STAPL: A High Productivity Programming Infrastructure for Parallel & Distributed Computing

Lawrence Rauchwerger

Parasol Lab, Dept of Computer Science

Texas A&M University

http://parasol.tamu.edu/~rwerger/

Motivation

- Parallel programming is Costly
- Parallel programs are not portable
- Scalability & Efficiency is (usually) poor
- Dynamic programs are even harder
- Small scale parallel machines: ubiquitous

Design Goals

- Development Environment for Parallel and Distributed Computing
- Inter-operable with Sequential Programs
- Extensible by end-user >> open ended
- Composable
- Portable to any platform
- High Productivity Environment

Our Approach: STAPL

- **STAPL** = parallel components library
  - Extensible, open ended
- **STAPL =** parallel superset of **STL**
  - Sequential inter-operability
- Layered architecture: User – Developer - Specialist
  - Extensible
  - Portable (only lowest layer needs to be specialized)
- High Productivity = components have (almost) sequential interfaces.

STAPL Specification

++ STL Philosophy

++ Shared Object View

>>> User Layer: No explicit communication

>>> Machine Layer: Architecture dependent code

++ Distributed Objects,

no replication, coherence

++ Portable efficiency

Runtime System provides a uniform interface to the underlying architecture.

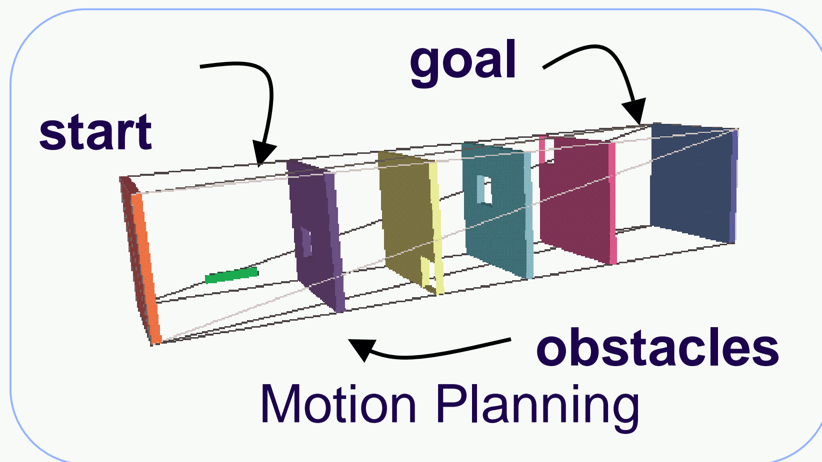
++ Concurrency & Communication Layer

SPMD (for now) parallelism, synchronization

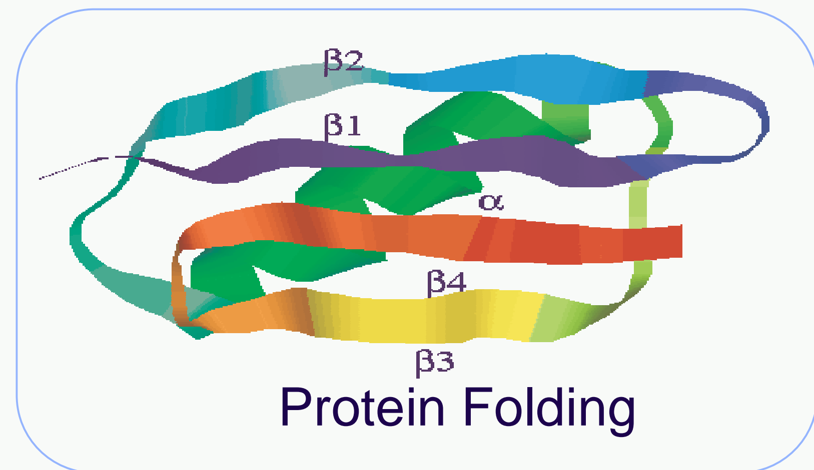
STAPL Applications

- **Motion Planning**

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



Motion Planning



Protein Folding

- **Molecular Dynamics**

A discrete event simulation that computes interactions between particles.

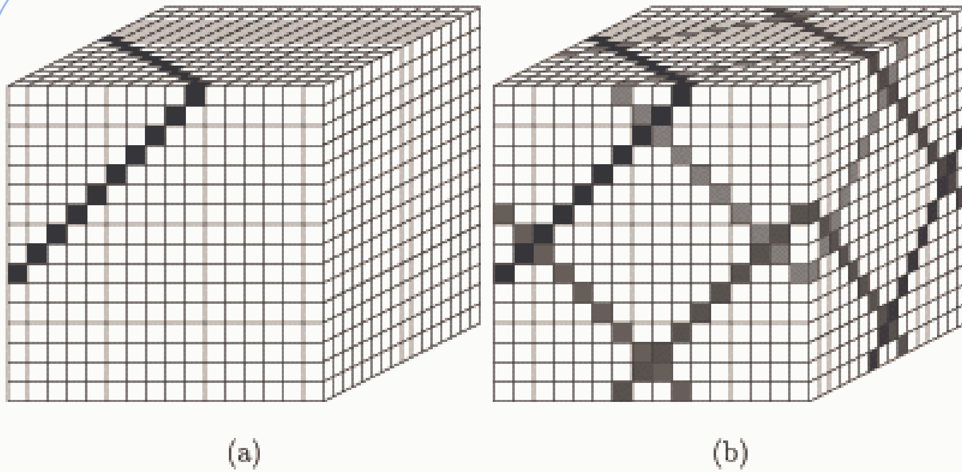
STAPL Applications

- **Particle Transport Computation**

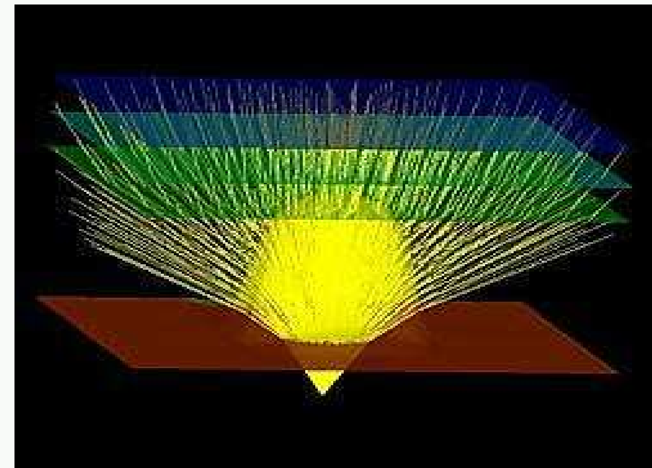
Efficient Massively Parallel Implementation of Discrete Ordinates Particle Transport Calculation.

- **Seismic Ray Tracing**

Simulation of propagation of seismic rays in earth's crust.

