Center for Exascale Radiation Transport

STAPL Developments
Lawrence Rauchwerger
CERT Fall 2015 TST Meeting
Texas A&M, December 2-3, 2015
Project Components and Integration

Improved Predictive Capability for RT

UQ

Experiments

Simulations

PDT
TAXI
STAPL

Subgrid Models

Numerics

Transport Algorithms

Comp Sci Algorithms

Verification appears in all components
Outline

• STAPL Programming Model Review

• Communication Reduction in Graph Algorithms

• Out of Core Graph Processing

• Runtime Support for Nested Parallelism

• Kripke Status
Programming Model with STAPL

- STAPL Programming Model.
  - High Level of Abstraction ~ similar to C++ STL
  - Fine grain expression of parallelism – can be coarsened
  - Implicit parallelism – Serialization is explicit
  - Distributed Memory Model (PGAS)
  - Algorithms defined by
    - Data Dependence Patterns (Library)
    - Distributed containers
    - Execution policies (scheduling, data distributions, etc.)
  - Algorithm run-time representation: Data Flow Graphs (PARAGRAPHS)
Programming Model with STAPL

• Data parallelism:
  o Apply work-function on data
  o Work function: (Parallel | Serial ) function (algorithm)

• Task Parallelism:
  o Compose functions (algos) using operators
e.g., parallel, serial, enum, etc.
  o Compose using “flows” (arbitrary dependences)

• STAPL = imperative + functional

Values flow along data flow graph and/or stored in containers.
Scalable Graph Processing

• Big Data Graphs
  o Large – Billions of vertices and edges
  o Often small-world scale-free characteristics
  o Power-law degree distribution creates Hubs

• Challenges
  o Communication bottlenecks due to hubs and cross-edges
  o Severe work-imbalance due to power-law
  o May not fit in RAM of single machine
  o Use off-core storage (disk)
Algorithmic Communication Reduction

- Reduce duplication and minimize number of messages and volume of data communicated
  - Exploit Algorithmic Redundancy in outgoing and incoming messages
- Reuse fine-grained algorithms without modification
- Dramatically improves performance and scalability

190% - 850% improvement

Graph Algorithms on Cray XE6 (scale=32, 12288 cores)

Graph500 Input
- 4 Billion Vertices
- 64 Billion Edges
- 12,288 cores
Eliminating Algorithmic Redundancy

Reducing Out-Degree Communication

- Algorithmic Redundancy
  - Many graph algorithms send the same data to all neighbors
  - Examples: BFS, CC, Centrality, PageRank, k-Core decomposition, etc.

- Eliminate communication due to algorithmic redundancy
  - Send a single copy of the message to each location
  - Benefits vertices with high out-degree
Eliminating Algorithmic Redundancy
Reducing High In-Degree Communication

- Extend approach for high in-degree vertices
  - Co-locate computation on source location
    - Distributes computation
    - Reduces imbalance
  - Local reduction of hub vertex messages
    - Minimize volume of data communicated
Using the Hierarchy
Reducing Out-Degree Communication

- **Hierarchy creation to match machine hierarchy**
  - Graph partitioned among “locations” (processors, nodes, etc.)
  - Each location-subgraph represented with “super-vertex”
  - Eliminate edges between two locations in the same level of hierarchy
  - Replace with “super-edges” in the next level of hierarchy
  - Apply recursively to higher levels

- **Communication**
  - Apply updates to local vertices
  - Remote updates sent to hierarchical parent
  - Received updates applied (scatter) to hierarchical children
Performance at Scale

- 131,000 cores on IBM Blue Gene/Q
- Similar trend across systems, algorithms and graphs
- Faster network, slower cores
Communication Reduction
Breadth-First Traversal (BFS) [Light Comm.]

- ~1/3 bytes communicated (33.8 GB ➞ 10.3 GB)
- 2.1x performance improvement, increases with scale
- Improved load-balance helps performance
Communication Reduction
Breadth-First Traversal (BFS) [Light Comm.]

- ~1/3 bytes communicated (33.8 GB → 10.3 GB)
- 2.1x performance improvement, increases with scale
- Improved load-balance helps performance
Asynchronous Subgraph Paging

- On-demand paging of subgraphs using asynchronous updates
  - Eliminate random disk-reads, minimize random writes
  - Effectively utilizes available RAM
  - Scales well from small tablets to large distributed machines
  - Single algorithm implementation runs on all systems
Asynchronous Subgraph Paging

- On-demand paging of subgraphs using asynchronous updates
  - Eliminate random disk-reads, minimize random writes
  - Effectively utilizes available RAM
  - Scales well from small tablets to large distributed machines
  - Single algorithm implementation runs on all systems

![Running Times Across Platforms, G500-24](chart)

- Android (1GB) 2 cores, STAPL GL: 770 sec, GraphChi: 26 sec
- PC (8GB) 4 cores, STAPL GL: 122 sec, GraphChi: 122 sec
- Cray XE6 (256G) 128 cores, STAPL GL: 0.7 sec, GraphChi: 128 sec
Hybrid Paradigm Implementation

• For each superstep:
  o Load subset of active subgraphs to RAM and process
  o Asynchronously forward neighbor-updates to their home-locations
    • **Apply** updates to vertices in active subgraphs
    • **Store** updates to inactive subgraphs (buffered), and **apply** when the subgraphs are loaded
Asynchronous Subgraph Paging

- Asynchronous deferred writes ("Lazy Push" model)
- Graph split into subgraphs
  - Each owned by a location (home-location)
  - Locations may own multiple subgraphs

- Subgraphs may be held in RAM (loaded) or stored on disk (inactive)

- Asynchronous distributed graph container

- Vertices globally addressable using a 2-level distributed directory
In-Memory Performance

- Little to no overhead vs. in-memory libraries
- Comparable to shared-memory libraries
- Faster than distributed libraries
Out-of-Core Performance

Graph500 Web-Graph (4GB RAM)

- BFS
- CC
- PageRank
- k-core

Graph500 Input
17 Million Vertices
268 Million Edges

- 2-core PC with 4GB RAM
- 2.5–6x better performance
Most STAPL developers do not interact directly with it.

