Efficient Parallel Containers With Shared Object View in STAPL

(Standard Template Adaptive Parallel Library)

Ping An, Alin Jula, Gabriel Tanase, Paul Thomas Nancy Amato, Lawrence Rauchwerger

Parasol
Smarter computing. Texas A&M University
STAPL Overview

- STAPL is a C++ parallel library designed as a superset of the Standard Template Library (STL).

<table>
<thead>
<tr>
<th>STL</th>
<th>Container</th>
<th>pContainer</th>
<th>Runtime:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iterator</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Algorithms</td>
<td>pAlgorithms</td>
<td>Toolboxes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Performance Optimization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>System Profiling</td>
</tr>
</tbody>
</table>

```cpp
vector<int> vec(100);
vector<int>::iterator it = vec.begin();
int tmp = *it;
It++;
... 
sort(vec.begin(), vec.end());
pVector<int> pvec(100);
pRange pr(&pvec);
p_sort(pr);
```
STAPL Overview

- STAPL simplifies parallel programming by automating
  - data partitioning, distribution, and communication
  - scheduling and parallel execution

Example using MPI {
  //Prepare Data in Proc 0
  if(myid == 0)
    MPI_send(...);
  else
    MPI_recv(...);
  //Work on local data,
  //fetch remote ones using send/recv
}

Example using STAPL {
  pVector<int> pvec1(100);
  //use default distribution
  pVector<int> pvec2(100,dist_info);
  //use given distribution_info
  //Work on local and remote data
  //with NO explicit communications
}
STAPL Design Goals

- **Ease of use**
  - Present a Shared Object View.
  - STL Compatible interface
  (No explicit communication).

- **Portable efficiency**
  - Runtime System that provides a uniform interface to the underlying architecture.
  (Same code for different kinds of machines).
STAPL Programming Model

SPMD (Single Program Multiple Data) Programming Model

```
void stapl_main() {
    pList<int> mylist(4,0);
    rmi_fence();
}
```

**Process 0:**
- `pList_Construction`
- Build part of `pList`:
  - (GID 0, Data 0);
  - (GID 1, Data 0);
- Wait at global Fence;

**Process 1:**
- `pList_Construction`
- Build part of `pList`:
  - (GID 2, Data 0);
  - (GID 3, Data 0);
- Wait at global Fence;
Applications

- **Motion Planning**
  Probabilistic Roadmap Methods for motion planning with application to protein folding, intelligent CAD, animation, robotics, etc.

- **Molecular Dynamics**
  A discrete event simulation that computes interactions between particles.
Applications

- **Particle Transport Computation**
  Efficient Massively Parallel Implementation of Discrete Ordinates Particle Transport Calculation.

- **Seismic Ray Tracing**
  Simulation of propagation of seismic rays in earth’s crust.

Particle Transport Simulation

Seismic Ray Tracing
pContainer Introduction

A pContainer is a distributed data structure with parallel 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**
  - A set of base classes with basic functionality
  - New pContainers can be derived from Base classes with extended and optimized functionality
PContainer status

- Base Infrastructure
- STL
  - pVector
  - pList
  - pHashMap
- Non STL
  - pArray
  - pGraph
  - pTree will be added soon
Layered Architecture in pContainer Design

- pContainer provides different views for users with different levels of expertise.
  - User can view the pContainer as stored in a single address space with interfaces similar to sequential containers.
  - Or advanced users can be exposed to a partitioned view of the data.
    - The partitioned view can be used to further optimize the program in terms of local/remote accesses.
pContainer: Basic Design

Three major components:

- Base pContainer
  - Base methods for access and manipulation
- pRange
- Base Distribution Manager
- Base Sequential Container Interface
Base pContainer Class

template< class Container_Part, class Distribution_Manager >
class BasePContainer  {
   //major attributes
   Distribution_Manager distribution;
   vector<Container_Part> pc_part_collection;

   //major methods for Shared Object View
   virtual void AddElement(Data);
   virtual Data GetElement(GID);
   virtual void SetElement(GID,Data);
   virtual void DeleteElement(GID);

