Table of Contents

- Introduction to Parallelism
- Introduction to Programming Models
- Shared Memory Programming
- Message Passing Programming
- Shared Memory Models
  - Cilk
  - TBB
  - HPF -- influential but failed
  - Chapel
  - Fortress
  - Stapl
- PGAS Languages
- Other Programming Models

HPF – High Performance Fortran

- History
  - High Performance Fortran Forum (HPFF) coalition founded in January 1992 to define set of extensions to Fortran 77
  - V 1.1 Language specification November, 1994
  - V 2.0 Language specification January, 1997
- HPF
  - Data Parallel (SPMD) model
  - Specification is Fortran 90 superset that adds FORALL statement and data decomposition / distribution directives

The HPF Model

- Execution Model
  - Single-threaded programming model
  - Implicit communication
  - Implicit synchronization
  - Consistency model hidden from user
- Productivity
  - Extension of Fortran (via directives)
  - Block imperative, function reuse
  - Relatively high level of abstraction
  - Tunable performance via explicit data distribution
  - Vendor specific debugger

The HPF Model

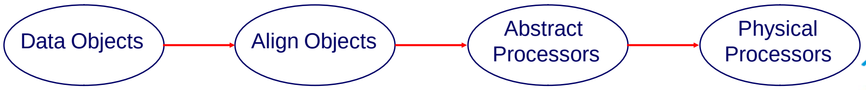
- Performance
  - Latency reduction by explicit data placement
  - No standardized load balancing, vendor could implement
- Portability
  - Language based solution, requires compiler to recognize
  - Runtime system and feature vendor specific, not modular
  - No machine characteristic interface
  - Parallel model not affected by underlying machine
  - I/O not addressed in standard, proposed extensions exist

HPF - Concepts

- DISTRIBUTE - replicate or decompose data
- ALIGN - coordinate locality on processors
- INDEPENDENT - specify parallel loops
- Private - declare scalars and arrays local to a processor

Data Mapping Model

- HPF directives - specify data object allocation
- Goal - minimize communication while maximizing parallelism
- ALIGN - data objects to keep on same processor
- DISTRIBUTE - map aligned object onto processors
- Compiler - implements directives and performs data mapping to physical processors
  - Hides communications, memory details, system specifics



Data Objects → Align Objects → Abstract Processors → Physical Processors

HPF

Ensuring Efficient Execution

- User layout of data
- Good specification to compiler (ALIGN)
- Quality compiler implementation

Simple Example (Integer Print)

```
INTEGER, PARAMETER :: N=16
   INTEGER, DIMENSION(1:N):: A,B
   !HPF$ DISTRIBUTE(BLOCK) :: A
   !HPF$ ALIGN WITH A :: B
   DO i=1,N
   A(i) = i
   END DO
   !HPF$ INDEPENDENT
   FORALL (i=1:N) B(i) = A(i)*2
   WRITE (6,*) 'A = ', A
   WRITE (6,*) 'B = ', B
   STOP
   END
```

Output:

```
0: A = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
0: B = 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
```

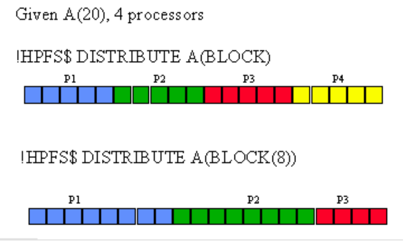
HPF Compiler Directives

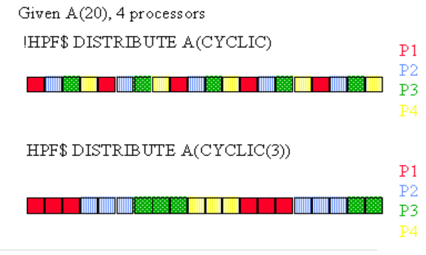
`trigger-string hpf-directive`

- `trigger-string` - comment followed by HPF$
- `hpf-directive` - an HPF directive and its arguments
  - **DISTRIBUTE, ALIGN, etc.**

HPF - Distribute

- **`!HPF$ DISTRIBUTE object (details)`**
  - distribution details - comma separated list, for each array dimension
    - BLOCK, BLOCK(N), CYCLIC, CYCLIC(N)
  - object must be a simple name (e.g., array name)
  - object can be *aligned to*, but not aligned





HPF - ALIGN

- **`!HPF$ ALIGN alignee(subscript-list) WITH object(subscript-list)`**
- **`alignee`** - undistributed, simple object
- **`subscript-list`**
  - All dimensions
  - Dummy argument (int constant, variable or expr.)
  - :
  - *

