Hierarchical Locality in HCAF
(Hierarchical Coarray Fortran)

Scott K. Warren
Rice University
August 13, 2014
Hierarchical Locality in HCAF

- **Overview**
- Model of Hierarchy
- Hierarchical Abstractions
- Tiling Pattern Specifications
Hierarchical Coarray Fortran (HCAF)

• **Motivation**
  - Large parallel computers are deeply hierarchical
  - Applications must *exploit* this hierarchy, not ignore it

• **Approach**
  - Language exposes hierarchy, programmer exploits it
  - Exposed hierarchies automagically mapped to hardware

• **Goals**
  - Explicit hierarchical locality in PGAS model
  - Dynamic task and data parallelism
  - Portable performance across machine topologies
  - In the spirit of Fortran
  - Extension of Rice CAF 2.0 with few incompatibilities

• **Disclaimers**
  - This work is preliminary
  - Still some pending design issues
  - No implementation yet
  - Irregular codes and heterogenous hardware are TBD
HCAF Design Principles

- **Optimizable** and *manually controllable*
  - Programmer makes *high-level decisions*, can *intervene at low level* if necessary
  - Compiler is responsible for most performance details

- **Explicit hierarchical locality**
  - *Single hierarchy model* for hardware, teams, coarrays, task/data parallelism
  - *Hierarchy abstraction* for locality-aware programming in a hardware-independent way

- **Single programming model** across all hierarchy levels (“H-PGAS into the node”)
  - Teams & coarrays on sets of cores *across or within nodes*
  - Async, do-parallel, collectives on any team *across or within nodes*

- **Mixed global-view & local-view programming**
  - *Hierarchical tiling* supports both element-wise & tile-wise access (global and local view)
  - *Relative locality* redefines coarray *local-vs-remote* distinction to *within-vs-outside current locale*

- **Strong typing** and *statically known locality*
  - *Type system* captures *hierarchical structure* of teams and coarrays
  - Static *correctness checking* of hierarchy references (e.g. subscript rank)
  - Static *locality-aware optimization*
  - *Dynamic hierarchy* supported by runtime checking
Related Work

- **Hierarchically Tiled Arrays and HPF**
  - HTA's are *hierarchical*, but *dynamic tiling* \(\Rightarrow\) no static optimization
  - HPF has *static tiling info* \(\Rightarrow\) aggressive optimization, but *not hierarchical*
  - HCAF: *hierarchical tiling with static info for locality optimization*

- **Hierarchical Place Trees and Titanium Hierarchical Teams**
  - HPTs model locality only *intra-node* and are *global & fixed at startup*
  - Titanium teams are *programmable & modular*, but model only *inter-node* locality
  - HCAF: *programmable, modular teams extending inter-node to intra-node*

- **Topology Mapping**
  - Two approaches: *graph-based* (LibTopoMap) and *tree-based* (TreeMatch, Rubik)
  - TreeMatch maps *arbitrary-size trees*, but trees are *unordered*
  - Rubik uses *Cartesian topologies* but maps *same-size trees*
  - HCAF: *maps arbitrary-size trees with Cartesian topologies*

- **Dynamic Parallelism & Work Stealing** (X10, Habanero, HotSLAW et al)
  - *Locality-aware fork-join parallelism* + parallel loops based on fork-join
  - Sophisticated inter- and intra-node *hierarchical work stealing* algorithms
  - HCAF: *same, but with more static info for locality optimization*
Opportunity: Statically-known Hierarchical Tiling

[Our] current implementation as a library forces to use dynamic analysis techniques to determine the communication patterns required when data is to be shuffled among processors. A compiler could calculate statically those patterns when they are regular enough, and generate a code with less overhead.


(emphasis added)

Cross-component optimization is essential to attain reasonable performance. For languages like HPF, compilers synthesize message passing communication operations and manage local buffers. Interprocedural analysis can reduce the frequency and volume of communication significantly. In the HTA library, communication optimization is in the hands of the programmer. A possible concern is that the programmer may not use the library efficiently.