   //Partitioned Shared Object View for optimizations
   virtual bool IsLocal(GID);
   virtual Location LookUp(GID);  
}
**Example Derived pContainer**

**pHashMap** is built by:

- Deriving
  HashMapPart from BaseContainerPart

- Instantiate
  BasePContainer with Base Distribution Manager and HashMapPart

- Derive pHashMap from BasePContainer
Base Distribution Manager

- Base Distribution Manager is responsible for finding the memory location of a remote element.
  - Each pContainer element has a unique global identifier (GID)

- Local element: Thread 0 needs an element with GID 2

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>GID</td>
<td>0</td>
</tr>
<tr>
<td>Data</td>
<td>7</td>
</tr>
</tbody>
</table>

- Remote element: Thread 0 needs an element with GID 4

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>GID</td>
<td>0</td>
</tr>
<tr>
<td>Data</td>
<td>7</td>
</tr>
</tbody>
</table>
Base Distribution Manager

Question: How to keep track of the location information of distributed data?

Simple approach: Use a complete table to translate names to the locations.

Problems:
1. Store the table in one processor will create accessing bottleneck and memory problems.
2. Replicating in all processors will lead to storage problems and extra maintenance overhead.
Base Distribution Manager

Our approach: Distributed Distribution Management

- Unique global Identifier (GID) is associated with each element
- The location map stores the home thread for each GID
  - it is distributed across all threads
  - each thread stores the home thread info for a fixed subset of GIDs
    - e.g., thread owning locmap(GID) is GID % numthreads
- Accessing a remote element is a two step process
  - First ask location map for the element’s home thread
  - Then ask element’s home thread for the element
Distribution Management
Filling Information:

Location Information is stored into the Element_Location_Map and it is updated when an element is added/removed/migrated.

Step 1: AddElement(Data, GID);

Step 2: Calculate thread owner ID; ② \{ GID 2 \% 3 procs = TID 2 \}

Step 3: Fill in Location Information of GID in TID;
Distribution Management Query:

Step 1: Is_Local(GID) ?
Step 2: Is_In_Location_Cache(GID)?
    \{ TID = \text{TID} \mod 3 = 2 \}
Step 3: Calculate thread owner ID;
Step 4: Query Location_Map(GID);
Step 5: Cache in Location_Cache(GID).
Step 6: Access element with GID 5
Decentralized Distribution Information

- Evenly divide the array into segments
- Each processor is responsible for knowing the location of one segment

The algorithm:

Example: Processor 0 needs element with GID = 5

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td><strong>Data</strong></td>
</tr>
<tr>
<td>GID</td>
<td>0</td>
</tr>
<tr>
<td>Data</td>
<td>~</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location Cache</td>
</tr>
<tr>
<td>GID</td>
</tr>
<tr>
<td>Proc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>GID</td>
</tr>
<tr>
<td>Proc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location Cache</td>
</tr>
<tr>
<td>GID</td>
</tr>
<tr>
<td>Proc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>GID</td>
</tr>
<tr>
<td>Proc</td>
</tr>
</tbody>
</table>
class **BaseDistribution**  

    //major attributes
    hash_map<GID, Location, hash<GID>> element_location_map;
    hash_map<GID, Location, hash<GID>> element_location_cache;

    //method for Element Locating, called by pContainer Lookup
    Location Lookup(GID);

    //major methods for Location bookkeeping
    void Add2LocationMap(GID, Location);
    void UpdateLocationMap(GID, Location);
    void DeleteFromLocationMap(GID);
Base Sequential Container

- pContainer is composed of sequential containers
- Base Sequential Container Interface/Part provides an uniform interface to easily build pContainers from different sequential containers
- The developer has to write this class in order to get a minimal pContainer
class BaseSequentialContainerPart {
    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;
}
Integrating Sequential Container

By multiple inheritance

By using as object

(a)