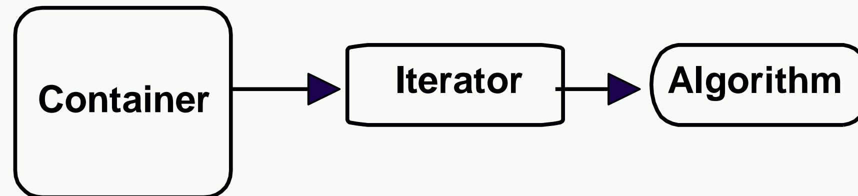


Particle Transport Simulation



Seismic Ray Tracing

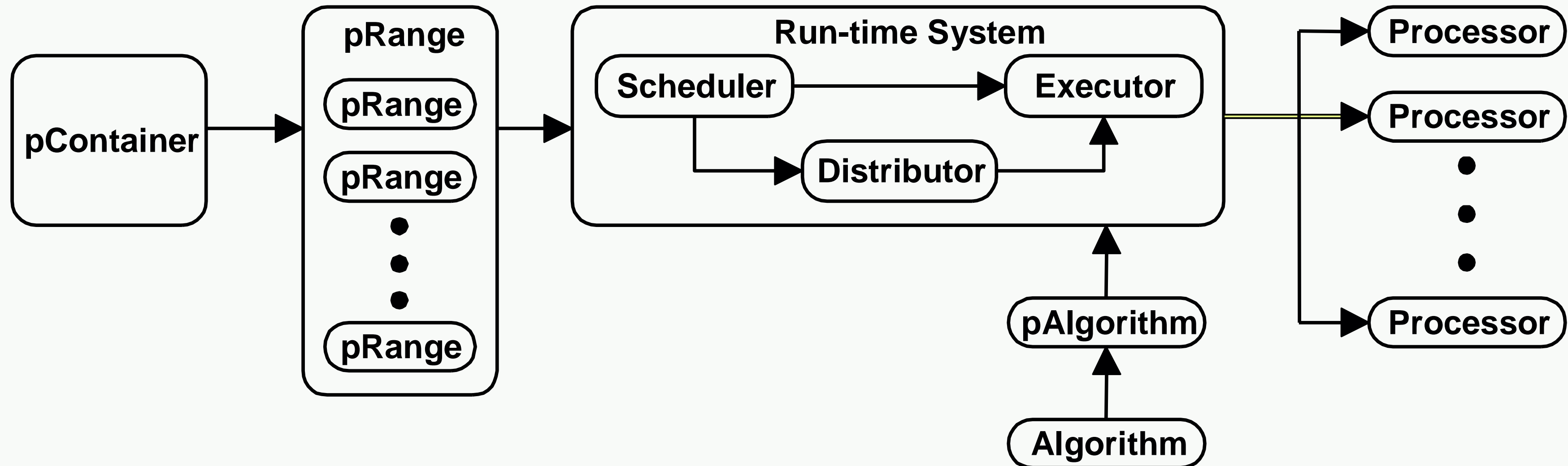
STL Overview



- Data is stored in Containers
- STL provides generic Algorithms
- Iterators bind Algorithms to Containers
  - Generalized pointers

STAPL Overview

Parasol



- Data is stored in pContainers
  - `p' equivalents of all STL containers & more (e.g., pGraph)
- STAPL provides generic pAlgorithms
  - `p' equivalents of all STL algorithms & more (e.g., list ranking)
- pRanges bind pAlgorithms to pContainers
  - Similar to STL iterators, but also support parallelism

STL vs STAPL Code

STL Code

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

STAPL Code

```
pVector<int>  pv;
… initialization of 'pv' …
psort( pv.get_pRange() );
```

Overview

- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
- Applications using STAPL

pContainer Overview

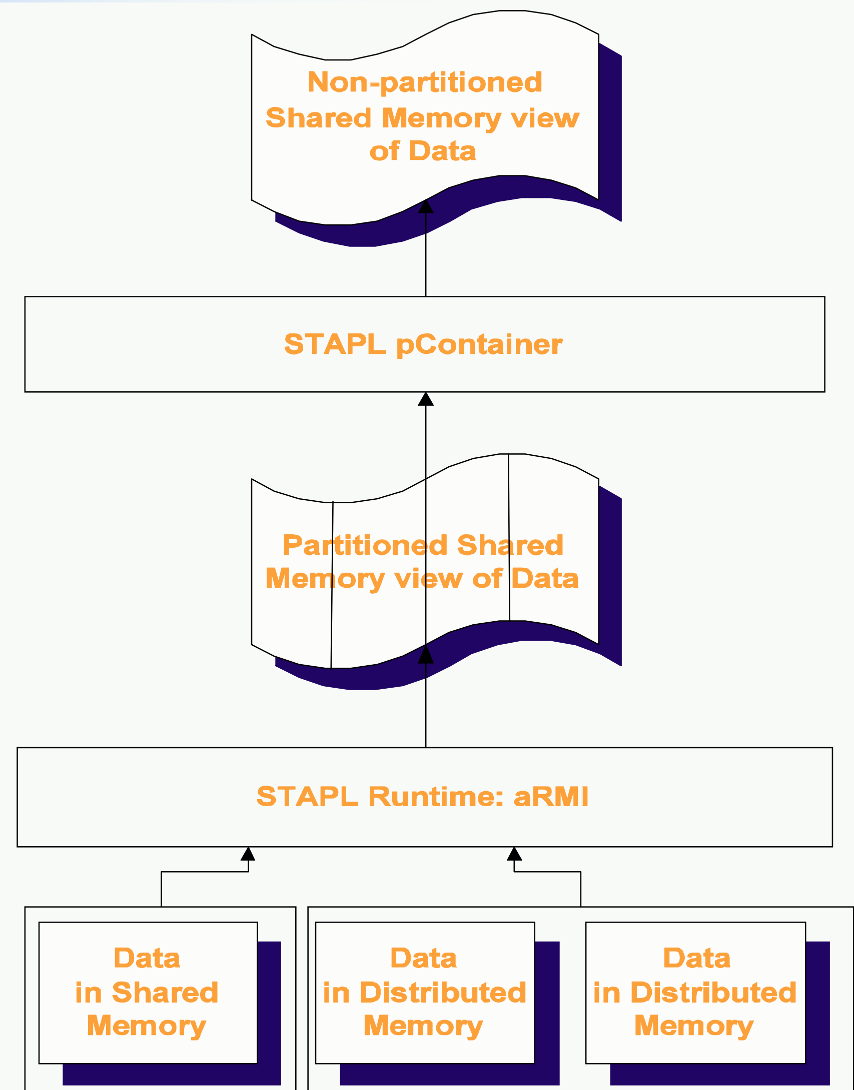
pContainer: A distributed data structure with parallel (thread-safe) methods

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

pContainer: Layered Architecture Design

pContainer provides different views for users with different levels of expertise

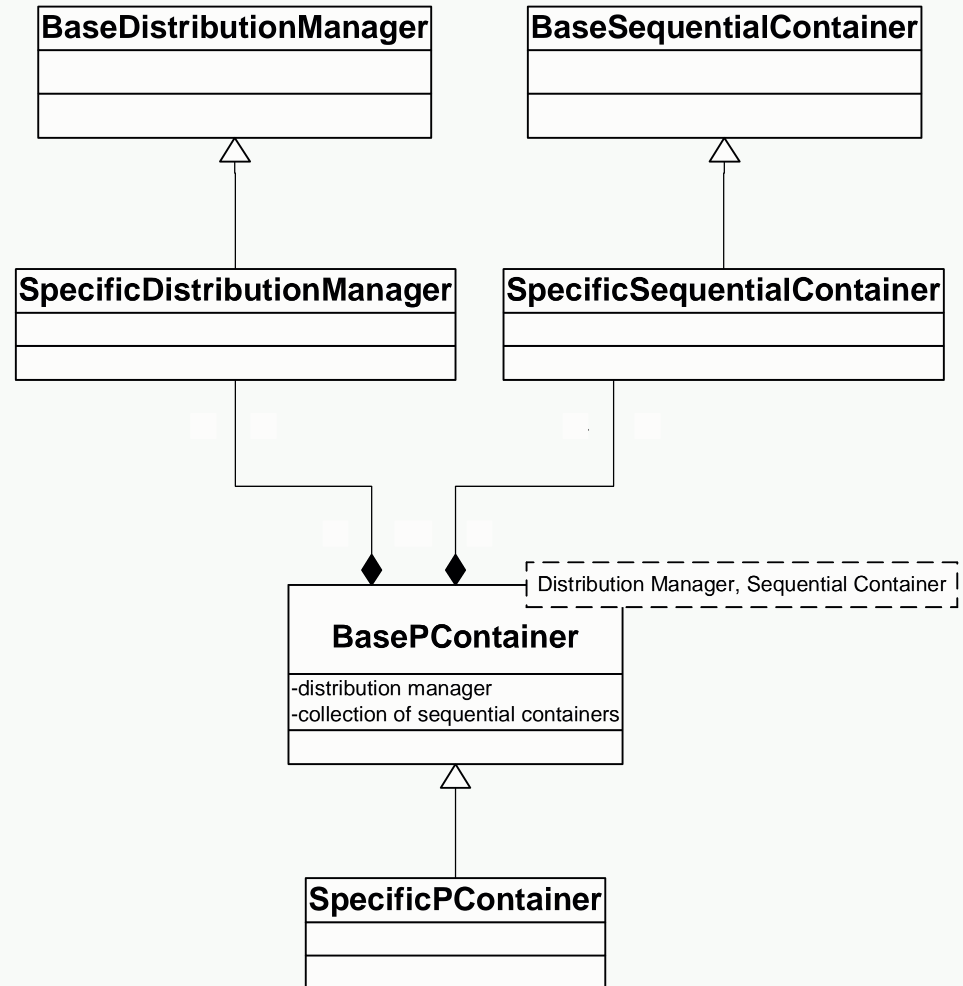
- Basic User pContainer view:
  - stored in a single address space
  - interfaces similar to STL containers
- Advanced User pContainer view:
  - partitioned view of the data
  - optimized performance by exploiting knowledge of distribution (e.g., preference to access local data)



pContainer: Basic Design

Three major components:

