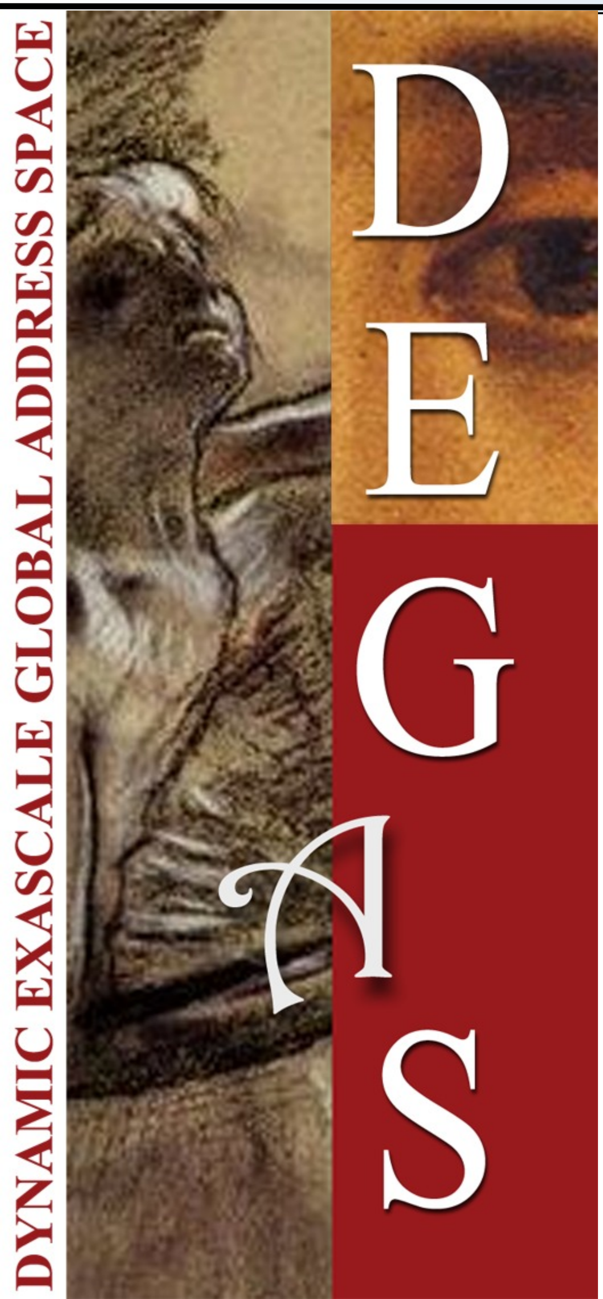


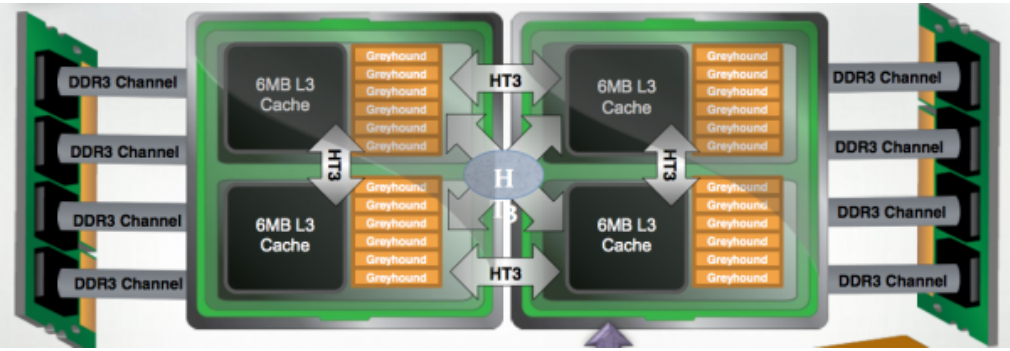
Towards a Portable Model for Mapping Locality to Hierarchical Machines

