaded Software Challenges

- Code may look easy, but can be quite subtle
  - Especially for unstructured, asynchronous operations like graph algorithms

- Two fundamental difficulties
  - Race conditions cause hard-to-recognize bugs
    - Nondeterministic code very hard to debug
    - Hard to know that the code is correct (or not)
  - Memory hot-spots limit scalability
    - If many threads hit on same memory simultaneously, performance is throttled
      - Fundamental impact on algorithm & data structure design
    - However, fine grained synchronization on MTA/XMT is a big improvement over traditional shared memory machines
Multithreaded Graph Library (MTGL)

- **Software engineering**: Encapsulate most challenging details
- **Generic programming through the visitor design pattern**
  - General idea, advocated for graph algorithms by Lumsdaine (Boost GL)
  - Algorithms are customized by invoking “visitor” functions at key times
    - E.g. when seeing a vertex for the first time, etc.
  - Visitors are a clean way to support complex filtering
  - Visitors also support generic algorithmic functionality
    - E.g. one thread-safe breadth-first-search code can compute connected components, spanning tree, shortest paths, betweenness centrality, etc.
- **Parallelism**
  - Loop-based, but highly dynamic and adaptive
  - “Futures” support recursive parallelism and thread virtualization
How to Code for XMT?

- Design large systems by abstracting away multithreading subtleties.

- Carefully design software layers
  - Lower layers handle parallelization (requires expert knowledge)
  - Top layers allow for customization of functionality (non-experts allowed)

- Generic programming through the visitor design pattern
  - General idea, advocated for graph algorithms by Lumsdaine (Boost GL)
  - Algorithms are customized by invoking “visitor” functions at key times
    - E.g. when seeing a vertex for the first time, etc.
  - Visitors are a clean way to support complex filtering
  - Visitors also support generic algorithmic functionality
    - E.g. one thread-safe breadth-first-search code can compute connected components, spanning tree, shortest paths, betweenness centrality, etc.
Connected Components Performance

Power Law Graph (highly unstructured)
MTGL Status

• ~10,000 lines of C++ (simpler use of language than STL (e.g. no iostream))

• **Thread-Safe Primitives**
  – Dynamic array
  – HashSet, HashTable
  – Tuple Datatypes, Timing Objects
  – Synchronization Wrappers, Counting Sort, Comparison & Copier Objects, etc.

• **Graph Layer**
  – Graph, Vertex and Edge objects
  – Parallel Search
  – Breadth-First Search
  – Induced subgraph operator, Masking for algorithms as filters
  – Various graph generators and readers, Graph metrics
• Algorithm Layer
  – Several algorithms for connected components
  – Strongly connected components
  – S-T shortest path
  – Semantic subgraph isomorphism
  – Sparse matrix-vector multiplication and conjugate gradients
  – Single-Source Shortest-Path through Delta stepping and Thorup algorithm
  – Random walks
  – Connection subgraphs (in progress)
  – Betweenness Centrality (in prototype)
  – Network simplex algorithms (next year)
Lessons for HPC Languages

• Application characteristics that are problematic for MPI include
  – Fine-grained parallelism
  – Highly adaptive computation
  – Non-BSP structure
  – Time-varying amount of parallelism

• Graphs algorithms require support for a different style of parallelism
  – Global address space
  – Nested parallelism, e.g. through Futures
  – Thread virtualization
    • Many more threads than processors to facilitate load-balance
  – Support for fine-grained parallelism
    • Latency tolerance, not just amortization as with MPI
    • Light-weight synchronization mechanisms

• No main-stream approach currently has the necessary capabilities
  – Complex interplay between languages and architecture