- Base pContainer
- Base Distribution Manager
- Base Sequential Container Interface



pContainer: Base Distribution Manager

- Unique Global Identifier (GID) associated with each element
- Each pContainer has a Distributed Location Map (Directory)
  - stores home thread for each GID
  - 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

pContainer:
Summary and Future work

- pContainer is easy to program with. No explicit communication is necessary.
- Currently we have pVector, pArray, pList, pGraph, and pHashMap, pMatrix. 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.

Overview

- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
- Applications using STAPL

pRange

- pRange is the parallel counterpart of STL iterator:
  - Binds pAlgorithms to pContainers
  - Provides an abstract view of a scoped data space
    - data space is (recursively) partitioned into subranges
- 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

Parasol

- pRange provides
  - shared object view of distributed data structure
  - 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
  - Data Dependence Graphs (DDGs) provide partial order for processing subranges

pRange Example

Thread 0

Thread 1

Graph Vertex

Graph Edge

Subrange

- 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

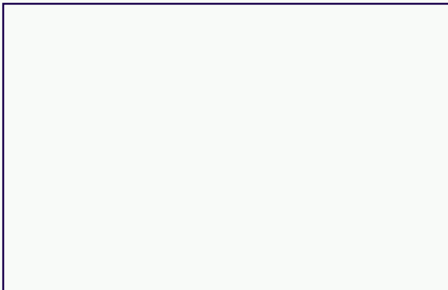
pRange–pContainer Interface

- pRange has `redistribute` method that will call `container->` `redistribute(pGraph<pair<int,Boundary>>)`
  - Reorganizes data in pContainer to match the boundaries.
- pRange constructors call `container->getDistributionVersion()`
  - Used to determine pRange validity before reuse later in code.

pRange

- Each subspace is disjoint and could be itself a pRange (Nested parallelism)

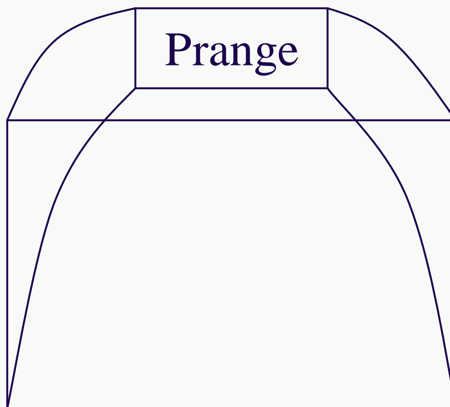
Data Space



```
stapl::pRange<stapl::pVector<int>::iterator>
   dataRange(segBegin, segEnd);
dataRange.partition();
stapl::pRange<stapl::pVector<int>::iterator>
   dataSubrange = dataRange.get_subrange(3);
dataSubrange.partition_like
(<0.25,0.25,0.25,0.25> * size);
```

pRange

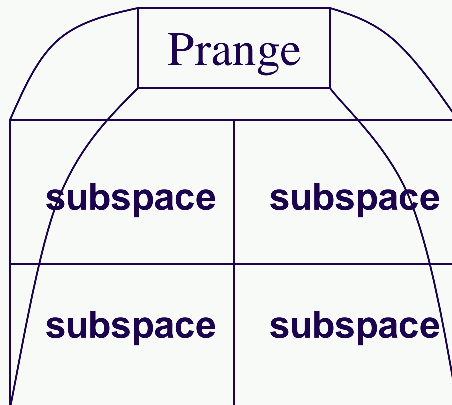
- Each subspace is disjoint and could be itself a pRange (Nested parallelism)




```
stapl::pRange<stapl::pVector<int>::iterator>
   dataRange(segBegin, segEnd);
dataRange.partition();
stapl::pRange<stapl::pVector<int>::iterator>
   dataSubrange = dataRange.get_subrange(3);
dataSubrange.partition_like
(<0.25,0.25,0.25,0.25> * size);
```

pRange

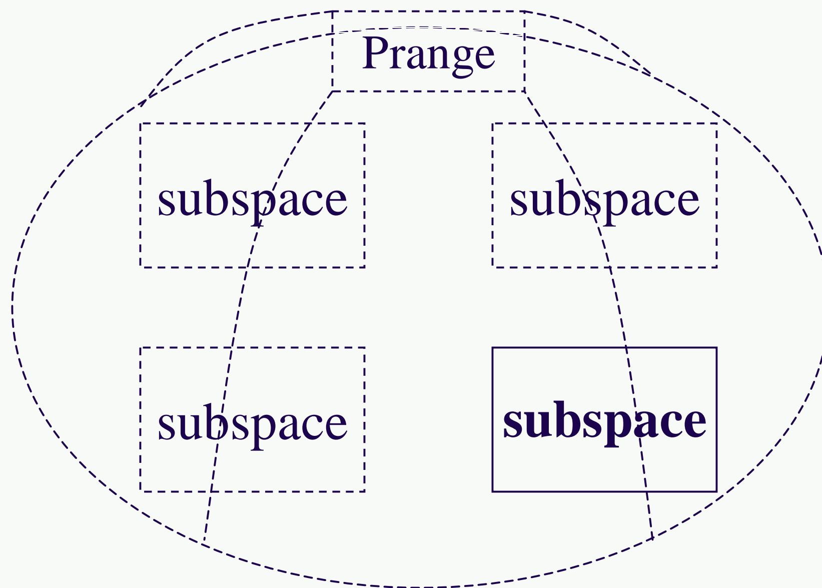
- Each subspace is disjoint and could be itself a pRange (Nested parallelism)




```
stapl::pRange<stapl::pVector<int>::iterator>
   dataRange(segBegin, segEnd);
dataRange.partition();
stapl::pRange<stapl::pVector<int>::iterator>
   dataSubrange = dataRange.get_subrange(3);
dataSubrange.partition_like
(<0.25,0.25,0.25,0.25> * size);
```

pRange

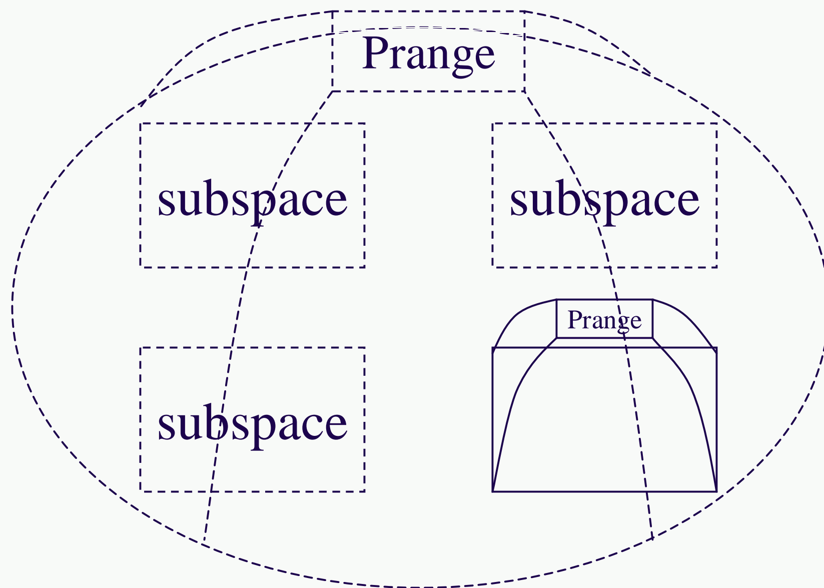
- Each subspace is disjoint and could be itself a pRange (Nested parallelism)




```
stapl::pRange<stapl::pVector<int>::iterator>
   dataRange(segBegin, segEnd);
dataRange.partition();
stapl::pRange<stapl::pVector<int>::iterator>
   dataSubrange = dataRange.get_subrange(3);
dataSubrange.partition_like
(<0.25,0.25,0.25,0.25> * size);
```