Amir Kamil and Katherine Yelick
Lawrence Berkeley Lab
Berkeley, CA, USA

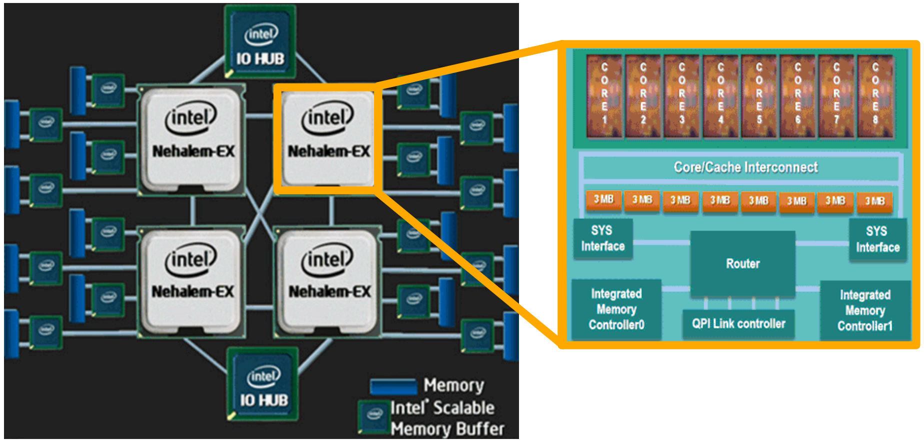
June 24, 2015

Hierarchical Machines

- Parallel machines have hierarchical structure



Dual Socket AMD MagnyCours

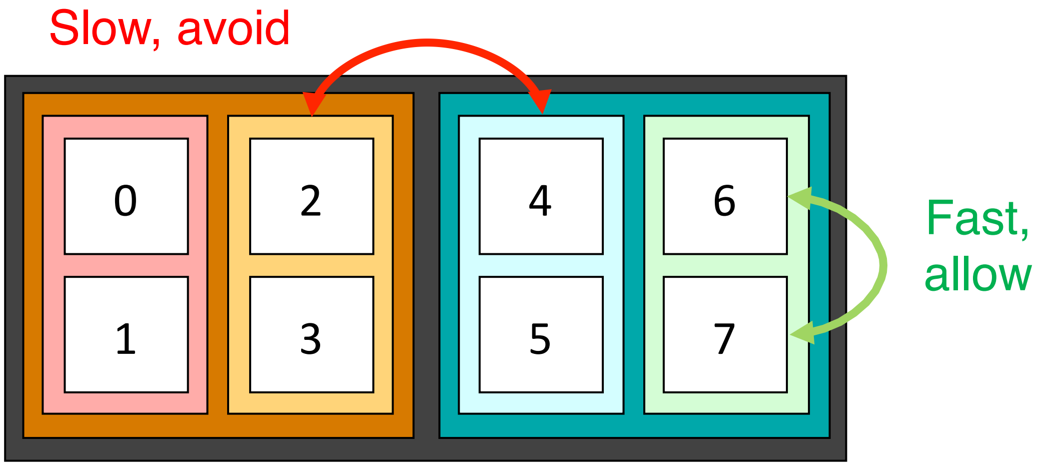


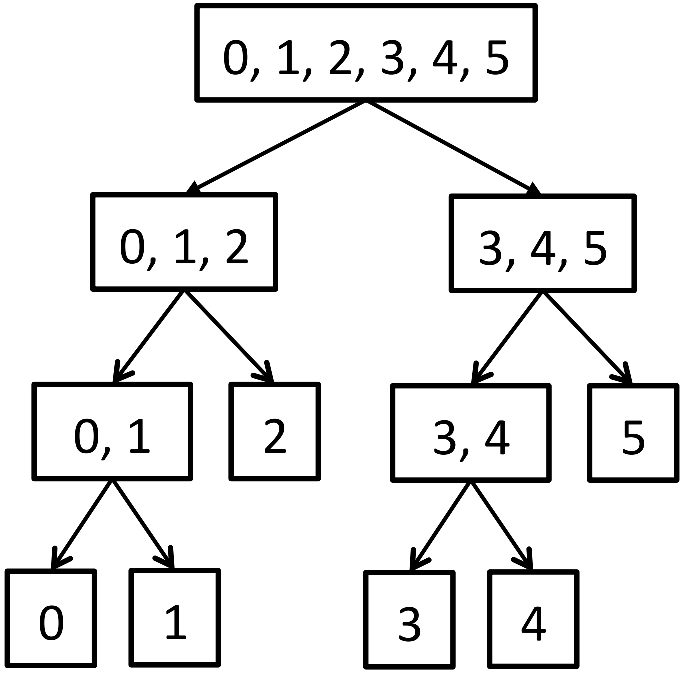
Quad Socket Intel Nehalem EX

- Expect this hierarchical trend to continue with manycore

Application Hierarchy

- Applications can reduce communication costs by adapting to machine hierarchy

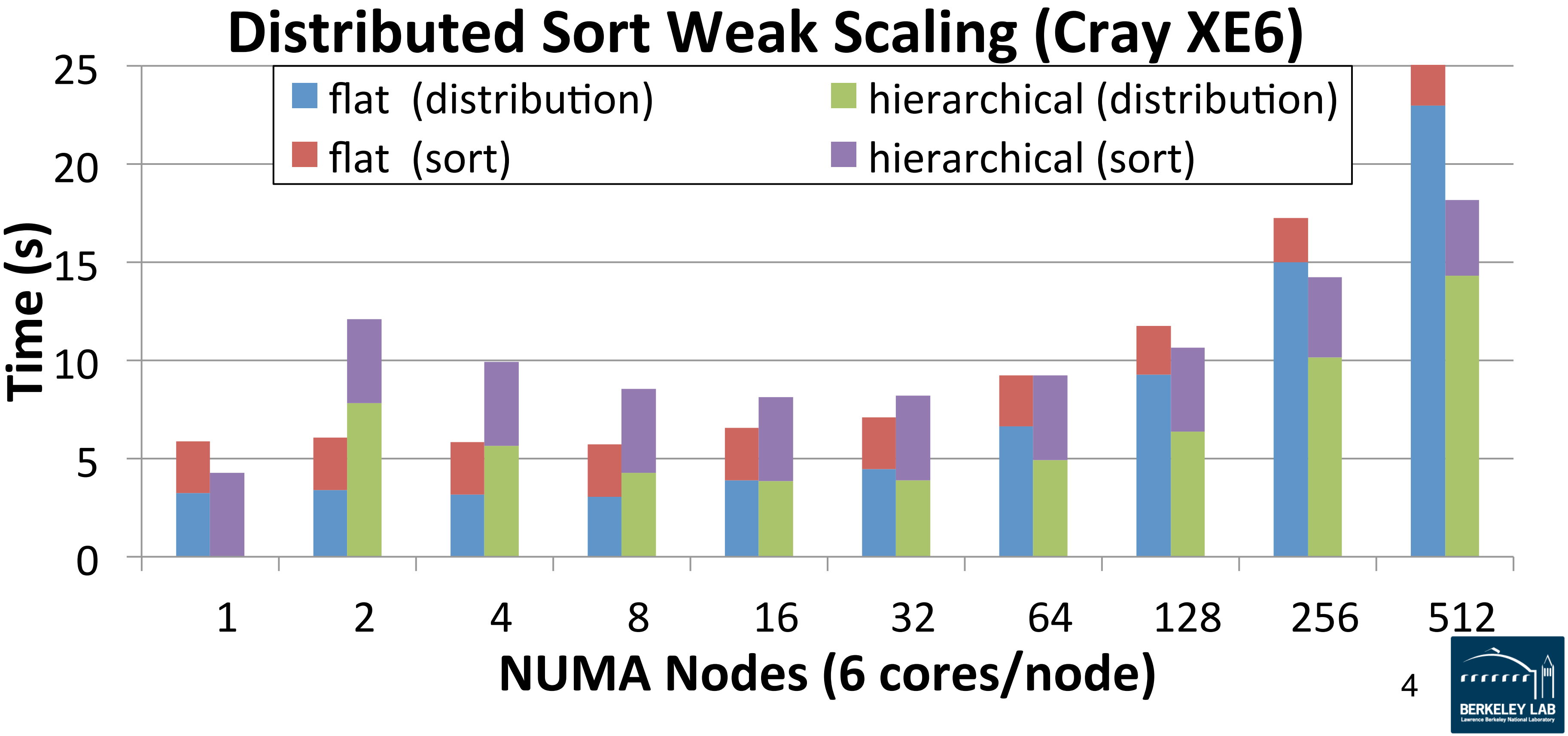




- Applications may also have inherent, algorithmic hierarchy
  - Recursive algorithms
  - Composition of multiple algorithms
