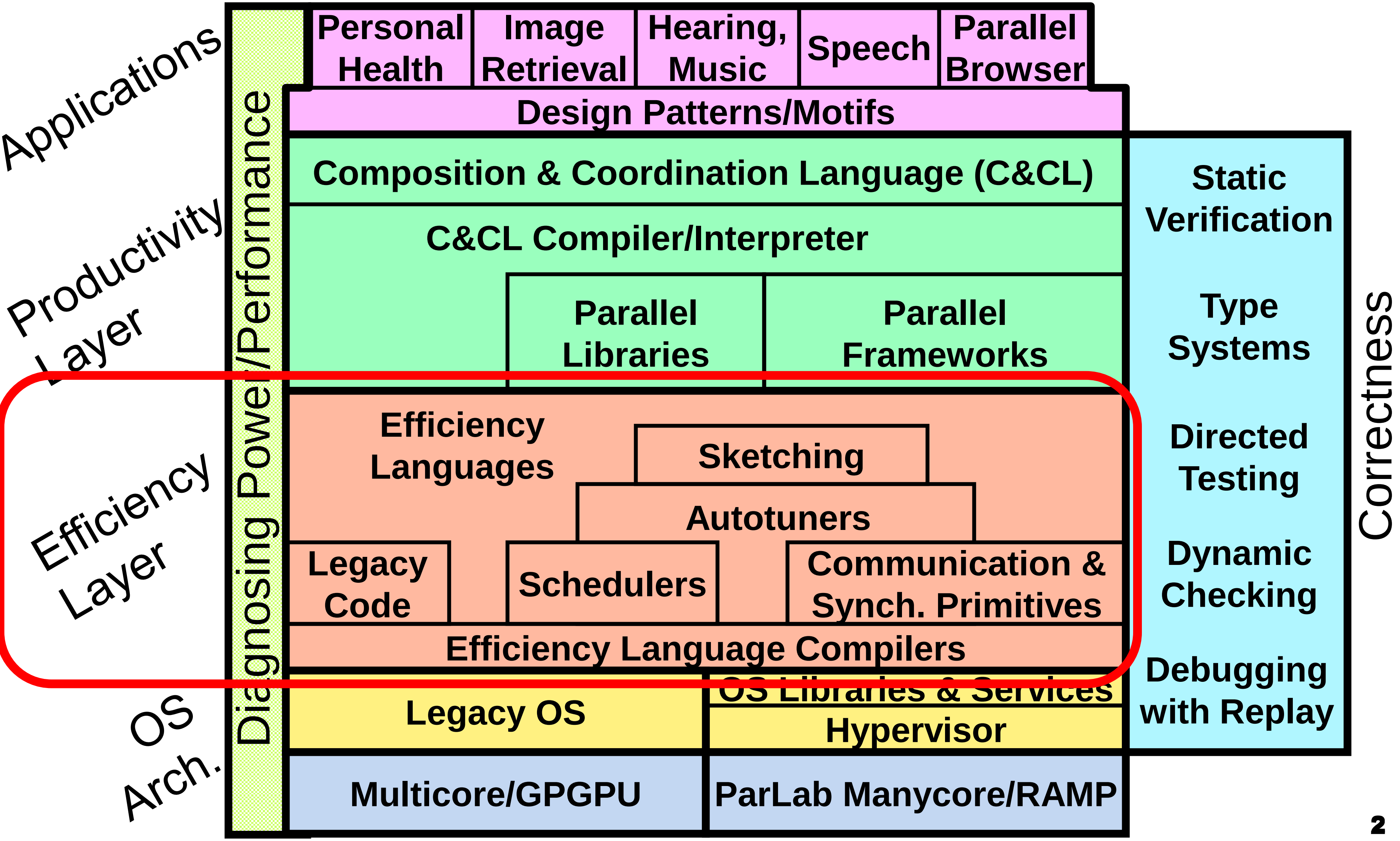
Hierarchical Parallelism in a Partitioned Address Space Model

Amir Kamil and Katherine Yelick
Par Lab Retreat
June 2, 2011

Par Lab Research Overview

Easy to write correct programs that run efficiently on manycore



Applications

Productivity Layer

Efficiency Layer

OS

Arch.

Diagnosing Power/Performance

Personal Health

Image Retrieval

Hearing, Music

Speech

Parallel Browser

Design Patterns/Motifs

Composition & Coordination Language (C&CL)

C&CL Compiler/Interpreter

Parallel Libraries

Parallel Frameworks

Efficiency Languages

Sketching

Autotuners

Legacy Code

Schedulers

Communication & Synch. Primitives

Efficiency Language Compilers

Legacy OS

OS Libraries & Services

Hypervisor

Multicore/GPGPU

ParLab Manycore/RAMP

Static Verification

Type Systems

Directed Testing

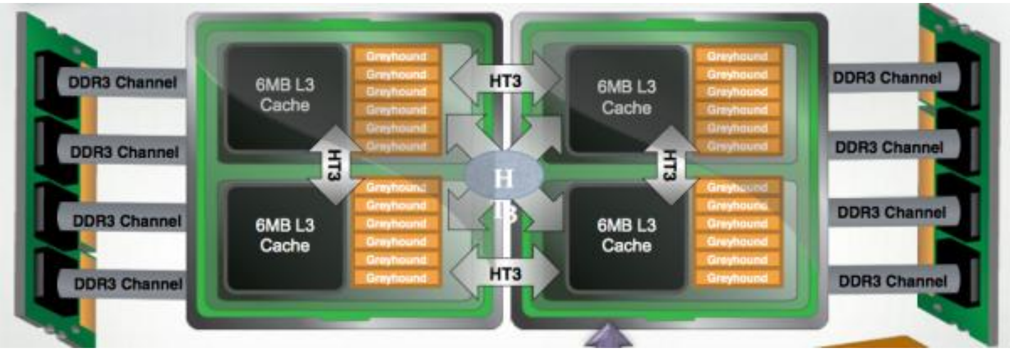
Dynamic Checking

Debugging with Replay

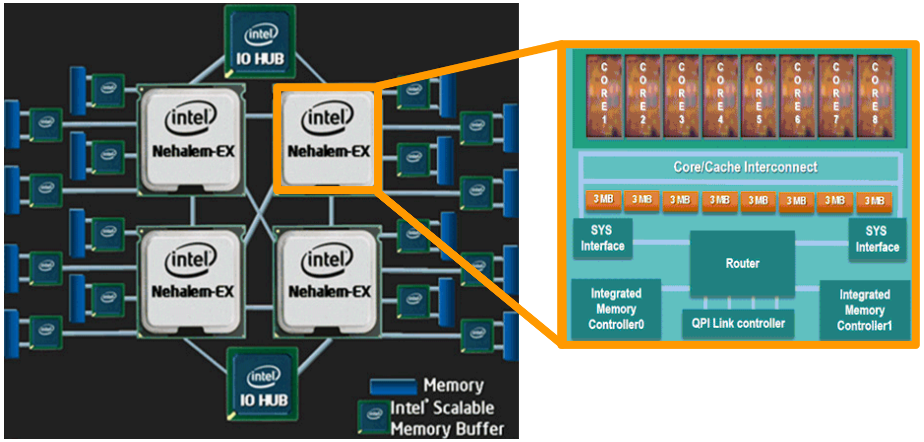
Correctness

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
  - Locality-awareness: minimize communication between distant threads, allow communication between nearby threads
- Applications may also have inherent, algorithmic hierarchy
  - Recursive algorithms
  - Composition of multiple algorithms
  - Hierarchical division of data