pRange

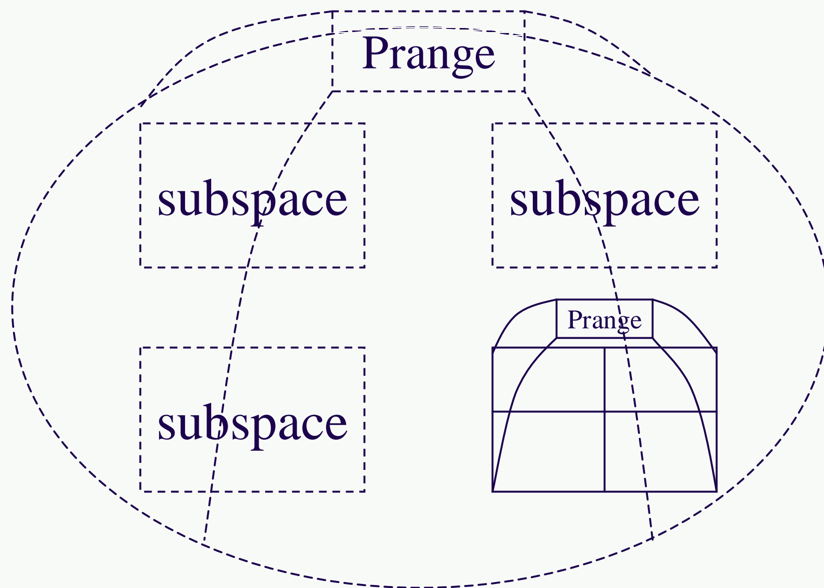
- Each subspace is disjoint and could be itself a pRange (Nested parallelism)




```
stapl::pRange<stapl::pVector<int>::iterator>
   dataRange(segBegin, segEnd);
dataRange.partition();
stapl::pRange<stapl::pVector<int>::iterator>
   dataSubrange = dataRange.get_subrange(3);
dataSubrange.partition_like
(<0.25,0.25,0.25,0.25> * size);
```

pRange

- Each subspace is disjoint and could be itself a pRange (Nested parallelism)




```
stapl::pRange<stapl::pVector<int>::iterator>
   dataRange(segBegin, segEnd);
dataRange.partition();
stapl::pRange<stapl::pVector<int>::iterator>
   dataSubrange = dataRange.get_subrange(3);
dataSubrange.partition_like
(<0.25,0.25,0.25,0.25> * size);
```

Overview

- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
- Applications using STAPL

pAlgorithms

Parasol

- 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

Overview

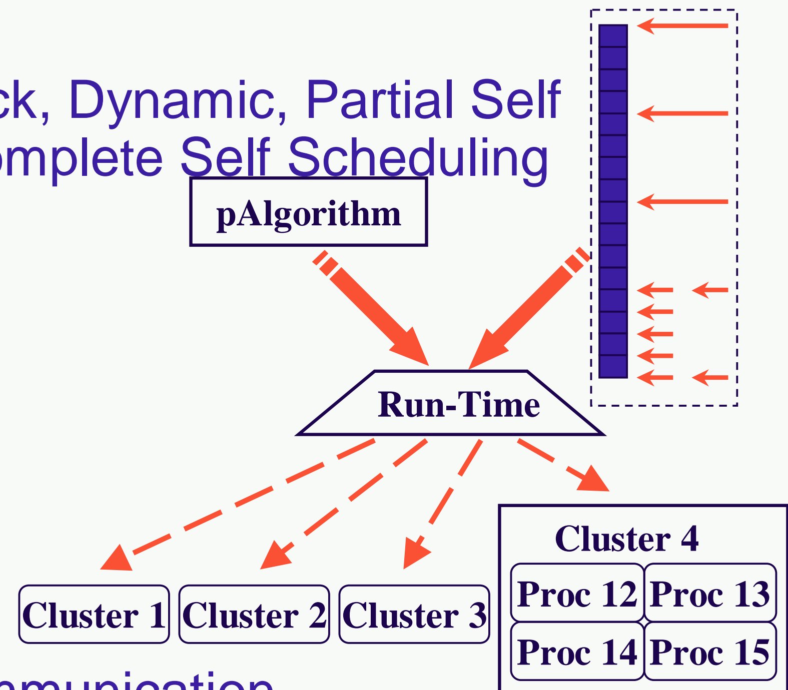
- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
- Applications using STAPL

STAPL Run-Time System

- Support for different architectures
  - HP V2200, SGI Origin 2000/3800, Linux Clusters
- Support different parallel execution models:
    - OpenMP, Pthreads, MPI

STAPL Run-Time System

- Scheduler
  - Determine an execution order (DDG)
  - Policies:
    - Automatic: Static, Block, Dynamic, Partial Self Scheduling, Complete Self Scheduling
    - User defined
- Executor
  - Execute DDG
    - Processor assignment
    - Synchronization and Communication



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 Overview

- Comparison of Communication Model
  - RMI can be as easy/natural as shared memory and as efficient as message passing
- ARMI Programming Interface
- ARMI Runtime Support
- Experiments

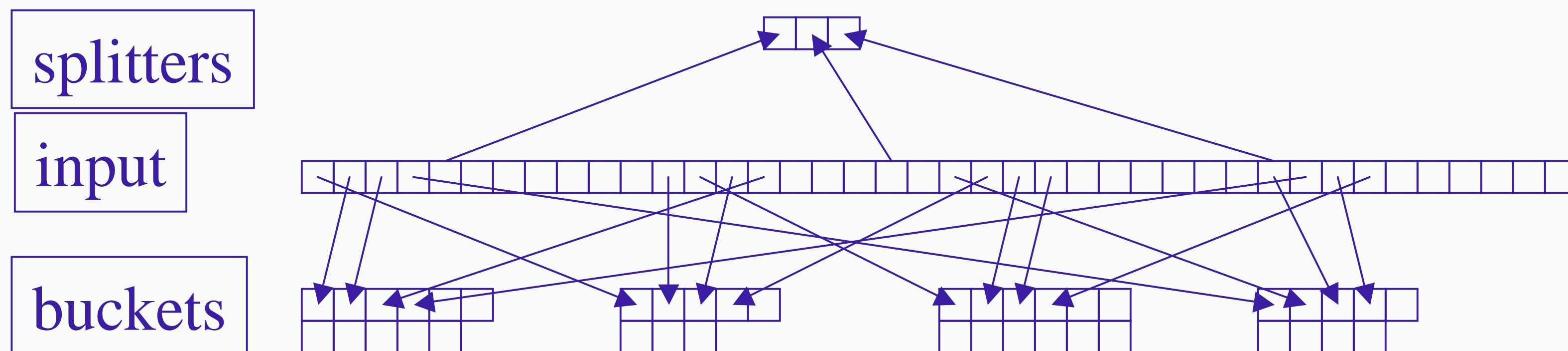
ARMI Goals & Contributions

Higher level of abstraction enables programmer to concentrate on algorithmic requirements and STAPL RTS selects implementation

- communication library natural to OO programs
  - RMI supports data hiding, encapsulation, etc
- portable efficiency
  - Programmer expresses maximum (fine-grained) parallelism and STAPL adaptively coarsens communication (and parallelism) to fit underlying architecture.
  - multi-protocol (e.g., MPI, OpenMP)
- flexible primitives enabling expressive programming
  - blocking and non-blocking communication primitives
  - Point to point, one-sided, and group communication

Case Study: Sample Sort

- Common parallel algorithm for sorting based on distributing data into buckets
- Algorithm:
  - sample a set of splitters
  - send elements to appropriate bucket based on splitters (e.g., elements less than splitter 0 are sent to bucket 0)
  - sort each bucket