  - Hierarchical division of data

Example: Hierarchical Sort in Titanium

- Hierarchical sort adapts to machine hierarchy by using sample sort between shared-memory domains
- Within a shared-memory domain, it runs divide-and-conquer merge sort



Hierarchy Mapping

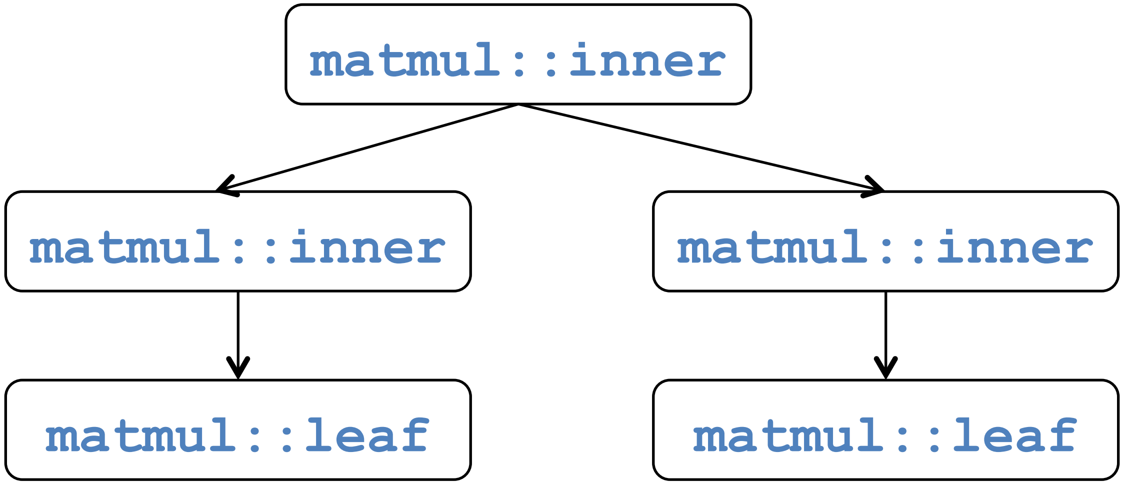
- Program's view of hierarchy must be mapped onto the actual hierarchy of a machine in a portable manner
- Ideal features of mapping facility:
  - Mapping should only affect performance, not correctness
  - Changing the mapping should require few if any changes to source code
    - E.g. Chapel's domain maps
  - High-level default mappers should be provided
    - E.g. Divide into fast-communication domains
  - Users should be able to write their own mappers
    - E.g. Map a binary tree onto the machine
  - Changing the mapping should be sufficient to port code to a new machine

Overview

- Goal is to design a hierarchy model in UPC++ that makes it easy to express and map application-level hierarchy onto a machine
- We survey some existing approaches to see what we can learn
  - Existing models include Sequoia, Legion, Titanium, Hierarchical Place Trees, and HCAF
  - Approach must be applicable to UPC++'s *SPMD* +*Async* model of execution
- We present a high-level strawman proposal for hierarchy in UPC++