(summarized)
Opportunity: Machine-independent Explicit Locality

- **Locale** denotes a relatively compact *subset of hardware*
- **Team** provides *abstraction of hardware* subset with desired topology
- **Coarray** exposes *data locality* for explicit management by application
- **Map** $M_1$ *distributes coarray* over application topology
- **Map** $M_2$ *embeds application topology* into physical topology
Hierarchical Localilty in HCAF

- Overview
- **Model of Hierarchy**
  - Resource hierarchies
  - Hierarchy maps
  - Hierarchy patterns
- Hierarchical Abstractions
- Tiling Pattern Specifications
Hierarchy: Basic Concepts

- **Hierarchy** here means *recursive partitioning*
  - ... of a finite set
  - Each set in the hierarchy has an associated partition into *subsets*
  - A hierarchy may be viewed as a *tree of sets* in two ways
  - Consider the hierarchy \{ \{1\}, \{2\}, \{3\}, \{4\}, \{5\} \}
  - \(T_1\) has nodes labeled with *included* sets
  - \(T_2\) has leaves labeled with *owned* sets;
    an interior node's included set is the union of its children's included sets
  - *We use \(T_1\) for natural global / local view, but \(T_2\) describes hardware*
  - HCAF uses hierarchies to represent *locality*
    - A subtree denotes a *neighborhood* of things relatively close together
    - A node's children subdivide it into *smaller, closer* neighborhoods
- **Tiling** here means *rectangular partitioning*
  - ... of a rectangular \(n\)-dimensional grid into *tiles*, also rectangular
  - A tiling may be *nonaligned, aligned, or regular* [1]
- **Hierarchical tiling** means *recursive rectangular partitioning*
  - Each tile is partitioned into a set of *sub-tiles*
  - Can be viewed as a hierarchy or tree — with *rectangular structure*

Cartesian Resource Hierarchies

- The structure underlying locales, teams, and coarrays
- A **Cartesian resource hierarchy** is a tuple \((V, E, \{A_r\}, \mathcal{K})\) where
  - \((V, E, A)\) is a rooted attributed tree with \(A = \{A_r\} \cup \{\mathcal{K}\}\)
  - Each \(A_r\) is a resource attribute function of type \(R_r\)
  - \(\mathcal{K}\) is the topology function which assigns to each interior node \(n \in V\) with children \(C_n\) a Cartesian topology \(\mathcal{K}(n)\) for \(C_n\)
- A **resource attribute function of type \(R\)** is some \(f : V \rightarrow \mathcal{P}(R)\) where
  - \(R\) is a finite set of resource elements and \(\mathcal{P}(R)\) is the power set of \(R\)
  - \(\forall n \in V\) with children \(C_n\) : \(\{f(c) \mid c \in C_n\}\) is a partition of \(f(n)\)
  - \(\forall\) leaf \(n \in V\) : \(f(n)\) is a singleton
- A **Cartesian topology for \(V\)** is a function \(t : \mathcal{D}_k \rightarrow V\) where
  - \(t\) is one-to-one (need not be onto)
  - \(\mathcal{D}_k = \prod_i [L_i, U_i]\) is a \(k\)-dimensional Cartesian domain (ie with rank \(k\))
  - \(\{L_i\}\) and \(\{U_i\}\) are the lower and upper bounds of \(\mathcal{D}_k\)
  - The shape of the topology is \((U_1 - L_1, U_2 - L_2, \ldots, U_k - L_k)\)
Characterization of Cartesian Hierarchies

