SmartApps: An Application Centric Computing Paradigm

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

http://parasol.tamu.edu
The New Computing Challenge

- **Today’s Applications**: Bio, Multi-physics, etc
  - Time Consuming => need optimized parallel codes
  - Dynamic/Irregular => need automatic optimization
- **Today’s Systems**: Heterogeneous, Parallel, Distributed, General multi-purpose
- portability => need high-level software tools & libraries
  - efficiency => need automatic optimization techniques
- **The Challenge**: Easy to Use & High Performance
Today: System Centric Computing

Classic avenues to performance:

- Parallel Algorithms
- Static Compiler Optimization
- OS support
- Good Architecture

WHAT’s MISSING?

- Compilers are conservative
- OS offers generic services
- Architecture is generic

No Global Optimization

- No matching between Application/OS/HW
- Intractable for the general case
Our Approach: SmartApps
Application Centric Computing

Application (algorithm)

Compiler (static) + run-time techniques

SmartApp

Compiler (run-time)

OS (modular)

Architecture (reconfigurable)

Development, Analysis & Optimization

Application-Centric Computing

HW

OS

Compiler

Application

Run-time System: Execution, Analysis & Optimization

Input Data

Application Control

Instance-specific optimization

Compiler + OS + Architecture + Data + Feedback
SmartApps Architecture

STAPL Application

Static STAPL Compiler
Augmented with runtime techniques

Compiled code + runtime hooks

Get Runtime Information
(Sample input, system information, etc.)

Compute Optimal Application and RTS + OS Configuration

Recompute Application and/or Reconfigure RTS + OS

Execute Application
Continuously monitor performance and adapt as necessary

Predictor & Optimizer

Configurer

Predictor & Evaluator

Adaptive Software
Adaptive RTS+ OS
Runtime tuning (w/o recompile)

Predictor & Evaluator

Predictor & Optimizer

Small adaptation (tuning)

Large adaptation (failure, phase change)

Advanced stages
development stage
Toolbox
SmartApps written in STAPL

- **STAPL** (Standard Template Adaptive Parallel Library):
  - Collection of generic parallel algorithms, distributed containers & run-time system (RTS)
  - Inter-operable with Sequential Programs
  - Extensible, Composable by end-user
  - Shared Object View: No explicit communication
  - Distributed Objects: no replication/coherence
  - High Productivity Environment
STAPL:

- Shared Object View
- Distributed Objects: no replication/coherence
- SPMD Internal Model – Serial External Model
- Portable efficiency
  - Runtime System provides a uniform interface to the underlying architecture.
- Layered Architecture:
  - Apps user – No communication visible
  - STAPL programmer – Experienced Programmer
  - RTS Layer – Only for developers
STL Overview

- Data is stored in **Containers**
- **STL** provides generic **Algorithms**
- **Iterators** bind Algorithms to Containers
  1. Generalized pointers
• Data is stored in \texttt{pContainers}
  • `p` equivalents of all STL containers & more (e.g., \texttt{pGraph})

• STAPL provides generic \texttt{pAlgorithms}
  • `p` equivalents of all STL algorithms & more (e.g., list ranking)

• \texttt{pRanges} bind \texttt{pAlgorithms} to \texttt{pContainers}
  • Similar to STL iterators, but also support parallelism
The STAPL Programming Environment

- User Code
  - pAlgorithms
    - pRange
      - RTS + Communication Library (ARMI)
        - Interface to OS (K42)
        - OpenMP/MPI/pthreads/native
STL vs STAPL Code

STL Code

```cpp
vector<int> v;
... initialization of `v` ...
sort( v.begin(), v.end() );
```

STAPL Code

```cpp
pVector<int> pv;
... initialization of `pv` ...
pSort( pv.get_pRange() );
```
STAPL Overview

- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
pContainer Overview

**pContainer**: A 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)

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

- **Extendability**
  1. A set of base classes with basic functionality
  2. New pContainers can be derived from Base classes with extended and optimized functionality
pContainer Design Principles

- Shared Object View of Distributed Data Structure
  1. User code can assume single address space
     1. No explicit communication required
     2. Local/remote accesses identified and handled internally by STAPL
     - Sophisticated users can access & manipulate distribution

- Portable, Scalable, and Efficient
  - Goal: minimum overhead over STL containers
    - De-centralized distribution management (scalable)
    - No data replication so coherence is not an issue
  - No user code modification required to port to different systems

- Extendability
  - base classes provide basic functionality
  - new pContainers can be derived from base
  - specialized pContainers can be derived from existing pContainers
pContainer Layered Architecture

pContainer provides different views for users with different needs/levels of expertise

- **Basic User view:**
  - a single address space
  - interfaces similar to STL containers

