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

* Adapted from presentation by Janet Salowe - http://www.nbcx.rutgers.edu/hpc/hpf1.2/*
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

- **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
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
  - DISTRIBUT, ALIGN, etc.

HPF - Distribute

- !HPF$ DISTRIBUT 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.)
  - `::`
  - `*`

**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**

```f77
REAL a(3,3), image(3*2,3*2)
DO i = 1, 3
  DO j = 1, 3
      a(i,j) = image(i*2,j*2)
      & image(i*2+1,j*2)
      & image(i*2,j*2+1)
      & image(i*2+1,j*2+1)
  END DO
END DO
```

**HPF**

```hpf
REAL a(3,3), image(3*2,3*2)
REAL, DIMENSION(MY::) :: centroid
HPF DISTIBUTE(BLOCK,BLOCK) : image
HPF ALIGN centroid(i,j) WITH image(i*2,j*2)
DO i = 1, 3
  DO j = 1, 3
      a(i,j) = image(i*2,j*2)
      & image(i*2+1,j*2)
      & image(i*2,j*2+1)
      & image(i*2+1,j*2+1)
  END DO
END DO
```
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:

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

HPF Example - Matrix Multiply

```hpf
PROGRAM ABmult
IMPLICIT NONE
INTEGER, PARAMETER :: N = 100
INTEGER DIMENSION (N,N) :: A, B, C
!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)
```

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**

```chapel
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
    ```chapel
    var x : single int;
    begin
      x = foo(); // sub computation spawned
    end
    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`

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

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

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

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

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

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

**alternate data-driven usage:**

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

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

- 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

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, ~] }
```

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

\[
\begin{align*}
&\text{for } i-1:m, j-1:n \text{ do } \\
&a[i,j] := b[i] \ c[j] \\
&\text{end } \\
&\text{for } i-\text{seq}(1:m) \text{ do } \\
&\text{print } a[i,j] \\
&\text{end}
\end{align*}
\]

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

\[
\text{seq}(<5, \text{seq}(<1,2>), \text{seq}(<3,4>)>)
\]
Fortress - Explicit Task Parallelism

- **Tuple expressions**
  - comma separated exp. list executed concurrently
    
    $(\text{foo}(), \text{bar}())$

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

```fortress
   do
       \text{foo}()
   also do
       \text{bar}()
   end
```

Fortress - Explicit Task Parallelism

- **Spawn expressions (futures)**

```fortress
   \_ = \text{spawn} do
   \_ = \_ end
   \_ = \_ \text{val}() //return value, block if not completed
   \_ = \_ \text{ready}() //return true iff \_ completed
   \_ = \_ \text{wait}() //block if not completed, no return
   \_ = \_ \text{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

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

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

\[
\text{obj.region()} \quad //\text{region where object obj is located}
\]
\[
\text{r.isLocalTo(s)} \quad //\text{is region r in region tree rooted at s}
\]
Fortress - Latency Reduction

- Explicit data placement with Distributions

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Name for distribution chosen by system.</td>
</tr>
<tr>
<td>Sequential</td>
<td>Equivalent to Sequential. Arrays are allocated in one contiguous piece of memory.</td>
</tr>
<tr>
<td>Local</td>
<td>Blocked into chunks of size 1.</td>
</tr>
<tr>
<td>Par</td>
<td>Blocked into roughly equal chunks.</td>
</tr>
<tr>
<td>Blocked(n)</td>
<td>Blocked into <em>n</em> roughly equal chunks.</td>
</tr>
<tr>
<td>Subdivided</td>
<td>Chopped into 2^n-sized chunks, recursively.</td>
</tr>
<tr>
<td>Interleaved(d1, d2, ... dn)</td>
<td>The first <em>n</em> dimensions are distributed according to <em>d1</em> ... <em>dn</em> with subdivisions alternating among dimensions.</td>
</tr>
<tr>
<td>Joined(d1, d2, ... dn)</td>
<td>The first <em>n</em> dimensions are distributed according to <em>d1</em> ... <em>dn</em>, subdividing completely in each dimension before proceeding to the next.</td>
</tr>
</tbody>
</table>

```plaintext
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