STAPL
Standard Template Adaptive Parallel Library

Lawrence Rauchwerger
Antal Buss, Harshvardhan, Ioannis Papadopoulous,
Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas,
Xiang Xu, Mauro Bianco, Nancy M. Amato

http://parasol.tamu.edu/stapl

Dept of Computer Science and Engineering, Texas A&M
Motivation

- Multicore systems: ubiquitous
- Problem complexity and size is increasing
  - Dynamic programs are even harder
- Programmability needs to improve
- Portable performance is lacking
  - Parallel programs are not portable
  - Scalability & Efficiency is (usually) poor
A library of parallel components that adopts the generic programming philosophy of the C++ Standard Template Library (STL)

**Application Development Components**
- pAlgorithms, pContainers, Views, pRange
- Provide Shared Object View to eliminate explicit communication in application

**Portability and Optimization**
- Runtime System (RTS) and Adaptive Remote Method Invocation (ARMI) Communication Library
- Framework for Algorithm Selection and Tuning (FAST)
Three STAPL Developer Levels

- **Application Developer**
  - Writes application
  - Uses pContainers and pAlgorithms

- **Library Developer**
  - Writes new pContainers and pAlgorithms
  - Uses pRange and RTS

- **Run-time System Developer**
  - Ports system to new architectures
  - Writes task scheduling modules
  - Uses native threading and communication libraries

![Diagram of STAPL Developer Levels]
Applications Using STAPL

- Particle Transport - PDT
- Bioinformatics - Protein Folding
- Geophysics - Seismic Ray Tracing
- Aerospace - MHD
  - Seq. “Ctran” code (7K LOC)
  - STL (1.2K LOC)
  - STAPL (1.3K LOC)
pContainers: Parallel Containers

- **Container** - Data structure with an interface to maintain and access a collection of generic elements
  - STL (vector, list, map, set, hash), MTL\(^1\) (matrix), BGL\(^2\) (graph), etc.

- **pContainer** - **distributed** storage and **concurrent** methods
  - Shared Object View
  - Compatible with sequential counterpart (e.g., STL)
  - Thread Safe
  - Support for user customization (e.g., data distributions)
  - Currently Implemented: pArray, pVector, pList, pGraph, pMatrix, pAssociative

---

1] Matrix Template Library  2] Boost Graph Library
pContainer Framework

Concepts and methodology for developing parallel containers

– pContainers - a collection of base containers and information for parallelism management

– Improved user productivity
  • Base classes providing fundamental functionality
    ◦ Inheritance
    ◦ Specialization
  • Composition of existing pContainers

– Scalable performance
  • Distributed, non replicated data storage
  • Parallel (semi-random) access to data
  • Low overhead relative to the base container counterpart
**pContainer Framework Concepts**

- **Base Container**: data storage
  - sequential containers (e.g., STL, MTL, BGL)
  - parallel containers (e.g., Intel TBB)

- **Data Distribution Information**
  - Shared object view
  - Global Identifier, Domain, Partition, Location, Partition Mapper

---

```
p_array pa(6)

0 1 2 3 4 5
a b c d e f
```

---

**Base Containers**

```
<table>
<thead>
<tr>
<th>Location 0</th>
<th>Location 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d</td>
<td>a b c d</td>
</tr>
<tr>
<td>e f</td>
<td>d e f</td>
</tr>
</tbody>
</table>
```

---

**Data Distribution**

```
<table>
<thead>
<tr>
<th>Info_0</th>
<th>Info_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c</td>
<td>d e f</td>
</tr>
</tbody>
</table>
```

---

**User Level**

```
-----------------------------
|                           |
-----------------------------
```

pContainer Interfaces

- Constructors
  - Default constructors
  - May specify a desired data distribution

- Concurrent Methods
  - Sync, async, split phase

- Views

```cpp
stapl_main(){
    partition_block_cyclic partition(10); //argument is block size
    p_matrix<int> data(100, 100, partition);
    p_generate(data.view(), rand());
    res=p_accumulate(data.view(),0);
}
```
pGraph Methods

- Performance for add vertex and add edge asynchronously
- Weak scaling on CRAY using up to 24000 cores and on Power 5 cluster using up to 128 cores
- Torus with 1500x1500 vertices per processor
pGraph Algorithms

- Performance for find_sources and find_sinks in a directed graph
- Weak scaling on CRAY using up to 24000 cores and on Power 5 cluster using up to 128 cores
- Torus with 1500x1500 vertices per processor
Views

- A View defines an abstract data type that provides methods for access and traversal of the elements of a pContainer that is independent of how the elements are stored in the pContainer.