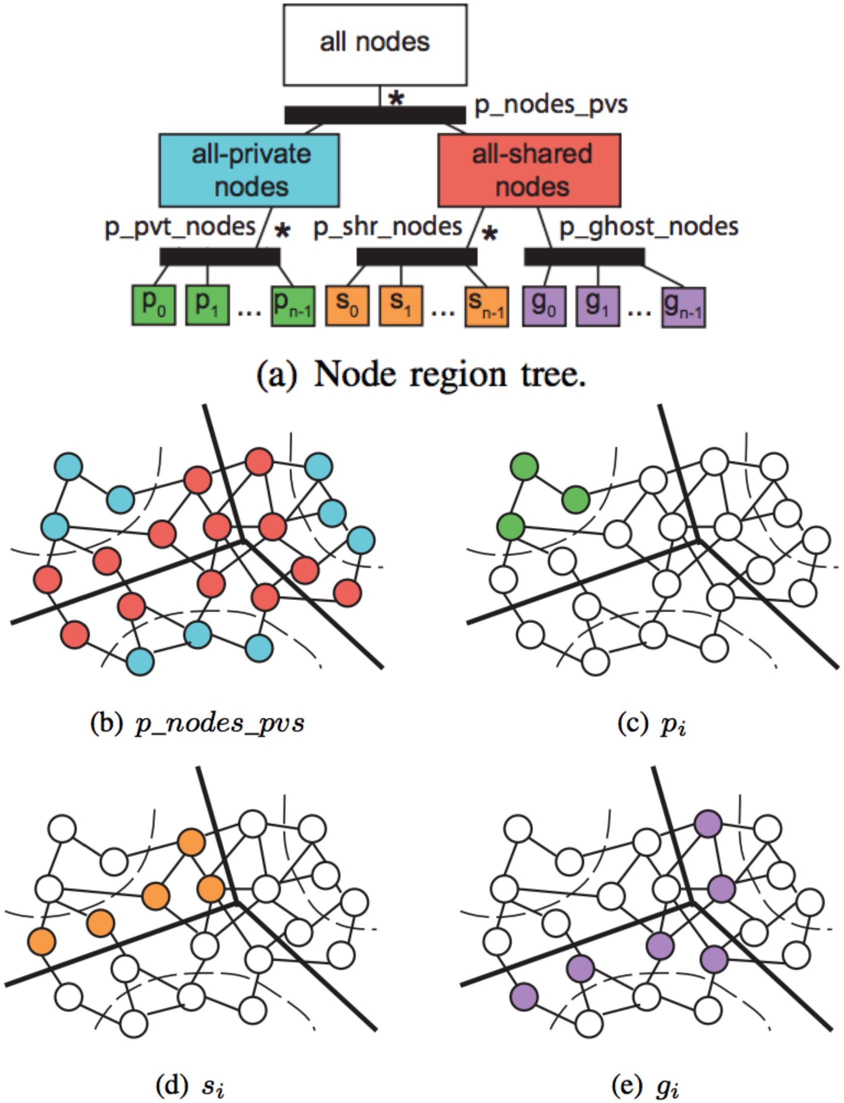
Sequoia Model

- Programmer specifies *inner* tasks and *leaf* tasks
  - Inner tasks decompose computation into smaller pieces
  - Leaf tasks perform actual computation
  - Communication restricted to arguments, return values
- A *machine file* describes the structure of a particular machine
- A *mapping file* maps a task hierarchy onto a machine
  - Also determines depth, width of hierarchy and task parameters



Legion Model

- Legion based on division of data into memory *regions* and execution into *tasks*
  - Tasks declare the regions they access and required access properties
  - Subtasks' regions and access properties must be subset of parents'
- A *mapper* maps regions and tasks onto machine at runtime
  - Simple default mapper provided
  - API provided to allow custom mappers to be written



(a) Node region tree.