Sample Sort: Shared-Memory

```
...
std::vector< std::vector<int> > buckets( p );
std::vector< lock > locks( p );
...
...fork threads...
...
for( int i=0; i<size; i++ ) {
  int dest = //...appropriate bucket...
  locks[dest].lock();
  buckets[dest].push_back( input[i] );
  locks[dest].unlock();
}
barrier();
...
```

Sample Sort: Message Passing

```
...
std::vector< int > bucket;
std::vector< std::vector<int> > outBuckets( p );
...
for( int i=0; i<size; i++ ) {
  int dest = //...appropriate bucket...
  outBuckets[dest].push_back( input[i] );
}
for( int i=0; i<p; ++i )
  if( i != my_id )
    Send( outBuckets[i], i, ... );
for( int i=0; i<p; ++i )
  if( i != my_id )
    Recv( bucket, ... );
...
```

Sample Sort: STAPL

Parasol

```
...
stapl::pvector< vector<int> > buckets( p );
...
for( int i=0; i<size; i++ ) {
  int dest = //...appropriate bucket...
  stapl::armi_async(dest,pv_handle,push_back,input[i]);
}
stapl::armi_fence();
...
```

abstracts whether implementation:

- atomically processes each element, as in shared memory
- buffers all elements and sends at once, as in message passing,
- or, some combo of above, as determined for each platform

ARMI Overview

- Comparison of Communication Models
- ARMI Programming Interface
  - blocking & non-blocking communication and synchronization primitives
  - built-in support for collective & group operations
  - use in STAPL pContainers and pAlgorithms
- ARMI Runtime Support
- Experiments