(b)
pArray: Introduction

- array:
  - data structure with fixed (unchangeable) size

<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>

- Elements can be accessed randomly using their index
  - array[5] = 45;
- Arrays are useful for numerically intensive applications
- C++ vector allows insertion and deletion of elements in the middle and thus is hard to optimize
pArray: Basic Design

- pArray is constructed by deriving from the base classes of the pContainer

- Three Major Components:
  - Array Part
  - Array Distribution
  - pArray
Optimized pArray Distribution manager

- The size of an array is fixed and the distribution doesn’t change, thus enabling us to implement an optimized distribution manager.
  - Array Distribution information is stored in a vector of pair `<<Start_Index, Size>, Processor ID>`
  - Each processor has a copy of the Distribution vector

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

### Processor 0

<table>
<thead>
<tr>
<th></th>
<th>GID</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td>~</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
<tr>
<td><strong>Distribution Vector</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>(Start_Index, Size):PID</code></td>
<td>(0, 2):0</td>
<td>(2, 4):1</td>
<td>(6, 2):0</td>
<td></td>
</tr>
</tbody>
</table>
## Array Distribution

### Optimized vs. Default

<table>
<thead>
<tr>
<th></th>
<th>Optimized Distribution Management</th>
<th>Default Distribution Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROs</strong></td>
<td>Each processor has information about the location of each element, no need to request information remotely</td>
<td>Location information is distributed and not duplicated, save space</td>
</tr>
<tr>
<td><strong>CONs</strong></td>
<td>Distribution information is duplicated, potentially space consuming</td>
<td>May need to look up the location information of an element remotely, slower</td>
</tr>
</tbody>
</table>
Array Part

- As a wrapper over the sequential STL container valarray
- Has all of functionality of the valarray
Performance Results

- **Scalability** is the ability of a program to exhibit good speed-up as the number of processors used is increased.
- Scalability = Time running on 1 Processor/Parallel Running Time

Running Prefix Sums for a pArray of 1,000,000 elements on 1 to 6 processors
Performance Results

pVector is a similar to pArray data structure with a dynamic size (new elements can be added and deleted at runtime)

- Running Prefix Sums on 1,000,000 elements using pArray and pVector
- pArray is faster due to less overhead

![Graph showing comparison between pArray and pVector](image-url)
Parallel Graph Container

- Graph data structure is widely used by applications in different domains:
  - ASCI
  - Motion Planning
  - STAPL components(pRange)

- STAPL provides a sequential graph data structure

- Through integration with STAPL pContainer design the pGraph provides:
  - support for efficient data distribution
  - load balancing
  - efficient data access
pGraph Representation

a) Original graph to be distributed on two processors

b) Distributed graph with ghost nodes

c) Distributed graph with Distribution Manager (no ghost nodes)
STAPL pGraph

Adjacency list representation

<table>
<thead>
<tr>
<th>Vertices</th>
<th>List of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 6</td>
</tr>
<tr>
<td>2</td>
<td>3, 7</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>Null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertices</th>
<th>List of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- Remote edges (highlighted) are stored as regular edge
- No need for ghost vertices/edges
- Distribution Manager associated with pGraph will provide support to decide if a vertex/edge is remote
Parallel DFS Algorithm

```cpp
class _dfs{
    pGraph g;
    Visitor visit;
    void run(Vertex v){
        //color the vertex
        visit(v);
        for all successors s of v
            stapl::async_rmi(g.location(s), handle,
                             &_dfs::run, s);
        //if vertex s is local ::run is executed locally
        //else ::run is executed remote
    }
};
void dfs(pGraph pg,Visitor visitor,Vertex start_vertex){
    _dfs(pg,visitor);
    if(pg.IsLocal(start_vertex)
        dfs.run(start_vertex);
    stapl::rmi_fence();
}
pDFS

Benefits from using the primitives

- Code size reduces dramatically compared with MPI implementation
- Async RMI allows to overlap communication with computation
- No need for explicit communication aggregation
- No need for termination detection algorithm (provided by fence)
- The parallel graph traversal are not fully parallel, but a lot of graph applications can take advantage by running multiple traversals simultaneously
- Similar to DFS we have implemented parallel BFS, topological_traversal
pGraph Methods

- pGraph construction (Add, Delete vertices Edges)
- Input/Output (two file formats)
- Integrated partition tools like Metis and Chaco
- Set/Get Vertex/Weight Field (hides the local/remote placement of vertices/edges)
- Distribute/Redistribute
- Basic Traversals: DFS, BFS, Topological
- Parallel Algorithms:
  - Strongly Connected Components
  - Connected Components.
pGraph methods

