Center for Exascale Radiation Transport

Toward Exascale Computing with STAPL
Lawrence Rauchwerger
Parasol Lab, Computer Science and Engineering
Outline

1. State of PDT: Peta
2. Intro to STAPL
3. Roadmap from Peta to Exa
4. Taking STAPL from Peta to Exa
5. Building an Exa PDT on top of STAPL using TAXI
6. Beyond PDT: Our contributions to Comp Sci at Exascale
Exa – Peta ... where is PDT now?

- Previously developed PDT using PTTL library
  - Scale to 128k cores

- Currently developing PDT using STAPL library
  - STAPL is general purpose parallel library
  - 75 K LOC in PDT only
  - Scales beyond 393k processors on BG/Q (Sequoia)

- Exploit space (geometry) level parallelism
  - Sequence of parallel sweeps across the *rectangular* grids with Pipelined directions
  - Asynchronous (step wise) communication
  - No fault tolerance
  - Homogeneous computer system (BG/Q) – no accelerators
STAPL: Standard Template Adaptive Parallel Library

A library of parallel components that adopts the generic programming philosophy of the C++ Standard Template Library (STL).

- **STL**
  - *Iterators* provide abstract access to data stored in *Containers*.
  - *Algorithms* are sequences of instructions that transform the data.

- **STAPL**
  - *pViews* provide abstracted access to distributed data stored in *pContainers*.
  - *pAlgorithms* specified by *PARAGRAPHs*, parallel task graphs that transform the input data.
    - Can use existing *PARAGRAPHs*, defined in collection of common parallel patterns.
    - *Extensible* - users can define new patterns.
pViews & pContainers

- A pView defines an abstract data type (ADT) for the collection of elements it represents.
  - Example: Matrix View of the elements in a pVector

- Provides data access operations to pAlgorithms
- Allows element ordering independent of stored order
- pContainer : pView + Data storage
View Example

- Print elements of Matrix
  - row-wise or column-wise?
  - Implement several print methods…
  - Use one generic print method with different views

Matrix

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Rows view

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Columns view

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

print(View v)
for i=1 to v.size()
do print(v[i])

Output
1,2,3,4,5,6,7,8,9

Output
1,4,7,2,5,8,3,6,9
Peta: How do we do it now? Using STAPL

- STAPL = Standard Template Adaptive Parallel Library
- 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: Task Graphs (PARAGRAPhS)* (and Marv showed you some examples for sweeps)
inner_product(View1 v1, View2 v2) {
    return map_reduce(
        multiplies(), plus(),
        v1, v2
    );
}

• inner_product() specified by PARAGRAPH.

• Employs map_reduce parallel pattern.

• Defines a new pattern we can use to \textit{compose} a \textit{nested} PARAGRAPH.
Matrix Vector Multiplication

matvec(View2D A, View1D x) {
    using functional::inner_product;
    return map_func(inner_product(), full_overlap(x), A.rows());
}

View transformations and PARAGRAPh reuse in composition enable an exact, succinct specification of matvec task graph.
Example: NAS CG in STAPL

cg_iteration(View2D A, View1D p, Ref rho, ...) {
    q       = A * p;
    alpha   = rho / inner_product(q, p);
    new_z   = z + alpha * p;
    new_r   = r - alpha * q;
    new_rho = inner_product(new_r, new_r);
    beta    = new_rho / rho;
    new_p   = new_r + beta * p;
...
}

- Operator overloads call pAlgorithms: A * p \rightarrow\text{matvec}(A, p)
- Sequence composition is non blocking:
  \textit{Specification proceeds concurrently with execution.}
- \textit{NO Barriers -- Only point-to-point communication/synchro}
- For simplicity / space, we next consider just the first two statements.
Example: Sequence Composition - CG

Matvec() pAlgorithm on 2D_view of pMatrix and 1D_view of pArray.

\[ q = A \times p; \]

Inner product of two 1D_view views whose scalar result is divisor of dividend rho.

\[ \text{alpha} = \frac{\rho}{\text{inner_product}(q, p)}; \]

Expressive syntax quickly yields nested/hierarchical PARAGRAPHs.
NAS CG Peta- Scalability

STAPL implementation versus NAS MPI implementation.

BG/Q
Sequoia @ LLNL.

![Graph showing scalability comparison between NPB and STAPL implementations on NAS CG and LLNL BlueGene/Q.](image)
NAS EP Peta – Scalability

- LLNL BlueGene/Q System
  - 16-core PowerPC A2 processor per node
  - 16GB RAM per node
  - Nodes connected in 5-D torus

- NAS EP
  - Transforms stream of uniformly distributed random numbers into normally distributed stream.
  - Combines statistics of each processor’s output stream to validate.