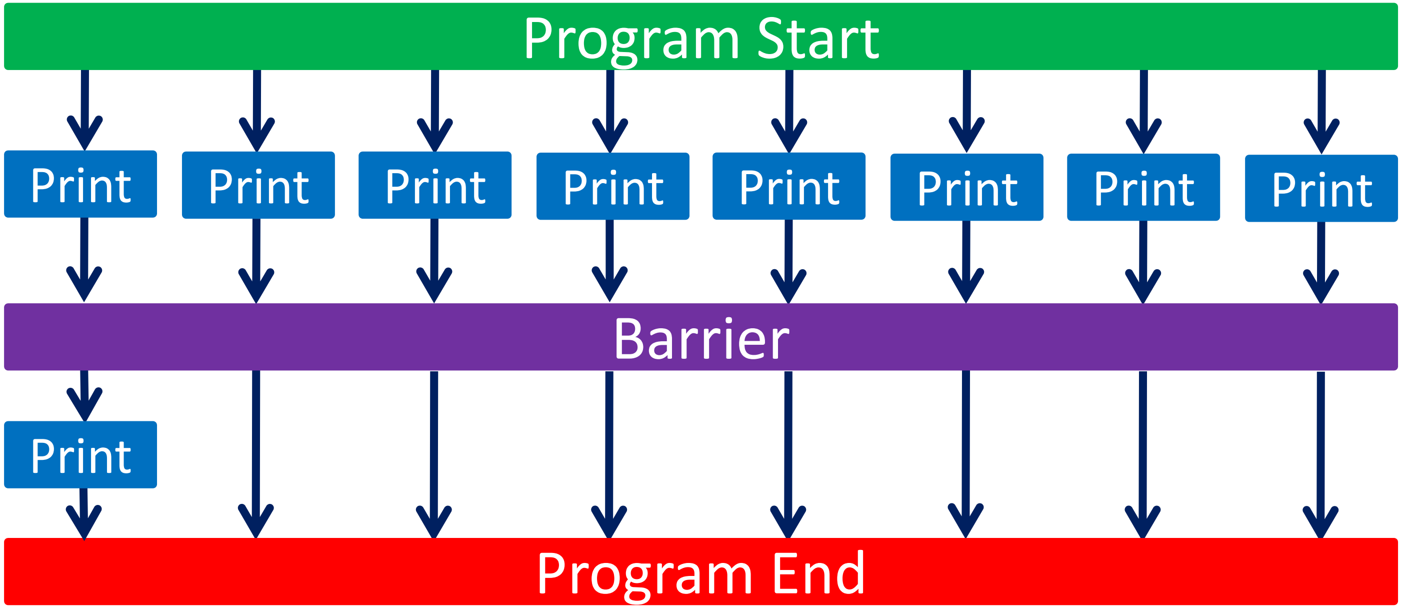
Locality is King

- Programming model must expose locality in order to obtain good performance on large-scale machines
- Possible approaches
  - Add locality hints to multithreaded languages or frameworks (e.g. TBB, OpenMP)
  - Spawn tasks at specific locality domains (X10, Chapel)
  - Use static number of threads matched to specific processing cores (SPMD)
    - Most Par Lab efficiency layer codes use this approach

Single Program, Multiple Data

- Single program, multiple data (SPMD): fixed set of threads execute the same program image

```
public static void main(String[] args) {
   System.out.println("Hello from thread "
                               + Ti.thisProc());
   Ti.barrier();
   if (Ti.thisProc() == 0)
      System.out.println("Done.");
}
```



Program Start

Print Print Print Print Print Print Print Print

Barrier

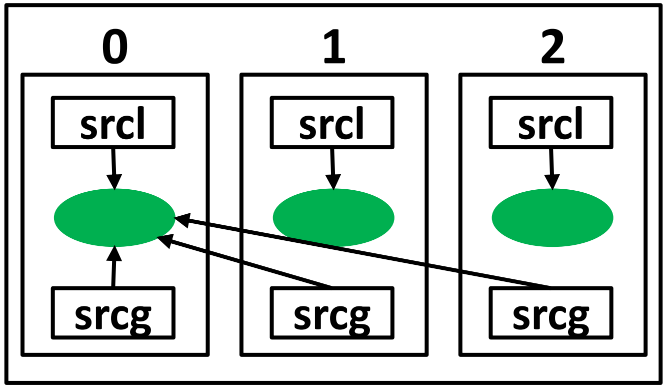
Print

Program End

Partitioned Global Address Space

- Partitioned global address space (PGAS) abstraction provides illusion of shared memory on non-shared memory machines
- Pointers can reference local or remote data
  - Location of data can be reflected in type system
  - Runtime handles any required communication

```
double[1d] local srcl = new double[0:N-1];
double[1d] srcg = broadcast srcl from 0;
```