ARMI Communication Primitives

armi_async

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

```
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

Overview

- pContainers
- pRange
- pAlgorithms
- RTS & ARMI Communication Infrastructure
- Applications using STAPL

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 in G'* there is a path from *u* to *v* and a path from *v* to *u 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 Algorithm - top level

pSCC(G)

If G is empty then return

trim() G in forward direction

//topological_traversal with vertices of indegree zero as

//starting points

If G is not empty then

trim() G in backward direction *//topological_traversal*

select pivot V from the live vertices

mark *Pred(G,v)* and *Succ(G,v) //parallel_bfs traversal*

SCC(G,v) = Pred(G,v) ∧ Succ(G,v)

Do in parallel

pSCC(*Pred(G,v)-SCC(G,v)*)

pSCC*(Succ(G,v)-SCC(G,v)*)

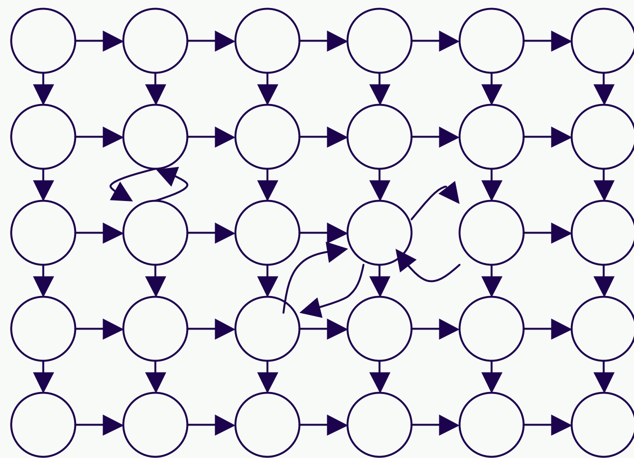
pSCC*(Rem(G,v)*)

Endif

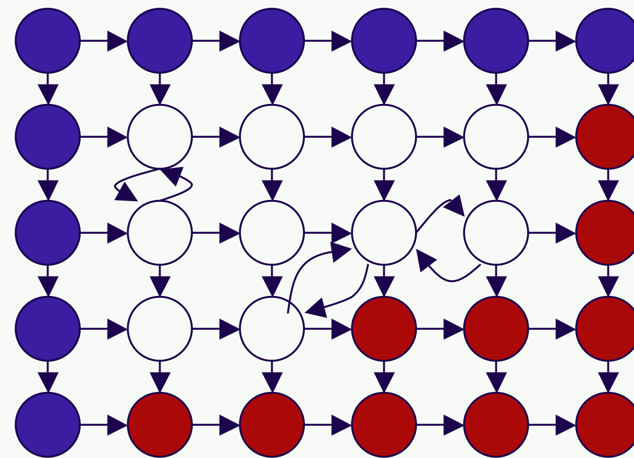
pSCC Example

Parasol

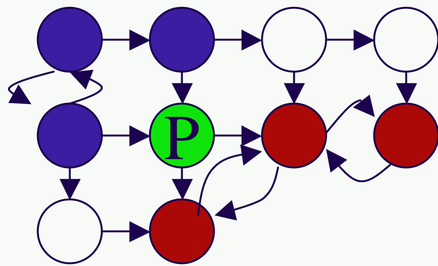
Original Graph



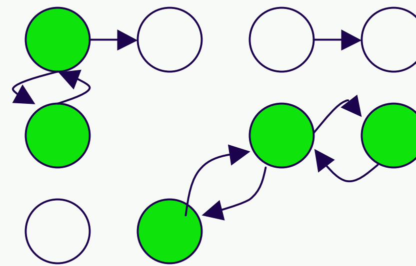
Forward and backward trim



Pivot Selection and marking



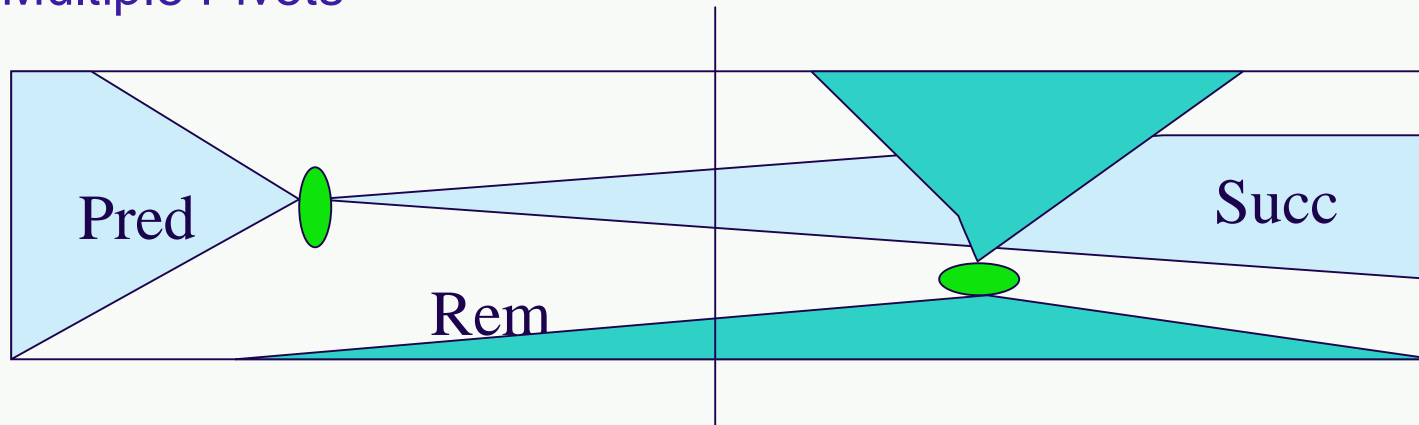
Next iteration



Improvements

Parasol

- Multiple Pivots



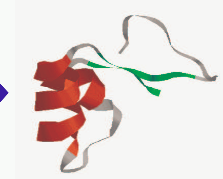
- Information reuse (Graphs corresponding to meshes that deforms in time)
- Apply sequential SCC algorithm when the component is local

Protein Folding Problem

- Shawna Thomas and Nancy Amato

  Texas A&M University
- Study the folding process, how the protein folds into its final, native 3D structure
- Different from protein structure prediction
  – Given the amino acid sequence,

  predict the 3D structure

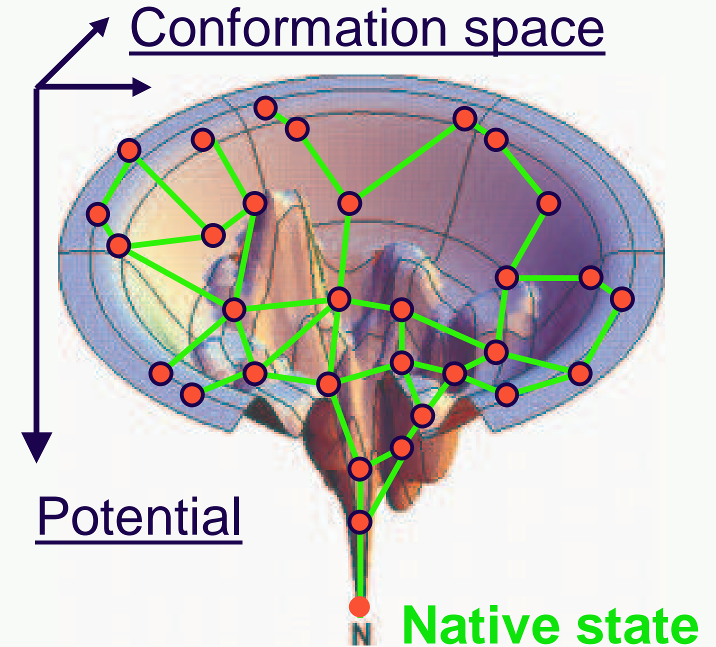
TTCCPSIVARSNFNVCRLPGTPEALCATYTGCIIIPGATCPGDYAN 

- Why important?
  - Insight into the folding process could help develop better structure prediction algorithms
  - Some diseases are caused by misfolded proteins

Folding Landscape

Parasol

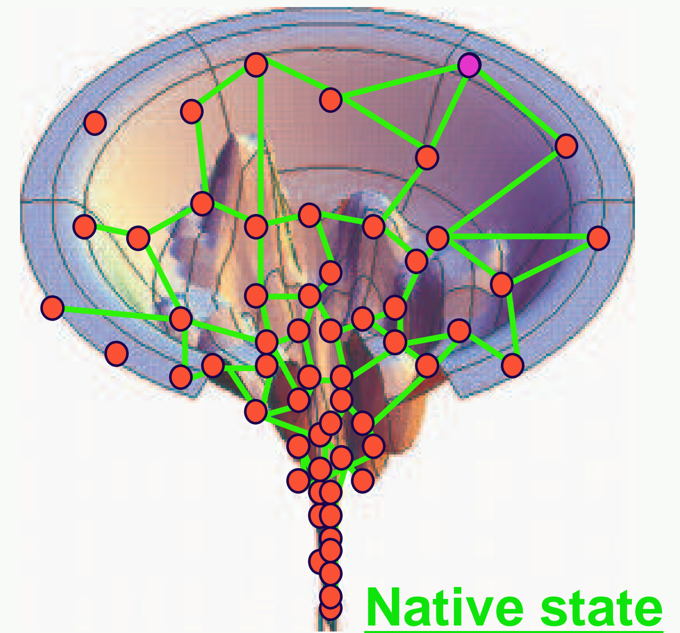
- Each conformation is associated with a potential energy
- The set of all possible conformations forms a landscape
- Different proteins ⇔ different landscapes ⇔ different folding behaviors
- Approximate the landscape by
  - Sampling conformations (graph nodes)
  - Connecting neighboring conformations to indicate feasible transitions (graph edges)
- Can extract folding pathways from this graph



Takes 95% of running time; Can compute each edge in parallel

Parallel Protein Folding Algorithm

1. Sample nodes
2. **For** each node q in graph **pardo**
3. Let Nk be k closest neighbors of q
4. **For** each q’ in Nk **do**
5. **If** can connect q and q’
6. **Then** add edge (q,q’) to graph



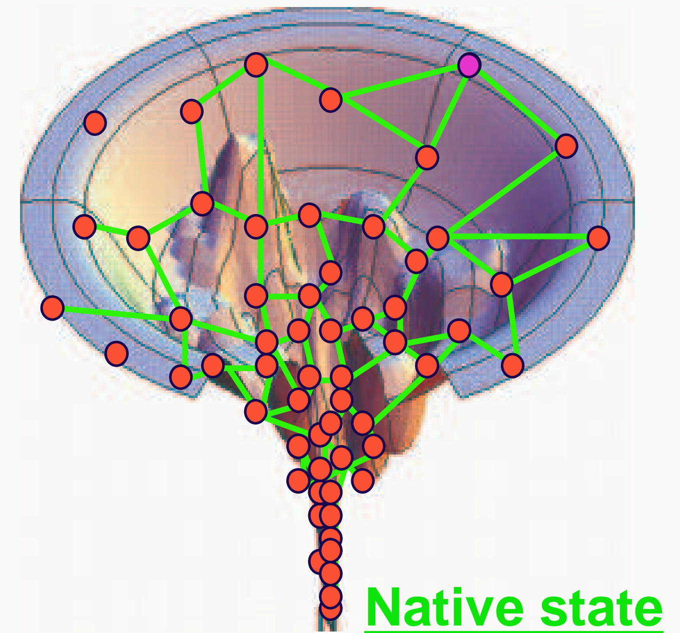
Implementation

- Sequential code implemented in C++
- Used STAPL to parallelize the existing code
  - Allows for easy transition from sequential to parallel