- STAPL implementation scales as well as native Fortran+MPI to one million cores.
Where is PDT Now? In PETA

Model

- Simple estimate of execution time
  \[ \text{Time} = \text{Computation} + \text{Communication} \]
- Uses seq. exec. time for computation
- MPI bandwidth/latency measured at execution startup
- Assumes messages processed immediately

Input

- Size of spatial domain increased with core count, maintains size of cells and mean-free paths
- Number of unknowns/core kept constant
From Peta to Exa: Not so fast...because:

- Amdahl’s law effect 1000 stronger
  \(\text{(Reduce (global) communication(synchronizations))}\)
  \(\text{More asynchrony, More parallelism !!!}\)

- Load imbalance on many cores is much worse
  \(\text{(Load balancing/scheduling)}\)

- More processors \(\Rightarrow\) More failures
  \(\text{Fault tolerant SW (app + system) at every level}\)

- Communications too complex to be specified explicitly
  \(\text{(raise level of abstraction)}\)

- System too complex for simple models
  \(\text{(need adaptive model)}\)
Our Roadmap from Peta to Exa

Immediate Plans:

- Exa scalable STAPL
- Fault tolerant STAPL → fault tolerant DSL & PDT
- TAXI: A Domain Specific Library (DSL) for Rad. Transport (built on top of STAPL)

Longer Term:

- Adaptive STAPL → Adaptive PDT (all levels!)
  - Tunable granularity: Fine ↔ Coarse Grain Algorithms
  - Communication aggregation
  - Load balancing
  - Use a fraction of processors for monitor/control performance

- Study Approximate methods for TAXI (and STAPL) to improve scalability, in context of UQ
Some features for Exa-scalable STAPL

- Asynchronous Algorithms
- Nested/Hierarchical parallelism (parallel algorithms)
- Extension to heterogeneous architectures - GPUs
- Special support for
  - AMR (space/angle)
  - Arbitrary grids, sparse data structures
- Adaptive behavior
  - Granularity control of tasks (data + work)
    - Fine \(\rightarrow\) Coarse Grain Parallel Algorithms Morphing
  - Communication/Synch aggregation AND Customization (pt2pt)
- Algorithmic Composition for Productivity & Performance
  (skeleton library + composition operators)
Asynchronous Algorithms & Communication

➢ Asynchrony ➔ Latency Hiding

➢ Asynch communication: STAPL: ARMI comm. Library
  ▪ Asynch active messages – never waits for a return value.
  ▪ Futures – place holders for return values not yet computed but needed for current evaluation (increases asynchrony).
  ▪ Recursively nested communication subgroups (and subcontainer registration) ➔ locality, load balancing + affinity, work reduction (efficiency)

➢ Asynchronous Algorithms – not an easy task …
Asynch Algos: Tunable BFS

- Level-synchronous BFS
  - Barrier after every level across all locations
  - Used in PBGL

- Asynchronous BFS
  - Removing sync may improve performance
  - High out-degree results in multiple tasks visiting the same vertex, repeating work

- Tunable BFS
  - Allows user to specify degree of asynchrony
  - Can outperform fully (sync or async) versions
Asynch Algos: Tunable BFS

- Scalability on a large graph
  - Synthetic road network
  - Multiple (8 or 32) copies of European road network joined to form a single large graph
  - Experiment ran on NERSC Cray XE6 system (hopper)

- KLA outperforms Level-Sync
  - Removing sync improves performance
  - “Ideal” line plots linear scalability
  - Level-Sync performance degrades on higher core counts as cost of global synchronization increases
Nested, Hierarchical Algos and RT

- Nested parallelism:
  \[ \text{While}\{\text{forall}\ (\text{reduce}\ (\text{sweep}\ (\ldots)))\}\}\]

- Hierarchical parallelism (algos): nested and mapped onto the machine memory hierarchy
  \[ \text{forall}\ (\text{view}_i, \text{forall}\ (\text{view}_j, \text{wf}\{\}))\text{ where}\]
  \[ \text{view}_i = U\{\text{view}_j\} \text{ and mapped hierarchically on machine hierarchy (Locales)}\]

- Support for various Runtime Systems (MPI/OpenMP/Pthreads…recursive constructs)

- Nested/Hierarchical \(\Rightarrow\) Latency reduction (locality) + Expressivity (and productivity)
Support for Heterogeneous Architectures

- Storage: STAPL is distributed and GPU means another address space (Locale)
- Algos+Code: GPUs use different code, algorithms than CPU (needs engineering)
- STAPL: Will enable easy memory tracking— all data structures have GIDs.
- STAPL will enable simpler programming but not make compiler/user level decisions.
Adaptive Graph-based Data-Structures