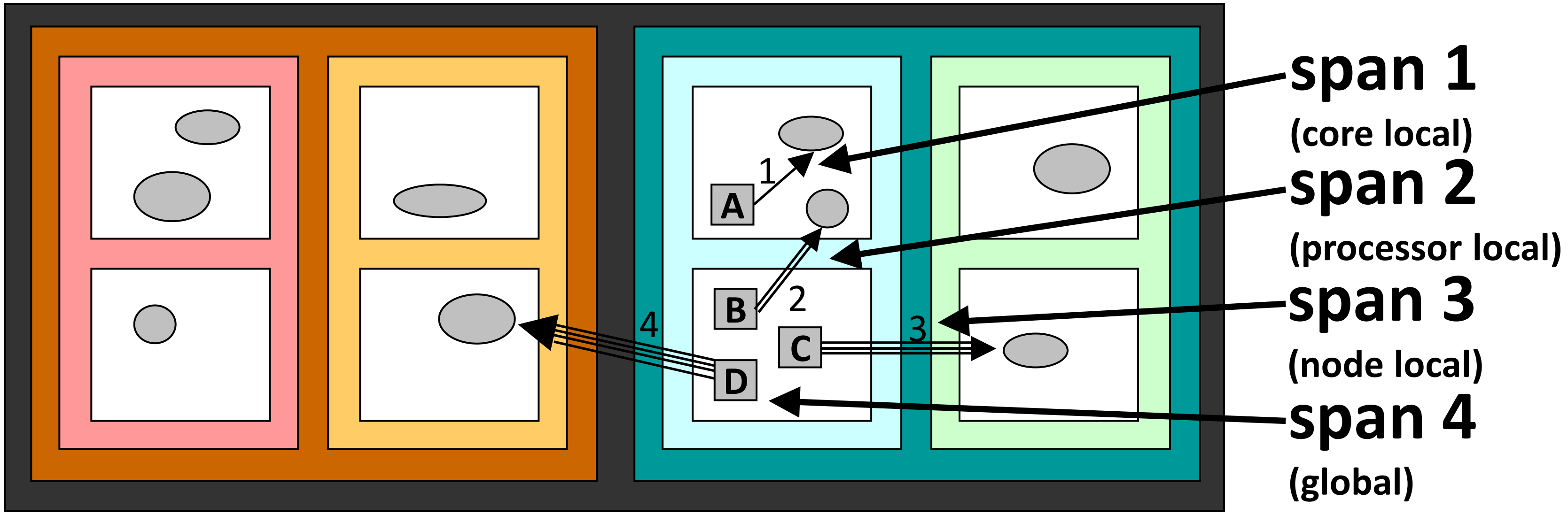


EECS
Electrical Engineering and
Computer Sciences

BERKELEY PAR LAB

Hierarchical Memory

- Previous work extended PGAS model to hierarchical arrangement of memory spaces (SAS'07)
- Pointers have varying *span* specifying how far away the referenced object can be
  - Reflect communication costs



Hierarchical Computation

- Challenge: PGAS memory model expresses locality, but hard to express hierarchical computation in SPMD execution model
- Approach: Introduce programming constructs to incorporate hierarchy in SPMD/PGAS model
    - Address both machine and algorithmic hierarchy
    - New constructs should be safe and easy to analyze
- Implementation done in Titanium language
    - SPMD/PGAS dialect of Java
    - Runtime layer based on GASNet

Outline

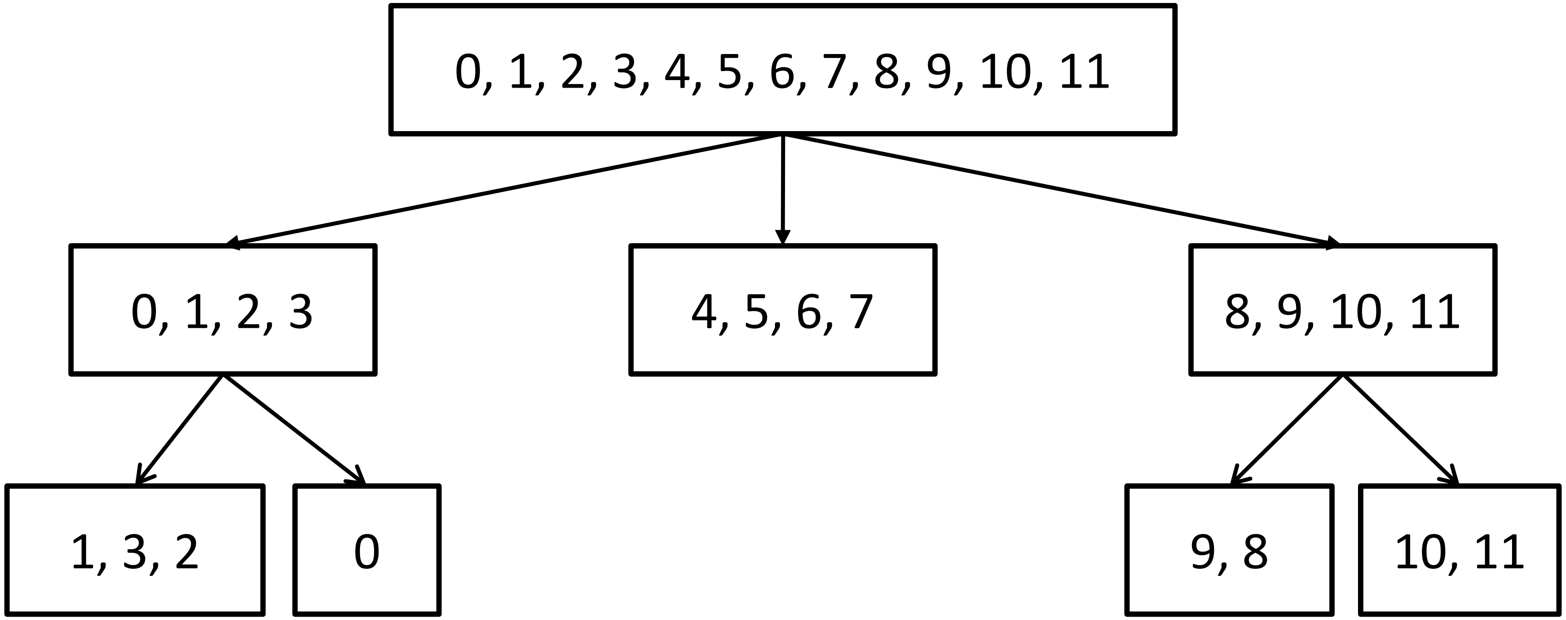
- Language Extensions
- Case Study: Sorting
- Conclusions and Future Work

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
- Teams should be safe to use
  - Should minimize new ways to program erroneously, e.g. deadlocking