- A \textit{d-uniform} hierarchy is one where every leaf has depth \( d \)
- A \textit{d-ranked} hierarchy is one where
  - Every leaf node has depth \( \geq d \)
  - \( \forall d' < d \exists k_{d'} \text{ s.t. every node of depth } d' \text{ has a topology of rank } k_{d'} \)
  - Then the \textit{d-rank} of the hierarchy is \( (k_0, k_2, \ldots k_{d-1}) \)
- A \textit{ranked} hierarchy is \( d \)-ranked and \( d \)-uniform for some \( d \); then \( (k_0, k_2, \ldots k_{d-1}) \) is its rank
- A \textit{d-regular} hierarchy is one where
  - The hierarchy is \( d \)-ranked
  - \( \forall d' < d \exists S_{d'} \text{ s.t. every node of depth } d' \text{ has a topology of shape } S_{d'} \)
  - Then the \textit{d-shape} of the hierarchy is \( (S_0, S_2, \ldots S_{d-1}) \)
- A \textit{regular} hierarchy is \( d \)-regular and \( d \)-uniform for some \( d \); then \( (S_0, S_2, \ldots S_{d-1}) \) is its shape
- HCAF uses these properties for security and efficiency:
  - Locales and teams are ranked; coarrays are regular (but not sections)
  - \textit{Types} of hierarchical objects have \textit{d-rank type parameters} for type checking and optimization of subscripts and loops (and additional info about distribution and communication)
Tiled Resource Hierarchies

- A **tiled resource hierarchy** is a tuple \((V, E, \mathcal{K}, \{A_t\}, \mathcal{T})\) where
  - \((V, E, \mathcal{K}, \{A_t\})\) is a Cartesian resource hierarchy
  - \(A_t \in \{A_r\}\) is the **tiled resource** of type \(R_t\)
  - \(\mathcal{T}\) is the **tiling function**, a resource attribute assigning to each node \(n \in V\) a Cartesian topology \(\mathcal{T}(n)\) for \(A_t(n)\) which satisfies certain conditions
  - \(R_t\) is the set of **tiled elements**, \(A_t(n) \subset R_t\) is the **tile** at \(n\), and \(\mathcal{T}(n)\) is the **element topology** at \(n\)
  - \(\mathcal{T}(n)\) specifies an index tuple for each tile element of \(n\)'s tile
  - \(\mathcal{T}\) must satisfy **tiling conditions** at every \(n \in V\) with children \(C_n\):
    - \(\{\mathcal{T}(c) \mid c \in C_n\}\) is a partition of \(\mathcal{T}(n)\), viewing the functions as sets of pairs
    - The tile at \(n\) has rank \(k\) and bounds \([L_i]\) and \([U_i]\) of \(D_k\), where \(\mathcal{T}(n) : D_k \rightarrow V\)
    - Thus a given tile element has the same indices at every level of tiling;
      HCAF uses this convention for subscripting teams and coarrays
  - **Rank** and **shape** are defined for both **elements** and **tiles** at a node:
    - We use **rank**, **shape**, and **size** for the **element-wise** topology at a node
    - We use **corank**, **coshape**, and **cosize** for the **tile-wise** topology at a node
Hierarchy Maps

- A hierarchy map $M$ from $G$ to $H$ is a tuple $(G, H, m)$ where
  - $m : V_G \rightarrow \mathcal{P}(V_H)$ is descendant-preserving, i.e.
    if $p, q \in V_G$ and $p$ is a descendant of $q$, then
    $\forall r \in m(p) \exists s \in m(q)$ such that $r$ is a descendant of $s$
  - This preserves our notion of locality (relative closeness)
  - Cartesian topologies are not preserved, but should be “respected”
- Hierarchy maps adapt an application’s virtual hierarchies to fit the current job’s hardware hierarchy
  - A hierarchical team is mapped to a set of processors (with corresponding hierarchical structure)
  - A hierarchical coarray is mapped to a set of memories (with corresponding hierarchical structure)
  - Hierarchy map composition provides modularity: e.g. if $H$ is the hardware and $G$ is a team passed to a library, the library realizes its preferred team structure $G_2$ by composing a new map with $G$’s existing map:
    $G_2 \rightarrow G \rightarrow H$
- Goodness of maps and finding good ones are TBD
  - But there are many relevant papers & working systems
Hierarchy Map Examples

