# Hierarchical Parallelism in a Partitioned Address Space Model


EECS
Electrical Engineering and Computer Sciences

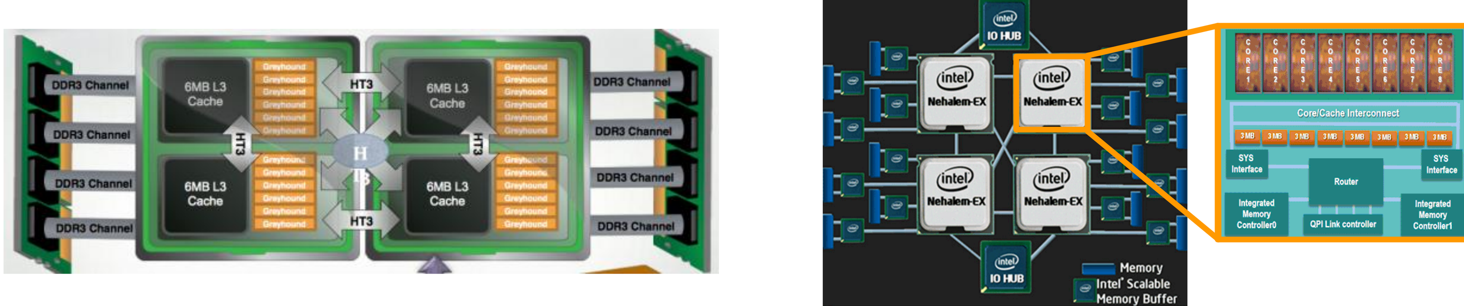
Amir Kamil and Katherine Yelick

BERKELEY PAR LAB

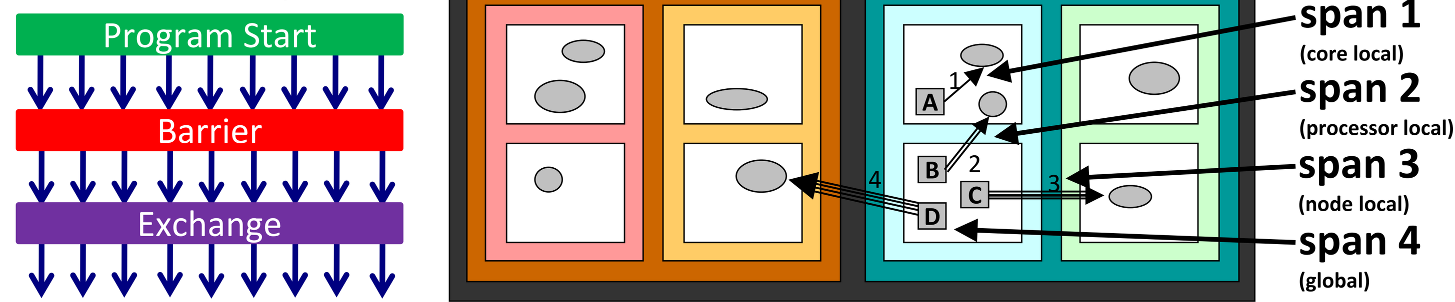
PARALLEL COMPUTING LABORATORY

## Hierarchical Machines and Algorithms

- Parallel machines are hierarchical



- Algorithms also may be hierarchical
  - Examples: merge sort, Barnes-Hut particle simulation
- Efficiency layer language must expose locality to users while also allowing general parallelism
- Our approach: start with parallelism model that has locality, add more general parallelism constructs
  - Single-Program, Multiple-Data parallelism model
    - Fixed number of threads, collective operations for synchronization, communication
  - Partitioned Global Address Space memory model
    - Threads can access data anywhere, locality encoded in pointer type




- Implementation in Titanium language, using GASNet runtime

## Hierarchical Thread Teams

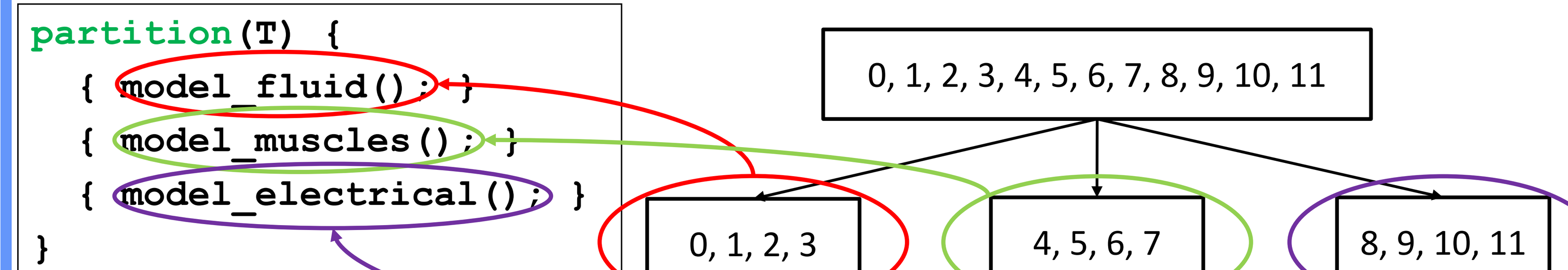
- Thread *teams* are basic units of cooperation
  - Groups of threads that cooperatively execute code
  - Collective operations over teams
- Teams should be hierarchical
  - Match hierarchical nature of machines, algorithms
- Team data structure