Team Data Structure

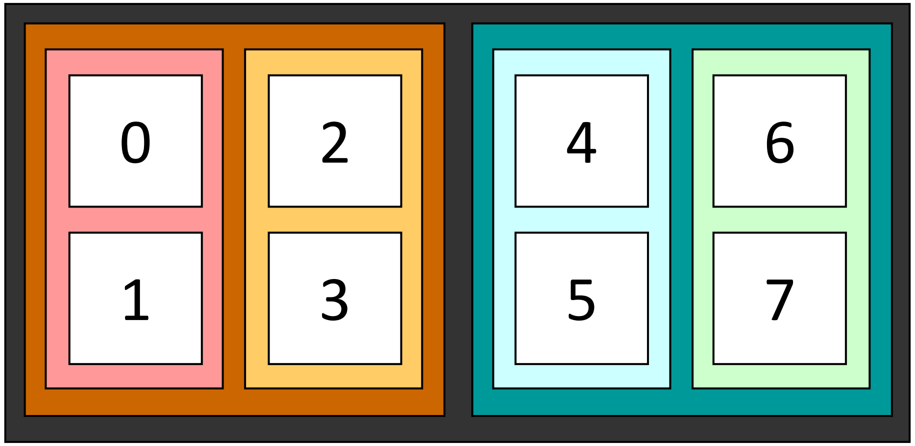
- Threads comprise teams in tree-like structure
  - Allow arbitrary hierarchies (e.g. unbalanced trees)
- First-class object to allow easy creation and manipulation
  - Library functions provided to create regular structures (e.g. even division of threads, block-cyclic)

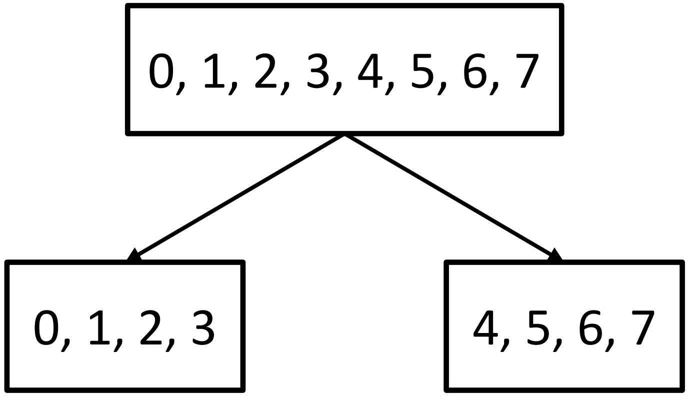


Machine Structure

- Need to provide mechanism for querying machine structure and thread mapping at runtime
  - Right now, we provide a function for constructing a team that distinguishes between threads that share memory and those that don't

```
Team T = Ti.defaultTeam();
```




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

0, 1, 2, 3

4, 5, 6, 7

Language Constructs

- Thread teams may execute distinct tasks

```
partition(T) {
   { model_fluid(); }
   { model_muscles(); }
   { model_electrical(); }
}
```

- Threads may execute the same code on different sets of data as part of different teams

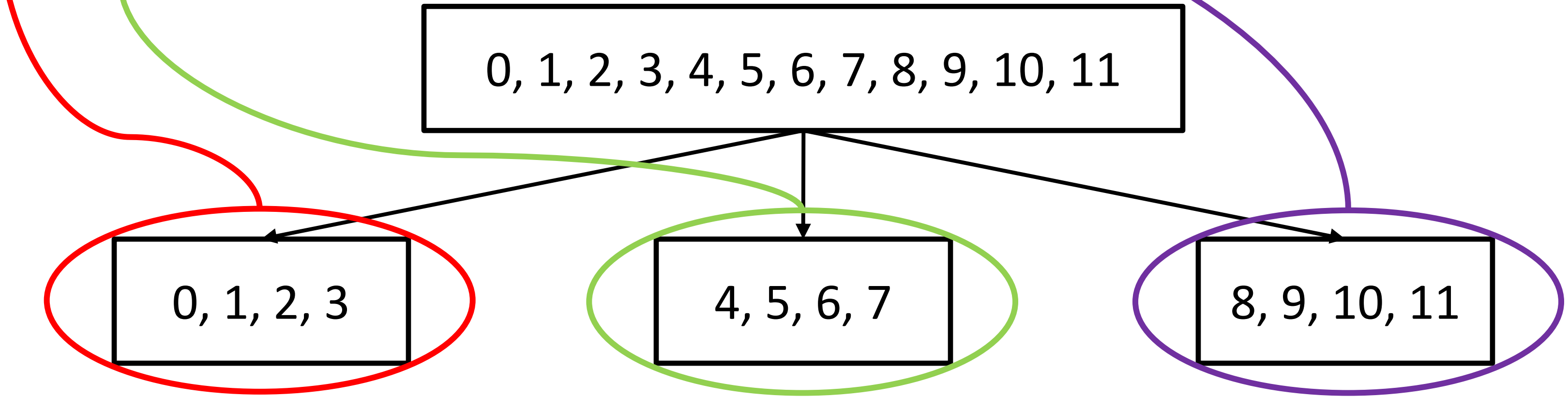
```
teamsplit(T) {
  row_reduce();
}
```

- Lexical scope prevents some types of deadlock
  - Constructs can be nested, but actual execution team is determined by innermost construct

Partition Semantics

- First subteam of **`T`** executes **`model_fluid()`**, second executes **`model_muscles()`**, third executes **`model_electrical()`**

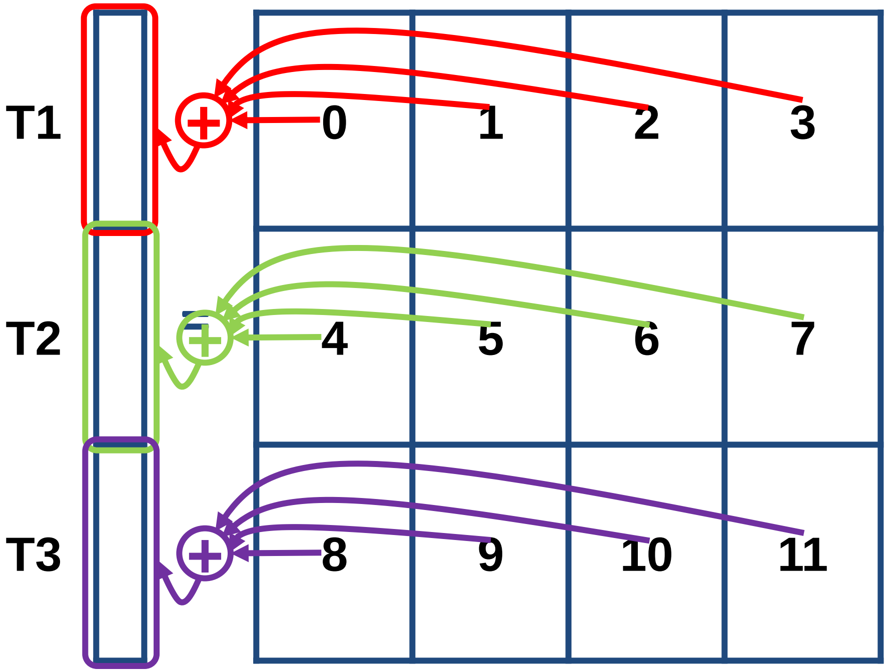
```
partition(T) {
   { model_fluid(); }
   { model_muscles(); }
   { model_electrical(); }
}
```