- AMR (space and angle): STAPL hierarchical, nested data structures
- **Load Balancing** for above:
  - *Adaptive work-stealing Task (Data+Work) migration when work/vertex is unknown*
  - *Bulk-synchronous redistribution when work/vertex known*
Peta to Exa: Fault tolerance via STAPL

- STAPL – distributed \textbf{virtualized} system makes it easier
- Fault tolerant STAPL components $\rightarrow$ Fault tolerant composed program
- Fault Detection: extend ARMI + other techniques
- Fault Recovery: Distributed Checkpoints + Task graph replication
  - Groups of re-work processors/memory (plenty of them)
  - Paragraphs with built-in replication/redundancy
  - See Manteuffel’s coarse grain replication of data

Open Question:
- Fault resilient algorithms: error $\iff$ fault tolerance
Further Research: Program composition

- Algorithmic skeletons + compositional operators
  
  \[
  \text{Transport} = \\
  \text{While(}!\text{converged}\text{)}(\text{Forall(energy)}(\text{Reduce(Sweep(grid_view)})))
  \]

- Merging of skeletons  \(\rightarrow\) less global synchronizations

- Composition of attributes (e.g., fault tolerance, performance models, …)

- Granularity control: Fine-grain specification, coarsen task graph (static/dynamic) \(\rightarrow\) expressive, virtualized, higher performance

- Dependence graph transformations  \(\rightarrow\) polymorphic, adaptive algorithms

- For Example: Sweep \(\leftrightarrow\) Block-Jacobi :

- Single composed parametric polymorphic Algo
Open Question: Approximate Computation

- Increased Asynchrony requires
  - tolerance of stale info
  - otherwise approximating it
  - Example: use of old data in sweeps on re-entrant graphs

- Relaxation of dependences to keep computation local

- Non-determinism

- Tradeoff: Algorithm induced error $\leftrightarrow$ performance (parallelism)

- UQ in the presence of approximate computation
TAXI library will contain data structures and algorithms for radiation transport

- Extend STAPL data structures (Graph -> Grid)
- Composition of Algorithms (skeletons) into transport specific algorithms (simultaneous sweeps)
- BiCG, etc
- Composition of building blocks will allow Transport exploration
Beyond PDT and TAXI: Contributions to Exascale Issues in CompSci

- Exa-scaled parallel *generic* library STAPL

- Answers to many general questions:
  - AMR/Arbitrary Grids
  - Fault tolerant STAPL Library and trade-offs with speed
  - Hierarchical/Heterogeneous parallelism mapped on H/H Machines
  - Dynamic Load Balancing
  - Transformation between Fine-Coarse grain of algorithms
  - Asynchrony, (weaker) memory consistency and programming productivity tradeoffs.

- How to build a useful DSL

- Make peta scale good for general use.

Almost nothing presented is Transport exclusive!
Backup Slides
STAPL graph library outperforms Boost

Parallel Boost Graph Library (PBGL) Graph 500 reference implementation

Algorithms used:
- BFS kernel from Graph 500
- PageRank

Throughput for PageRank in SGL vs PBGL on Hopper
Weak scaling with $2^{17}$ vertices per location.

Throughput for BFS in SGL vs PBGL on Hopper
Weak scaling -- Mega Traversed Edges per Second
Example - STAPL Inner Product

array<int> arr1(100);
array<int> arr2(100);

view_1D view1(arr1);
view_1D view2(arr1);

x = inner_product(view1, view2);

In specification, user can be unaware of data distribution and locality.
STAPL Components

- **Application Development**
  - *Shared object view eliminates explicit communication in application.*

- **pAlgorithm Development**
  - *PARAGRAPHER a function of:*
    - User specified operation
    - Dependence Pattern
    - pViews
  - **Dependence Pattern**
    - Necessary and sufficient dependence information.
    - Input independent specification of dependencies.
    - Parametrically expanded as function of pView size.
STAPL Components

- **pContainer Development**
  - **pContainer Composition** (pContainer of pContainers).
  - **Parallel Container Framework (PCF)** provides components to develop a new pContainer and allows new components to be introduced.

- **Portability and Optimization**
  - **Runtime System (RTS)**
    - Adaptive Remote Method Invocation (ARMI)
    - Task Execution and Scheduling
  - **Physical processors abstracted to locations**
  - **Framework for Algorithm Selection and Tuning (FAST)**
SGL: STAPL Graph Library

- Implements a Distributed Parallel Graph
- Shared object (unified address) view
- Graph View: ADT (Abstract Data Type)
- Recursive Hierarchical Graph View
  - Expresses levels of resolution (scale) well
  - Compact, scalable – used in PDT
SGL: Scalable Graph Algorithms

- Scalability of KLA BFS was shown earlier

- SGL provides several algorithms
  - BFS
  - k-core
  - PageRank
  - etc.