(b) *p_nodes_pvs*

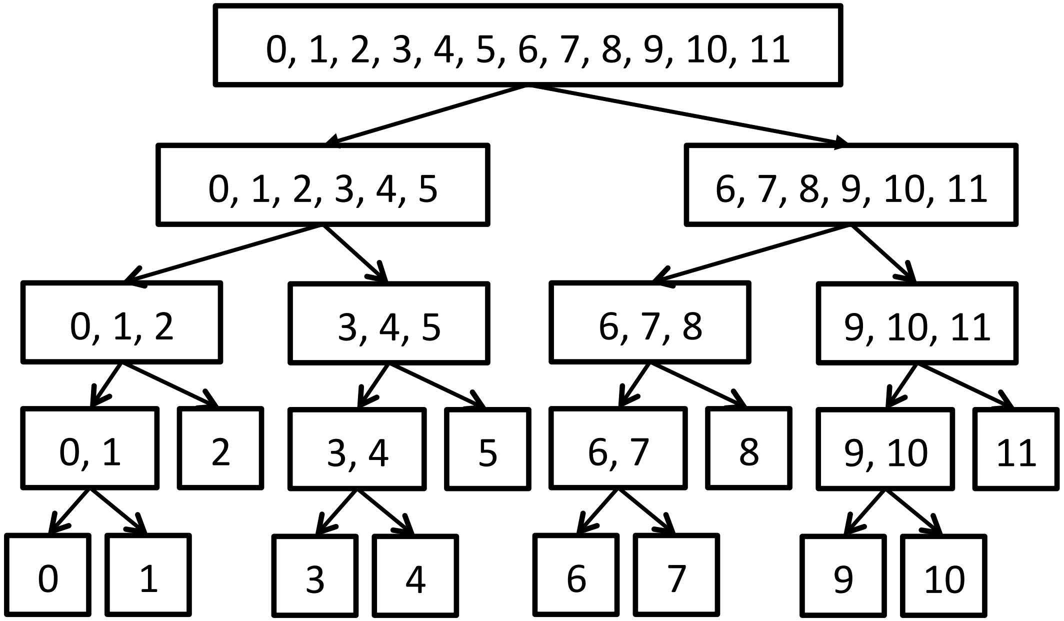
(c) p_i

(d) s_i

(e) g_i

Titanium Model

- Hierarchical *teams* of cooperating threads
- Application determines appropriate hierarchy and explicitly maps data and execution accordingly
  - Runtime provides a machine-based hierarchy for reference
- Dynamically scoped language constructs for executing on teams

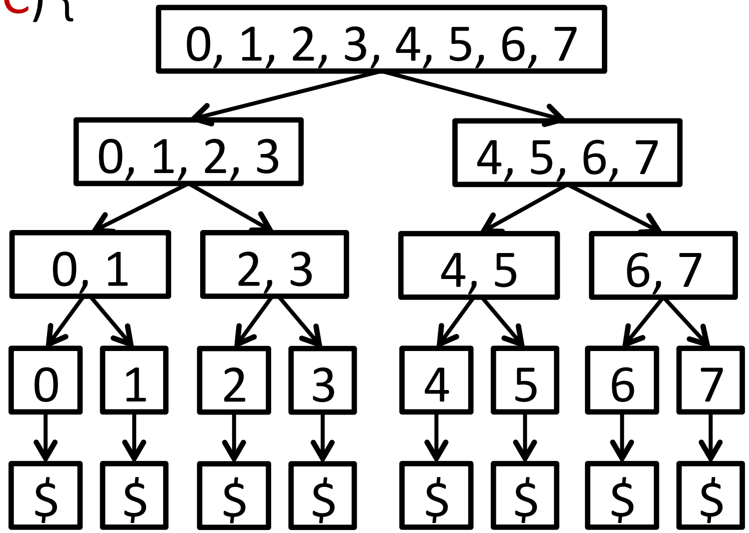


```
team t = Ti.defaultTeam();
teamsplit(t) {
 sampleAndDistribute(data);
 team t2 =
   binaryTree(Ti.currentTeam());
 teamsplit(t2) {
   mergeSort(data);
 }
}
```

