STAPL: The Standard Template Adaptive Parallel Library

Lawrence Rauchwerger
http://parasol.tamu.edu/~rwerger
Parasol Lab, Dept of Computer Science, Texas A&M
SmartApps: Written in STAPL

- **STAPL:** Parallel components library
  - Extensible, open ended
  - Parallel superset of **STL**
  - Sequential inter-operability

- Layered architecture: User – Developer - Specialist
  - Extensible
  - Portable (only lowest layer needs to be specialized)

- **High Productivity Environment**
  - Components have (almost) sequential interfaces.
STAPL Specification

- **Shared Object View**
  - User Layer: No explicit communication
  - Machine Layer: Architecture dependent code

- **Distributed Objects**
  - no replication
  - no software coherence

- **Portable efficiency**
  - Runtime System virtualizes underlying architecture.

- **Concurrency & Communication Layer**
  - SPMD (for now) parallelism
STAPL Overview

Adaptive Framework

User Application Code

pAlgorithms

Views
pContainers

pRange

Run-time System

ARMI Communication Library

Scheduler

Executor

Performance Monitor

Pthreads

OpenMP

MPI

Native
STAPL Overview

- **pContainers** - parallel equivalent of STL container.
- **pView** - analogous to STL iterator, provides a generic interface to pContainer for pAlgorithms.
- **pAlgorithms** - parallel STL equivalents and more (graph traversals, parallel prefix, etc).
- **pRange** - bind possibly heterogeneous work and data together in a uniform manner for execution. Used to implement pAlgorithms.
- **RTS & ARMI** – Run-time system and communication primitives.
- **STAPL Applications** - particle transport, protein folding.
## Related work

<table>
<thead>
<tr>
<th>Features/Project</th>
<th>STAPL</th>
<th>Charm++</th>
<th>PSTL</th>
<th>HTA</th>
<th>POOMA</th>
<th>Titanium</th>
<th>pBGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language/Library</td>
<td>Lib</td>
<td>Lang</td>
<td>Lib</td>
<td>Lib</td>
<td>Lib</td>
<td>Lang</td>
<td>Lib</td>
</tr>
<tr>
<td>Memory Address Space</td>
<td>Shared</td>
<td>Shared/ Part</td>
<td>Shared</td>
<td>Shared</td>
<td>Shared/ Part</td>
<td>Shared/ Part</td>
<td></td>
</tr>
<tr>
<td>Programming Model</td>
<td>SPMD/ MPMD</td>
<td>MPMD</td>
<td>SPMD</td>
<td>SPMD</td>
<td>SPMD</td>
<td>SPMD/ MPMD</td>
<td>MPMD</td>
</tr>
<tr>
<td>Generic Data Type/ Generic Algorithms</td>
<td>Y/Y</td>
<td>Y/N</td>
<td>Y/Y</td>
<td>Y/N</td>
<td>Y/N</td>
<td>Y/Y</td>
<td>Y/Y</td>
</tr>
<tr>
<td>Reuse Seq Containers</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Framework for pContainers</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Data Structures</td>
<td>A,V,L,G, M</td>
<td>V</td>
<td>V,L</td>
<td>M</td>
<td>A</td>
<td>A</td>
<td>G</td>
</tr>
<tr>
<td>(Array, Vector,List, Graph, Matrix)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Views</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Data Partition/ Mapping</td>
<td>Y/Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Outline

- pContainers and Views
- pRanges
- pAlgorithms
- ARMI
- Algorithm Selection Framework
- Applications Using STAPL
STAPL pContainers

pContainer: A generic, distributed data structure with parallel (thread-safe) methods

- **Ease of Use**
  - Shared Object View
  - Handles data distribution and remote data access internally (no explicit communication)
  - Interface Equivalent with Sequential Counterpart
  - Thread Safe with well defined consistency model

- **Efficiency**
  - De-centralized distribution management
  - OO design to optimize specific containers
  - Minimum overhead over STL containers

- **Extendability**
  - A set of base classes with basic functionality
  - New pContainers can be derived from Base classes with extended and optimized functionality
pContainer Modules

- **Base pContainer**
  - Interface to sequential containers that actually store data
  - Provides default interfaces, views, and functionality for distributed data structures
  - One representative per computational thread, the union is the pContainer.