- Weak scaling of PageRank on 3D Toroid
  - Experiment run on NERSC Cray XE6 system (hopper)
Hierarchical graph algorithms

- Hierarchical graph
  - Supervertices contain subgraphs
  - A single edge represents all cut edges

- Hierarchical graph algorithm
  - Operates on supervertices
  - Supervertex update processes fine-grain subgraph

- Algorithm can differ across levels of hierarchy
  - Process subgraph sequentially
  - Process subgraph recursively

- Pointer Jumping benefits from hierarchical expression.
Hierarchical graph algorithms

- Algorithm used on subgraph can use external library
- Green-Marl
  - Graph analysis DSL from Stanford
  - Shared-memory only
- Hierarchical graph has subgraph vertices on a shared-memory node
- Green-Marl used to process subgraphs
SGL Comparison to Other Libraries

- Boost Parallel Graph Library (PBGL)
- Graph 500 reference implementation
- Algorithms used:
  - BFS kernel from Graph 500
  - PageRank
- Evaluated on:
  - Cray XE6 at NERSC
  - Opteron cluster at TAMU

STAPL performs well at all scales.
SGL Comparison to Other Libraries

- GraphLab
- MultiThreaded Graph Library (MTGL)
- Algorithms used:
  - BFS over various graphs
  - BFS kernel from Graph 500
- Evaluated on
  - Cray XE6 at NERSC
  - Opteron cluster at TAMU

STAPL performs well at all scales.
Load Balancing for Parallelization of Sampling-based Motion Planning Algorithms

- Motion planning problem
  - Compute feasible paths through an environment for a movable object.
  - Applications range from robotics to CAD to protein folding
  - Best methods are sampling-based planners
  - Samples connected to neighbors to form a “roadmap” graph, implemented using STAPL pGraph

- Uniform Spatial subdivision techniques
  - Scale well
  - Prone to imbalance due to heterogeneity of spatial regions assigned to processors

- Two approaches studied
  - Bulk-synchronous redistribution
  - Adaptive work-stealing
Experimental Results

- Rebalancing outperforms work stealing when estimate of work per vertex is accurate.

- Work stealing is better when load estimate is inaccurate.
  - Repartitioning increased imbalance in this case
pViews

- A pView defines an abstract data type for the collection of elements it represents.
  - Example: Matrix View of the elements in a pVector

- Provides data access operations to pAlgorithms

- Allows element ordering independent of stored order
Print elements of Matrix

- row-wise or column-wise?
- Implement several print methods…
- Use one generic print method with different views

Output
1,2,3,4,5,6,7,8,9

Output
1,4,7,2,5,8,3,6,9

print(View v)
for i=1 to v.size() do
  print(v[i])
pContainers: Parallel Containers

- **pContainer**: Data storage + Abstract Data Type
  - *In STAPL, ADT is pView (LCPC 2010)*
  - **Sequential Containers**
    - STL (vector, list, map, set, hash), MTL\(^1\) (matrix), BGL\(^2\) (graph), etc.

- **pContainer properties**
  - *Distributed non-replicated storage*
  - *Concurrent, thread safe methods*
  - *Shared Object View*
  - *Support for user customization (e.g., data distributions)*

---

\(^1\) Matrix Template Library \(^2\) Boost Graph Library
pGraph

- Collection of vertices and edges

\[
\text{stapl::graph<DIRECTED/UNDIRECTED, MULTI/NONMULTIEDGES, [VertexProperty, EdgeProperty, Partition, Traits]>
}
\]

- Unique descriptors; No ghost nodes - no data replication
- Support for different storage : Boost Graph, STAPL sequential graph
- pGraph is designed to use different graph partitions

---

**Diagram:**

- Vertices and edges depicted with numbers and arrows.
- Example of graph structure with vertex properties.
- Partitioning of vertices across different locations.

---

**References:**

- [Graph Theory](http://example.com/graph_theory)
- [STAPL Graph Library](http://example.com/stapl)

---

**Location Assignments:**

- Location i: 1, 4, 7
- Location j: 2, 5
- Location k: 3, 6
- **Notes:**
  - Unique descriptors ensure no data replication.
  - Support for different storage formats.
  - Designed for use with various graph partitions.
pGraph Inputs

- **Graph500 input**
  - Designed to simulate small-world, scale-free graphs (e.g. internet, social networks)

- **Texas Road Network**
  - All roads and intersections of Texas state highways

- **Torus**
  - $x \times y$ vertices per processor, $2 \times x \times y$ edges per processor
  - $x$ cross boundary edges

![Diagram of a torus graph]