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
0, 1, 2, 3 | 4, 5, 6, 7 | 8, 9, 10, 11
1, 3, 2 | 0 | 9, 8 | 10, 11

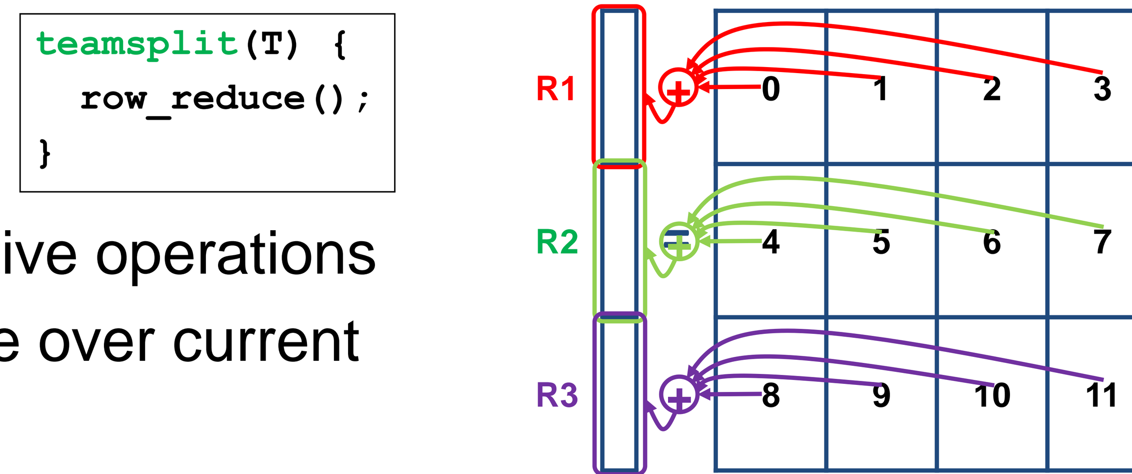
- Library functions provided to facilitate team creation
- Need to provide ability to query machine structure
  - At the moment, we provide a function to create a team hierarchy that loosely matches the machine

## Team Usage

- Thread teams may execute different code



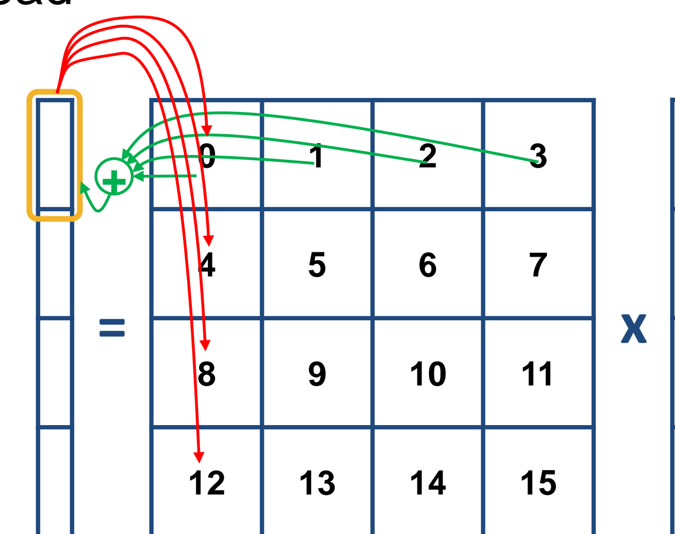

- Threads may execute same code as part of different teams of cooperating threads
- Collective operations operate over current team