HPT Model

- Hierarchical place trees (HPT) model hierarchy of resources
  - Places can have memory units, execution units, or both
- An *execution configuration* specifies the structure of a particular machine
- Application maps data, execution onto configuration

```
void MatMul(double[.] A, double[.] B, double[.] C) {
  if (here.isLeafPlace()) {
   for (point [i, j, k] : [myA, myB, myC])
    C[i,j] += A[i,k] * B[k,j];
  } else {
   dist d = here.getCartesianView(2);
   finish ateach (point p : d)
    MatMul(block(A, d)|p, block(B, d, 0)|p,
                block(C, d, 1)|p);
  }
}
```



0, 1, 2, 3, 4, 5, 6, 7

0, 1, 2, 3

4, 5, 6, 7

0, 1

2, 3

4, 5

6, 7

0

1

2

3

4

5

6

7

$

$

$

$

$

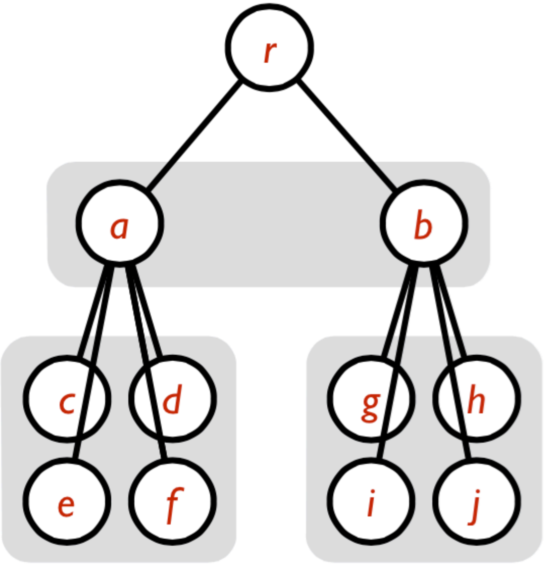
$

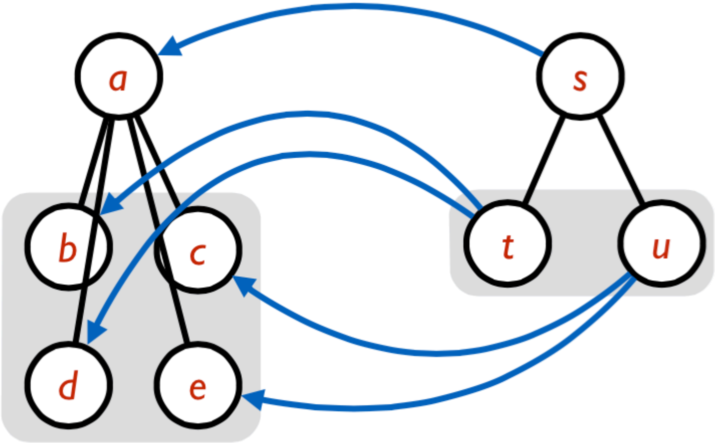
$

$

Proposed HCAF Model

- Hierarchy in HCAF based on *Cartesian resource hierarchies*
  - Tree with Cartesian topology at each level
- Application statically expresses hierarchy using Cartesian extension of hierarchical teams
- HCAF compiler models machine using Cartesian extension of HPTs
- Goal is to map application hierarchy onto machine hierarchy using compiler analysis





Strawman Proposal for Hierarchy in UPC++

- Hierarchical place tree (HPT) represents machine

```
hpt h = get_full_hpt();
```

  - Structure can be specified at program startup, modified at runtime, or divided into subsets of machine

- Mapper maps a user-level structure onto an HPT

```
mapper m1 = fast_comm_mapper();
mapper m2 = k_ary_tree_mapper(2);
```

- Hierarchical team represents user's view of execution and is mapped to an HPT

```
team t1(h, m1); // fast-communication domains
team t2(h, m2); // binary tree
```

