Introduction to HCAF: Hierarchical Coarray Fortran

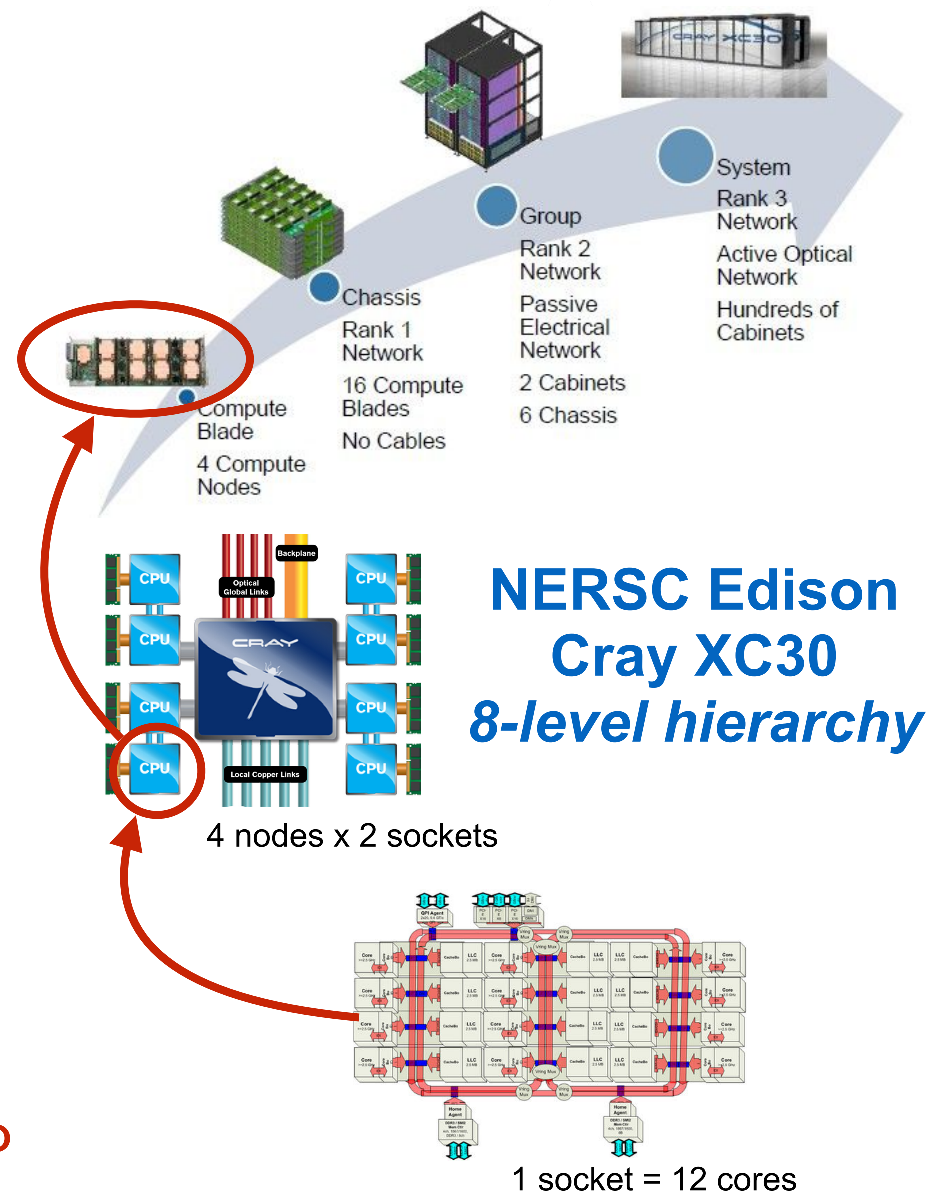
Scott K. Warren
Rice University

Outline

- Overview
- HCAF Hierarchy Model
- Hierarchical Abstractions
- Language Constructs

Hierarchical Coarray Fortran (HCAF)

- Motivation
  - Large parallel computers are deeply hierarchical
  - Applications must ***exploit*** this hierarchy, not ignore it
- HCAF 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
- Approach
  - Language exposes hierarchy, programmer exploits it
  - Exposed hierarchies automagically mapped to hardware
- **Disclaimers**
  - **This work is preliminary**
  - **Still some pending design issues**
  - **No implementation yet**
  - **Irregular codes and heterogenous hardware are TBD**



HCAF Goals

Hierarchical locality, PGAS, and dynamic parallelism in the spirit of Fortran

What do I mean by "spirit of Fortran" ?

- Focus on dense array handling
- Emphasis on performance
- Strong type checking
- Aggressive static optimization

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*** ⇒ no static optimization
  - HPF has ***static tiling info*** => 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-image*** locality
  - **HCAF:** ***programmable, modular teams extending inter-node to intra-image***
- *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

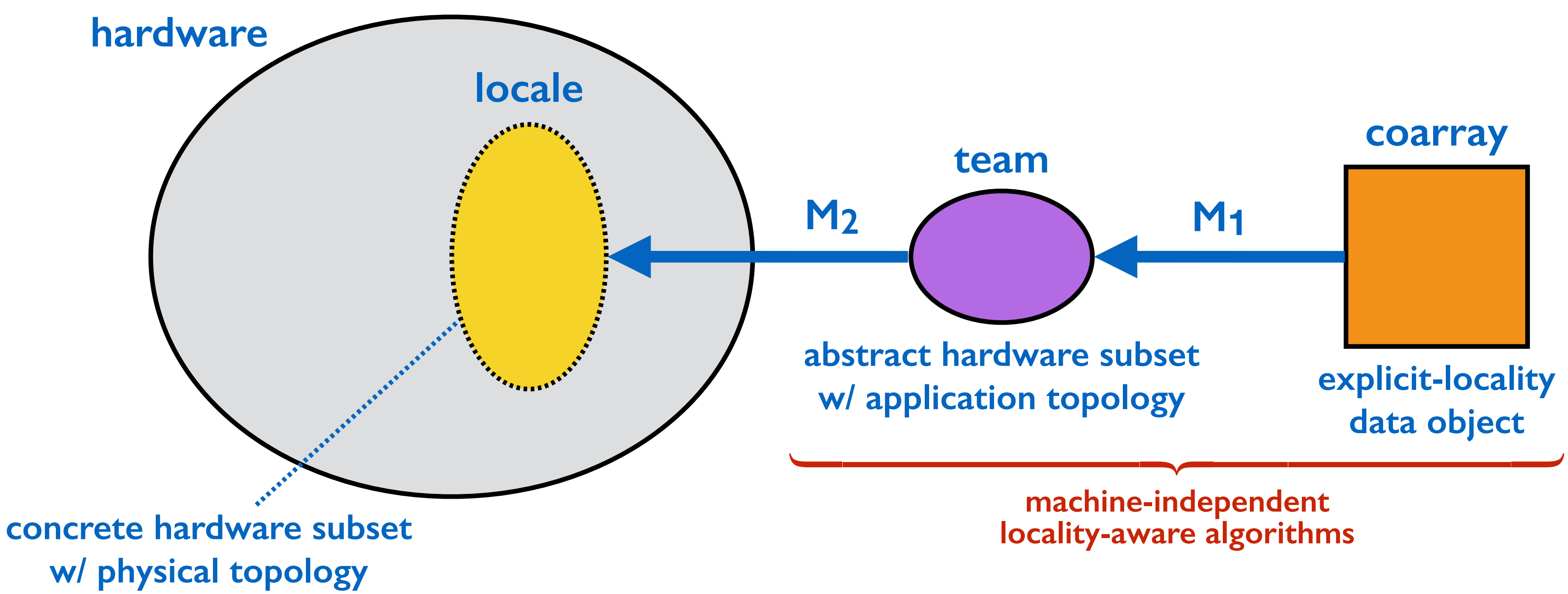
[Our] current implementation as a library forces to use dynamic analysis techniques to determine the communication patterns required when data is to be shuffled among processors. **A compiler could calculate statically those patterns when they are regular enough, and generate a code with less overhead.**

"Programming for Parallelism and Locality with Hierarchically Tiled Arrays", Bikshandi et al, 2006.
(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.**

"Optimization Techniques for Efficient HTA Programs", Fraguela, Bikshandi, et al, 2012.
(summarized)

Opportunity: Machine-independent Explicit Locality



all hierarchical

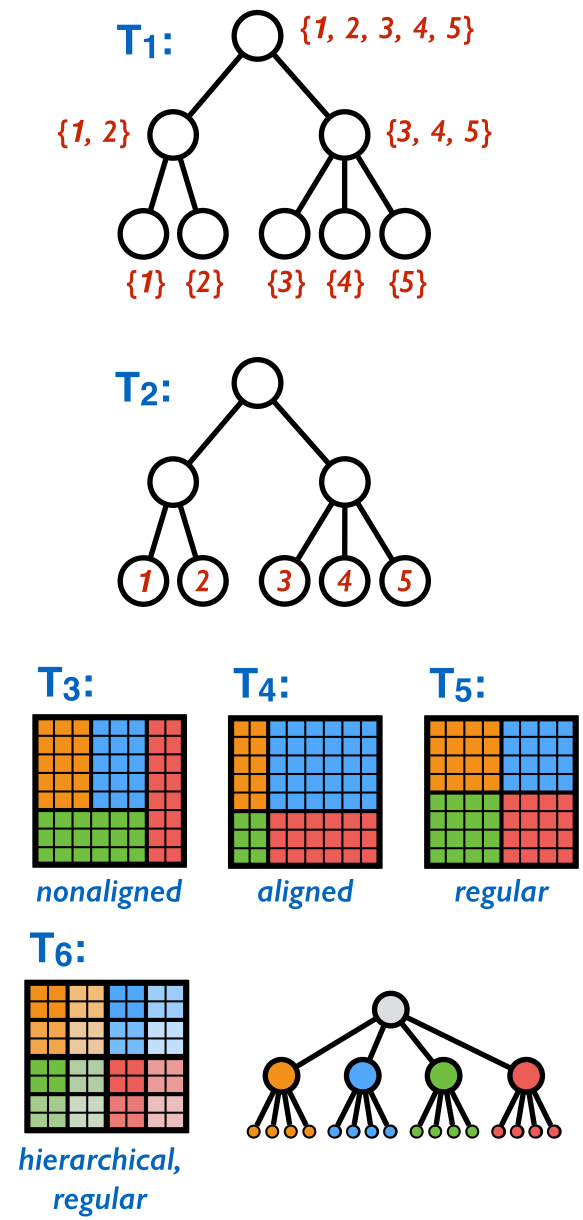
- *Locale* denotes a relatively compact ***subset of hardware***
- *Team* provides ***abstraction of hardware subset*** with desired topology
- *Coarray* ***exposes data locality*** for explicit management by application
- *Map* M_1 ***distributes data*** over application topology
- *Map* M_2 ***embeds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

Outline

- Overview
- **HCAF Hierarchy Model**
  - **Resource hierarchies**
  - **Hierarchy maps**
  - **Hierarchy patterns**
- Hierarchical Abstractions
- Language Constructs
- Implementation Ideas

Hierarchy: Basic Concepts

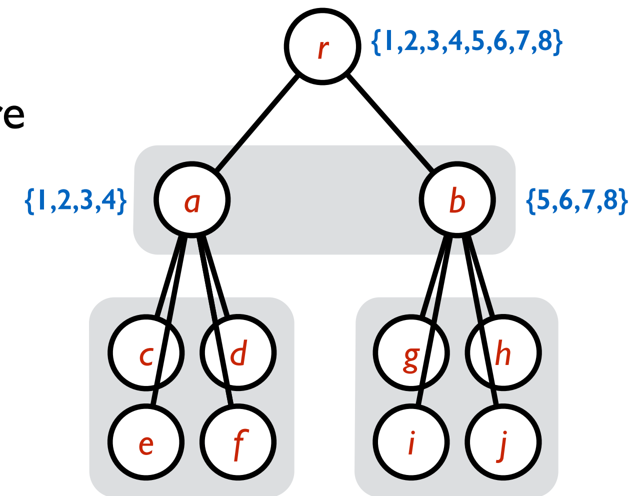
- ***Hierarchy*** here means ***recursive partitioning***
  - … of a finite set
  - Each ***set*** in the hierarchy has an associated partition into ***subsets***
  - A hierarchy may be viewed as a ***tree of sets*** in two ways
    - Consider the hierarchy { { {1}, {2} }, { {3}, {4}, {5} } }
    - T_1 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***



[1] P. Furtado and P. Baumann. Storage of multidimensional arrays based on arbitrary tiling, *15th International Conference on Data Engineering*, pp.480–489, 1999.