Teamsplit Semantics

- Each subteam of **`T`** executes **`row_reduce()`** on its own

```
teamsplit(T) {
   row_reduce();
}
```



Outline

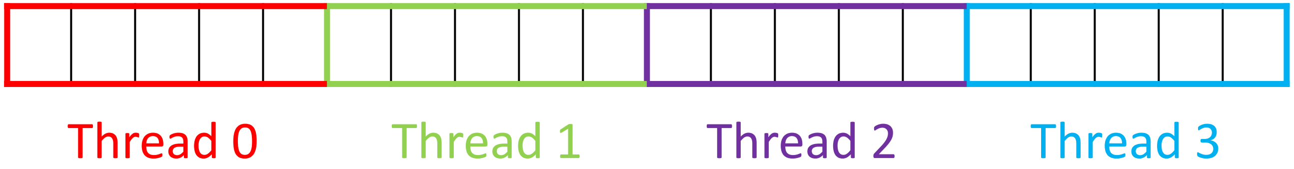
- Language Extensions
- Case Study: Sorting
- Conclusions and Future Work

Sorting

- Distributed sorting application using new hierarchical constructs
- Three pieces: sequential, shared memory, and distributed
  - Sequential: quick sort reused from Java 1.4 library
  - Shared memory: sequential sort on each thread, merge results from each thread
  - Distributed memory: sample sort to distribute elements among nodes, shared memory sort on each node

Shared Memory Sort

- Divide elements equally among threads

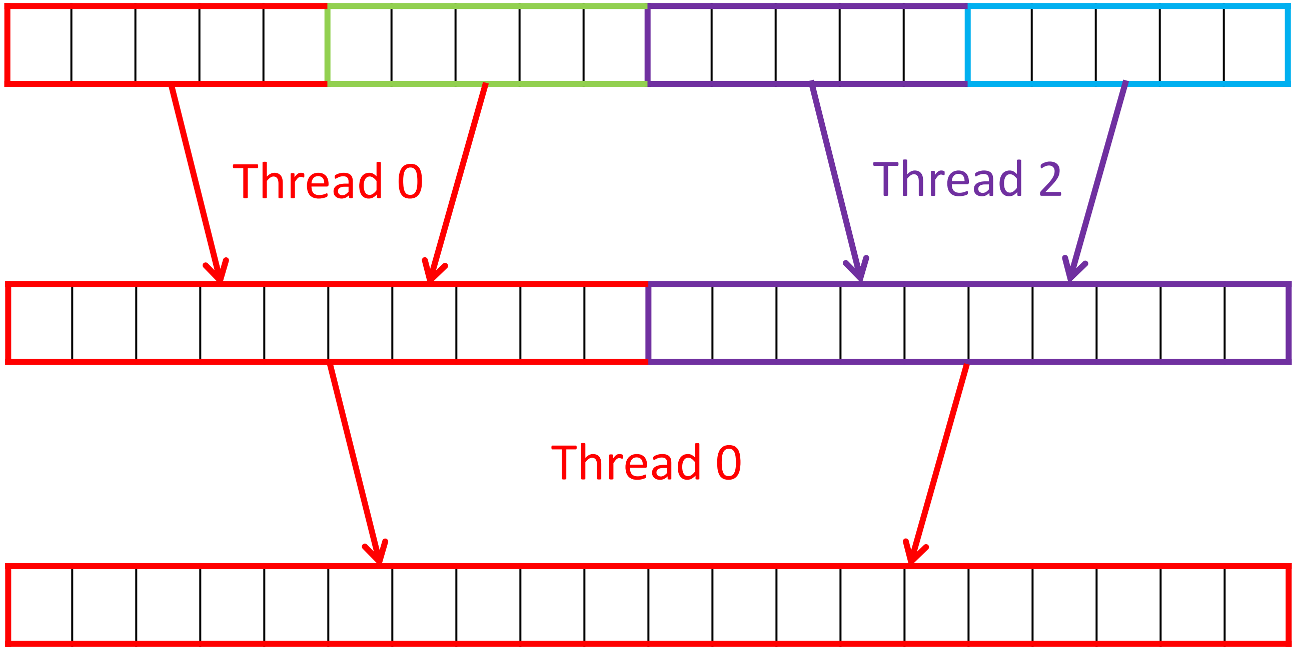


Thread 0 Thread 1 Thread 2 Thread 3

  - No communication required due to shared memory
  - No copying required thanks to Titanium's array views
- Each thread calls sequential sort to process its elements

Shared Memory Merge

- Merge in parallel



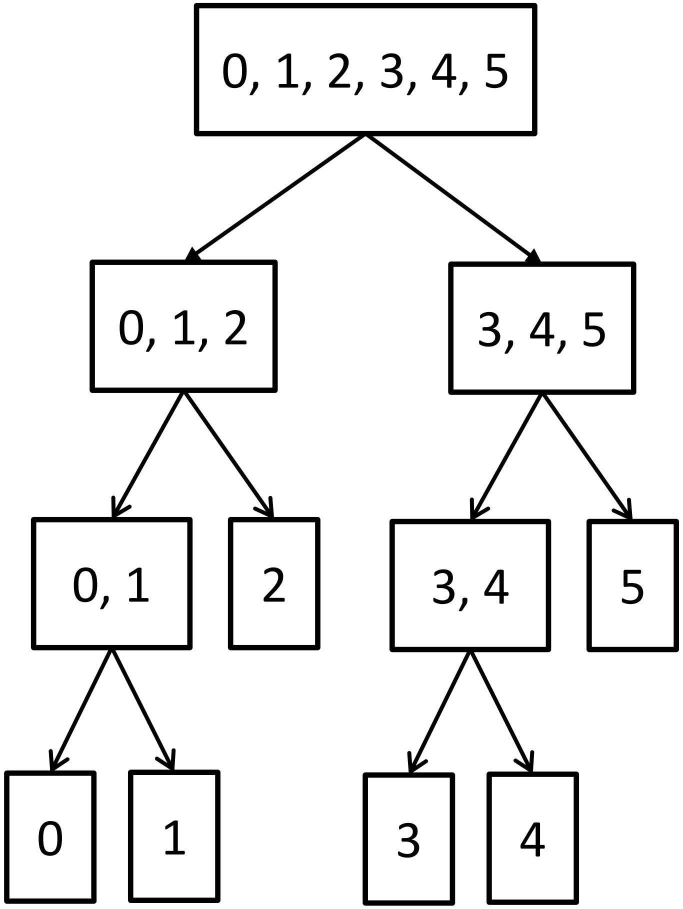
  - Number of threads approximately halved in each iteration
    - Non-trivial to determine which threads perform merge when number of threads is not power of two