- **Advanced User view:**
  - access to data distribution info & can use to optimize methods
  - can provide customized distributions that exploit knowledge
pContainer Major Components

- Base Sequential Container
  - STL Containers used to store data
- Distribution Manager
  - provides shared object view
- BasePContainer
pContainer Major Components: Base Sequential Container

- Developer must implement operations in Base Sequential Container Interface

```cpp
class BaseSequentialContainer {
    virtual void AddElement(const Data&, GID) = 0;
    virtual const Data& GetElement(GID) const = 0;
    virtual void SetElement(GID, const Data&) = 0;
    virtual void DeleteElement(GID) = 0;
    virtual bool ContainElement(GID) const = 0;
}
```
pContainer major Components: Distribution Manager

- Distribution Manager
  - provides shared object view
  - Functionality in Base:
    - local/remote tests
    - Methods to monitor Distribution (balance, locality, ...)
  - Methods to re-Distribute
- All aspects Customizable
pContainer major Components:

Base pContainer

template< class Container_Part, class Distribution_Manager >
class BasePContainer { 

//major attributes

  Distribution_Manager distribution;
  vector<STL_Containers> stl_container_collection;

//major methods providing Shared Object View

  virtual void AddElement(Data);
  virtual Data GetElement(GID);
  virtual void SetElement(GID, Data);
  virtual void DeleteElement(GID);
  virtual bool IsLocal(GID);
  virtual Location LookUp(GID);   
}
Example: pHashMap

- Derive sHashMap from BaseContainer and include STL HashMap

- Derive pHashMap from BasePContainer
  - Instantiate with sHashMap and Base Distribution Manager
pContainer Outline

- STAPL pContainers
  - STL Containers & STAPL pContainers
  - Design Principles & Basic Design
  - Distribution Management
Distributed Distribution Manager

- Distribution Manager responsible for providing shared object view
  - Distinguishing local/remote elements
  - Finding location of remote elements (so know who to send RMI request to)

- Default pContainer Distribution Manager:
  - Each element has unique global identifier (GID)
  - Location Map stores home thread for each GID
    - Known, fixed distribution, e.g., thread owning locmap(GID) is GID % numthreads
  - Two Steps to access remote element
    - First ask LocationMap for element’s home thread
    - Then ask element’s home thread for element
Ex Default Base Distribution Manager: Thread 0 accesses element with GID 5

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pContainer</strong></td>
<td><strong>pContainer</strong></td>
<td><strong>pContainer</strong></td>
</tr>
<tr>
<td><strong>Data:</strong></td>
<td><strong>Data:</strong></td>
<td><strong>Data:</strong></td>
</tr>
<tr>
<td>GID</td>
<td>GID</td>
<td>GID</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Value</td>
<td>Value</td>
<td>Value</td>
</tr>
<tr>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
</tbody>
</table>

**Distribution**

**Location_Map: (if GIDmod3 =0)**

<table>
<thead>
<tr>
<th>GID</th>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Location_Cache:**

<table>
<thead>
<tr>
<th>GID</th>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

**Thread 1**

**Location_Map: (if GIDmod3=1)**

<table>
<thead>
<tr>
<th>GID</th>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Location_Cache:**

<table>
<thead>
<tr>
<th>GID</th>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Thread 2**

**Location_Map: (if GIDmod3=2)**

<table>
<thead>
<tr>
<th>GID</th>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Location_Cache:**

<table>
<thead>
<tr>
<th>GID</th>
<th>Process ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Step 1:** Is_Local(GID)?

**Step 2:** Is_In_Location_Cache(GID)?

{ \( TID = 5 \mod 3 = 2 \)}

**Step 3:** thread w/ GID in LocMap?

**Step 4:** ask LocMap thread for GID loc

**Step 5:** cache GID loc in LocCache

**Step 6:** Access element with GID 5
Ex Optimized Distribution Manager: pArray

- **STL vector**
  - allows insertion & deletion any where
  - Allow random access using index, e.g., A[5]
  - Hard to optimize b/c size can change, potential imbalance

- **STAPL pArray**
  - Fixed size – does not allow insertion/deletion of elements (only can change values)
  - Sequential container uses C++ valarray (not in STL)
  - Potential to optimize distribution manager

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13</td>
<td>98</td>
<td>56</td>
<td>45</td>
<td>0</td>
<td>45</td>
<td>77</td>
<td>38</td>
<td>23</td>
<td>52</td>
</tr>
</tbody>
</table>
pArray: Basic Design

- pArray Components:
  - sequential array container uses C++ valarray (not in STL)
  - Array Distribution
    - specialized
  - pArray
**pArray: Optimized Distribution Manager**

- Fixed size array enables optimized distribution
  - Array Distribution information is stored in a vector of pair `< <Start_Index, Size>, Processor ID>`
  - Each processor has a copy of the Distribution vector
  - Accessing remote element requires only one remote access (default method requires 2 remote accesses)