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 $(U_1 - L_1,\ U_2 - L_2,\ \ldots\ U_k - L_k)$



f :

$r \mapsto \{1,2,3,4,5,6,7,8\}$
$a \mapsto \{1,2,3,4\}$
$b \mapsto \{5,6,7,8\}$
$c \mapsto \{1\}$
$d \mapsto \{2\}$
$e \mapsto \{3\}$
$f \mapsto \{4\}$
$g \mapsto \{5\}$
$h \mapsto \{6\}$
$i \mapsto \{7\}$
$j \mapsto \{8\}$

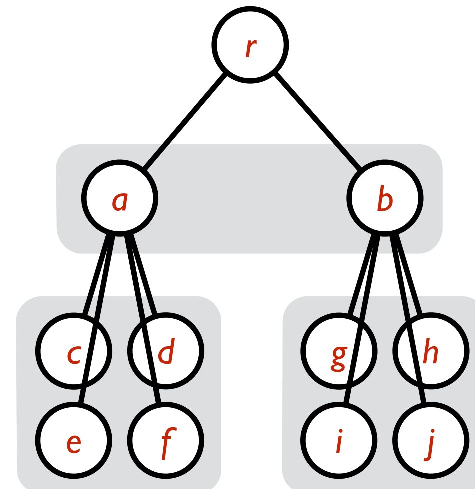
$\mathcal{K}(r)$:
$(1) \mapsto a$
$(2) \mapsto b$

$\mathcal{K}(a)$:
$(1,1) \mapsto c$
$(2,1) \mapsto d$
$(1,2) \mapsto e$
$(2,2) \mapsto f$

$\mathcal{K}(b)$:
$(1,1) \mapsto g$
$(2,1) \mapsto h$
$(1,2) \mapsto i$
$(2,2) \mapsto j$

Characterization of Cartesian Hierarchies

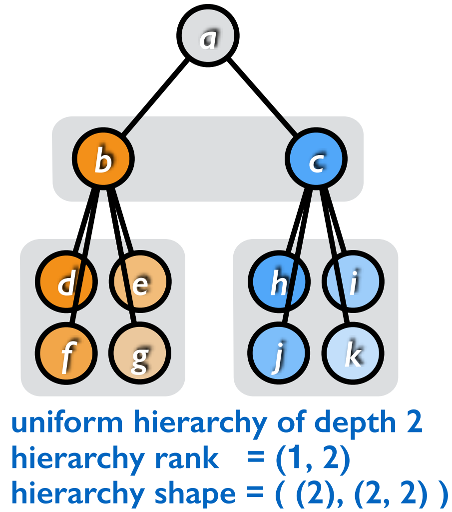
- A ***d-uniform*** hierarchy is one where every leaf has depth d
- *A* ***d-ranked*** hierarchy is one where
  - Every leaf node has depth $\geq d$
  - $\forall d' < d \;\; \exists k_{d'}$ s.t. every node of depth d' has a topology of rank $k_{d'}$
  - Then the ***d-rank*** of the hierarchy is $(k_0,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

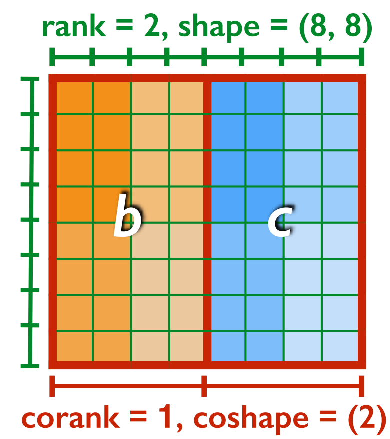


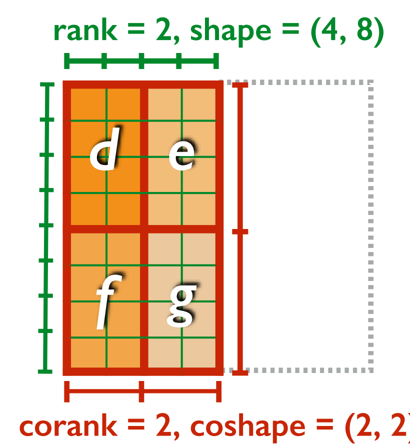
regular hierarchy of depth 2
hierarchy rank = (1, 2)
hierarchy shape = ((2), (2, 2))

Tiled Resource Hierarchies

- A ***tiled resource hierarchy*** is a tuple $(V, E, \mathcal{K}, \{A_r\}, \mathcal{T})$ where
  - $(V, E, \mathcal{K}, \{A_r\})$ 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 $\mathcal{D}_k$, where $\mathcal{T}(n) : \mathcal{D}_k \to V$
  - 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

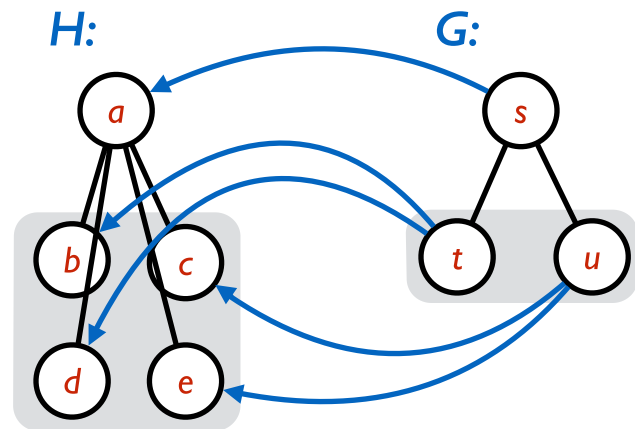




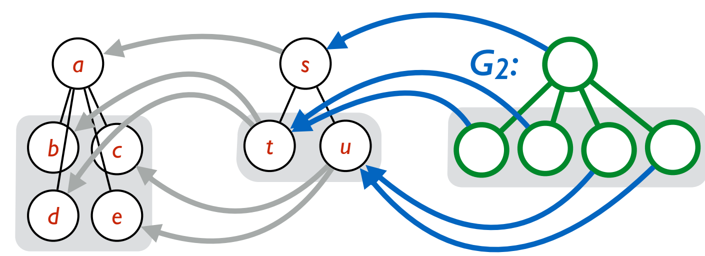


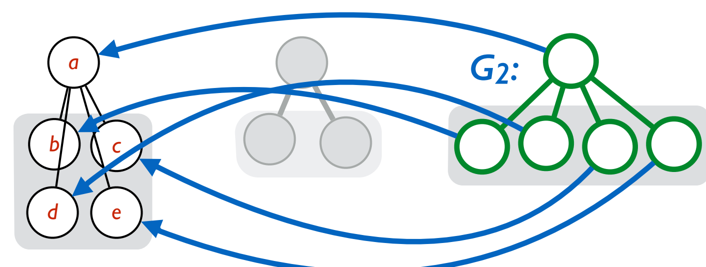
Hierarchy Maps

- 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

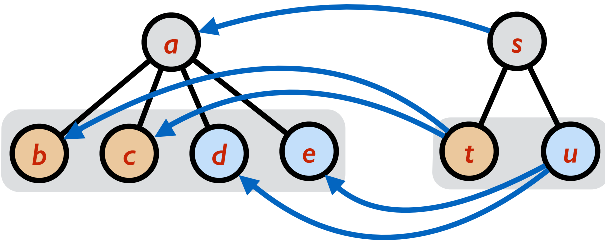


t* is a descendant of *s
***m*(*t*) = {*b*, *d*}, *m*(*s*) = {*a*}**
b* is a descendant of *a ✔
d* is a descendant of *a ✔