Shared Memory Hierarchy

- Team hierarchy is binary tree
- Trivial construction

```
static void divideTeam(Team t) {
  if (t.size() > 1) {
    t.splitTeam(2);
    divideTeam(t.child(0));
    divideTeam(t.child(1));
  }
}
```

- Threads walk down to bottom of hierarchy, sort, then walk back up, merging along the way
  - See poster for code



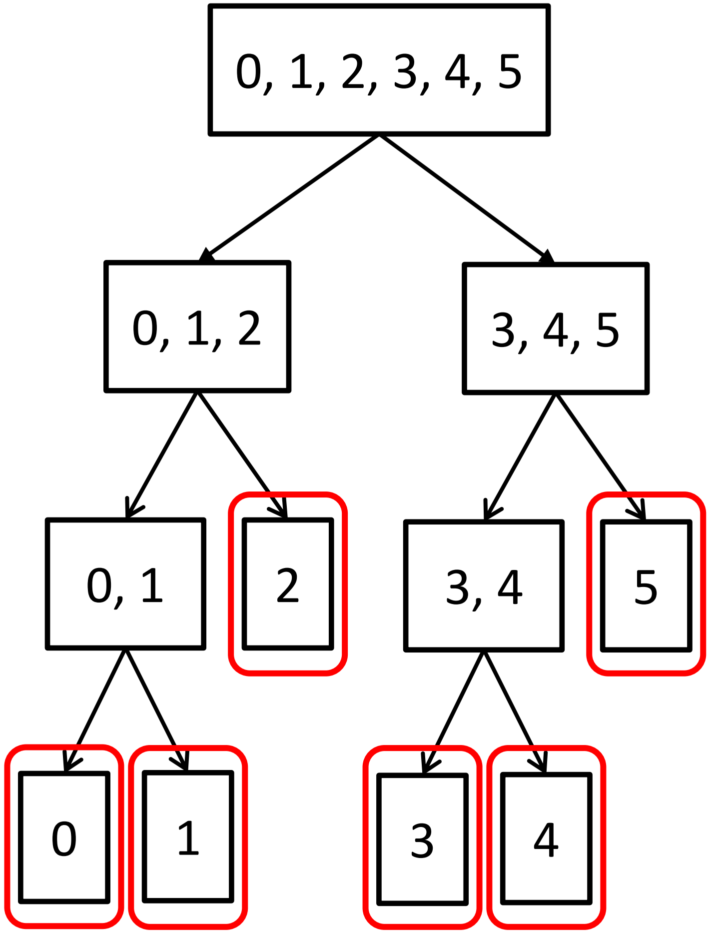
SMP Sort and Merge Logic

- Control logic for sorting and merging

```
static single void sortAndMerge(Team t) {
  if (Ti.numProcs() == 1) {
    allRes[myProc] = sequentialSort(myData);
  } else {
    teamsplit(t) {
      sortAndMerge(Ti.currentTeam());
    }
    Ti.barrier();
    if (Ti.thisProc() == 0) {
      int otherProc = myProc + t.child(0).size();
      int[1d] myRes = allRes[myProc];
      int[1d] otherRes = allRes[otherProc];
      int[1d] newRes = target(t.depth(), myRes, otherRes);
      allRes[myProc] = merge(myRes, otherRes, newRes);
    }
  }
}
```