- **Distribution Manager**
  - Interface to the machine
  - Mechanisms to locate data
  - Maps data to physical locations

- **pContainer View**
  - Interface to pAlgorithms
  - Generic access interface to the data
  - Specialized views, e.g., row or column views for matrices
pContainer Distribution Manager

- Responsible for data distribution on the machine
  - *Partition* – describes how the data space of a pContainer is partitioned in blocks
  - *Data Mapping* – maps a pContainer partition onto the machine
  - *Location Info* – provides mechanism for locating individual data elements
    - cooperates with Partition & Data Mapping
    - Default mechanism provides 2 step location of any element
      - Can be optimized, e.g., fixed mapping for static pArray
pContainer View

- Generic access mechanism to pContainer data
- STAPL equivalent of STL iterator, extended to support parallelism
- Support simultaneous multiple views without replicating data
  - E.g., row view and column view of the same pMatrix
- Hierarchical Views support heterogeneous systems and nested parallelism

Example

Gray areas - the pContainer physical partition.
Transparent areas - different logical views of the data.
pContainer

Data Traversal and Access

- Elements included in a view are accessed using view iterator
- View iterator accesses elements independent of physical location
  - Shared memory view
- View iterator provides the same interface as an STL iterator
  - Useful when reusing STL algorithms in parallel implementations

```cpp
Class WorkFunction {
  Void operator()(Subview s) {
    for (Subview::iterator si = s.begin(); si != s.end(); ++si) {
      foo(*si);
    }
    std::for_each(s.begin(), s.end(), foo);
  }
}
```

//allocate a pArray of size 4
pArray p_a (4);

View v = p_a.create_view (block_size=2);
//allocate a blocked view with block size 2
//two subviews will be created

pRange (v, work_function);
//associate a view with an work function;
//pRange will create two tasks corresponding to the //two subviews

p_for_all (pr);
//apply work function to data in subviews in parallel
Outline

- pContainers and Views
- pRanges
- pAlgorithms
- ARMI
- Algorithm Selection Framework
- Applications Using STAPL
pRange Overview

- Interface between pAlgorithms and pContainers
  - pAlgorithms expressed in terms of pRanges
  - pContainers provide Views of data used in pRanges

- Parallel programming support
  - Expression of computation as parallel task graph
  - Stores DDGs used in processing subranges

- Less abstract than STL iterator
  - Views in a pRange provide access to pContainer methods
pVector<int> points(22);
pRange<
  tuple<pVector<int>::view_type> >
  point_pr(points.create_view());
RandomFunctionObject rand(N); //produces vals in [0,N]
pGenerate(point_pr, rand);

Thread 0

Thread 1

pRange

Subrange

Subrange

Subview

Subview

Task

Task
pRange

- View of a work space
  - Set of tasks in a parallel computation
  - pRange Tasks set by pAlgorithm
  - pRange processed by Executor

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pRange

- Subrange
  - Subview
    - Task

- Subrange
  - Subview
    - Task
pRange Tasks

- Task in a pRange contains
  - Id of subrange it will operate on
  - Function object
- Multiple tasks can operate on the same subrange

![Diagram of pRange Tasks]

- **Task**
  - Subrange ID
  - Work Function

- **Thread 0**
  - Subrange
  - Subview
  - Task
  - Task

- **Thread 1**
  - Subrange
  - Subview
  - Task
  - Task
Task Dependencies

- One task may produce data used by another
- pRange can store a task dependence graph
  - Used by Executor to enforce processing order
- pRange provides methods to use commonly occurring dependence patterns.
  (e.g. recursive doubling, binary tree, no dependence)
Outline

- pContainers and Views
- pRanges
- pAlgorithms
- ARMI
- Algorithm Selection Framework
- Applications Using STAPL
pAlgorithms

- **pAlgorithm** is a set of parallel task objects
  - Input for parallel tasks specified by the pRange
  - (Intermediate) results stored in pContainers
  - ARMI for communication between parallel tasks

- **pAlgorithms in STAPL**
  - Parallel counterparts of STL algorithms provided in STAPL
  - Common parallel algorithms
    - Prefix sums
    - List ranking
  - pContainer specific algorithms
    - Strongly Connected Components (pGraph)
    - Euler Tour (pGraph)
    - Matrix multiplication (pMatrix)
pAlgorithms Example: p_accumulate

template<typename PRange, typename BinaryFunction>
class p_accumulate_w : public work_function_base<PRange>
{
public:
    void operator()(typename PRange::subview_set_type& subrange_data)
    {
        this->m_subrange_result = //Invoke STL Algorithm on subrange
            std::accumulate(at<0>(subrange_data)->begin(), at<0>(subrange_data)->end(), ...);
    }
};