HPF - ALIGN

Equivalent directives, with !HPF$ DISTRIBUTE A(BLOCK,BLOCK)

```
!HPF$ ALIGN B(:,:) WITH A(:,:)
!HPF$ ALIGN (i,j) WITH A(i,j) :: B
!HPF$ ALIGN (:,:) WITH A(:,:) :: B
!HPF$ ALIGN WITH A :: B
```

Example

Original F77

```
 ...
 REAL centre(N,N), image(N+2,N+2)
 ...
 DO i = 1, N
  DO j = 1, N
   centre(i,j) =
&    -image(i  ,j)-image(i  ,j+1)   -image(i  ,j+2)
&    -image(i+1,j)-image(i+1,j+1)*8.0-image(i+1,j+2)
&    -image(i+2,j)-image(i+2,j+1)   -image(i+2,j+2)
  END DO
 END DO
```

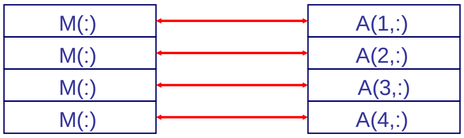
HPF

```
End result, Fortran 90 style
    REAL, DIMENSION(N,N) :: centre
    REAL, DIMENSION(N+2,N+2) :: image
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: image
!HPF$ ALIGN centre(i,j) WITH image(i+1,j+1)
    ...
     centre(:,:) =
    & -image( :N,  :N)-image( :N,  2:N+1)   -image( :N,  3:N+2)
    & -image(2:N+1,:N)-image(2:N+1,2:N+1)*8 -image(2:N+1,3:N+2)
    & -image(3:N+2,:N)-image(3:N+2,2:N+1)   -image(3:N+2,3:N+2)
```

HPF - Alignment for Replication

- Replicate heavily read arrays, such as lookup tables, to reduce communication
  - Use when memory is cheaper than communication
  - If replicated data is updated, compiler updates ALL copies
- If array M is used with every element of A:

```
INTEGER M(4)
INTEGER A(4,5)
!HPF$ ALIGN M(*) WITH A(i,*)
```



M(:)	A(1,:)
M(:)	A(2,:)
M(:)	A(3,:)
M(:)	A(4,:)

HPF Example - Matrix Multiply

```
PROGRAM ABmult
 IMPLICIT NONE
 INTEGER, PARAMETER :: N = 100
 INTEGER, DIMENSION (N,N) :: A, B, C
 INTEGER :: i, j
 !HPF$ DISTRIBUTE (BLOCK,BLOCK) :: C
 !HPF$ ALIGN A(i,*) WITH C(i,*)
 ! replicate copies of row A(i,*)
 ! onto processors which compute C(i,j)
 !HPF$ ALIGN B(*,j) WITH C(*,j)
 ! replicate copies of column B(*,j))
 ! onto processors which compute C(i,j)
 A = 1
 B = 2
 C = 0
 DO i = 1, N
 DO j = 1, N
 ! All the work is local due to ALIGNs
 C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
 END DO
 END DO
 WRITE(*,*) C
```

HPF - FORALL

- A generalization of Fortran 90 array assignment (not a loop)
- Does assignment of multiple elements in an array, but order not enforced
- Uses
  - assignments based on array index
  - irregular data motion
  - gives identical results, serial or parallel
- Restrictions
  - assignments only
  - execution order undefined
  - not iterative

```
FORALL (I=1:N) B(I) = A(I,I)
FORALL (I = 1:N, J = 1:N:2, J .LT. I) A(I,J) = A(I,J) / A(I,I)
```

Table of Contents

- Introduction to Parallelism
- Introduction to Programming Models
- Shared Memory Programming
- Message Passing Programming
- Shared Memory Models
  - Cilk
  - TBB
  - HPF
  - Chapel
  - Fortress
  - Stapl
- PGAS Languages
- Other Programming Models

Chapel

- The Cascade High-Productivity Language (Chapel)
  - Developed by Cray as part of DARPA HPCS program
  - Draws from HPF and ZPL
  - Designed for “general” parallelism
    Supports arbitrary nesting of task and data parallelism
  - Constructs for explicit data and work placement
  - OOP and generics support for code reuse

Adapted From:http://chapel.cs.washington.edu/ChapelForAHPCRC.pdf

---

The Chapel Model

- Execution Model
  - Explicit data parallelism with **`forall`**
  - Explicit task parallelism **`forall`**, **`cobegin`**, **`begin`**
  - Implicit communication
  - Synchronization
    - Implicit barrier after parallel constructs
    - Explicit constructs also included in language
  - Memory Consistency model still under development