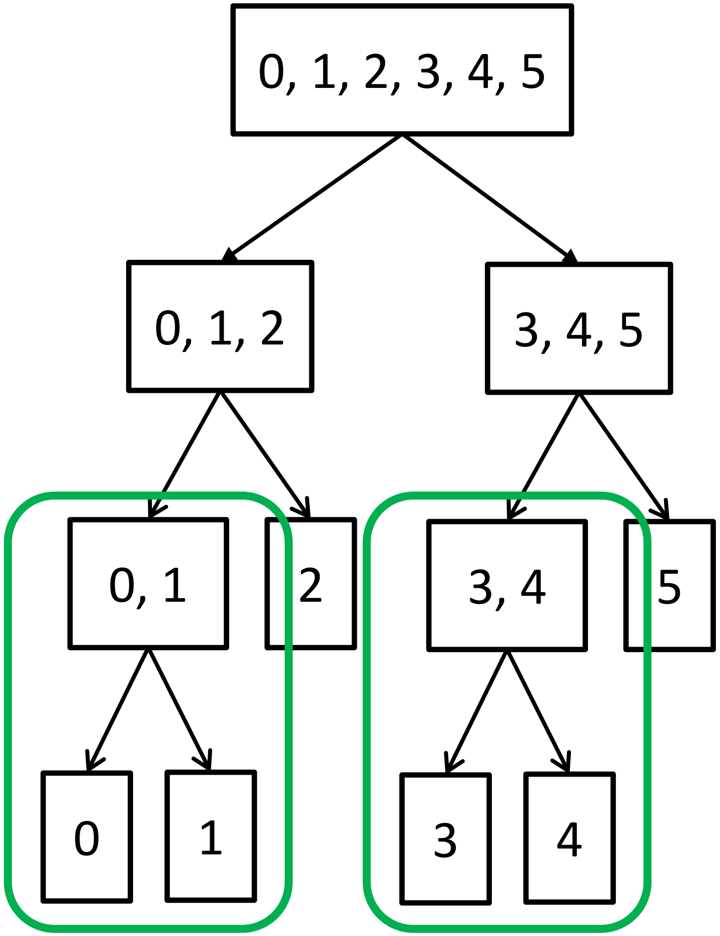
SMP Sort and Merge Example

- Phase 1: all threads sort



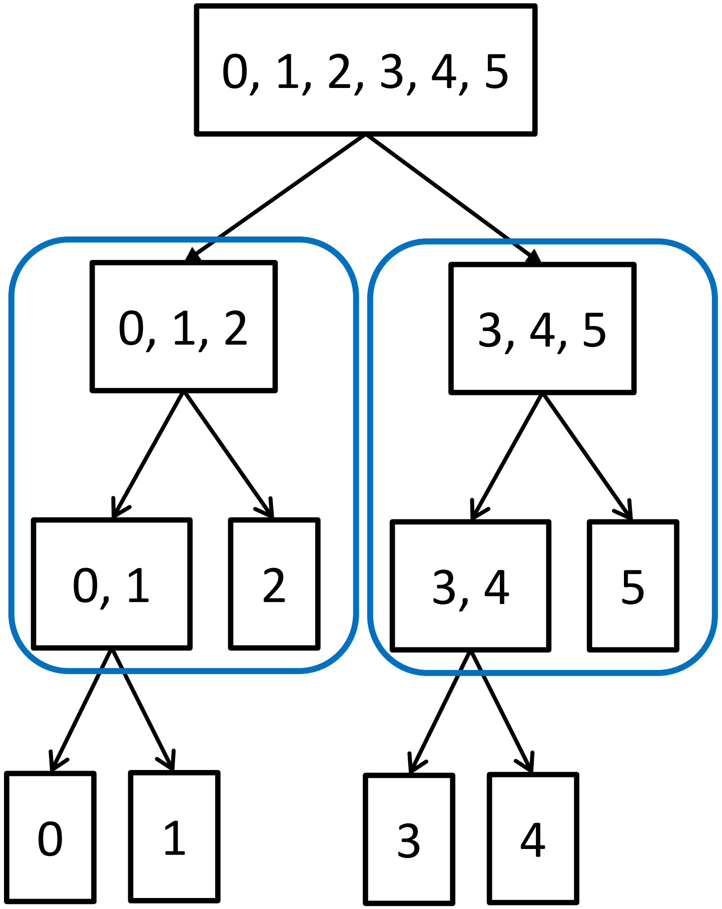
SMP Sort and Merge Example

- Phase 1: all threads sort
- Phase 2: t0 and t3 merge



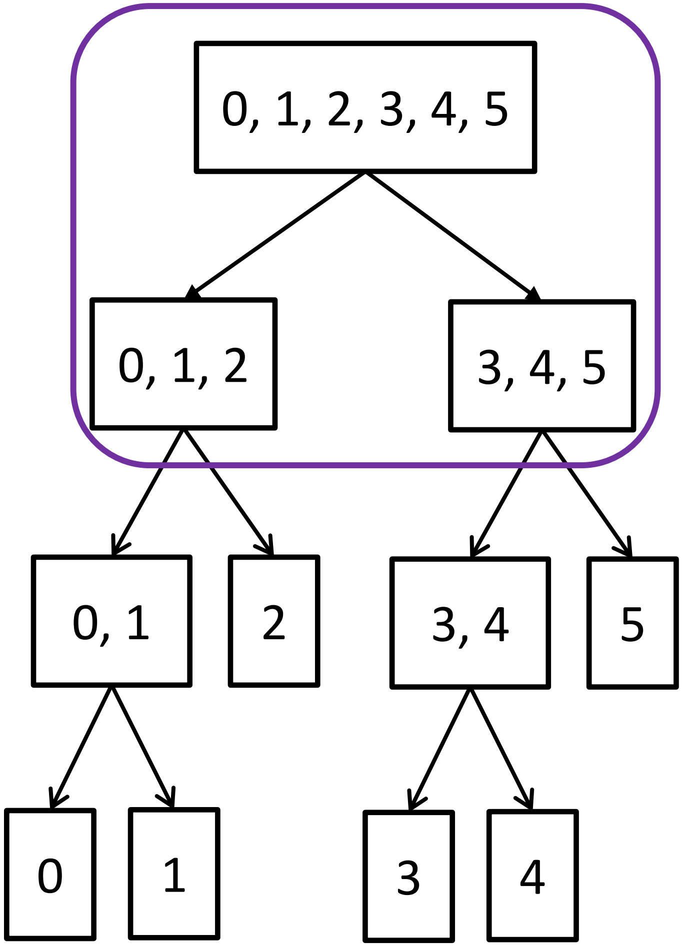
SMP Sort and Merge Example

- Phase 1: all threads sort
- Phase 2: t0 and t3 merge
- Phase 3: t0 and t3 merge



SMP Sort and Merge Example

- Phase 1: all threads sort
- Phase 2: t0 and t3 merge
- Phase 3: t0 and t3 merge
- Phase 4: t0 merges



SMP Sort Summary

- Hierarchical team constructs allow simple shared memory parallel sort implementation
    - Constructs facilitate expression of parallel recursive algorithms
- Implementation details
    - ~90 lines of code (not including test code, sequential sort)
    - 2 hours to implement (including test code) and test

Distributed Sort

- Existing sample sort written 12 years ago by Kar Ming Tang
  - Not very optimized, but it works (for small number of threads)
- Algorithm details
  - At end, elements on thread *i* sorted and less than any elements on thread *i+1*
  - Elements initially distributed across threads randomly
  - Threads exchange elements to satisfy above property
    - Lots of communication involved
  - Elements then sorted locally

CLUMPS Sort v0.1

- For clusters of SMPs, use sampling and distribution between nodes, SMP sort on nodes
  - Requires fewer messages than pure sample sort, so should scale better on large number of nodes
- Quick and dirty first version
  - Recycle old sampling and distribution code
  - Use one thread per node to perform sampling and distribution