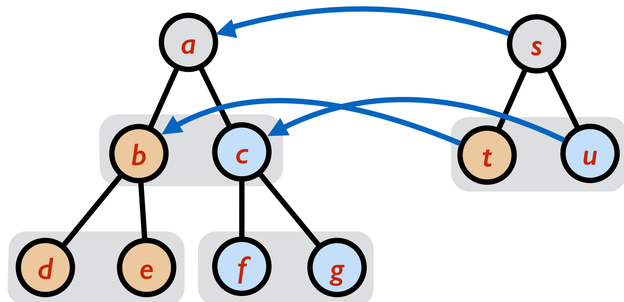




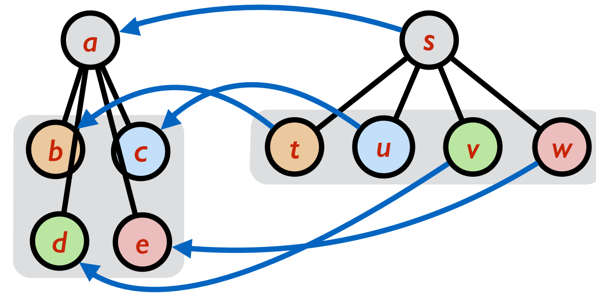
Hierarchy Map Examples



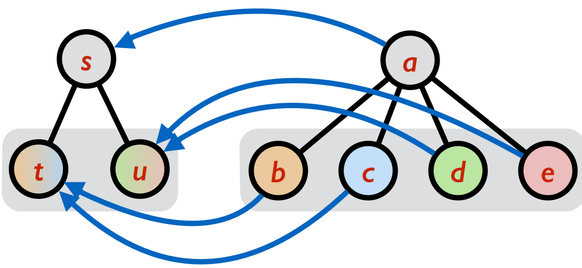
coarse to fine



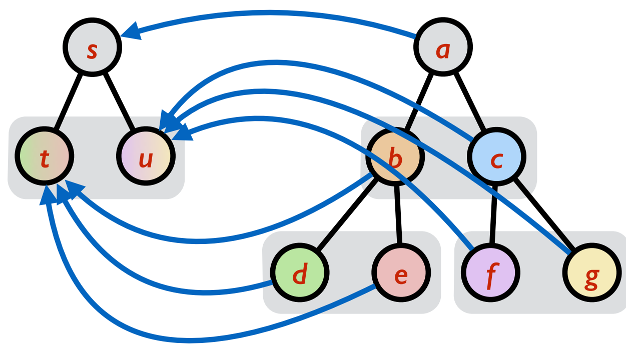
shallow to deep



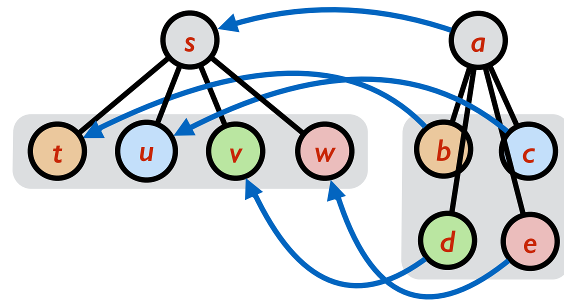
low to high rank



fine to coarse

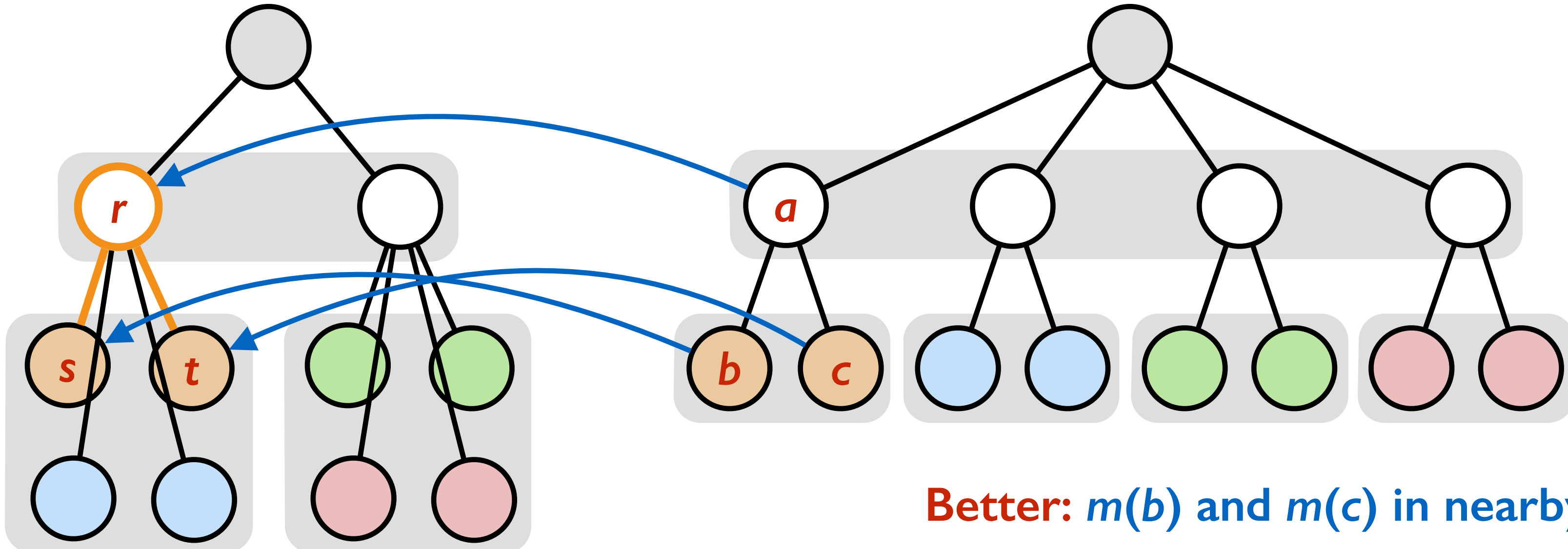


deep to shallow

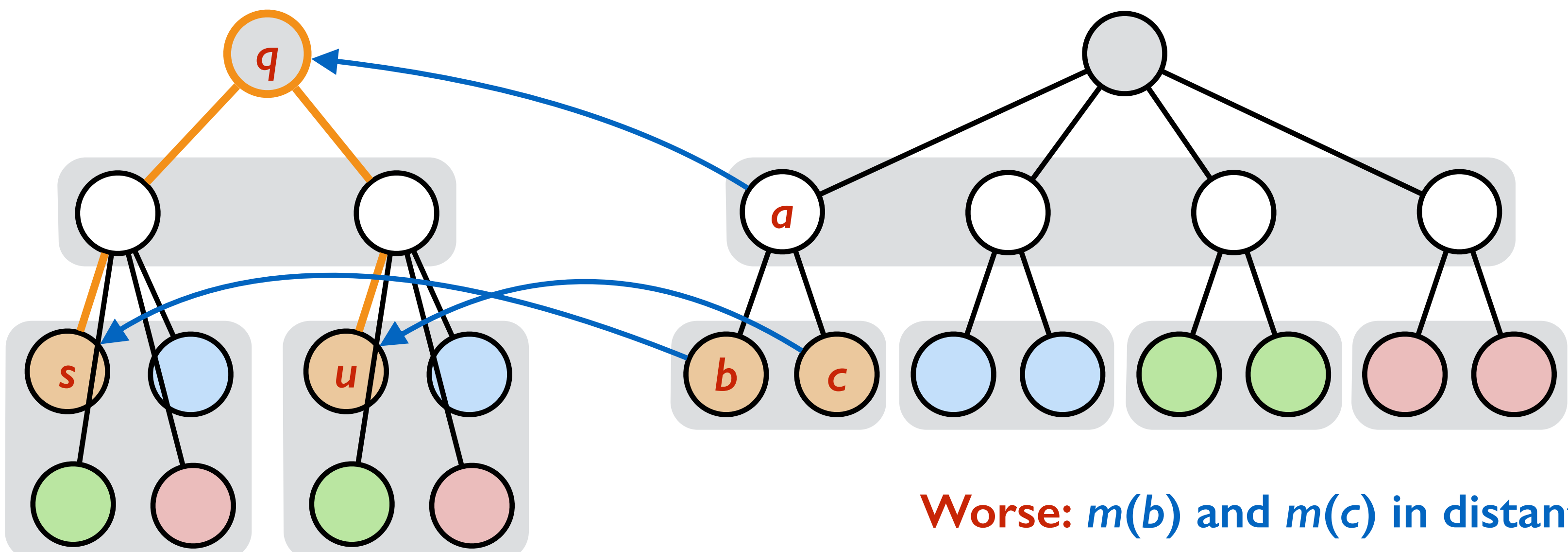


high to low rank

Goodness of Hierarchy Maps



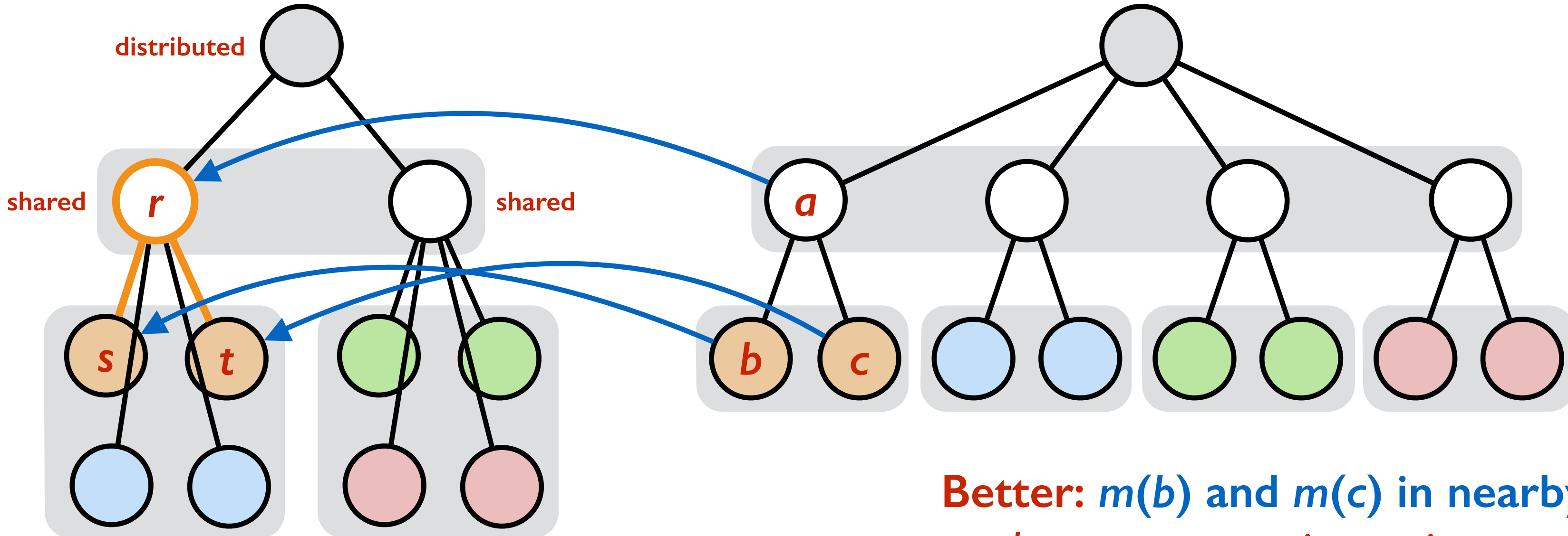
Better: $m(b)$ and $m(c)$ in nearby locale r



Worse: $m(b)$ and $m(c)$ in distant locale q

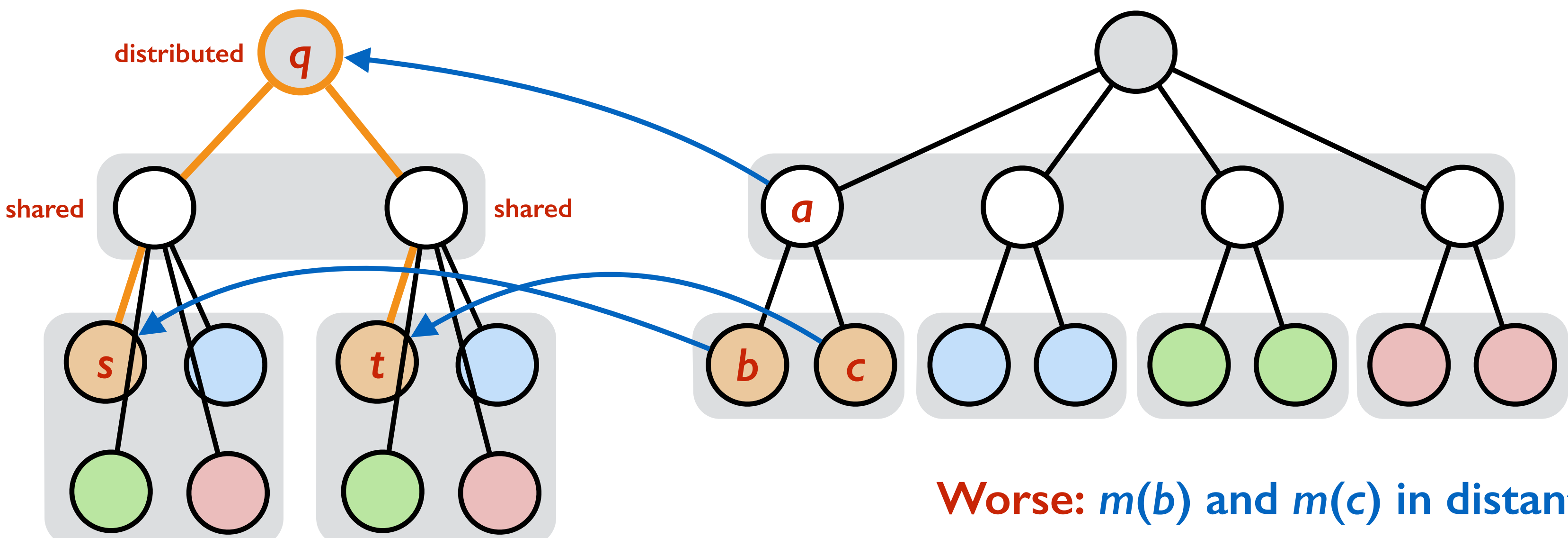
Goodness of Hierarchy Maps

(hierarchical team → hardware)