coarse to fine

shallow to deep

low to high rank

fine to coarse

deep to shallow

high to low rank
Goodness of Hierarchy Maps

Better: $m(b)$ and $m(c)$ in nearby locale $r$

Worse: $m(b)$ and $m(c)$ in distant locale $q$
Goodness of Hierarchy Maps
(hierarchical team → hardware)

Better: $m(b)$ and $m(c)$ in nearby locale $r$
$\Rightarrow b \leftrightarrow c$ communicate via memory access

Worse: $m(b)$ and $m(c)$ in distant locale $q$
$\Rightarrow b \leftrightarrow c$ communicate via messaging
Goodness of Hierarchy Maps

- Best mapping between a given pair of hierarchies may not be great
  - How serious this is depends on the situation
  - E.g. the map above may be fine if all target locales are shared-memory
- For best results: choose a source hierarchy that maps well to target
- HCAF’s answer for this is tiling patterns

Bad: $m(b)$, $m(c)$, and $m(d)$ in distant locale $q$
...and can’t do better!
Tiling Patterns

- A **tiling pattern** is a pair \( P = (R, M) \) where
  - \( R = (k_0, k_2, \ldots k_{d-1}) \) is a d-rank
  - \( M \) is a possibly infinite set of tiled resource hierarchies with d-rank \( R \), comprising all the matches of \( P \)
- A **matching function** is some \( \text{Match} : (P, \mathcal{D}_k, H_T) \mapsto (H_O, m) \) where
  - \( P = ( (k_0, k_2, \ldots k_{d-1}), M ) \) is the **tiling pattern to be matched**
  - \( \mathcal{D}_k \) is the **input domain**, a Cartesian domain with rank \( k = k_0 \)
  - \( H_T \) is the **target hierarchy**, a tiled resource hierarchy that the match result should conform to
  - \( H_O \in M \) is the **output hierarchy**, a tiled resource hierarchy satisfying:
    - \( H_R \in M \), i.e. the **output hierarchy matches the pattern** \( P \)
    - \( \text{Domain}(\mathcal{T}(r)) = \mathcal{D}_k \), where \( r \) is the root of \( H_O \);
      i.e. the top level tile of \( H_O \) is the input domain,
      i.e. the input domain is tiled by \( P \) to give the output hierarchy
    - \( m \) is the **output hierarchy map** from \( H_R \) to \( H_T \);
      i.e. a view of the output hierarchy as an abstraction of the target
- Of course we prefer that \( m \) be a **good** hierarchy map
Hierarchical Locality in HCAF

• Overview
• Model of Hierarchy
• **Hierarchical Abstractions**
  • *Locales*: machine topology
  • *Teams*: processor groups
  • *Coarrays*: data objects
• Tiling Pattern Specifications
Locales: Hierarchical Machine Topology

• **Locales** are units of computer hardware locality
  • Nested regions of a parallel computer containing computing resources which are *relatively close* in terms of *communication cost*
  • E.g. cores, dies, sockets, nodes, boards, chassis, cabinets, ...

• A locale is a Cartesian resource hierarchy \((V, E, A, \mathcal{K})\) where
  • \(V\) is the set of regions and \(E\) is the containment relation among them
  • \(A = \{\text{Procs, Mem}ss, \text{Comm}\}\) describes each locale’s computing elements
    • **Procs** : \(V \rightarrow \mathcal{P}(P)\) is the *processor resource function*
      • \(P\) is the set of processors (hardware threads)
      • \(\text{Procs}(e) = \{p_1, p_2, \ldots\}\) is the set of processors contained in locale \(e\)
    • **Mems** : \(V \rightarrow \mathcal{P}(M)\) is the *memory resource function*
      • \(M\) is the set of memories (RAMs or caches)
      • \(\text{Mems}(e) = \{m_1, m_2, \ldots\}\) is the set of memories contained in locale \(e\)
    • **Comm** : \(V \rightarrow \{\text{distributed, shared}\}\) is the *communication attribute function*
      • *distributed* and *shared* denote respectively communication via message passing and memory reference
      • \(\text{Comm}(e)\) is the worst-case communication kind among elements of \(e\)
      • Require that no *shared* locale has a *distributed* sub-locale