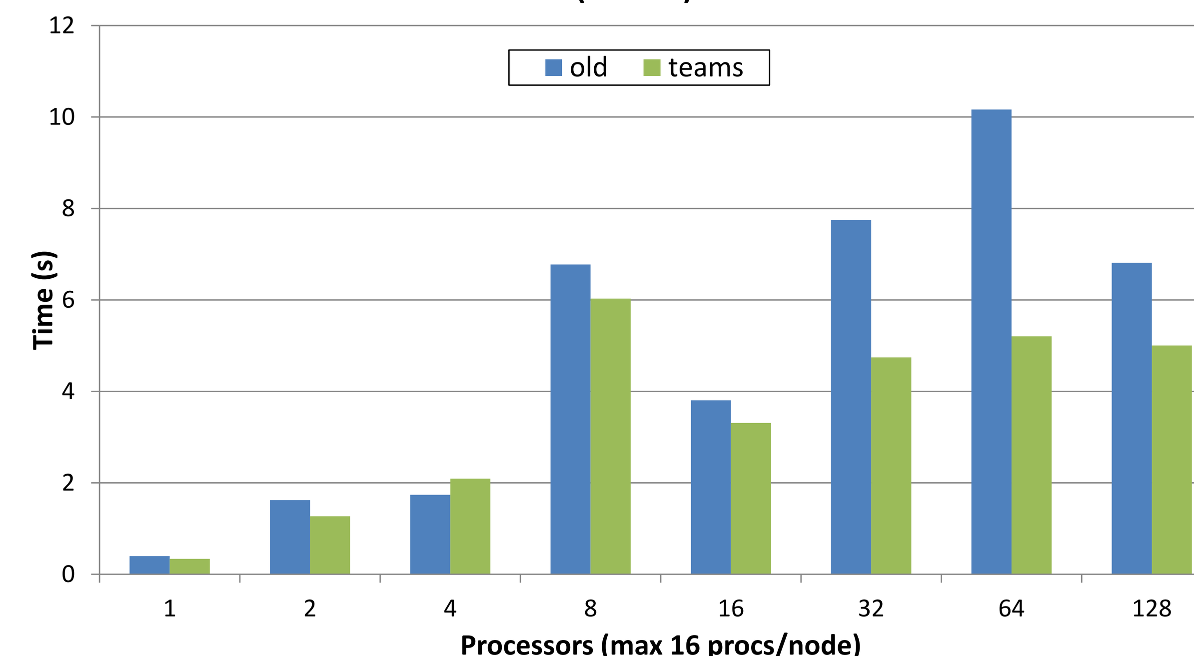



## Conjugate Gradient Library

- NAS conjugate gradient application originally written by Kaushik Datta
- Iterative solver with sparse matrix-vector multiply in each iteration
  - Matrix divided among threads by row and column
  - Result of one iteration is input of next
  - Reduction across rows, broadcast along columns
  - Original code had hand-written collectives; we use built-in team collectives instead

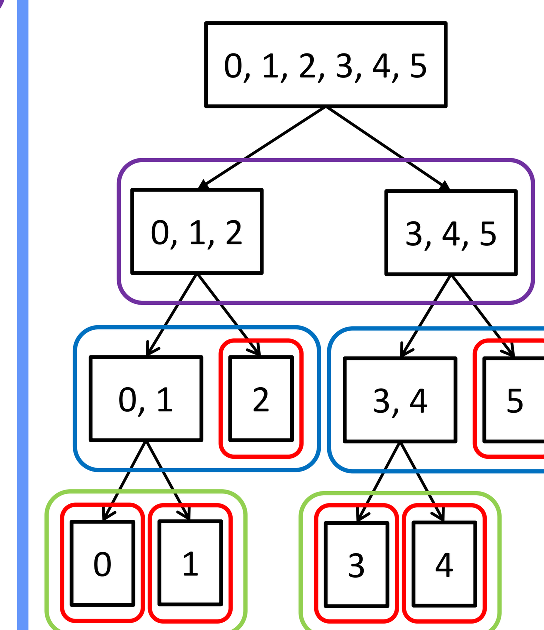
```
teamsplit(rowTeam) {
  Reduce.add(mtmp, myResults0, rpivot);
}
if (reduceCopy)
  myOut.copy(allResults[reduceSource]);
teamsplit(columnTeam) {
  myOut.vbroadcast(cpivot);
}
```




## Parallel Sorting Library

- Parallel sorting application using new hierarchical constructs
- SMP sort uses team constructs to merge results from different threads in parallel




```
void divideTeam(Team t) {
  if (t.size() > 1) {
    t.splitTeam(2);
    divideTeam(t.child(0));
    divideTeam(t.child(1));
  }
}
single void sortAndMerge(Team t) {
  if (Ti.numProcs() == 1) {
    sequentialSort(myData);
  } else {
    teamsplit(t) {
      sortAndMerge(Ti.currentTeam());
    }
    Ti.barrier();
    if (Ti.thisProc() == 0) {
      int otherProc = myProc + t.child(0).size();
      merge(data(myProc), data(otherProc));
    }
  }
}
```

- Initial distributed sort uses existing, unoptimized sample sort code to distribute elements among nodes, SMP sort to sort on each node
- 12 lines of code, 5 minutes to solution
- However, doesn't scale beyond 8 nodes

```
Team t1 = Ti.defaultTeam();
team.initialize(false);
Team t2 = new Team();
t2.splitTeamAll(t1.myChildTeam().myRank(),
                t2.myChildTeam().teamRank());
t2.initialize(false);
partition(t2) {
  { sampleSort(); }
}
teamsplit(t1) {
  keys = SMPSort.parallelSort(keys);
}
```

- New version uses all threads to distribute elements among nodes
- Still uses SMP sort to sort on each node, with no changes required to SMP code