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

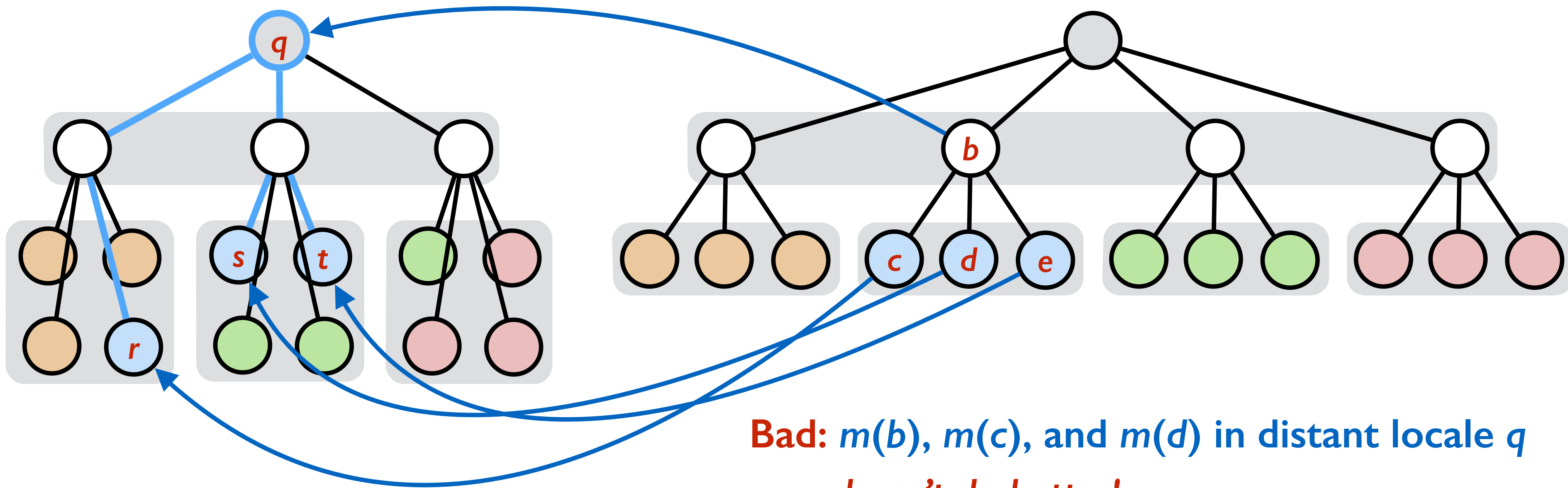
⇒ $b \leftrightarrow c$ communicate via **memory access**



Worse: $m(b)$ and $m(c)$ in distant locale q

⇒ $b \leftrightarrow c$ communicate via **messaging**

Goodness of Hierarchy Maps



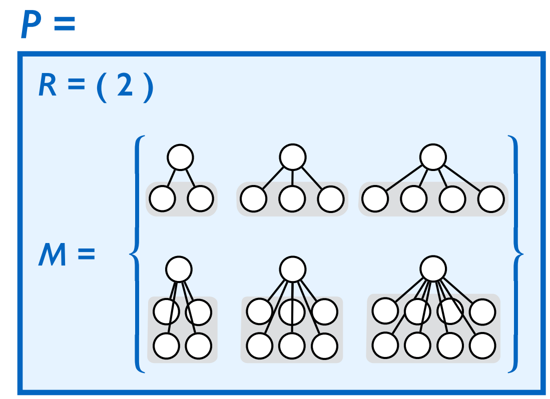
Bad: *m*(*b*), *m*(*c*), and *m*(*d*) in distant locale *q*

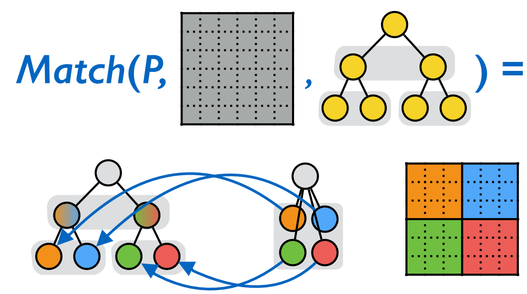
…and can't do better!

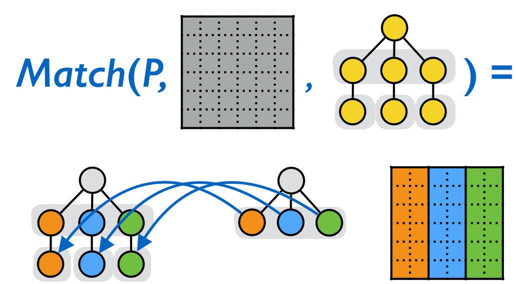
- Best mapping between a given pair of hierarchies may not be great
  - How serious this is depends on the situation
  - E.g. the map above may be fine if all target locales are shared-memory
- For best results: ***choose a source hierarchy that maps well to target***
- HCAF's answer for this is ***tiling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

Tiling Patterns

- A ***tiling pattern*** is a pair $P = (R, M)$ where
  - $R = (k_0, k_2, \ldots\ k_{d\text{-}1})$ 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 $Match : (P,\ \mathcal{D}_k\,,\ H_T) \mapsto (H_O, m)$ where
  - $P = (\ (k_0, k_2, \ldots\ k_{d\text{-}1}),\ \ M\ )$ 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***
    - $Domain(\ \mathcal{T}(r)\ ) = \mathcal{D}_k$, 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





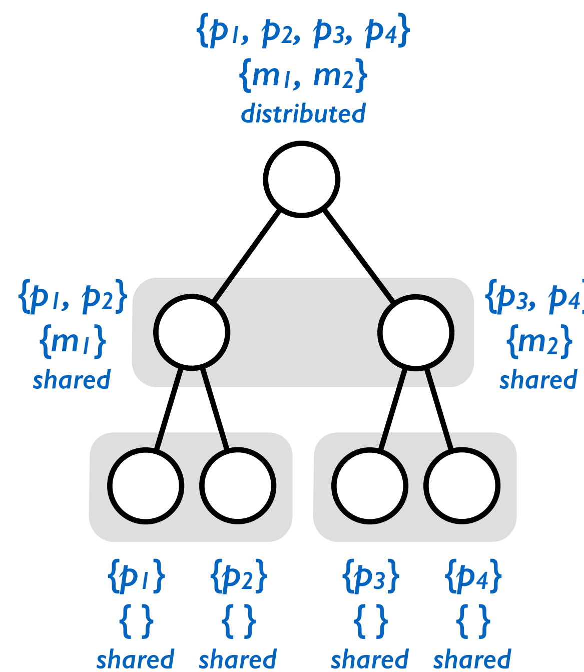


Outline

- Overview
- HCAF Hierarchy Model
- **Hierarchical Abstractions**
  - *Locales:* machine topology
  - *Teams:* processor groups
  - *Coarrays:* data objects
- Language Constructs

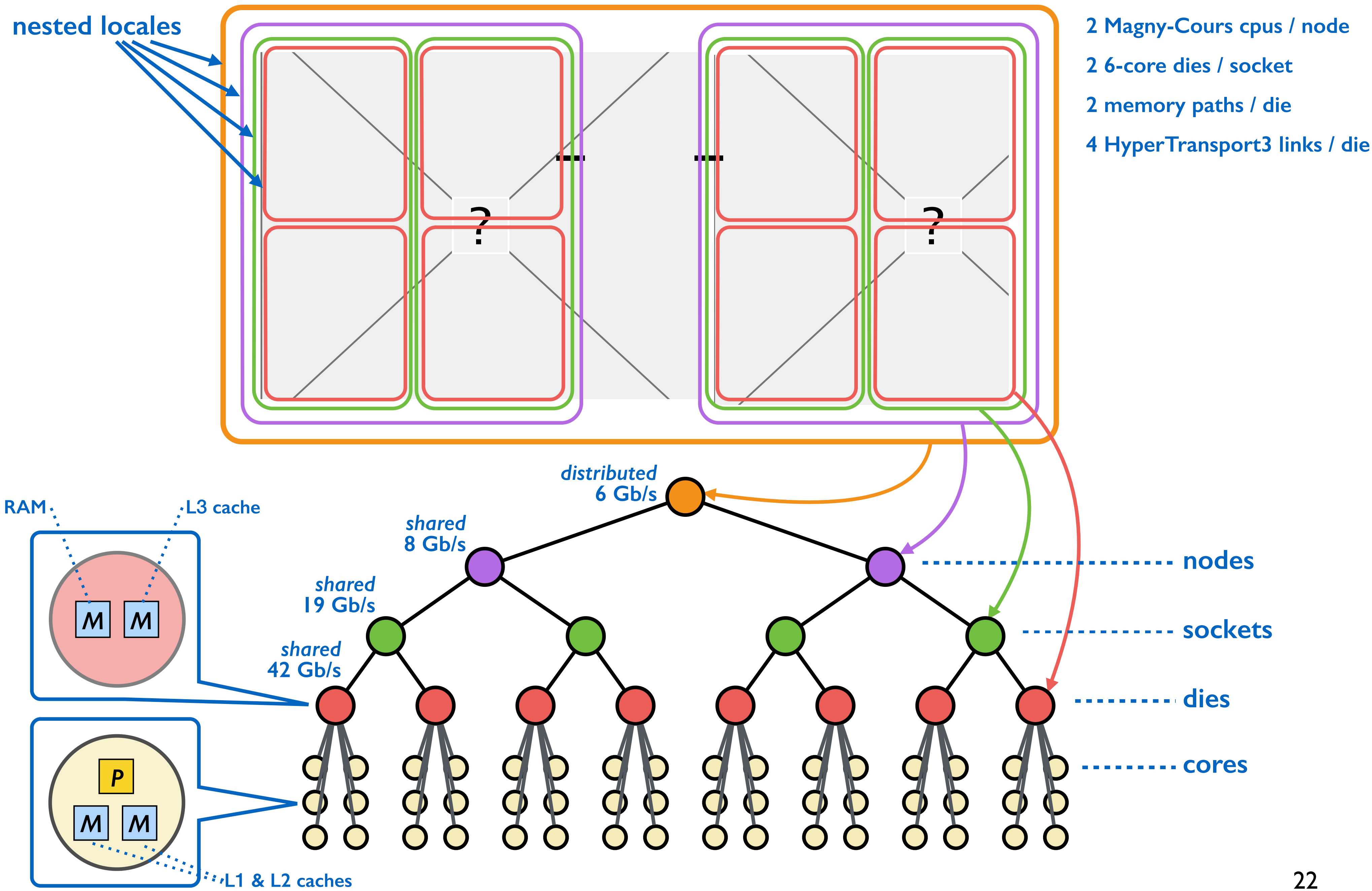
Locales: Hierarchical Machine Topology

- *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 = {*Procs, Mems, 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)
    - *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)
    - *Mems*(e) = $\{m_1, m_2, \ldots\}$ is the set of memories contained in locale e
  - ***Comm*** : $V \rightarrow$ {***distributed, shared***} is the ***communication attribute function***
    - *distributed* and *shared* denote respectively communication via message passing and memory reference
    - *Comm*(e) is the worst-case communication kind among elements of e
    - Require that no *shared* locale has a *distributed* sub-locale