```cpp
template <class T>
int pSetWeightField(VID v1, VID v2,
                     int (WEIGHT::* method)(T), T);

template <class T1, class T2>
int pSetWeightField(VID v1, VID v2,
                     int (WEIGHT::* method)(T1, T2), T1, T2);

template <class T1, class T2, class T3>
int pSetWeightField(VID v1, VID v2,
                     int (WEIGHT::* method)(T1, T2, T3), T1, T2, T3);
```
Parallel Strongly Connected Components algorithm using pGraph

- A **strongly connected component (SCC)** of a digraph is a maximal subgraph $G'$ such that for every two nodes $u, v \text{ in } G'$ there is a path from $u$ to $v$ and a path from $v$ to $u \text{ in } G'$
- Begin with a **work sub-optimal** sequential algorithm that exposes more parallelism
- SCCs identified by searches from `pivot’ vertices that mark the SCC
- Input dependent algorithm
  the number of iterations in the loop is proportional to the number of SCCs in the input pGraph.
pSCC

- Input dependent algorithm
  the number of iterations in the loop is proportional to the number of SCCs in the input pGraph.
- Number of iterations decreases when the number of processors increases.

Each thread picks a pivot (vertex) from which to search for the SCC which contains the pivot.

```pSCC (pGraph pgraph, SCC_List scc_list)
while( ! pgraph.Empty() ) {
  --- Trim vertices that are not in a SCC
  --- Pick a pivot vertex from the trimmed pgraph
  --- Mark the SCC that contains the pivot
  --- Remove the SCC from the pgraph and add it to scc_list
}```
**pSCC**

**pSCC(G)**

If G is empty then return

trim() G in forward direction

If G is not empty then

trim() G in backward direction

select pivot v from the live vertices

mark $\text{Pred}(G,v)$ and $\text{Succ}(G,v)$

$\text{SCC}(G,v) = \text{Pred}(G,v) \cap \text{Succ}(G,v)$

Do in parallel

pSCC($\text{Pred}(G,v)-\text{SCC}(G,v)$)

pSCC($\text{Succ}(G,v)-\text{SCC}(G,v)$)

pSCC($\text{Rem}(G,v)$)

Endif
pSCC Example

Original Graph

Forward and backward trim

Pivot Selection and marking

Next iteration
Analysis

Pred

scc

Succ

Rem

scc
Improvements

- Multiple Pivots

- Information reuse (Graphs corresponding to meshes that deforms in time)

- Apply sequential SCC algorithm when the component is local
pSCC Performance

![Graph showing pSCC Performance](image)
pSCC Experimental Results

**Graph with 10K Vertices, 20K Edges, and 338 SCC**

**Input Sensitivity of pSCC Algorithm**
Graphs with 10K vertices and 20K edges

- 4 procs
- 8 procs
- 12 procs

Number of Processors vs Running Time (sec.)

Number of SCC per Graph vs Running Time (sec.)
Conclusions and Future work

- pContainer is easy to program with. No explicit communication is necessary.
- Currently we have pVector, pArray, pList, pGraph, and pHashMap. pTree will be added soon.
- Smarter pContainers : Detect load imbalance automatically, redistribution if necessary.
- Adaptive pContainers: User specifies the requirements (like associativity). STAPL chooses the pContainer best suited for the application.
References


Base Distribution Manager

- Base Distribution Manager is responsible for finding the memory location of a remote element.
  - Each pContainer element has a unique global identifier (GID)

- Local element: Processor 0 needs an element with GID 2

- Remote element: Processor 0 needs an element with GID 4