template<typename PRange, typename T, typename BinaryFunction>
T p_accumulate(PRange& pr, T init, BinaryFunction binary_op)
{
    p_accumulate_w wf(pr, binary_op); //Instantiate Work Function
    pr.create_tasks(wf); //Assign a work function to prange
    tPRange::result_storage_type result_storage(pr); //Define temporary storage (one value per subrange)
    task_result_aggregator
        aggregator (result_storage.prange(), wf); //Create an aggregator for results of all subranges
    p_for_all(pr, result_storage.prange(), wf); //Apply work function to prange
    T result = binary_op(init, aggregator.result()); //Aggregate result
    return result;
}
pAlgorithms Example: p_accumulate

- Assign a work function to prange (i.e. addition)
- Define temporary storage (i.e. one value per subrange)
- Create an aggregator to aggregate results of all subranges
- Apply work function to prange
- Aggregate result

Subrange 0: 8 2 5 7
Subrange 1: 6 3
Subrange 2: 4 9 1

Temp. storage: 1 element per subrange

- Subrange 0: 22
- Subrange 1: 9
- Subrange 2: 14

Final result: 45
Outline

- pContainers and Views
- pRanges
- pAlgorithms
- RTS & ARMI
- Algorithm Selection Framework
- Applications Using STAPL
SmartApps RTS – not today

Application Specific Parameters

SmartApps Run-time System

Application

ARMI

Memory Management

K42 User-level Scheduler

Executor

Operating System (K42)
Outline

- pContainers and Views
- pRanges
- pAlgorithms
- ARMI
- Algorithm Selection Framework
- Applications Using STAPL
Algorithm Selection

Given multiple implementations of an abstract operation with a specified execution environment and input data, choose one that maximizes performance.

Abstract operation – sorting, matrix multiplication, convex hull.

Execution Environment
  Architecture – number of processors, memory interconnection, cache.
  OS/runtime - thread management, memory allocation, OS policies.

Input Data – data type, layout, size, other properties known to affect implementations (i.e., input disorder for sorting).

Performance – time to completion, power consumption.
Our Approach

- Create general framework for parallel algorithm selection.
  - Flexible specification of execution environment and input data parameters
  - Generic modeling interface – interchange different approaches. Use generic machine learning approaches.

- Integrated approach within STAPL.
  - Selection is transparent to end user.
  - Library adaptively choose the best algorithm at run-time from a library of implementation choices.
Overview of Approach

- **Given**
  Multiple implementation choices for the same high level algorithm.

- **STAPL installation**
  Analyze each algorithm's performance on system and create a selection model.

- **Program execution**
  Gather parameters, query model, and use predicted algorithm.
Model Generation

Goal
Create a model to predict the “winning” algorithm in each case.

No attempt is made to model the absolute performance of algorithms.

Generic interface enables learners to compete

- Implemented – decision tree, neural network, Bayes naïve classifier.
- Model selection based on estimated accuracies.
  (10-fold cross validation).
Experiments

- Investigated two operations
  - Parallel Sorting
  - Parallel Matrix Multiplication

- Three Platforms
  - 128 processor SGI Altix at TAMU (*Altix*).
  - 1152 node, 2 proc. Intel cluster at LLNL (*Linux Cluster*).
  - 68 node, 16 way IBM cluster at LLNL (*SMP Cluster*).
Parallel Sorting Algorithms

- **Sample Sort**
  - Low disorder in input minimizes processor communication.

- **Radix Sort N**
  - Performs best on an input with small range.
  - Works for integers only.

- **Column Sort**
  - Theoretically optimal and scales well.
  - Struggles on small inputs.
Parallel Sorting Attributes

Attributes used to model sorting decision

- Processor Count
- Data Type
- Input Size
- Max Value
  - Smaller value ranges favor radix sort by reducing passes
- Presortededness
  - Initial level of disorder known to greatly impact performance of sequential algorithms.
Sorting Attributes - Presortededness

- **Instance Generation**
  - Select initial disorder (sorted, reversed, random)
  - Randomly displace percentage of elements