$P = \{p_1, p_2, p_3, p_4\}$
$M = \{m_1, m_2\}$

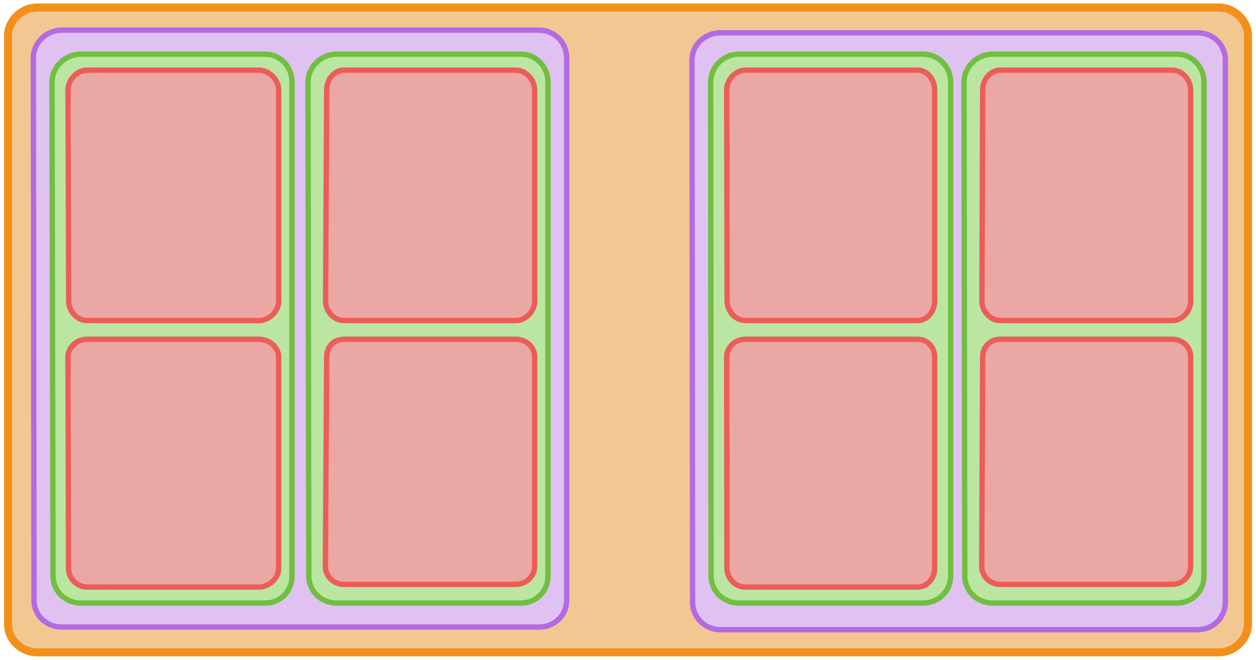
Example Locale: 2 Hopper 24-core Nodes



Locales and Hierarchical PGAS

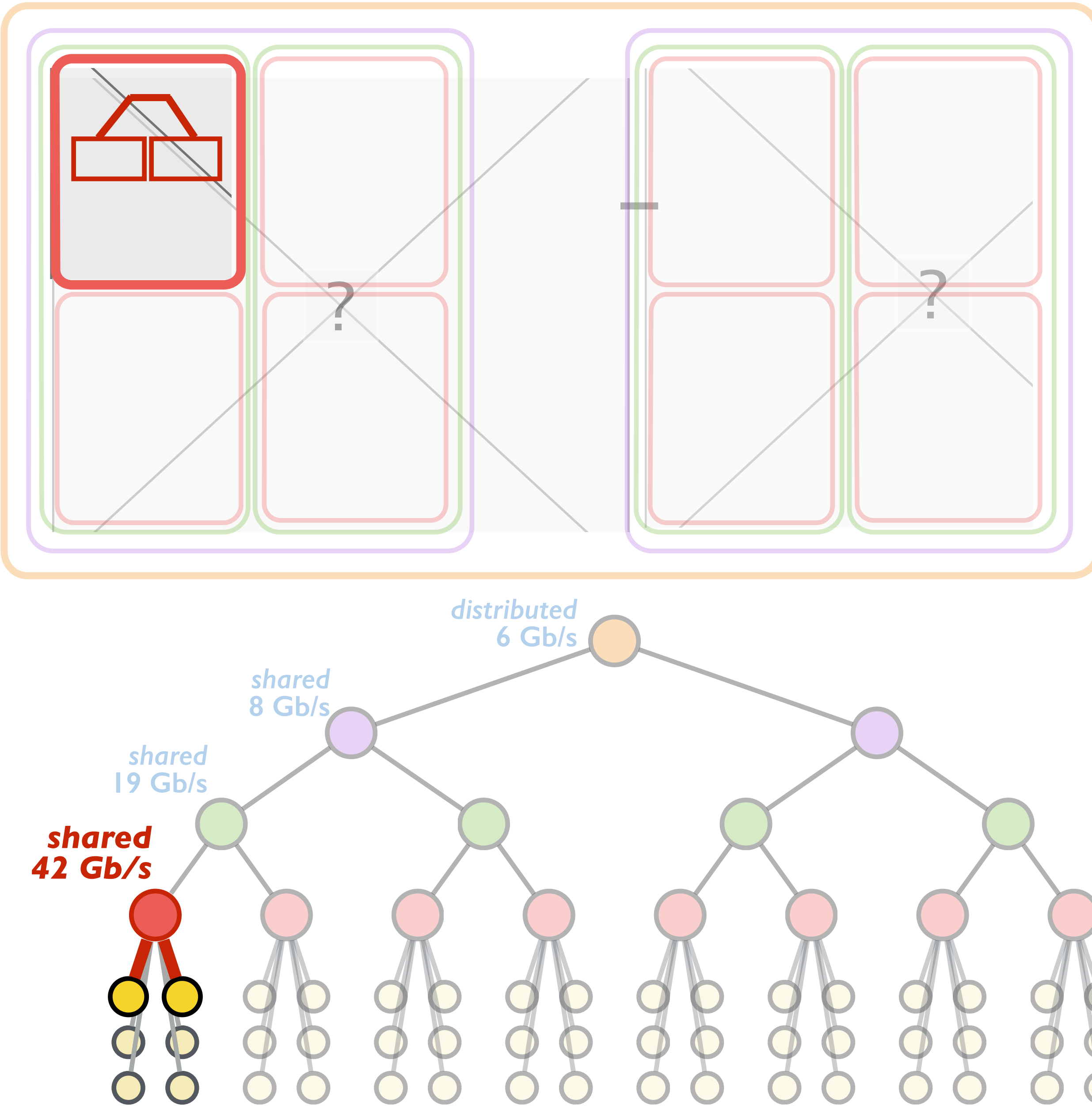
locales = hierarchically partitioned address spaces

smaller locale = closer elements = cheaper communication



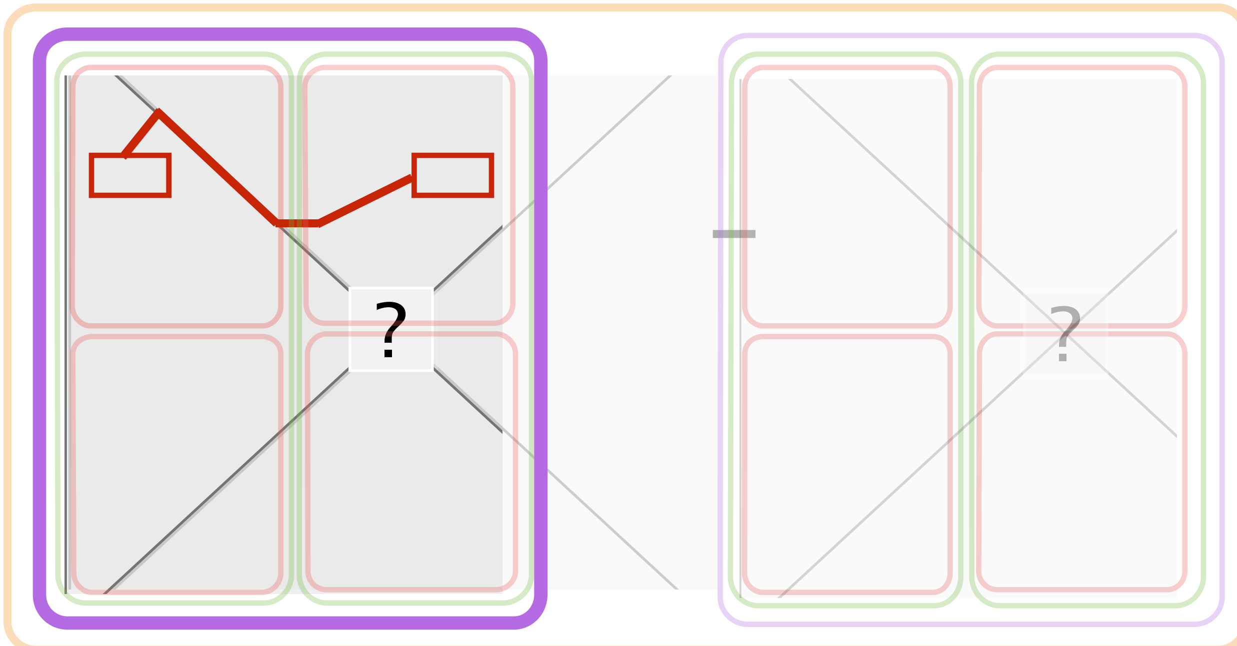
- Any processor can access any address space
- Speed of access is modeled by the *smallest enclosing locale* of a processor and the other processor or memory it accesses
- Equivalently, by the *lowest common ancestor node* in the corresponding Cartesian resource hierarchy

Locales and Hierarchical PGAS

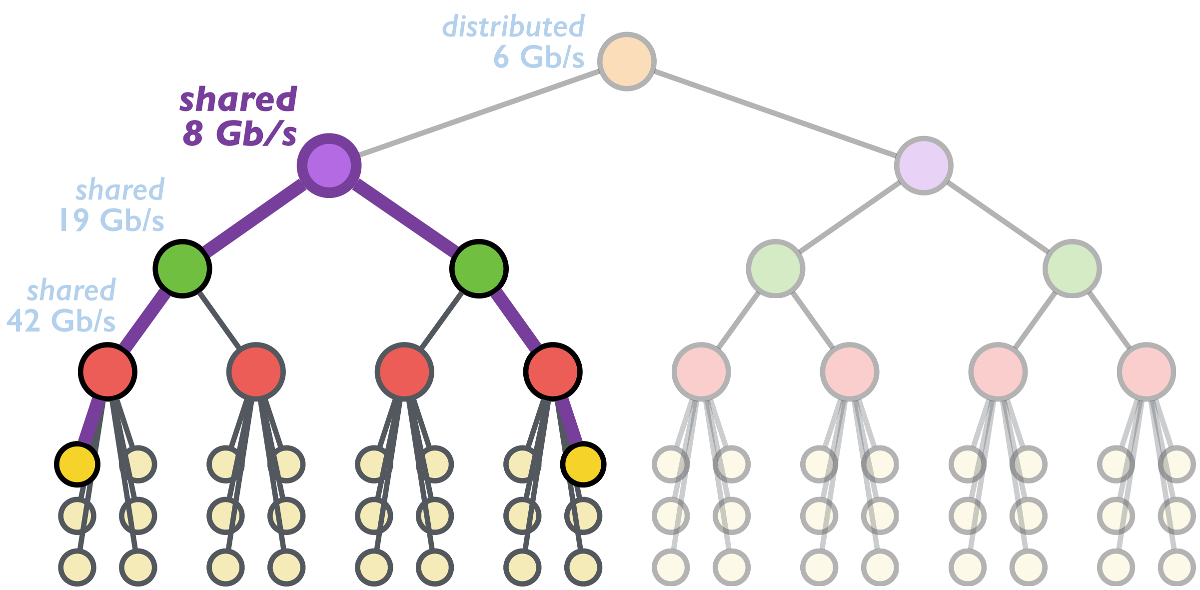


finest partition of address space = innermost locale = one die ⇒ shared-memory comm at 42 Gb/s