Chapel - Data Parallelism

- `forall` loop

  loop where iterations performed concurrently

  ```
  forall i in 1..N do
    a(i) = b(i);
  ```

  alternative syntax:

  ```
  [i in 1..N] a(i) = b(i);
  ```

Chapel - Task Parallelism

- `forall` expression

  allows concurrent evaluation expressions

  ```
  [i in S] f(i);
  ```

- **`cobegin`**

  indicate statement that may run concurrently

  ```
      cobegin {
       ComputeTaskA(…);
       ComputeTaskB(…);
   }
  ```

- **`begin`**

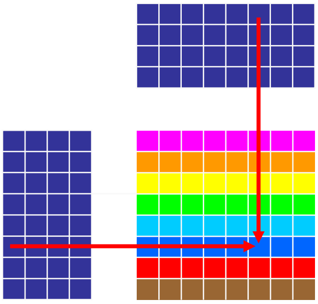
  spawn a computation to execute a statement

  ```
  begin ComputeTaskA(…); //doesn't rejoin
   ComputeTaskB(…);      //doesn't wait for ComputeTaskA
  ```

Chapel - Matrix Multiply

```
var A: [1..M, 1..L] float;
var B: [1..L, 1..N] float;
var C: [1..M, 1..N] float;

forall (i,j) in [1..M, 1..N] do
   for k in [1..L]
     C(i,j) += A(i,k) * B(k,j);
```



Chapel - Synchronization

- `single` variables
  - Chapel equivalent of **futures**
  - **Use** of variable stalls until variable **assignment**

```
var x : single int;
begin x = foo();   //sub computation spawned
var y = bar;
return x*y;        //stalled until foo() completes.
```

- `sync` variables
  - generalization of single, allowing multiple assignments
  - ***full / empty*** semantics, read ‘empties’ previous assignment
- `atomic` statement blocks
  - transactional memory semantics
  - no changes in block visible until completion

Chapel - Productivity

- New programming language
- Component reuse
  - Object oriented programming support
  - Type generic functions
- Tunability
  - Reduce latency via explicit work and data distribution
- Expressivity
  - Nested parallelism supports composition
- Defect management
  - ‘Anonymous’ threads for hiding complexity of concurrency
    no user level thread_id, virtualized

Chapel - Performance

- Latency Management
  - Reducing
    - Data placement - distributed domains
    - Work placement - `on` construct
  - Hiding
    - `single` variables
    - Runtime will employ multithreading, if available

Chapel - Latency Reduction

- Locales
  - Abstraction of processor or node
  - Basic component where memory accesses are assumed uniform
  - User interface defined in language
    - integer constant **numLocales**
    - type `locale` with (in)equality operator
    - array `Locales[1..numLocales]` of type `locale`

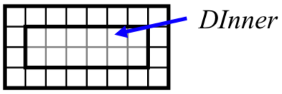
```
var CompGrid:[1..Rows, 1..Cols] local = ...;
```

CompGrid

Chapel - Latency Reduction

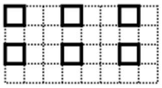
- Domain
  - set of indices specifying size and shape of aggregate types (i.e., arrays, graphs, etc)

```
var m: integer = 4;
var n: integer = 8;
var D: domain(2) = [1..m, 1..n];
var DInner: domain(D) = [2..m-1, 2..n-1]
```



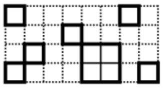
D

```
var StridedD: domain(D) = D by (2,3);
```



StridedD

```
var indexList: seq(index(D)) = ...;
var SparseD: sparse domain(D) = indexList;
```

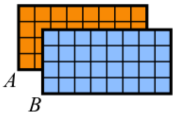


SparseD

Chapel - Domains

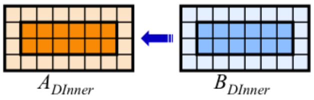
- Declaring arrays

  `var A, B: [D] float`



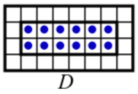
- Sub-array references

  `A(Dinner) = B(Dinner);`



- Parallel iteration

  `forall (i,j) in Dinner { A(i,j} = ...}`



Chapel - Latency Reduction

- Distributed domains
  - Domains can be *explicitly* distributed across locales

    `var D: domain(2) distributed(block(2) to CompGrid) = ...;`

  - Pre-defined
    - `block, cyclic, block-cyclic, cut`
  - User-defined distribution support in development