\[
\begin{align*}
V &= \{p_1, p_2, p_3, p_4\} \\
E &= \{m_1, m_2\} \\
A &= \{\text{Procs}, \text{Mems}, \text{Comm}\} \\
K &= \{\text{distributed, shared}\}
\end{align*}
\]
Example Locale: 2 Hopper 24-core Nodes

- 2 Magny-Cours cpus / node
- 2 6-core dies / socket
- 2 memory paths / die
- 4 HyperTransport3 links / die

Nested locales:
- 21 sockets
- 21 nodes
- 42 dies
- 84 cores

Shared bandwidths:
- 8 Gb/s
- 19 Gb/s
- 42 Gb/s

Distributed bandwidth:
- 6 Gb/s

Cache levels:
- L1 & L2 caches
- L3 cache

RAM: M M

Nodes: P M M

Sockets: M M

Dies: M M
locales = hierarchically partitioned address spaces
smaller locale = closer elements = cheaper communication

- Any processor can access any address space
- Speed of access is modeled by the smallest enclosing locale of a processor and the other processor or memory it accesses
- Equivalently, by the lowest common ancestor node in the corresponding Cartesian resource hierarchy
Locales and Hierarchical PGAS

finest partition of address space = innermost locale = one die
⇒
shared-memory comm at 42 Gb/s
Locales and Hierarchical PGAS

mid-level partition of address space
= mid-level locale
= one node
⇒ shared-memory comm at 8 Gb/s
Locales and Hierarchical PGAS

coareset partition of address space = top-level locale = two nodes ⇒ distributed-memory comm at 6 Gb/s

coarsest partition of address space = top-level locale = two nodes ⇒ distributed-memory comm at 6 Gb/s
Teams: Hierarchical Processor Groups

- **Teams** are groups of hardware processors (cores)
  - Nested sets of processors which are *relatively close* in communication cost
  - Teams specify sets of processors and inherit sets of memories
  - Teams serve as *abstract locales* to isolate application from hardware details

- A team is a Cartesian resource hierarchy \( T = (V, E, A, \mathcal{K}) \) where
  - \( V \) is the set of subteams and \( E \) is the containment relation among them
  - \( A = \{\text{Procs, Mems, Comm}\} \) just as for locales

- A team has a hierarchy map \( m : V_T \to \mathcal{P}(V_H) \) where
  - \( H \) is the hardware locale (root)
  - \( m(r) \) is typically a sub-locale of the hardware locale, where \( r \) the root of \( T \); it denotes the *machine subset implementing* \( T \)
  - \( \text{Procs}(r) \) is the team’s set of processors, possibly a subset of \( \text{Procs}(m(r)) \)
  - \( m \) describes how the team’s processors are distributed on the machine

- Consider a team as a *hierarchy of processors*, with its memories just inherited from its associated locale:
  - Require \( \forall t \in V : \text{Mems}(t) = \text{Mems}(m(t)) \)
  - These are the memories close to the team’s processors

- A team is mapped to hardware by the map \( m \)
Teams: Locality-aware Parallelism

• Teams are resources for parallel execution
  • Not a set of images or threads, but a set of processors (w/ nearby memories)
  • Basic unit of parallelism: spawn task on team (controls execution locality at arbitrary grain)
  • Team’s processors cooperate to execute in parallel all tasks spawned on it
  • Team’s memories hold tasks’ stack frames & heap-allocated objects (by default)