  - Provides portable efficiency to different systems (both shared memory and distributed memory) with no user code modification

Implementation using STAPL pVector

Parasol

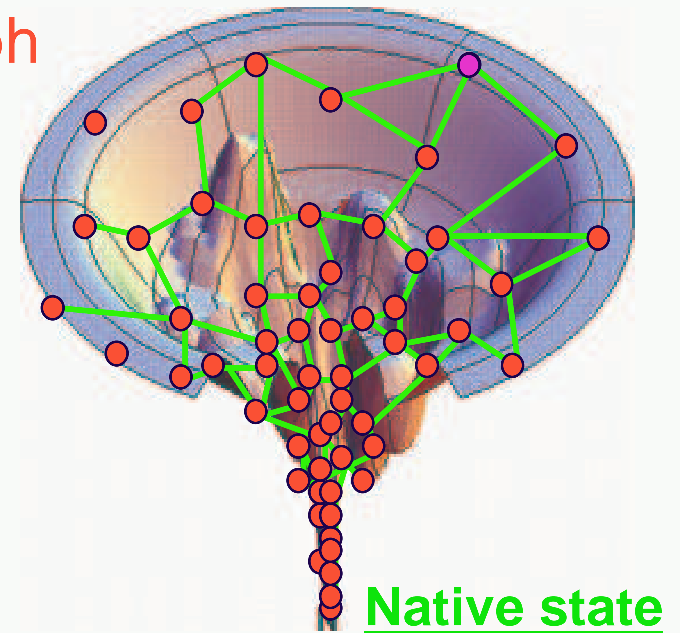
1. Sample nodes
2. Initialize pVector Edges
3. **For** each node q and its k closest neighbors q’ **pardo**
4. **If** can connect q and q’
5. **Then** add pair (q, q’) to Edges
6. Redistribute Edges to processor 0
7. Processor 0 adds edges to graph



pArray implementation is similar, but because it is fixed size, must record all edges and whether they are valid -> additional communication and space overhead over pVector implementation

Implementation using STAPL pGraph

1. Processor 0 samples nodes for pGraph
2. Redistribute pGraph evenly among all processors
3. **For** each node q in graph and its k closest neighbors q’ **pardo**
4. **If** can connect q and q’
5. **Then** add edge (q,q’) to pGraph

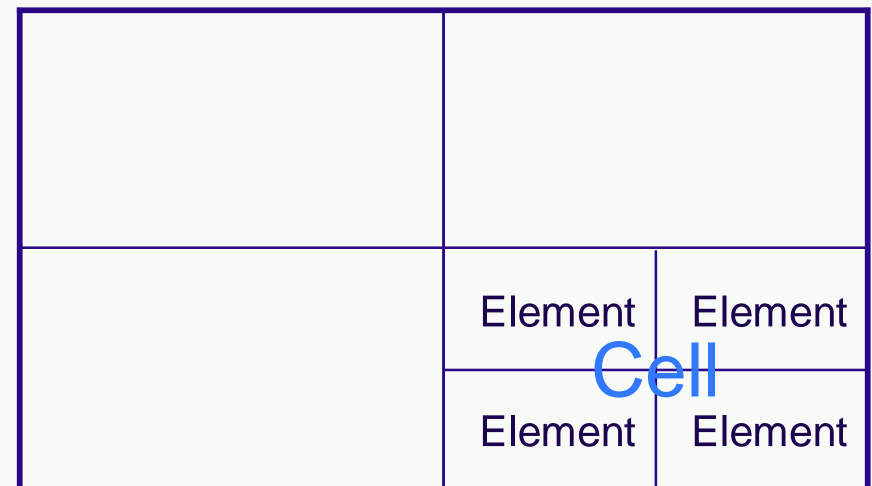


Particle Transport Problem

- Given a configuration at time T
  - Spatial domain divided into cells
  - Energy range divided into groups
  - Quadrature set chosen for angular integrals
  - Fixed Sources, Boundary Conditions, and Initial Conditions
- Compute particle flux at time $T + \Delta T$
  - Particle flux at time $T + \Delta T$
  - Reaction rates, etc., during time step

TAXI Data Structures

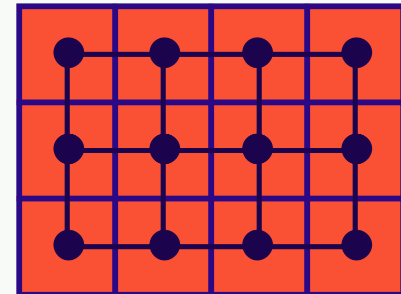
- Spatial domain decomposed into Cells
  - Cells are stored in a grid
  - Cell contains Elements
- Elements are the base spatial data structure
- Material definitions stored as in time-dependent neutronics problems
  - Number densities stored by cell
  - Microscopic cross sections stored by isotope



Grid Overview

Parasol

- Instantiation of ArbitraryPGrid
  - Derived from BaseSpatialGrid and pGraph
- Grid is templated on
  - cell type (brick, polyhedral, etc.)
  - element type (whole_cell, corner, ...)
- It contains instantiations of
  - cells (one per graph_vertex)
  - faces (really graph_edges)
- It contains methods for
  - putting outgoing-surface intensities into graph_edges
  - getting incoming-surface intensities from graph_edges



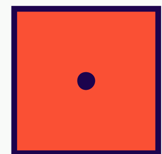
Graph Vetex

Graph Edge

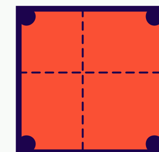
Cell

Cell Overview

- Cell class doesn't contain much.
  - list of isotopes and their number densities
  - list of elements
  - list of vertex indices
- The "element" concept gives us great generality and flexibility.
  - There is one element for each "fundamental" spatial unknown.
  - simple methods with one unknown per cell use the "whole-cell" element



  - methods with one unknown per vertex per cell use the "corner" element



Element Overview

- Elements contain:
  - volumetric sources (scattering and total)
  - the solution (angular moments for generating the scattering source; also the angular intensities in a time-dependent problem)