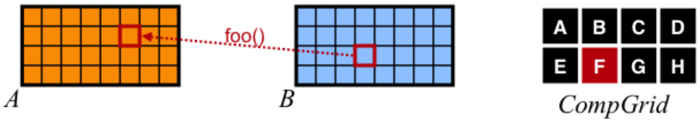
Chapel - Latency Reduction

- Work Distribution with **`on`**

```
cobegin {
  on TaskALocs do ComputeTaskA(...);
  on TaskBLocs do ComputeTaskB(...);
}
```

alternate data-driven usage:

```
forall (i,j) in D {
  on B(j/2, i*2) do A(i,j) = foo(B(j/2, i*2));
}
```



Chapel - Portability

- Language based solution, requires compiler
- Runtime system part of Chapel model. Responsible for mapping implicit multithreaded, high level code appropriately onto target architecture
- **locales** machine information available to programmer
- Parallel model not effected by underlying machine
- I/O API discussed in standard, scalability and implementation not discussed

Table of Contents

The Fortress Model

- Developed by Sun for DARPA HPCS program
- Draws from Java and functional languages
- Emphasis on growing language via strong library development support
- Places parallelism burden primarily on library developers
- Use of extended Unicode character set allow syntax to mimic mathematical formulas

```
trait EquivalenceRelation⟦T extends EquivalenceRelation⟦T, ∼⟧, opr ∼⟧
      extends { Reflexive⟦T, ∼⟧, Symmetric⟦T, ∼⟧, Transitive⟦T, ∼⟧ }
end
```

Adapted From: http://irbseminars.intel-research.net/GuySteele.pdf

The Fortress Model

Execution Model

- User sees single-threaded execution by default
  - Loops are assumed parallel, unless otherwise specified
- Data parallelism
  - Implicit with `for` construct
  - Explicit ordering via custom Generators
- Explicit task parallelism
  - Tuple and `do all` constructs
  - Explicit with `spawn`

The Fortress Model

Execution Model

- Implicit communication
- Synchronization
  - Implicit barrier after parallel constructs
  - Implicit synchronization of reduction variables in `for` loops
  - Explicit `atomic` construct (transactional memory)
- Memory Consistency
  - Sequential consistency under constraints
    - all shared variable updates in `atomic` sections
    - no implicit reference aliasing

Fortress - Data Parallelism

- `for` loops - default is parallel execution

```
for i←1:m, j←1:n do
  a[i,j] := b[i] c[j]
end
```

```
for i←seq(1:m) do
  for j←seq(1:n) do
    print a[i,j]
  end
end
```

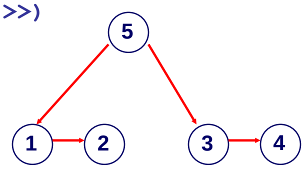
`1:N` and `seq(1:N)` are *generators*

`seq(1:N)` *is generator for sequential execution*

Fortress - Data Parallelism

- Generators
  - Controls parallelism in loops
  - Examples
    - Aggregates - `<1,2,3,4>`
    - Ranges - 1:10 and 1:99:2
    - Index sets - `a.indices` and `a.indices.rowMajor`
    - `seq(g)` - sequential version of generator `g`
  - Can compose generators to order iterations

`seq(<5,<seq(<1,2>), seq(<3,4>)>>)`



Fortress - Explicit Task Parallelism

- Tuple expressions
  - comma separated exp. list executed concurrently

```
(foo(), bar())
```

- **`do-also`** blocks
  - all clauses executed concurrently

```
do
   foo()
also do
   bar()
end
```

Fortress - Explicit Task Parallelism

- Spawn expressions (futures)

```
…
v = spawn do
   …
end
…
v.val()   //return value, block if not
completed
v.ready() //return true iff v completed
v.wait()  //block if not completed, no
return value
v.stop()  //attempt to terminate thread
```

Fortress - Synchronization

- `atomic` blocks - transactional memory
  - other threads see block completed or not yet started
  - nested **`atomic`** and parallelism constructs allowed
  - **`tryatomic`** can detect conflicts or aborts

```
sum : N := 0
accumArray⟦N extends Additive, nat x⟧(a : N[x]) : () =
  for i ← a.indices do
    atomic sum += a[i]
  end
```

```
do
   x: ℤ := 0
   y: ℤ := 0
   z: ℤ := 0
   atomic do
      x += 1
      y += 1
   also atomic do
      z := x + y
   end
   z
end
```

Fortress - Productivity

- Defect management
  - Reduction
    - explicit parallelism and tuning primarily confined to libraries
  - Detection
    - integrated testing infrastructure
- Machine model
  - *Regions* give abstract machine topology