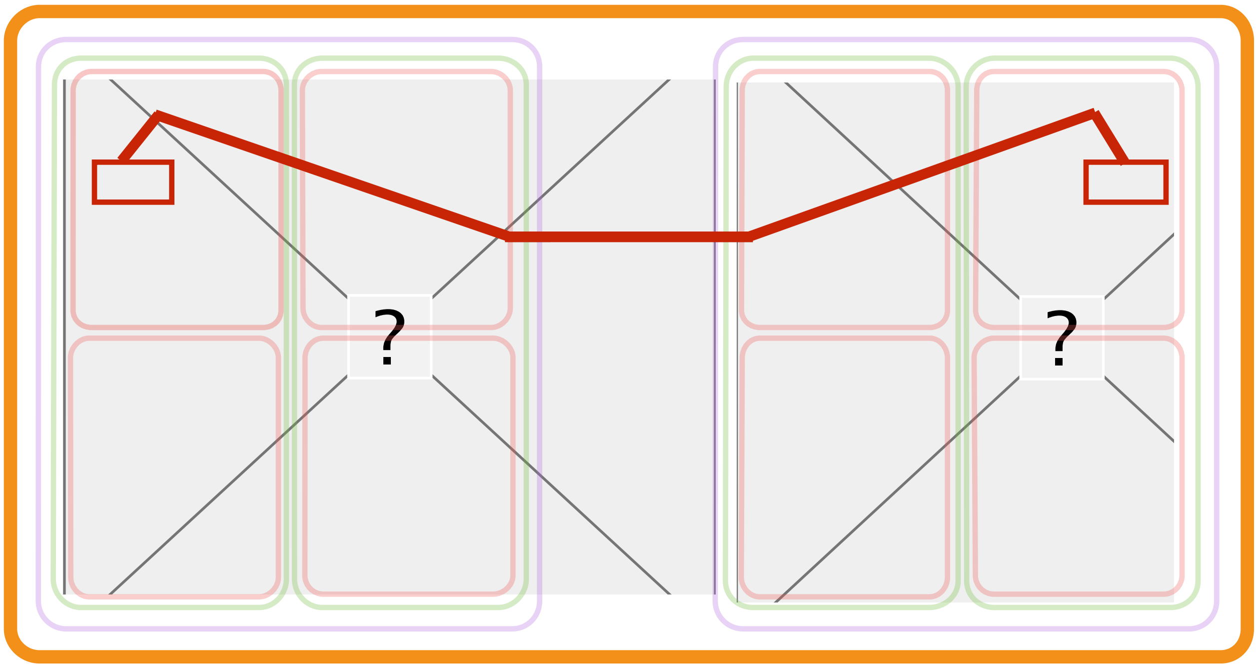
Locales and Hierarchical PGAS



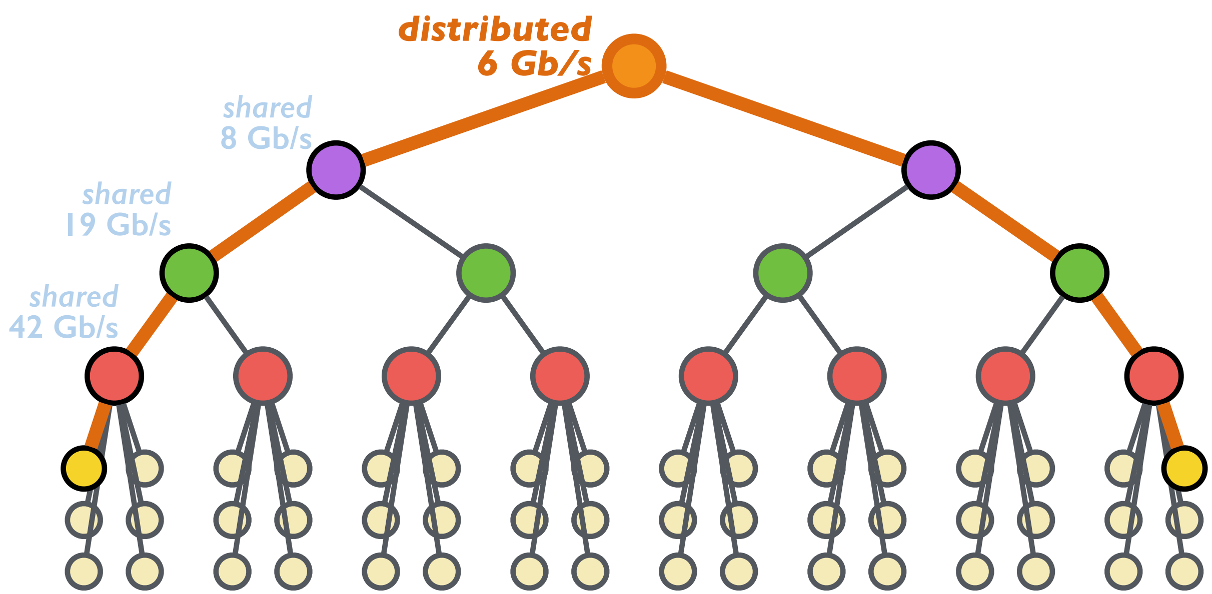
mid-level partition
of address space
=
mid-level locale
=
one node
⇒
shared-memory comm
at 8 Gb/s



Locales and Hierarchical PGAS

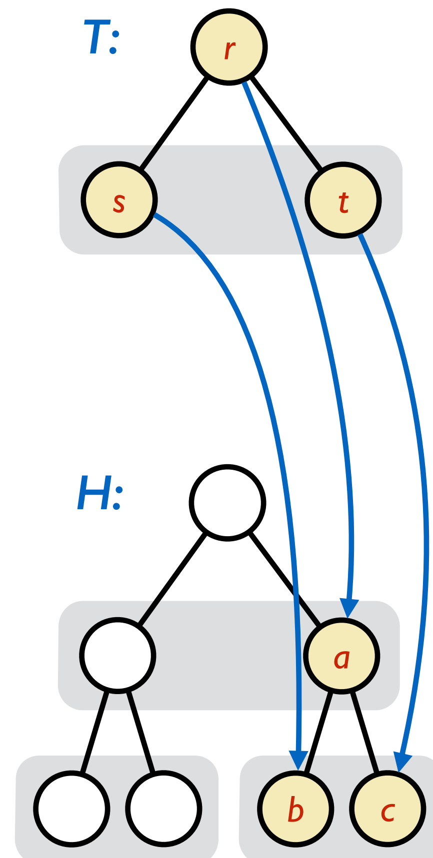


coarsest partition
of address space
=
top-level locale
=
two nodes
⇒
distributed-memory
comm at 6 Gb/s



Teams: Hierarchical Processor Groups

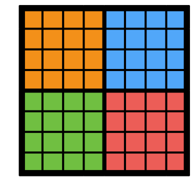
- ***Teams*** are groups of hardware processors (cores)
  - Nested sets of processors which are ***relatively close*** in communication cost
  - Teams ***specify*** sets of processors and ***inherit*** sets of memories
  - Teams serve as ***abstract locales*** to isolate application from hardware details
- 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
  - $A = \{Procs, Mems, Comm\}$ just as for locales
- A team has a hierarchy map $\boldsymbol{m} : V_T \rightarrow \mathcal{P}(V_H)$ 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***
  - ***Procs*(*r*)** is the team's set of processors, possibly a subset of *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 $\forall\ t \in V :\ Mems(t) = Mems(m(t))$
  - These are the memories close to the team's processors
- A team is mapped to hardware by the map ***m***



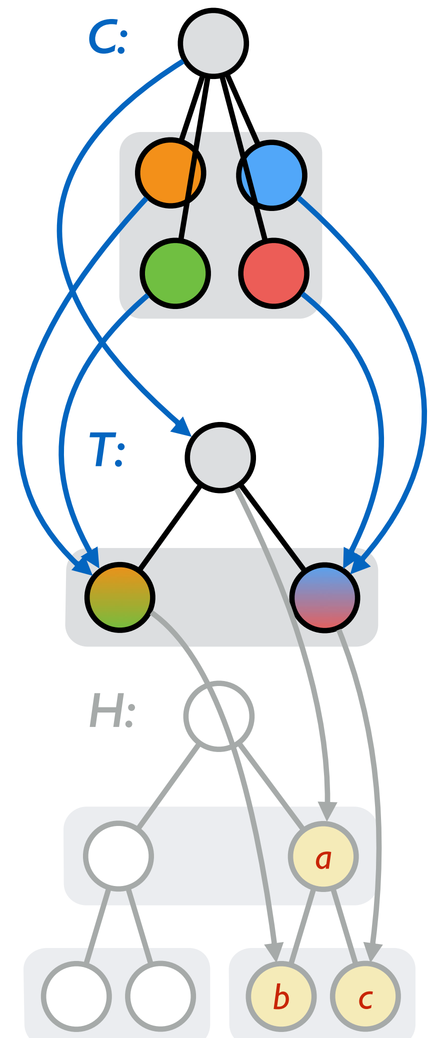
Teams: Locality-aware Parallelism

- 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
  - *Problem:* lexical closures — Habanero/Chapel style {`in`, `out`, `inout`} specifiers?
  - Implementation
    - Berkeley *HotSLAW;* Quintin & Wagner; Olivier & Prins; Saraswat, Paudel et al; etc