• Uniform model for all concurrency in HCAF
  • Task parallelism: like async/finish X10, Habanero, Chapel, CAF 2.0
  • Loop parallelism: iterations are spawned on current team like X10 ateach
  • Data parallelism: array intrinsics implemented as parallel loops
  • Both intra-node and inter-node spawning are supported

• Hierarchical work-stealing scheduler per team
  • Similar to place schedulers in Habanero’s Hierarchical Place Trees
  • Both distributed-memory and shared-memory work stealing are supported

• Implementation
  • Berkeley HotSLAW; Quintin & Wagner; Olivier & Prins; Saraswat, Paudel et al; etc
  • May be complicated by HCAF’s programmable teams
Coarrays: Hierarchical Data Objects

- **Coarrays** are tiled groups of storage locations (elements)
  - Nested tiles of elements which are *relatively close* in communication cost
  - Coarrays *specify* sets of elements and *inherit* processors and memories
  - Coarrays are *allocated on teams* and their tiles are placed in teams' memories

- A coarray is a tiled resource hierarchy \( C = (V, E, K, A, T) \) where
  - \( V \) is the set of sub-tiles and \( E \) is the containment relation among them
  - \( A = \{ \text{Elems}, \text{Procs}, \text{Mems}, \text{Comm} \} \) where \( \text{Elems} \mapsto \) storage locations in each tile
  - \( \text{Elems}(r) \) is the coarray's top level (global-view) tile and \( T(r) \) is the tile's shape

- A coarray has a hierarchy map \( m : V_C \rightarrow \mathcal{P}(V_T) \) where
  - \( T \) is the team on which \( C \) is allocated
  - \( m(r) \) is typically the root of the team, where \( r \) is the root of \( C \)
  - \( m \) describes how the coarray's tiles are distributed on the team

- Consider a coarray as a *hierarchy of elements*, with its processors and memories just inherited from its associated team:
  - Require \( \forall c \in V_C : \text{Procs}(c) = \text{Procs}(m(c)) \) and \( \text{Mems}(c) = \text{Mems}(m(c)) \)
  - These are the processors owning and memories storing the coarray

- A coarray is mapped to hardware by the composition \( C \rightarrow T \rightarrow H \)
Example: Coarray on Team on 2 Hopper Nodes

- Team and coarray hierarchies have same shape here, but this is not required.
- Each leaf coarray tile is allocated in one die’s memories and has 3 cores of the die assigned to it.
- Each 3-core leaf subteam is mapped to a die’s locale, which is the smallest locale enclosing its cores.
- The team is a 3-level 16-leaf abstraction of the 5-level 48-leaf hardware hierarchy.
Hierarchical Locality in HCAF

- Overview
- Model of Hierarchy
- Hierarchical Abstractions
  - Tiling Pattern Specifications
    - Level and dimension specs
    - Parameters and constraints
    - Distribution and communication specs
Tiling Pattern Specifications

• Problem:
  • Locality-aware applications and optimizers \textit{statically depend on hierarchy shape}
  • Hardware hierarchy is \textit{known only at runtime} (cf. machine type & job scheduler)
  • Need \textit{abstraction} to decouple application’s \textit{virtual hierarchies} from machine’s \textit{real hierarchy}
  • But manually mapping virtual to real is difficult

• Solution:
  • \textit{Tiling pattern} describes a \textit{set of desirable hierarchies}
  • Compiler \textit{statically optimizes} using properties common to all set members
  • Runtime \textit{dynamically chooses} desirable hierarchy with a \textit{good mapping} to hardware

• Tiling pattern specification defines:
  • \textit{Hierarchy rank} (first $d$ levels) and set of \textit{hierarchy coshapes}
  • Required \textit{communication kind} at each level (distributed vs shared memory)
  • Tile \textit{distributions} and Rubik-style \textit{tilts/shifts/etc}