**Lookup Process:**
Check if GID is in the ranges in local Distribution Vector

**Processor 0**

| **Data** | | | | |
|---|---|---|---|
| GID | 0 | 1 | 6 | 7 |
| Data | ~ | ~ | — | — |

**Distribution Vector**

`(Start_Index, Size):PID

| (0, 2):0 | (2, 4):1 | (6, 2):0 |

**Processor 1**

| **Data** | | | | |
|---|---|---|---|
| GID | 2 | 3 | 4 | 5 |
| Data | ~ | ~ | ~ | ~ |

**Distribution Vector**

`(Start_Index, Size):PID

| (0, 2):0 | (2, 4):1 | (6, 2):0 |
STAPL Overview

- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
pRange

pRange is the parallel counterpart of STL iterator:
- Binds pAlgorithms to pContainers
- Provides an abstract view of a data/work space
  - Data/work space is (recursively) partitioned into subranges
  - Controls and connects Data & Control (task) parallelism

pRange also interacts with RTS
- Scheduler/distributor decides how computation and data structures should be mapped to the machine
- Executor launches parallel computation, manages communication, and enforces dependences
pRange

- pRange provides
  - shared object view of distributed data structure + work
  - random access to a partition of the data space
    - View/access provided by iterators describing pRange boundary

pRanges are defined recursively

- Supports nested parallelism

pRanges are partitioned into subranges

- Automatically by STAPL based on machine characteristics, number of processors, partition factors, etc.
- Manually according to user-specified partitions

pRange can represent relationships among subspaces & work

- Data Dependence Graphs (DDGs) provide partial order for processing subranges
pRange Example

- Subrange boundary is a set of cut edges
- A pRange on each thread contains the subranges defined and allows communication between subranges
- DDGs can be defined on subranges and on vertices inside each subrange
STAPL Overview

- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
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
  - STAPL contains additional parallel algorithms
    - List ranking
    - Parallel Strongly Connected Components
    - Parallel Euler Tour
    - etc
STAPL Overview

- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
STAPL Run-Time System

- Support for different architectures
  - HP V2200, SGI Origin 2000/3800, IBM Regatta, BlueGeneL, Linux Clusters, Mac OS X, etc.

- Support different parallel execution models
  - Shared Memory (e.g., OpenMP, Pthreads)
  - Message Passing (e.g., MPI)
  - Mixed-Mode
STAPL Run-Time System

- **Scheduler**
  - Determine an execution order (DDG)
  - Policies:
    - Automatic: Static, Block, Dynamic, Partial Self Scheduling
    - User defined

- **Executor**
  - Execute DDG
    - Processor assignment
    - Synchronization and Communication
ARMI: STAPL Communication Infrastructure

ARMI: Adaptive Remote Method Invocation
- abstraction of shared-memory and message passing communication layer
- programmer expresses fine-grain parallelism that ARMI adaptively coarsens
- support for sync, async, point-to-point and group communication

ARMI can be as easy/natural as shared memory and as efficient as message passing
ARMI Overview

- ARMI Programming Interface
  - blocking & non-blocking communication and synchronization primitives
  - built-in support for collective & group operations
  - use in STAPL pContainers and pAlgorithms
ARMI Communication Primitives

armi_async

- statement: tell a thread something
- non-blocking: doesn’t wait for request arrival or completion method invocation

```c
template<class Class, class Rtn, class Arg1...>
void armi_async(int destThread, armiHandle handle,
    Rtn (*method)(Arg1...), Arg1 a1... )
```
ARMI Communication Primitives

- **armi_sync**
  - question: ask a thread something
  - blocking version
    - function doesn’t return until answer received from rmi
  - non-blocking version
    - function returns without answer
    - program can poll with rtnHandle.ready() and then access armi’s return value with rtnHandle.value()

- **collective operations**
  - armi_broadcast, armi_reduce, etc.
  - can adaptively set groups for communication
  - arguments always passed by value
ARMI Synchronization Primitives

- **armi_fence, armi_barrier**
  - tree-based barrier
  - implements distributed termination algorithm to ensure that all outstanding ARMI requests have been sent, received, and serviced

- **armi_wait**
  - blocks until at least one (possibly more) ARMI request is received and serviced

- **armi_flush**
  - empties local send buffer, pushing outstanding ARMI requests to remote destinations
ARMI in STAPL: pContainers

- **pContainer** = distributed set of sub-containers
  - pContainer methods abstract communication
  - decision between shared-memory/message passing made in communication infrastructure

- **Communication needs:**
  - **access**: access data in another sub-container
    - handled by armi_sync (need return value)
  - **update**: update data in another sub-container
    - handled armi_async
  - **group update**: update all sub-containers
    - handled by armi_broadcast
ARMI in STAPL: pAlgorithms

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

- **Communication needs:**
  - *merge results* from parallel tasks
    - e.g., `armi_reduce`
  - *message aggregation*:
    - handled by ARMI RTS using ARMI primitives
  - *enforcing data dependences*
    - handled by ARMI RTS using ARMI primitives
Compiled code + runtime hooks

Static STAPL Compiler
Augmented with runtime techniques

Get Runtime Information
(Sample input, system information, etc.)