Fortress - Productivity

Expressivity

- High abstraction level
  - Source code closely matches formulas via extended Unicode charset
  - Types with checked physical units
  - Extensive operator overloading
- Composition and Reuse
  - Type-based generics
  - Arbitrary nested parallelism
  - Inheritance by traits
- Expandability
  - ‘Growable’ language philosophy aims to minimize core language constructs and maximize library implementations

---

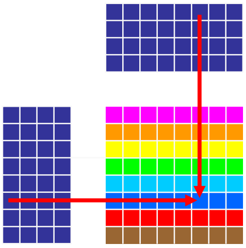
Fortress - Productivity

- Implementation refinement
  - Custom generators, distributions, and thread placement
- Defect management
  - Reduction
    - explicit parallelism and tuning primarily confined to libraries
  - Detection
    - integrated testing infrastructure
- Machine model
  - *Regions* give abstract machine topology

Fortress - Matrix Multiply

```
matmult(A: Matrix[/Float/],
        B: Matrix[/Float/])
        : Matrix[/Float/]
  A B
end

C = matmult(A,B)
```

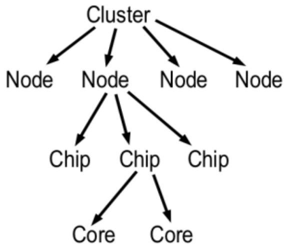


Fortress - Performance

- Regions for describing system topology
- Work placement with **`at`**
- Data placement with Distributions
- **`spawn`** expression to hide latency

Fortress - Regions

- Tree structure of CPUs and memory resources
  - Allocation heaps
  - Parallelism
  - Memory coherence
- Every thread, object, and array element has associated region



```
obj.region()   //region where object obj is located
r.isLocalTo(s) //is region r in region tree rooted at s
```

Fortress - Latency Reduction

- Explicit work placement with **`at`**

inside `do also`

```
do
    $v := a_i$
also at $a.region(j)$ do
    $w := a_j$
end
```

with `spawn`

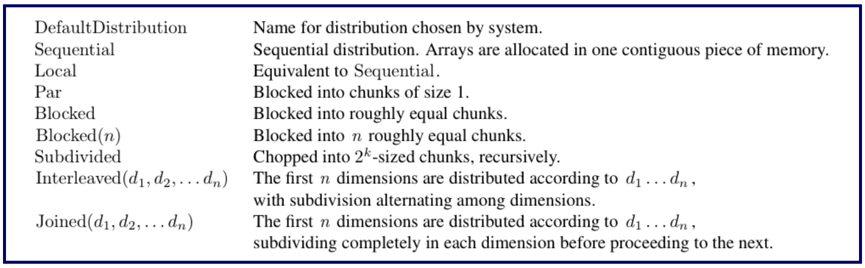
```
$v =$ spawn at $a.region(i)$ do
          $a_i$
     end
$w =$ spawn at $v.region()$ do
          $v.val() \cdot 17$
     end
```

lar block stmt

```
do
    $v := a_i$
    at $a.region(j)$ do
        $w := a_j$
    end
    $x = v + w$
end
```

Fortress - Latency Reduction

- Explicit data placement with Distributions



DefaultDistribution	Name for distribution chosen by system.
Sequential	Sequential distribution. Arrays are allocated in one contiguous piece of memory.
Local	Equivalent to Sequential.
Par	Blocked into chunks of size 1.
Blocked	Blocked into roughly equal chunks.
Blocked(n)	Blocked into n roughly equal chunks.
Subdivided	Chopped into 2^k-sized chunks, recursively.
Interleaved$(d_1, d_2, \ldots d_n)$	The first n dimensions are distributed according to $d_1 \ldots d_n$, with subdivision alternating among dimensions.
Joined$(d_1, d_2, \ldots d_n)$	The first n dimensions are distributed according to $d_1 \ldots d_n$, subdividing completely in each dimension before proceeding to the next.

```
a = Blocked.array(n,n,1); //Pencils along z axis
```

- User can define custom distribution by inheriting **Distribution** trait
  - Standard distributions implemented in this manner

Fortress - Portability

- Language based solution, requires compiler
- Runtime system part of Fortress implementation
- Responsible for mapping multithreaded onto target architecture
- **Regions** make machine information available to programmer
- Parallel model not affected by underlying machine