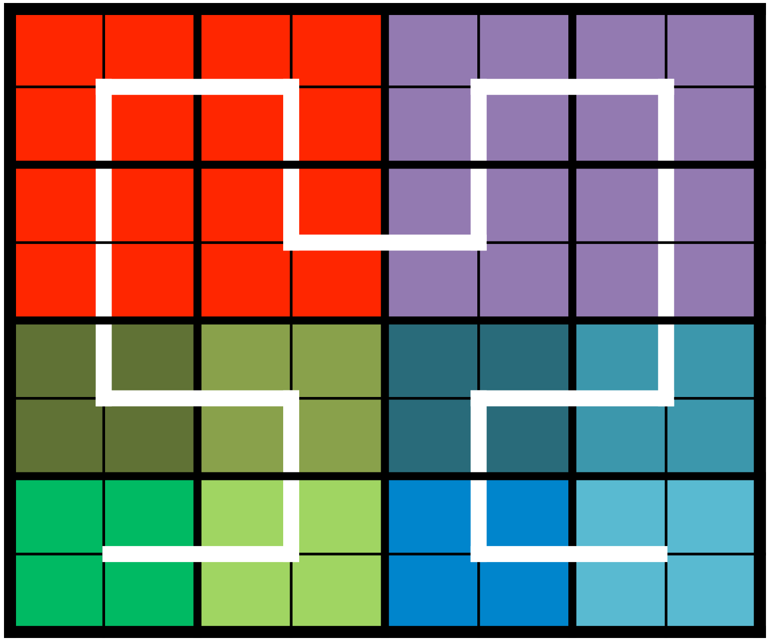
- Data structures map to HPT or team using multidimensional mappers

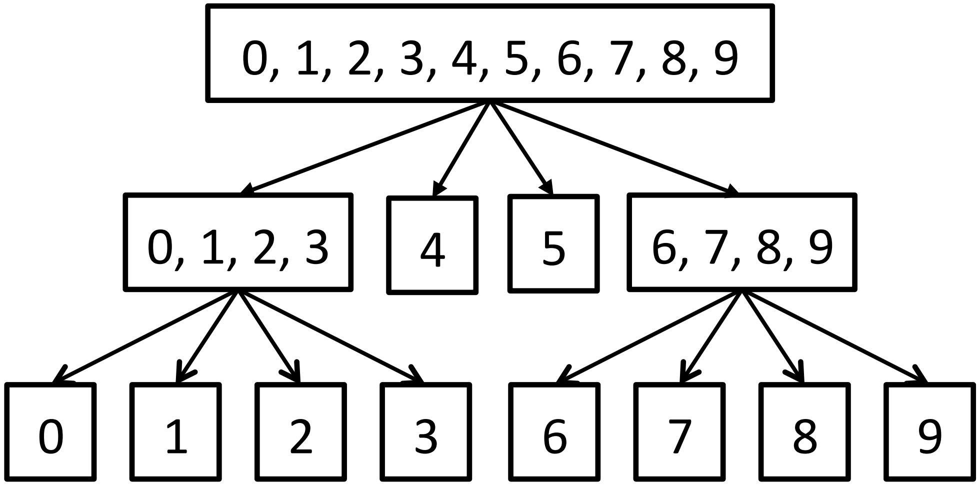
Example: Hierarchically Tiled Array

- An HTA is created over a rectangular index space, a hierarchy of tile sizes, an HPT or team, and a mapper

```
hta<T, N> array(RD(PT(0, 0), PT(8, 8)), tiling, hpt, mapper);
```

- Support regular (e.g. block-cyclic, diagonal) and user-defined mappings, as well as space-filling curves





Summary

- A hierarchical programming system must provide an expressive and portable means of mapping the programmer's view of hierarchy onto a machine
- Mapping should be easy to change to tune performance or port to a new machine
- Existing programming systems either impose a restricted programming model or require the user to manually map hierarchy onto the machine
- We are designing a model of hierarchy in UPC++ that incorporates the best ideas from existing systems in order to facilitate hierarchy mapping