- Most of the code is unaware of problem geometry or discretization method.
  - It just loops over elements
  - Work function processes the Cells in a STAPL pRange

TAXI Setup

- Energy and Angle objects constructed
- Grid partition decided
  - Specialized STAPL scheduling algorithms used
- Partition information distributed
- Grid constructed in parallel
  - Cells and Elements created and inserted
- pRanges built on grid
  - Specialized STAPL scheduler determines pRanges
  - pRange dependence graphs are directed

TAXI Algorithm

ITERATION LOOP FOR GROUPSET-TO-GROUPSET SCATTERING

STORE "OLD" INFORMATION

SET SOURCE : SCATTERING FROM OTHER GROUPSETS

WITHIN GROUP-SET SOLUTION

CONVERGENCE TEST

NO

YES

END LOOP

ITERATION LOOP FOR "WITHIN-GROUP SCATTERING" ITERATION

SET TOTAL SOURCE (USES EXECUTOR AND A WORK FUNCTION; EACH CELL INDEPENDENT)

SWEEP ALL "CHUNKS" (ALL (CS,AS) PAIRS FOR THIS GS; USES EXECUTOR)

TEST CONVERGENCE

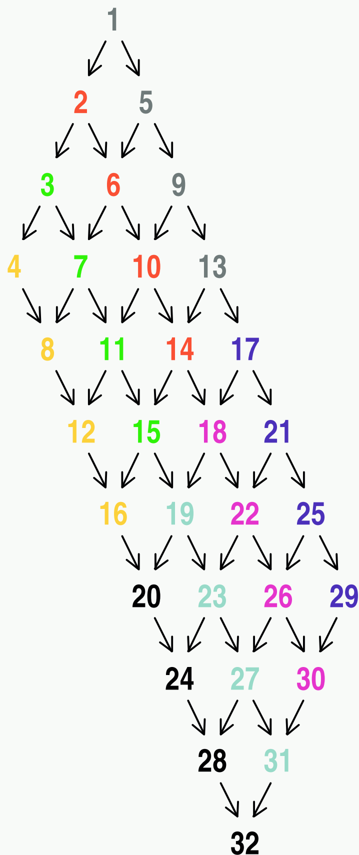
NO

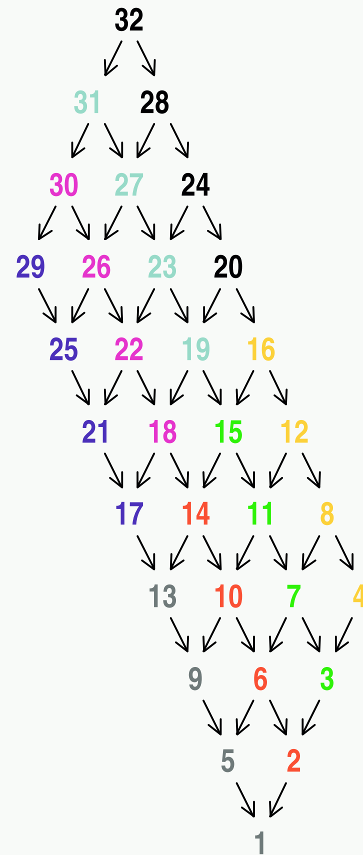
YES

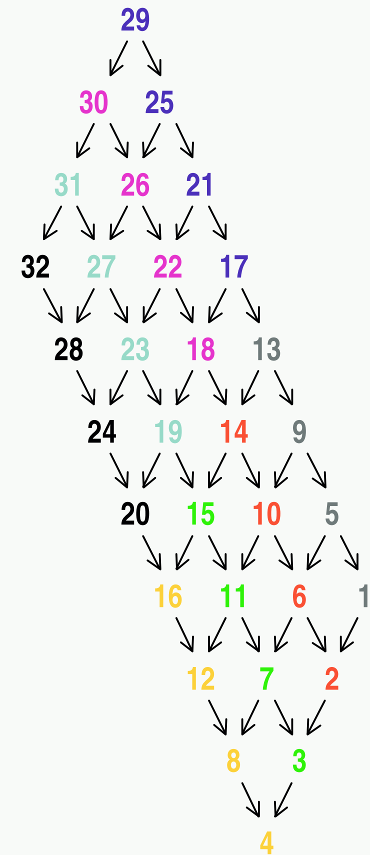
END LOOP

pRange Dependence Graph

Parasol







- Numbers are cellset indices
- Colors indicate processors

Adding a reflecting boundary

Parasol

angle-set B	angle-set A	angle-set C	angle-set D
4	1	32	29
3 8	2 5	31 28	30 25
2 7 12	3 6 9	30 27 24	31 26 21
1 6 11 16	4 7 10 13	29 26 23 20	32 27 22 17
5 10 15 20	8 11 14 17	25 22 19 16	28 23 18 13
9 14 19 24	12 15 18 21	21 18 15 12	24 19 14 9
13 18 23 28	16 19 22 25	17 14 11 8	20 15 10 5
17 22 27 32	20 23 26 29	13 10 7 4	16 11 6 1
21 26 31	24 27 30	9 6 3	12 7 2
25 30	28 31	5 2	8 3
29	32	1	4

A second reflecting boundary

angle-set B

4
3 8
2 7 12
1 6 11 16
5 10 15 20
9 14 19 24
13 18 23 28
17 22 27 32
21 26 31
25 30
29

angle-set A

1
2 5
3 6 9
4 7 10 13
8 11 14 17
12 15 18 21
16 19 22 25
20 23 26 29
24 27 30
28 31
32

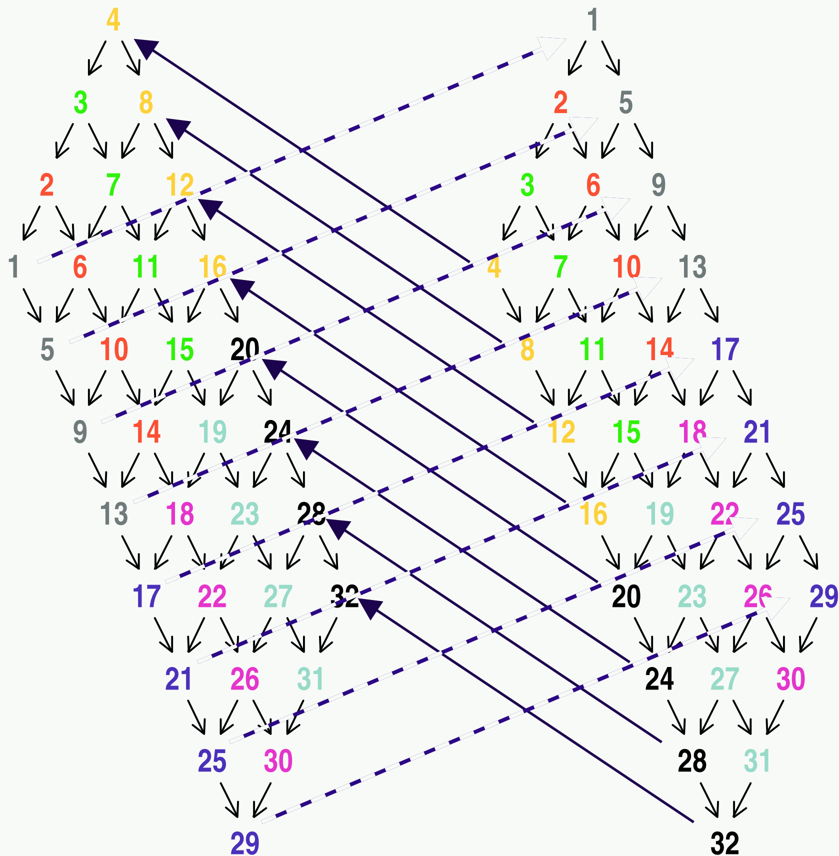
angle-set C

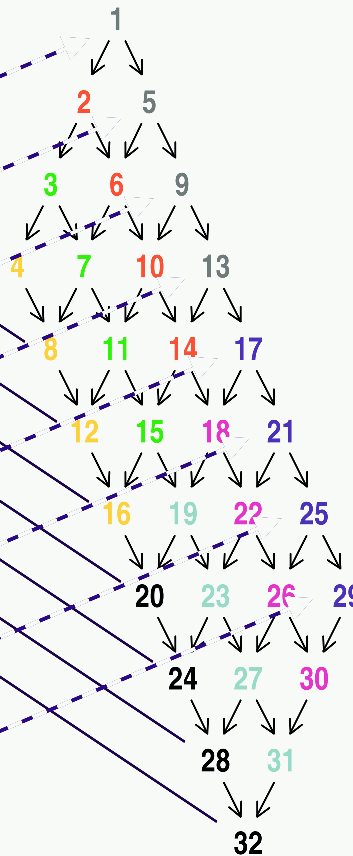
32
31 28
30 27 24
29 26 23 20
25 22 19 16
21 18 15 12
17 14 11 8
13 10 7 4
9 6 3
5 2
1

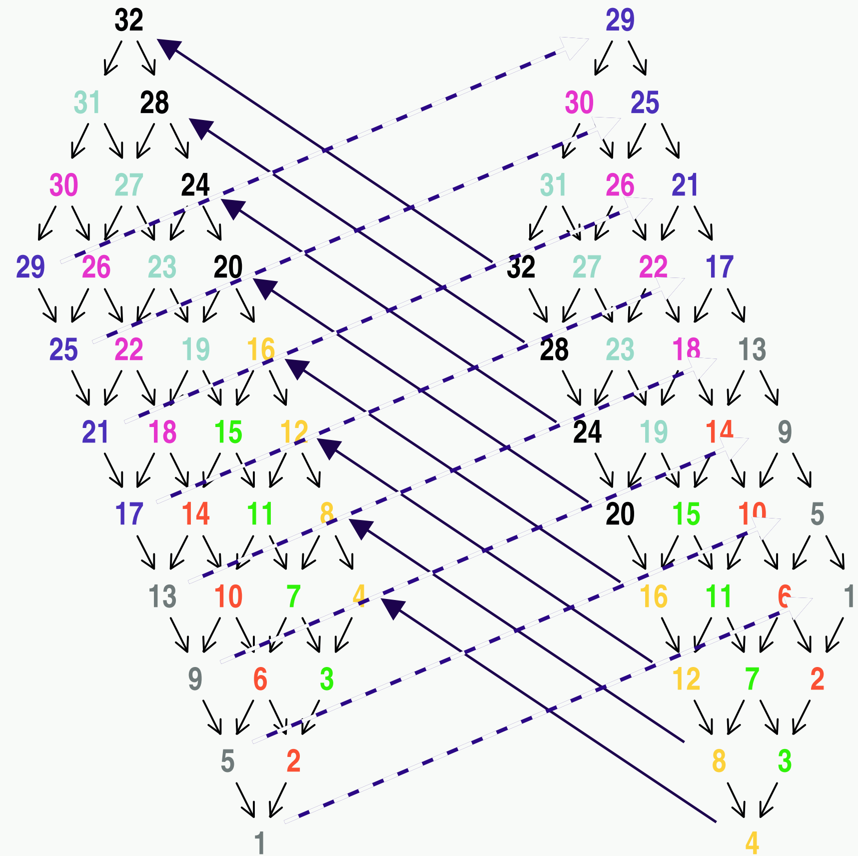
angle-set D

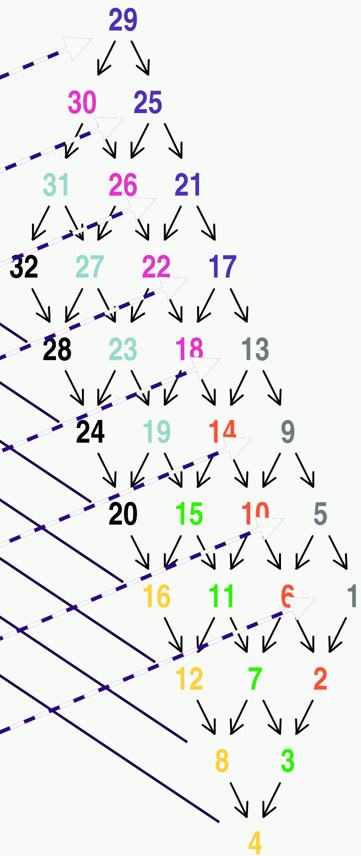
29
30 25
31 26 21
32 27 22 17
28 23 18 13
24 19 14 9
20 15 10 5
16 11 6 1
12 7 2
8 3
4

Opposing reflecting boundary









References

- **[1] "STAPL: An Adaptive, Generic Parallel C++ Library"**, Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato and Lawrence Rauchwerger, *14th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, KY, August, 2001.
- **[2] "ARMI: An Adaptive, Platform Independent Communication Library",** Steven Saunders and Lawrence Rauchwerger, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP),* San Diego, CA, June, 2003.
- **[3] "Finding Strongly Connected Components in Parallel in Particle Transport Sweeps",** W. C. McLendon III, B. A. Hendrickson, S. J. Plimpton, and L. Rauchwerger, in *Thirteenth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Crete, Greece, July, 2001.