Table of Contents

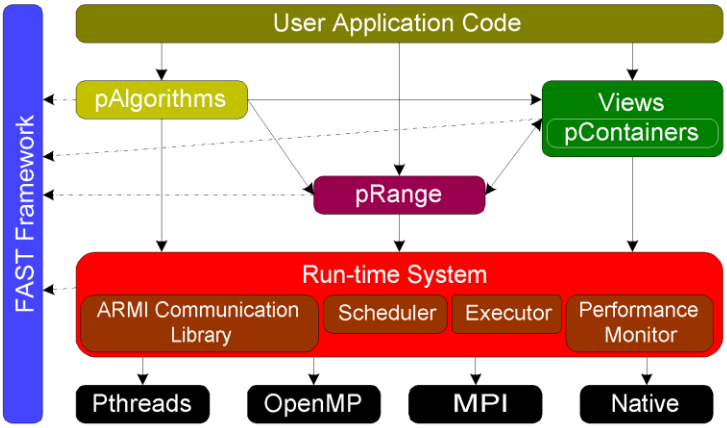
The STAPL Model

- Standard Adaptive Parallel Library
- Developed by Lawrence Rauchwerger, Nancy Amato, Bjarne Stroustrup and several grad students at Texas A&M
- Library similar and compatible with to STL
- Strong library development support
- Places parallelism burden primarily on library developers
- Commercial simple variant : Intel TBB

Adapted From: http://parasol.tamu.edu/stapl/

Standard Template Adaptive Parallel Library

A library of parallel, generic constructs based on the C++ Standard Template Library (STL).



FAST Framework

User Application Code

pAlgorithms

Views

pContainers

pRange

Run-time System

ARMI Communication Library

Scheduler

Executor

Performance Monitor

Pthreads

OpenMP

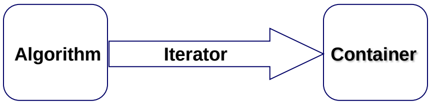
MPI

Native

Standard Template Library (STL)

Generic programming components using C++ templates.

- **Containers - collection of other objects.**
  - vector, list, deque, set, multiset, map, multi_map, hash_map.
  - Templated by data type. vector<int> v(50);
- **Algorithms - manipulate the data stored in containers.**
  - manipulate the data stored in containers.
  - count(), reverse(), sort(), accumulate(), for_each(), reverse().
- **Iterators - Decouple algorithms from containers.**
  - Provide generic ***element access*** to data in containers.
  - can define custom ***traversal*** of container (e.g., every other element)
  - count(vector.begin(), vector.end(), 18);



Algorithm

Iterator

Container

Execution Model

- Two models: User and Library Developer
- Single threaded – User
- Multithreaded – Developer
- Shared memory – User
- PGAS – Developer
- Data & task parallelism
- Implicit communications: User
- Explicit communications: Developer

Execution Model

- Memory Consistency:
  - Sequential for user
  - Relaxed for developer (Object level)
  - Will be selectable
- Atomic methods for containers
- Synchronizations: Implicit & Explicit

STAPL Components

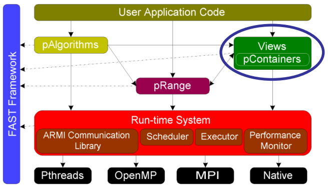
- Components for Program Development
  - pContainers, Views, pRange, pAlgorithms
- Run-time System
  - Adaptive Remote Method Invocation (ARMI)
  - Multithreaded RTS
  - Framework for Algorithm Selection and Tuning (FAST)

---

pContainers

Generic, distributed data structures with parallel methods.

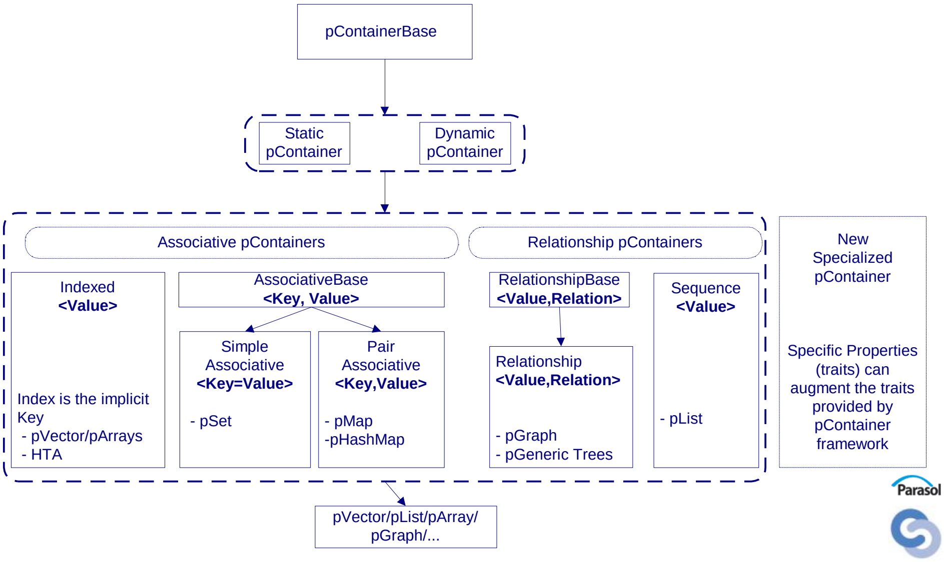
- Ease of Use
  - Shared object view
  - Generic access mechanism through Views
  - Handles data distribution and remote data access internally
  - Interface equivalent with sequential counterpart
- Efficiency
  - OO design to optimize specific containers
  - Template parameters allow further customization
- Extendability
  - New pContainters extend Base classes
- Composability
  - pContainers of pContainers