• Example: tiling pattern $P$ with hierarchy rank $(2,1)$

\begin{verbatim}
tiling :: P(N)
  [ N block, N cyclic(100) ]
  [ 2..32 ] shared
end tiling
\end{verbatim}
Level and Dimension Specs

- level spec: \([2, *]\)
- dimension specs:
  - rank = 2
  - shape = (4, 4)
  - corank = 0

* \(n\) divides into \(n\) tiles
* \(#n\) divides into tiles of size \(n\)
* * leaves dimension undivided
  - “tiles out” dimension

- \([2, 2]\)
  - tile rank = 2
  - tile shape = (2, 2)
  - corank = 2
  - coshape = (2, 2)

- \([2, *]\)
  - tile rank = 2
  - tile shape = (2, 4)
  - corank = 1
  - coshape = (2)

- \([*, 2]\)
  - tile rank = 2
  - tile shape = (4, 2)
  - corank = 1
  - coshape = (2)

- \([-, *]\)
  - tile rank = 1
  - tile shape = (4)
  - corank = 1
  - coshape = (4)

- \([- , 2]\)
  - tile rank = 1
  - tile shape = (2)
  - corank = 2
  - coshape = (4, 2)
Level and Dimension Specs

level spec: \([2, *]\)
dimension specs

- rank = 2
- shape = (4, 4)
- corank = 0
- [2, *] leaves dimension undivided
- [-, *] “tiles out” dimension

- \(n\) divides into \(n\) tiles
- \#n divides into tiles of size \(n\)

- tile rank = 2
- tile shape = (2, 2)
- rank = 2
- tile shape = (2, 4)
- tile rank = 2
- tile shape = (4, 2)
- tile rank = 1
- tile shape = (4)
- tile rank = 1
- tile shape = (2)
Parameters and Constraints

- **Parametrized** pattern specifies a *set of hierarchies*
  - *Parameters* are positive integer variables local to pattern
  - *Constraints* are arithmetic predicates over parameters
  - An *instantiation* is an assignment of values to parameters s.t. all constraints are satisfied

*Pattern matching:*
- Given hierarchy $H$, pattern $P$, and input tile $T$, find instantiation $P'$ of $P$ and $H' = tiling(T, P')$ s.t. $\exists$ “good” mapping $M : H' \rightarrow H$
- Result is $(H', M)$

- *Implicit parameter* $\equiv$ unnamed param + constraint
  - Range: $expr .. expr$

- *Extents* in dimension-specs are Fortran exprs
  - Treated like array bound expressions

- Dimension-specs have *lower* and *upper bounds*
  - Like array bounds: $extent : extent$
  - Empty lower bound $\equiv 1$, empty upper bound $\equiv$ any
    - $0 : 7..15 \Rightarrow 8 \leq n \leq 16$ elements indexed from 0
    - $: \Rightarrow n > 0$ elements indexed from 1
Distribution and Communication Specs

• **Distribution specifier** modifies dimension-spec
  • Specifies a dimension’s assignment of elements to tiles
    i.e partially specifies $T(c)$ at each child $c$ of tiled node
  • Classic distribution specs like HPF:
    - _block_ contiguous w/ extent $n$ or #$n$
    - _cyclic($k$)_ cyclic over $n$ w/ extent $k$
  • Additional distribution specs like Rubik
    - _tilt_ tile boundary tilted
    - _zigzag_ tile boundary zig-zagged
    - _zorder_ space filling curve
  • Default distribution is _block_, yields conventional tiling

• **Communication specifier** modifies level-spec
  • Specifies worst-case communication type at level
    ⇒ acts as a constraint in pattern matching
  • Types of communication:
    - _distmem_ message passing
    - _sharedmem_ memory access
    - _image_ SPMD program instance (shared)
    - _any_ unspecified (the default)