Coarrays: Hierarchical Data Objects

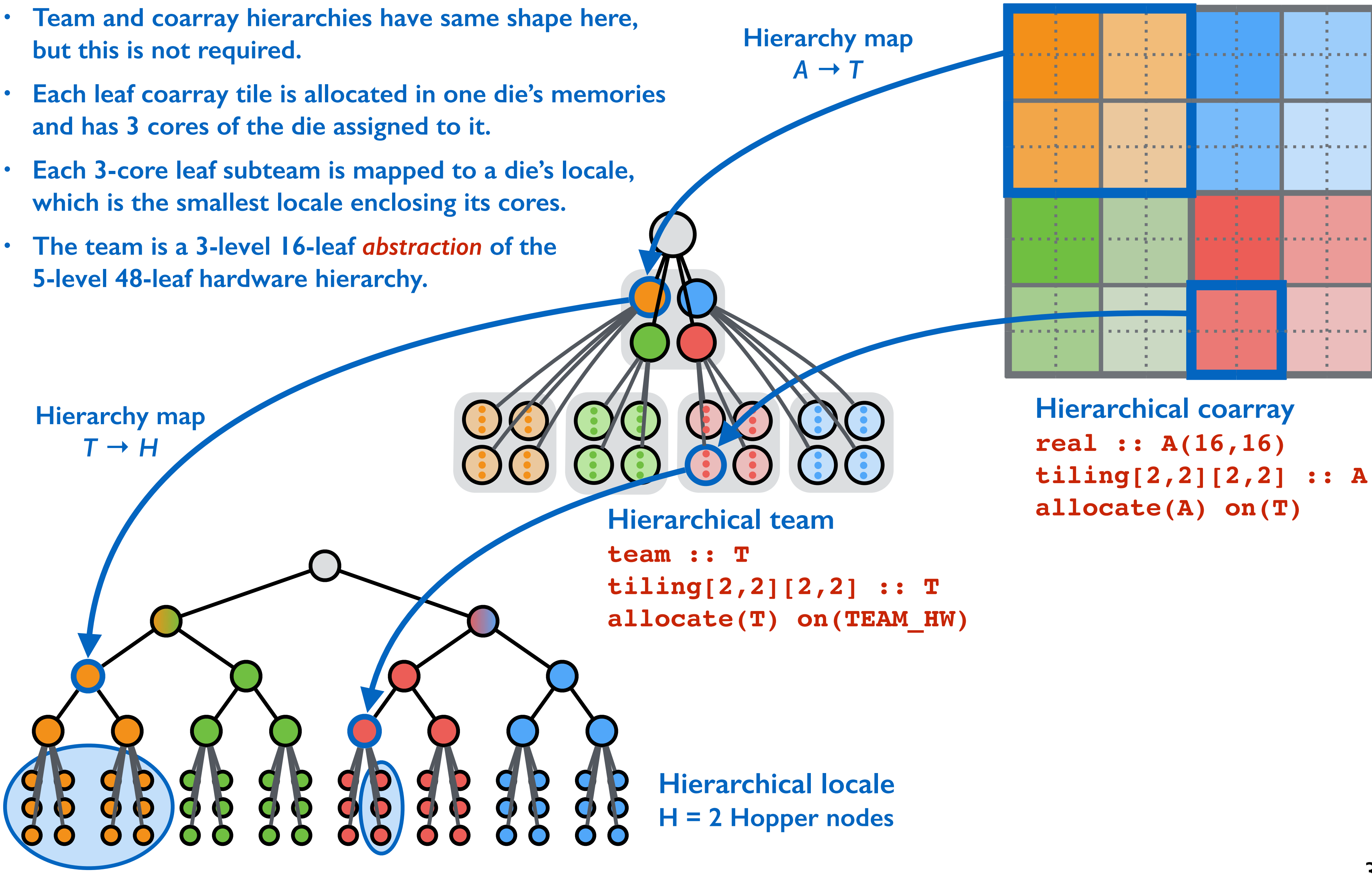


- ***Coarrays*** are tiled groups of storage locations (elements)
  - Nested tiles of elements which are ***relatively close*** in communication cost
  - Coarrays ***specify*** sets of elements and ***inherit*** processors and memories
  - Coarrays are ***allocated on teams*** and their tiles are placed in teams' memories
- A coarray is a tiled resource hierarchy $C = (V, E, \mathcal{K}, A, \mathcal{T})$ where
  - V is the set of sub-tiles and E is the containment relation among them
  - $A = \{Elems, Procs, Mems, Comm\}$ where $Elems \mapsto$ storage locations in each tile
  - $\boldsymbol{Elems(r)}$ is the coarray's top level (global-view) tile and $\boldsymbol{\mathcal{T}(r)}$ is the tile's shape
- A coarray has a hierarchy map $\boldsymbol{m : V_C \to \mathcal{P}(V_T)}$ where
  - T is the team on which C is allocated
  - $\boldsymbol{m(r)}$ is typically the root of the team, where r is the root of C
  - $\boldsymbol{m}$ 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 $\forall\, c \in V_C :\ Procs(c) = Procs(m(c))$ and $Mems(c) = Mems(m(c))$
  - These are the processors owning and memories storing the coarray
- A coarray is mapped to hardware by the composition $C \to T \to H$



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.



Outline

- Overview
- HCAF Hierarchy Model
- Hierarchical Abstractions
- **Language Constructs**
  - **Tiling patterns & generic hierarchy**
  - **Hierarchical teams**
  - **Hierarchical coarrays**
  - **Task, data, and SPMD parallelism**
  - **Example:** *Naive Matrix Multiply*

Tiling Patterns

- Problem:
  - Locality-aware applications and optimizers ***statically depend on hierarchy shape***
  - Hardware hierarchy is ***known only at runtime*** (cf. machine type & job scheduler)
  - Need ***abstraction*** to decouple application's ***virtual hierarchies*** from machine's ***real hierarchy***
  - But manually mapping virtual to real is difficult
- Solution:
  - ***Tiling pattern*** describes a ***set of desirable hierarchies***
  - Compiler ***statically optimizes*** using properties common to all set members
  - Runtime ***dynamically chooses*** desirable hierarchy with a ***good mapping*** to hardware
- Tiling pattern specifies:
  - ***Hierarchy rank*** (first ***d*** levels) and set of ***hierarchy coshapes***
  - Required ***communication kind*** at each level (distributed vs shared memory)
  - Tile ***distributions*** and Rubik-style ***tilts/shifts/etc***
- Example tiling pattern **'P'** with hierarchy rank (2,1):

```
tiling :: P(N)
  [ N block, N cyclic(100) ]
  [ 2..32 ] shared
end tiling
```

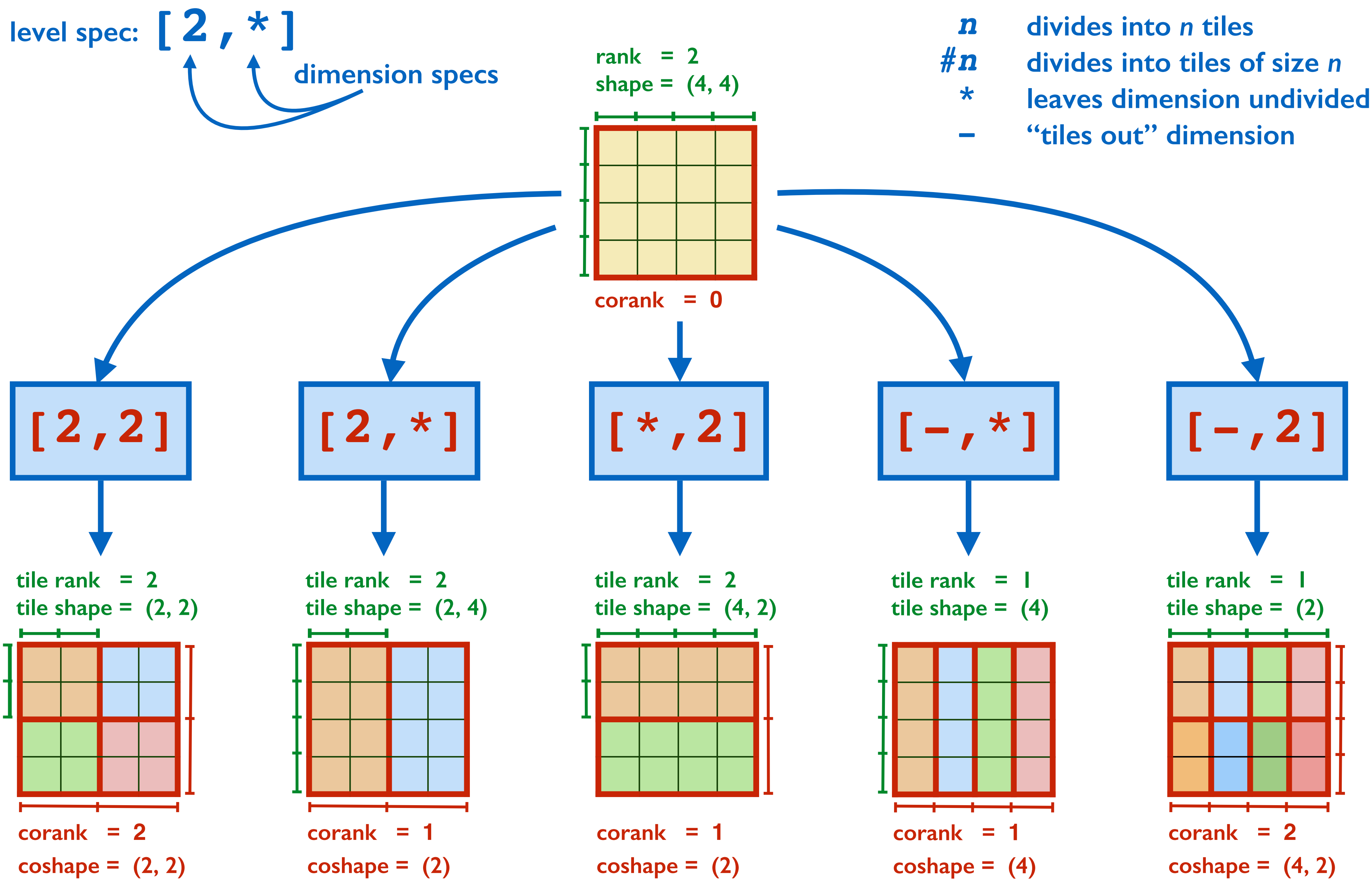
parameter

level specification

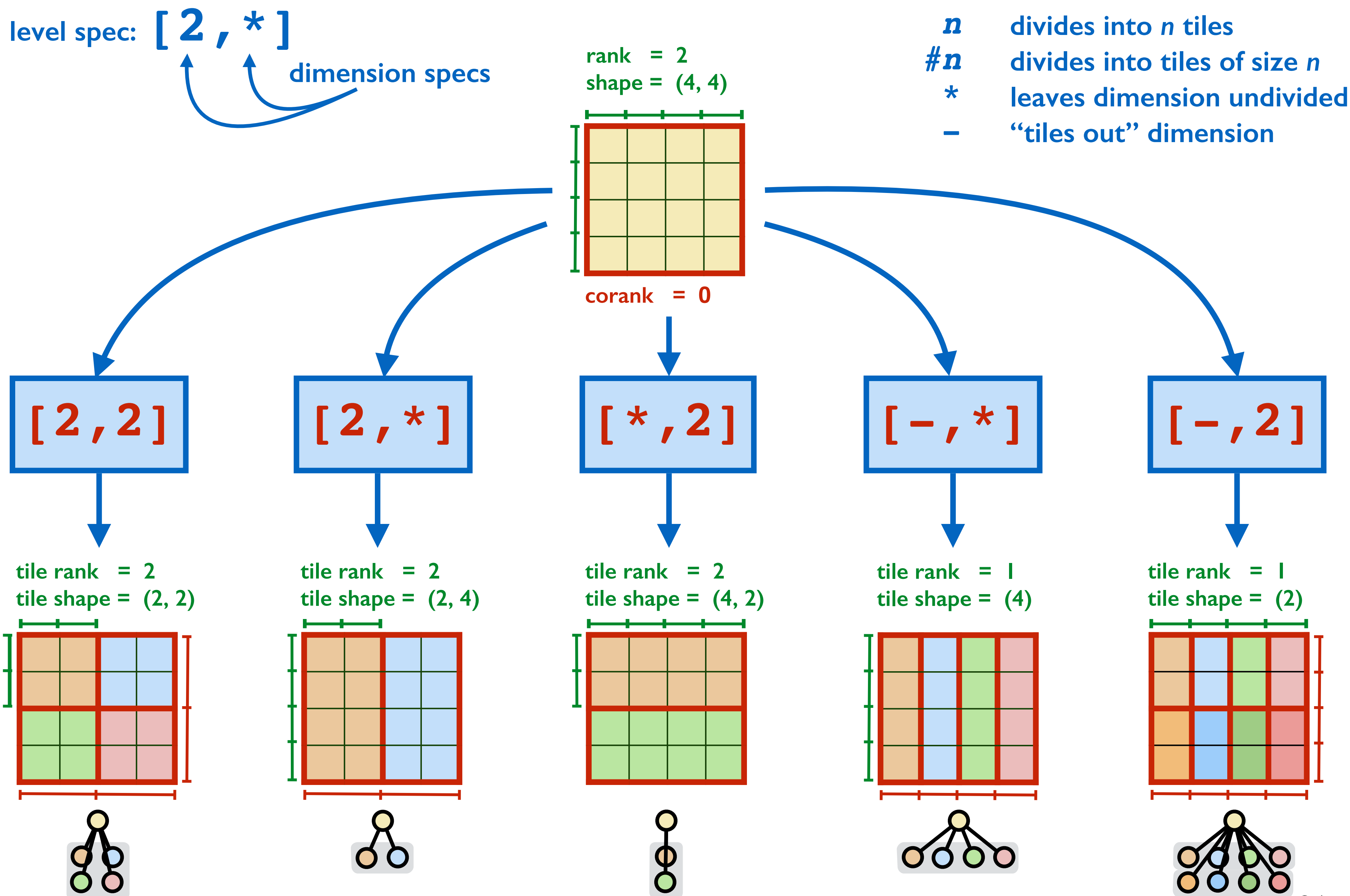
distribution

comm kind

Tiling Patterns: Level & Dimension Specs



Tiling Patterns: Level & Dimension Specs



Tiling Patterns: Parameters & 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 $M : H' \to H$
    - Result is (H', M)
- ***Implicit parameter*** ≡ 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 ≡ 1, empty upper bound ≡ *any*