Currently Implemented

pArray, pVector, pGraph, pMap, pHashMap, pSet, pList

pContainer Taxonomy



pContainerBase

Static pContainer

Dynamic pContainer

Associative pContainers

Relationship pContainers

Indexed **<Value>**

Index is the implicit Key
- pVector/pArrays
- HTA

AssociativeBase **<Key, Value>**

Simple Associative **<Key=Value>**

- pSet

Pair Associative **<Key,Value>**

- pMap
- pHashMap

RelationshipBase **<Value,Relation>**

Relationship **<Value,Relation>**

- pGraph
- pGeneric Trees

Sequence **<Value>**

- pList

New Specialized pContainer

Specific Properties (traits) can augment the traits provided by pContainer framework

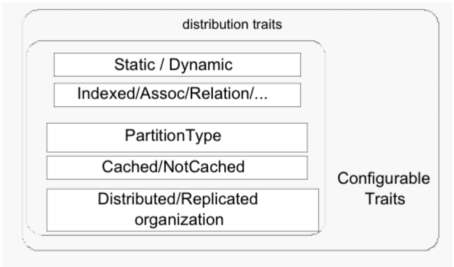
pVector/pList/pArray/ pGraph/...

pContainer Customization

Optional user customization through pContainer **Traits**.

- Enable/Disable Performance Monitoring.
- Select Partition Strategies.
- Enable/Disable Thread Safety.
- Select Consistency Models

```
class p_array_traits {
  Indexed, Assoc/Key=Index,
  Static,IndexedView<Static,...,
    Random>,
  DistributionManagerTraits,
  -u-Monitoring,
  -u-Relaxed
}
```



distribution traits

Static / Dynamic

Indexed/Assoc/Relation/...

PartitionType

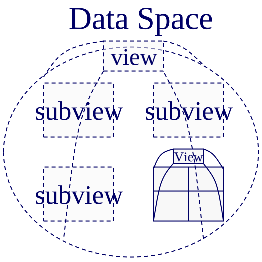
Cached/NotCached

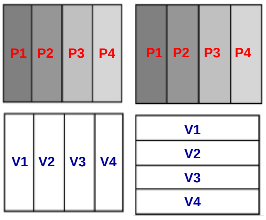
Distributed/Replicated organization

Configurable Traits

View

- STAPL equivalent of STL iterator, extended to allow for efficient parallelism.
- Focus on processing value range, instead of single item.
- Generic *access* mechanism to pContainer.
- Custom *traversal* of pContainer elements.
- Hierarchically defined to allow control of locality and granularity of communication/computation.

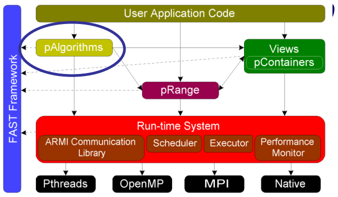




Gray -> the pContainer physical partition.
Transparent -> logical views of the data.

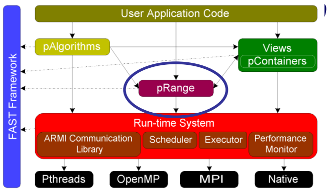
---

pAlgorithms



- 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)
  - Often, multiple implementations exist that are adaptively used by the library.

pRange

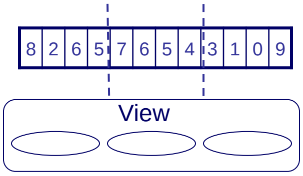


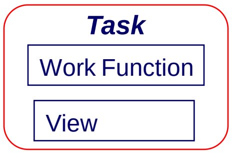
- pRange is a parallel task graph.
- Unifies work and data parallelism.
- Recursively defined as a tree of *subranges.*