- **Runtime Characterization**
  - Sort a sampling of the input data (sample sort)
  - Compute normalized average distance
    \[
    \frac{\sum |\text{index}_{\text{sorted}} - \text{index}_{\text{initial}}|}{\text{size(sample)}^2}
    \]

  \[
  \begin{array}{cccccccc}
  \text{List}_{\text{initial}} & 6 & 3 & 5 & 2 & 4 & 7 & 1 \\
  \end{array}
  \]

  \[
  \begin{array}{cccccccc}
  \text{List}_{\text{sorted}} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  \end{array}
  \]

  distance = 6-1 = 5

**Interpretation of dist_norm**

- Sorted
- Random
- Reversed
## Sorting: Creation of Problem Instances

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Training (TR)</th>
<th>Validation 1 (V1)</th>
<th>Validation 2 (V2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors*</td>
<td>2, 4, 8, 16, 32, 64</td>
<td>2, 4, 8, 16, 32, 64</td>
<td>2, 4, 8, 16, 32, 64</td>
</tr>
<tr>
<td>Data Type (DT)</td>
<td>int, double</td>
<td>int, double</td>
<td>int, double</td>
</tr>
<tr>
<td>Input Size (N)</td>
<td>(100K..20M)* P / sizeof(DT)</td>
<td>80M, 120M</td>
<td>(100K..20M)* P / sizeof(DT)</td>
</tr>
<tr>
<td>Max Value</td>
<td>N/1000, N/100, N/10, N, 3N, MAX_INT</td>
<td>N/1000, N</td>
<td>N/1000, N/100, N/10, N, 3N, MAX_INT</td>
</tr>
<tr>
<td>Input Order</td>
<td>sorted, reversed, random</td>
<td>sorted, reversed, random</td>
<td>sorted, reversed, random</td>
</tr>
<tr>
<td>Displacement</td>
<td>0..20% of N**</td>
<td>5% of N**</td>
<td>0..20% of N**</td>
</tr>
<tr>
<td># of Instances</td>
<td>1000 (Random)</td>
<td>120 / 144</td>
<td>1000 (Random)</td>
</tr>
</tbody>
</table>

*P = 64 only on Linux Cluster, SMP Cluster  
**only for sorted and reversed

Instance groups are disjoint.
Parallel Sorting: Experimental Results

Altix Selection Model

if \( p \leq 8 \) then
  \( \text{sort} = \text{“sample”} \)
else
  if \( \text{dist
norm} \leq 0.117188 \) then
    \( \text{sort} = \text{“sample”} \)
  else
    if \( \text{dist
norm} \leq 0.370483 \) then
      \( \text{sort} = \text{“column”} \)
    else
      \( \text{sort} = \text{“sample”} \)
  end if
end if

Altix Validation Set (V1) – 100% Accuracy

<table>
<thead>
<tr>
<th>N=120M</th>
<th>MaxElement = 120,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc:</td>
<td>4 8 16 32</td>
</tr>
<tr>
<td></td>
<td>4 8 16 32</td>
</tr>
<tr>
<td></td>
<td>4 8 16 32</td>
</tr>
<tr>
<td></td>
<td>4 8 16 32</td>
</tr>
</tbody>
</table>

- Radix
- Sample
- Column
- Adaptive

Normalized Execution Time

Adaptive Performance Penalty
Parallel Sorting Adaptive Selection (PPoPP ’05)

- Model obtains 99.7% of the possible performance.
- Next best algorithm (sample) provides only 90.4%.
Outline

- pContainers and Views
- pRanges
- pAlgorithms
- ARMI
- Algorithm Selection Framework
- Applications Using STAPL
Discrete Ordinates Particle Transport Computation: TAXI

- Important application for DOE
  - E.g., Sweep3D and UMT2K
- Large, on-going DOE project at TAMU

One sweep  
Eight simultaneous sweeps
TAXI Experimental Results

- uBGL - 1024 node, 2048 processor unclassified Blue Gene cluster at LLNL.

- Results shown for two machine configurations.
  - Virtual Mode - two independent computation processes per node.
  - Coprocessor - one processor per node for computation and one for communication.
TAXI Results on uBGL

![Graph showing uBGL Strong Scaling with lines for Coprocessor Overall, Coprocessor Sweep, Virtual Overall, and Virtual Sweep. The x-axis represents the number of nodes, and the y-axis represents speedup. The graph shows linear trends for all categories as the number of nodes increases.]
TAXI Results on uBGL

TAXI Execution Times on uBGL

- **Sweep**
- **Initialization**

<table>
<thead>
<tr>
<th>Coprocessor Nodes</th>
<th>Virtual Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
</tr>
</tbody>
</table>

Time (seconds)
STAPL – going corporate

- Intel library for multicores
- Microsoft