PARAGRAPH – Runtime representation of dependence graph. Coordinate creation, dependence enforcement, task placement, etc.

ARMI – Communication primitives:
  - Remote method invocation
  - Data marshalling / serialization
  - future / promises.

Scheduler / Dispatcher – Manage the ordering and execution of runnable tasks.
Application Driven Optimizations

• Task dependence graph has contextual information
  o Number of consumers
  o Number of consumers on different locations

• High level, application to runtime transfer of information.
  o Maintain appropriate abstraction.

• Zero-copy
  o move semantics transfer between locations

• Immutable sharing
  o locations share read-only access

• Algorithm guided message aggregation
  o Minimize amount of data sent between locations
Partial Evaluation of RMIs

- Algorithm may know something about the pattern of \texttt{p\_object} method invocations
  - For example, homogeneous vertex updates in graph
  - Destination location, container, and method invoked are constant

- Partial evaluation fixes all parameters except value passed
  \[
  f = \texttt{bind(\texttt{async\_rmi}, \texttt{dest}, \texttt{obj}, \&\texttt{A::recv, _1});}
  \]

- User function invokes RMI with different values
  \[
  f(10); // \texttt{call A::recv(10)}
  f(5); // \texttt{call A::recv(5)}
  \]

- Data transferred between locations minimized
  - Location, \texttt{p\_object} handle, and function pointer occur once in buffer
  - Each value passed also in buffer
  - Reduces runtime checks, increases message aggregation
Connected Components using Partial Evaluation

- Connected Components algorithm on Newman-Watts-Strogatz graph

**Diagram:**

IBM-BG/Q: Connected Components on Newman-Watts-Strogatz ($2^{10}$ vertices)

- From 1.5X at 32 cores to 1.7x at 128K cores. Improved scalability.
Kripke

- Sn transport mini-app from LLNL

- Designed to be a representative computational load of ARDRA
  - No real computation
  - Aggregation of discretized energy and angle support similar to PDT
  - Goal to explore different nesting orders of iteration over discretized domains

- Explicit interchange of iteration over energy, direction, and space
  - Kernel implemented for each nesting order to interchange loops

- Explicit MPI+OpenMP
  - Process-level sweep across spatial domain in MPI
  - OpenMP used in subdomain to process spatial, energy, and direction domains
  - Nesting of loops over domains changed to achieve all six combinations
Data Distribution and Loop Nesting

- Five dimensional space
  - 3-D space, 1-D energy, 1-D direction

- Spatial dimensions decomposed across MPI tasks

- One MPI task stores all groups and directions for subset of zones

- Computation
  For each group set
    For all direction sets
      Sweep over spatial domain

- Sweep processes elements of zoneset/groupset/directset

- Order of loops over elements differs for each nesting
Kripke Using STAPL

- Introduced zone sets to abstract across/within node
- Wavefront skeleton across zone sets distributed across nodes
- Parallelism within zoneset matches nesting
  - `zip(zip(wavefront(diamond_difference_op)))` for nesting with energy and direction on outside
  - `zip(wavefront(zip(diamond_difference_op)))`, etc. for other nests
- Wavefront skeleton across zone sets and within zone sets is the same code
  - Sweep within a sweep parallelization is unique to STAPL implementation
- Sliced views provide access to multiarray required by nesting
  - Provides access to hyperplanes of the elements in a container
  - DGZ kernel is `zip(zip(wavefront(diamond_difference_op)))`
  - Requires 1D view of 1D views of 3D views
  - Underlying container is a 5D multiarray
Status

• Implemented DGZ, GZD, and ZGD kernels
  o Remaining kernels have similar structure
  o Intended to reduce time spent propagating interface changes

• Execution using mixed-mode and nested parallelism is working

• Using DGZ kernel to profile overhead and evaluate impact of improvements
Issues Identified

• Degree of parallelism spawned

• Multiarray element access
  o Using slicing inside container to hoist address computation
  o Avoiding use of tuple for multidimensional id to reduce memory allocations

• Heap Allocation
  o Time spent in new/delete a significant sequential overhead

• Read-only container access
  o DGZ distribution results in locations on node remotely requesting sigt
  o Container lacks elements in direction domain
Issues Identified

• Distribution of Containers for Scattering
  o LTimes and LPlusTimes are map operations
  o Distribution across zones*groups provides good locality
  o Sweep requires different distribution depending on nesting
  o Requires redistribution of rhs and psi between scattering and sweep

• Task Graph Creation
  o Includes cost of coarsening transformation on views
  o Data is static, no need to rebuild task graph on every invocation
  o PARAGRAPh persistence amortizes these costs

• Formation of result of nested computation
  o Need more scalable transfer of results between nested PARAGRAPhS on zonesets
Summary

• Mapping computation on to system improves performance

• Nested parallelism distributes work at several levels

• Additional Developments
  o DSL for Skeleton composition in C++
    • Allows inline specification of algorithm composition
    • Eliminates need for additional tool to generate C++ from flow specification

• Kripke
  o Implemented v1.0 and running correctly in mixed-mode with nested parallelism
  o Overhead (sequential and parallel) being identified and addressed
    • Heap allocations
    • View coarsening
    • Returning result of nested computation
    • Redistribution between scattering and sweep