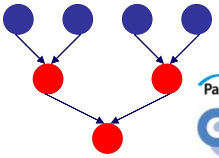
---

pRange -- Task Graphs in STAPL

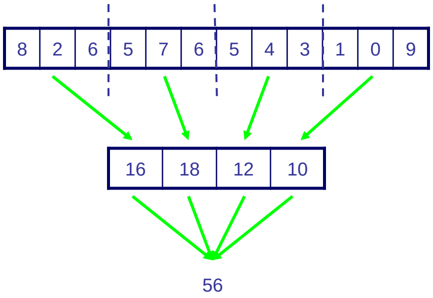
- Data to be processed by pAlgorithm
  - View of input data
  - View of partial result storage
- Work Function
  - Sequential operation
  - Method to combine partial results
- Task
  - Work function
  - Data to process
- Task dependencies
  - Expressed in Task Dependence Graph (TDG)
  - TDG queried to find tasks ready for execution







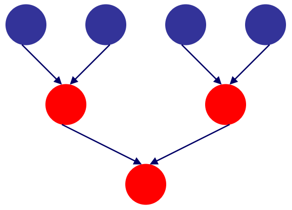
Task graph of pAlgorithm



A task is a work function and the set of data to process.

● = Find sum of elements

● = Combine partial results

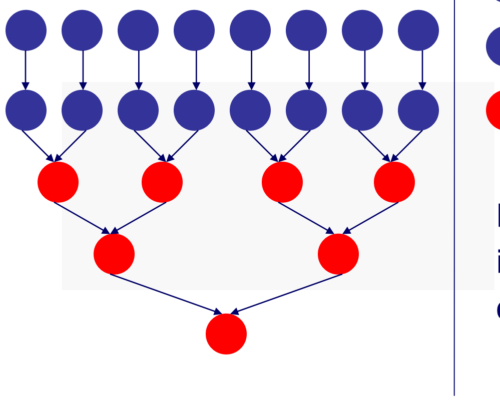


Tasks aren't independent.

Dependencies specify execution order of tasks.

Composing Task Graphs

- Increases amount of concurrent work available
- Forms a MIMD computation
- Dependencies between tasks specified during composition



● = Add 7 to each element

● = Find sum of elements

● = Combine partial results

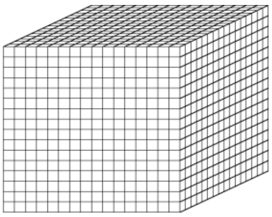
Dependencies only needed if tasks process the same data

Simple Dependence Specification

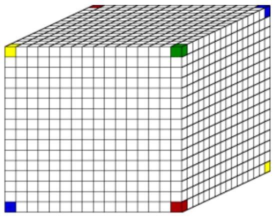
- Goal: Developer concisely expresses dependencies
  - Enumeration of dependencies is unmanageable
- Common patterns will be supported in pRange
  - Sequential – sources depend on sinks
  - Independent – no new dependencies needed in composed graph
  - Pipelined – dependencies follow a regular pattern

Discrete Ordinates Particle Transport Computation

- Important application for DOE
  - E.g., Sweep3D (3D Discrete Ordinates Neutron Transport) and UMT2K (Unstructured Mesh Transport)
- Large, on-going DOE project at TAMU to develop application in STAPL (TAXI)

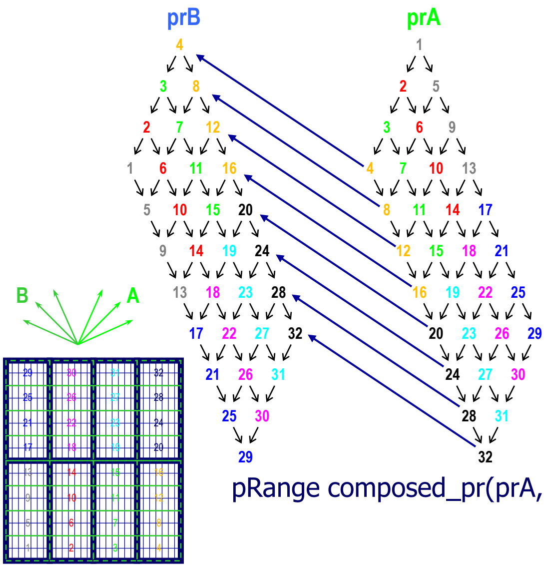


One sweep



Eight simultaneous sweeps

Pipeline Pattern Example



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

`pRange composed_pr(prA, prB, Pipeline(4,32,4));`

---

pRange Summary

- Binds the work of an algorithm to the data
- **Simplifies** programming task graphs
  - Methods to create tasks
  - Common dependence pattern specifications
  - Compact specification of task dependencies
  - Manages task refinement
  - Simple specification of task graph composition
- Supports multiple programming models
  - Data-parallelism
  - Task-parallelism