  `0 : 7..15` ⇒ $8 \le n \le 16$ elements indexed from 0

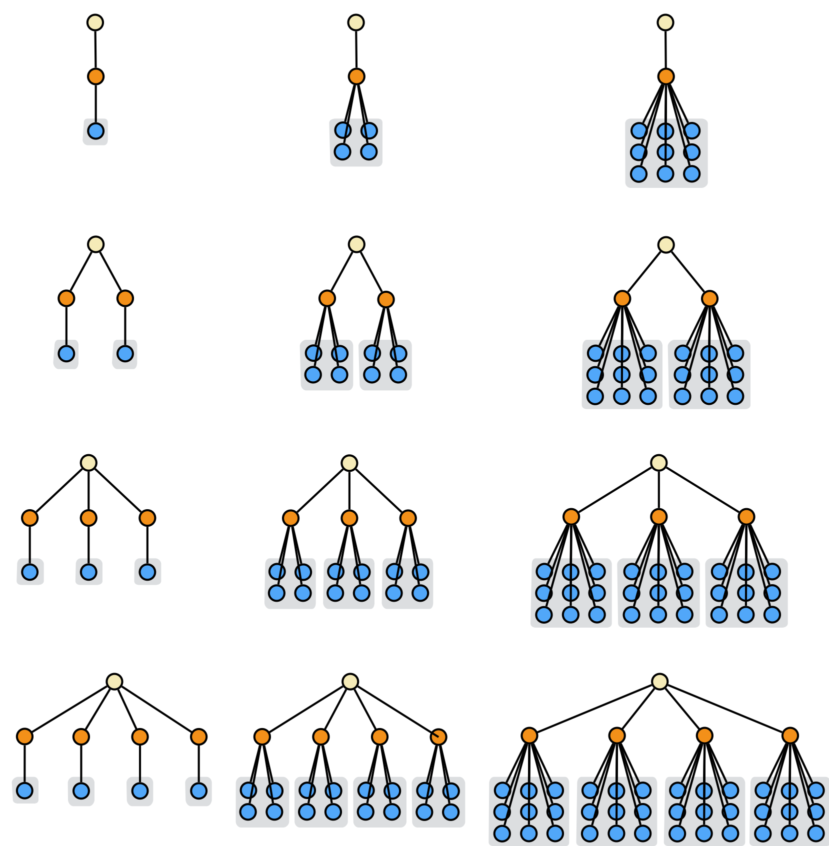
  `:` ⇒ $n > 0$ elements indexed from 1

```
tiling :: P( N )
   [ 1..4 ]
   [ N, N ]
where
   N <= 3
end tiling
```

explicit parameter

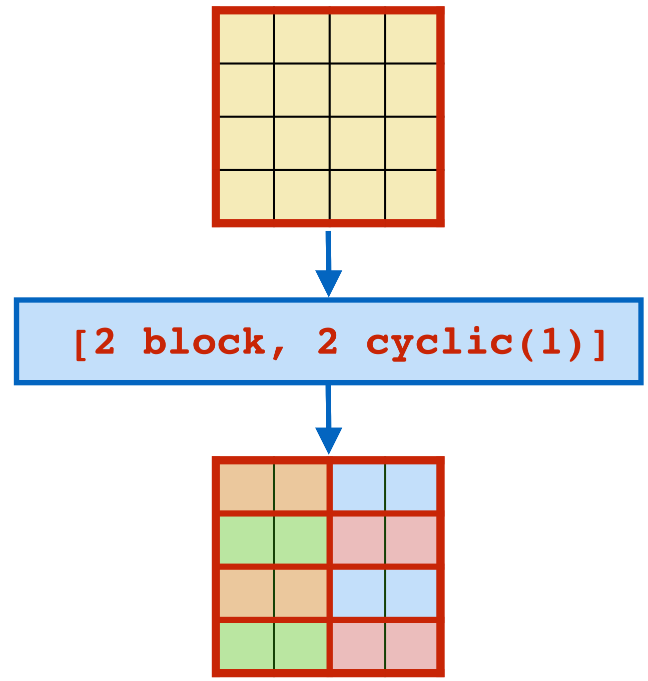
implicit parameter & constraint

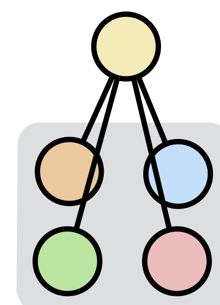
constraint



Tiling Patterns: Distribution & Comm Specs

- ***Distribution specifier*** modifies dimension-spec
  - Specifies a dimension's assignment of elements to tiles
    i.e partially specifies $\mathcal{T}(c)$ 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:
    - **`distributed`** message passing
    - **`shared`** memory access
    - **`image`** SPMD program instance (shared)
    - **`any`** unspecified (the default)





Generic Hierarchy Operations

- *H* may be a *locale*, a *team*, or a *coarray* (some operations require hierarchy be *regular*)
  - Shape and size
    - `codepth(H)` — number of tiling levels (0 ⇒ leaf)
    - `corank(H)`, `corank(H,k)` — number of tile dimensions at top or specified tiling level
    - `coshape(H)`, `coshape(H,k)` — tuple of tile extents at top or specified tiling level
    - `cosize(H)`, `cosize(H,k)` — total number of tiles at top or specified tiling level
    - `rank(H)`, `rank(H,k)` — number of element dimensions at top or specified tiling level
    - `shape(H)`, `shape(H,k)` — tuple of element extents at top or specified tiling level
    - `size(H)`, `size(H,k)` — total number of elements at top or specified tiling level
  - Access
    - `H[i,j,…]` — tile access
    - `H[l:u:s,…]` — tile section access
    - `H(i,j,…)` — element access
    - `H(l:u:s,…)` — element section access
  - Locality
    - `locale(H)` — opaque id of hardware locale to which `H` maps
    - `locale_info(id)` — description of hardware locale identified by `id`
  - Mapping
    - `map_hierarchy(H,T)` — new hierarchy by tiling `H` with tiling pattern `T`
      `map_hierarchy(H,H2)` — new hierarchy by tiling `H` with tiling pattern of `H2`

Constructs: Hierarchical Teams

- A *team* is a cartesian hierarchy of *processors* (not SPMD instances)
  - Team's processors cooperate to execute tasks spawned dynamically on the team
  - Team's processors communicate and synchronize via *collectives* as in CAF 2.0
  - Teams are characterized by the (*worst case*) kind of communication available between processors
    - *Distributed-memory team*: communication by message passing
    - *Shared-memory team*: communication by memory access
    - *Image team*: communication by global variables — a shared-memory team within an execution of the SPMD program
    - Team characterization is determined by locale to which it is mapped
- Team variable declarations
  - Recall that HCAF hierarchy types include a partial characterization of the hierarchy (i.e. of top ***d*** levels)
  - So HCAF's type `team` is parametrized by a tiling rank:

`team, tiling[:,:]    :: t1`	tiling rank is (2)
`team, tiling[:,:][:] :: t2`	tiling rank is (2,1)
`team, tiling(T)      :: t3`	tiling rank is T's rank

  - Default tiling rank is "any":

`team :: t`	rank is ( )

  - Team sub-typing by tiling rank subsumption:

`t  = t1`	✔ since ( ) is a prefix of (2)
`t1 = t2`	✔ since (2) is a prefix of (2,1)
`t2 = t1`	✘ since (2,1) **is not** a prefix of (2)

  - Allows static type checking of team variable uses

Constructs: Hierarchical Teams (2)

- Team construction
  - Predefined team values:

    `TEAM_HARDWARE` precisely describes hierarchy of current job's machine partition

    `TEAM_WORLD` as in CAF 2.0 (all processors, partitioned into image teams)

    `TEAM_DEFAULT` as in CAF 2.0
  - By splitting with a tiling pattern:

    `t3 = tile_map(TEAM_HARDWARE, T)`
  - By CAF 2.0's notion of team splitting?
    - *Don't know how to make this work hierarchically*
- Team usage
  - Allocate a coarray on a team
  - Perform collectives on a team
  - Team-oriented control structures:
    - `with team t` as in CAF 2.0
    - `with subteam t` our version of Titanium `team_split` statement
    - `select subteam t` our version of Titanium `partition` statement

      ```
       case <stmt> …
      end select
      ```

Data Parallelism: Parallel Loops

- Explicit data parallelism via *loops*
  - Iterating over *element* indices of a coarray:

```
do parallel( i, j in A )
  <statement list>
end
```

  - Iterating over *tile* indices of a coarray:

```
do parallel( tile i, j in A )
  <statement list>
end
```

  - Iterating over part of a coarray:

```
do parallel( tile j in A[k,:] )
  <statement list>
end
```

  - Loop indices can be omitted ⇒ rank-independent data parallelism

```
do parallel( tile in A )
  <statement list>
end
```

- Locality via hierarchy mapping:
  - Iterations of body are all spawned at once into implicit `finish`
  - Each iteration is spawned on the subteam owning the indexed element or tile

Task Parallelism: Async & Finish

- Two forms of `async`, analogous to Fortran's two forms of `if`
  - `async(t) <statement>`
  - ```
    async(t)
      <statement-list>
    end async
    ```
  - Can supply data reference instead of team ⇒ spawns on team owning the data
- Two forms of `finish`
  - `finish <statement>`
  - ```
    finish
      <statement-list>
    end finish
    ```
- Additional *event* argument signals completion
  - `async(team = t, event = e) <statement>`
  - ```
    async(team = t, event = e)
      <statement list>
    end async
    ```
    ```

What about SPMD Parallelism?

- What are the essential differences between task and SPMD parallelism?
  - **Number of "program images"**
    - Number of instantiations of libraries, number of copies of global variables
    - This matters because *program scope* is special!
  - **Degree of parallelism present at startup**
    - Task parallel => 1 control thread; SMPD parallel => P control threads
    - This matters because of data parallel loops:
      - are they *fork/joins* ?
      - or are they *collectives* ?
- Can SPMD be modeled as implicit an *initial fork* and a *final join* ?
- Can a data-parallel loop on a *distributed-memory* team have the same semantics as a data-parallel loop on a *shared-memory* team?
  - if not, how can we have one programming model with uniform semantics throughout the machine hierarchy?
- See following example for more insight

Example: Naive Matrix Multiply

```
program main

  tiling :: T(m1, m2)
    [m1, m1]
    [m2, m2] shared
  end

  real, dimension(:,:), tiling(T) :: A, B, C
  integer, parameter :: n = 1000

  allocate( A(n,n), B(n,n), C(n,n) ) on(TEAM_HARDWARE)
  ! initialize A and B somehow …
  C = 0

  call matmul(A, B, C)

end program
```

Example: Naive Matrix Multiply cont'd

```
subroutine matmul(A, B, C)
  real :: A(:,:)[#], B(:,:)[#], C(:,:)[#]
  integer :: n

  select tiling( A )
    case [:,:]
      n = cosize(A, 1)
      do parallel( tile i, j in C )
        do k = 1, n
          call matmul(A[i,k], B[k,j], C)
        end do
      end do
    case []
      n = size(A, 1)
      do parallel( i, j in C )
        do k = 1, n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
        end do
      end do
 end select

end subroutine
```