Compute Optimal Application and RTS + OS Configuration

Recompute Application and/or Reconfigure RTS + OS

Execute Application
Continuously monitor performance and adapt as necessary

Predictor & Evaluator

Predictor & Optimizer

Predictor & Evaluator

Predictor & Optimizer

Adaptive Software
Adaptive RTS + OS
Runtime tuning (w/o recompile)

Small adaptation (tuning)

Large adaptation (failure, phase change)

STAPL Application

DataBase

Smart Application

Get Runtime Information
(Sample input, system information, etc.)

Compute Optimal Application and RTS + OS Configuration

Recompute Application and/or Reconfigure RTS + OS

Execute Application
Continuously monitor performance and adapt as necessary

Predictor & Evaluator

Predictor & Optimizer

Predictor & Optimizer

Adaptive Software
Adaptive RTS + OS
Runtime tuning (w/o recompile)

Small adaptation (tuning)

Large adaptation (failure, phase change)
Algorithm Adaptivity in STAPL

- **Problem:** Parallel algorithms highly sensitive to:
  - Architecture – number of processors, memory interconnection, cache, available resources, etc
  - Environment – thread management, memory allocation, operating system policies, etc
  - Data Characteristics – input type, layout, etc

- **Solution:** adaptively choose the best algorithm from a library of options at run-time
Adaptive Framework

- Currently implemented for:
  - Parallel sorting
  - Matrix multiplication
  - Reduction algorithm
## Case Study - Adaptive Sorting

<table>
<thead>
<tr>
<th>Sort</th>
<th>Strength</th>
<th>Weakness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>Theoretically time optimal</td>
<td>Many passes over data</td>
</tr>
<tr>
<td>Merge</td>
<td>Low memory overhead</td>
<td>Poor scalability</td>
</tr>
<tr>
<td>Radix</td>
<td>Extremely fast</td>
<td>Integers only</td>
</tr>
<tr>
<td>Sample</td>
<td>Two passes over data</td>
<td>High memory overhead</td>
</tr>
</tbody>
</table>
Performance: Adaptive Sorting

V2200

Power Challenge

Origin 2000

Performance on 10 million integers
Performance: Adaptive Sorting

**Origin 2000**

```c
if (data_type == INTEGER)
    radix_sort();
else if (num_procs < 5)
    merge_sort();
else
    column_sort();
```
Case Study: Adaptive Reductions

**Reduction**: update operation via associative and commutative operators: \( x = x \circ expr \)

```
FOR i = 1 to M
    sum = sum + B[i]
```

```
DOALL i = 1 to M
    p = get_pid()
    s[p] = s[p] + B[i]
    sum = s[1]+s[2]+...+s[#proc]
```

**Irregular Reduction**: updates of array elements through indirection.

```
FOR i = 1 to M
```

- Bottleneck for optimization.
- Many parallellization transformations (algorithms) were proposed and none of them always delivers the best performance.
SmartApps Architecture

STAPL Application

Static STAPL Compiler
Augmented with runtime techniques

Compiled code + runtime hooks

STAPLSTAPL
Application

Advanced stages
development stage

Toolbox

Get Runtime Information
(Sample input, system information, etc.)

Compute Optimal Application and RTS + OS Configuration

Recompute Application and/or Reconfigure RTS + OS

Predictor & Evaluator

Predictor & Optimizer

Predictor & Evaluator

Execute Application
Continuously monitor performance and adapt as necessary

Predictor & Optimizer

Adaptive Software Adaptive RTS + OS
Runtime tuning (w/o recompile)

Configurer

Configurable

Smart Application

Large adaptation (failure, phase change)

Small adaptation (tuning)
SmartApps to RTS to OS

Specialized Services from Generic OS Services
- OS offers one *size fits all* services.
- IBM K42 offers customizable services
- We want customized services BUT.... we do not want to write them

Interface between SmartApps(RTS) & OS(k42)

- Vertical integration of Scheduling/Memory Management
Collaborative Effort:

• STAPL, STAPL Compiler,
• RTS - K42
• Applications
• Validation on DOE extreme HW
  BlueGene, possibly PERCS

Texas A&M (Parasol, Nuclear Eng., Geo) + IBM + LLNL + LANL +