- Example: print the elements of a matrix

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

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>

Output

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

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>

Output

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

- Build and execute task graphs to perform computation
  - Task graphs in STAPL are called pRanges

- Easy to develop
  - Work functions look like sequential code
  - Work functions can call STAPL pAlgorithms
  - pRange factories simplify task graph construction

- STAPL pAlgorithms accelerate application development
  - Basic building blocks for applications
  - Parallel equivalents of STL algorithms
  - Parallel algorithms for pContainers
    - Graph algorithms for pGraphs
    - Numeric algorithms/operations for pMatrices
Parallel Find

- Find first element equal to the given value

```cpp
View::iterator p_find(View view, T value)
    return map_reduce(
        view,
        find_work_function(value),
        std::less());
```

```cpp
View::iterator find_work_function(View view)
    if (do_nesting())
        return p_find(view, value)
    else
        return std::find(view.begin(), view.end(), value)
    endif
end
```
Parallel Sample Sort

- pAlgorithm written using sequence of task graphs.

\[
p\text{_sort}(\text{View view, Op comparator})
\]

// handle recursive call
if (view.size() <= get_num_locations())
    reduce(view, merge\_sort\_work\_function(comparator));

sample\_view = map(view, select\_samples\_work\_function());

// sort the samples
p\_sort(sample\_view, comparator);

// partition the data using the samples
partitioned\_view = map(view, full\_overlap\_view(sample\_view),
    bucket\_partition\_work\_function(comparator));

// sort each partition
map(partitioned\_view, sort\_work\_function(comparator));
Scalability of pAlgorithms

Execution times for weak scaling of pAlgorithms on data stored in different pContainers on CRAY XT4.
The STAPL Runtime System (RTS)...

- Abstracts platform resources
  - threads, mutexes, atomics
- Provides consistent API and behavior across platforms
- Configured at compile-time for a specific platform
  - Hardware counters, different interconnect characteristics
- Adapts at runtime at the runtime environment
  - Available memory, communication intensity etc.
  - Provides interface for calling functions on distributed objects
    - ARMI – Adaptive Remote Method Invocation
- There is one instance of the RTS running in every process
  - So it is distributed as well
ARMI: Adaptive Remote Method Invocation

- Communication service of the RTS
- Provides two degrees of freedom
  - Allows transfer of data, work, or both across the system
  - Used to hide latency
- Used to call a function on a distributed object anywhere on the system
- Supports a mixed-mode operation (MPI+threads)

// Example of ARMI use
async_rmi(destination, p_object, function, arg0, ...);
r = sync_rmi(destination, p_object, function, arg0, ...);
The STAPL RTS: Major Components

STAPL Runtime System

Resource Reuse
- Thread Pool
- Memory Manager

Hardware Monitoring
- Hardware Counters
- System resource usage
- Interfaces with TAU and OS provided system calls

Adaptivity
- Different Communication methods
- RMI request aggregation and combining
- Feedback to pContainers / pRange
- Resource acquisition before needed
- Load monitoring and thread allocation
- Detection if on shared memory or have to use MPI

ARMI

Scheduler
- Execution of RMIs
- Consistency Models

Executor
- Execution of tasks provided by the pRange

Multithreaded Support
FAST Architecture

- Framework for parallel algorithm selection
- Developer specifies important parameters, metrics to use
- Developer queries model produced to select implementation
- Selection process transparent to code calling algorithm
Parallel Sorting: Experimental Results

Attributes for Selection Model
- Processor Count
- Data Type
- Input Size
- Max Value (impacts radix sort)
- Presortedness

SGI Altix Selection Model

```java
if p \leq 8 then
    sort = “sample”
else
    if dist_norm \leq 0.117188 then
        sort = “sample”
    else
        if dist_norm \leq 0.370483 then
            sort = “column”
        else
            sort = “sample”
        end if
    end if
end if
```

SGI Altix Validation Set (V1) – 100% Accuracy

Adaptive Performance Penalty
• Model obtains 99.7% of the possible performance.
• Next best algorithm (sample) provides only 90.4%.
PDT: Developing Applications with STAPL

- Important application for DOE
  - E.g., Sweep3D and UMT2K
- Large, on-going DOE project at TAMU to develop application in STAPL
- STAPL precursor used by PDT in DOE PSAAP center
pRanges in PDT: Writing new pAlgorithms

- pRanges are sweeps in particle transport application
- Reflective materials on problem boundary create dependencies
- Composition operator will allow easy composition

```
zip(prA, prB, Zipper(4,32,4));
```
Sweep Performance

- Weak scaling keeps number of unknowns per processor constant.
- Communication increases with processor count.
- KBA Model shows performance of perfectly scheduled sweep.
- Divergence after 2048 processors due to non-optimal task scheduling.
Conclusion

- STAPL allows productive parallel application development

- pContainers and pAlgorithms
  - Application building blocks
  - Simplify development
  - Extensibility enables easy development of new components

- Composition of pContainers and pAlgorithms enable reuse

- RTS and FAST provide portability and adaptability