CLUMPS Sort v0.1 Code

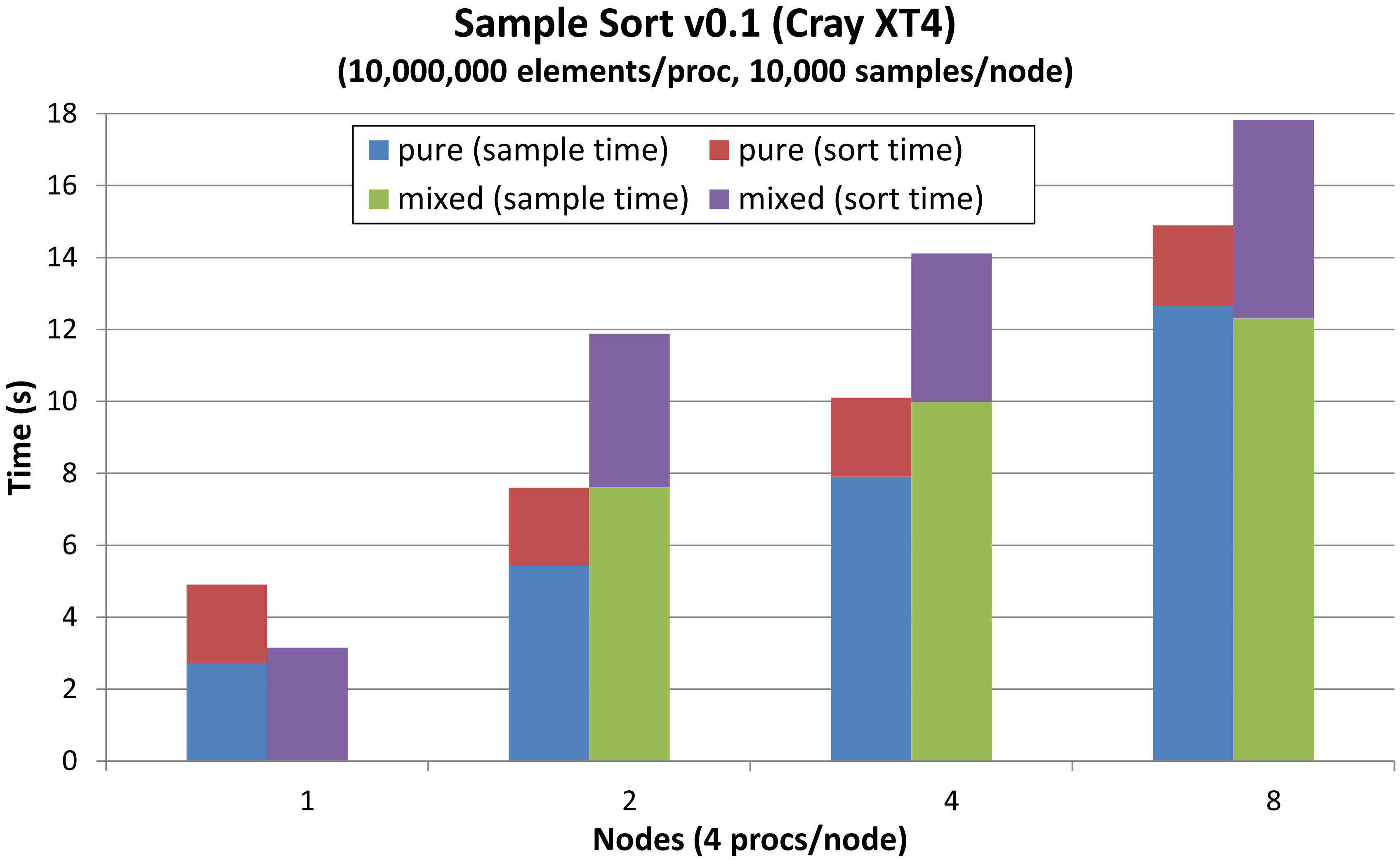
- Code for v0.1

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

- 12 lines of code, 5 minutes to solution!

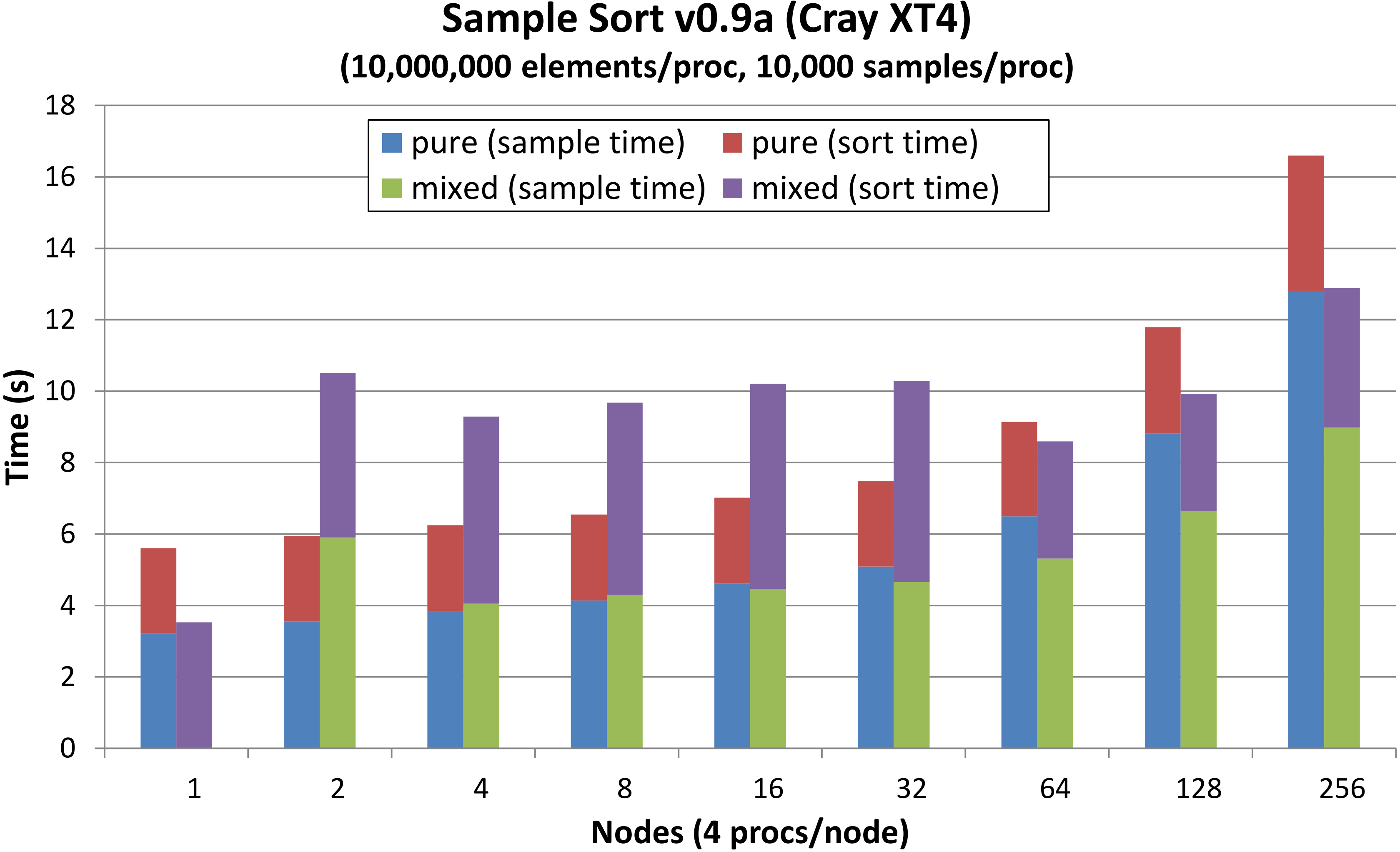
CLUMPS Sort v0.1 Results

- And it works!



CLUMPS Sort v0.9a Results

- New and improved version with more parallel distribution



Code Composition

- Sampling/distribution code composes cleanly with SMP sort – no changes to latter required
- Team constructs designed to facilitate composition
  - Existing calls **`Ti.thisProc()`**, **`Ti.numProcs()`** return thread ID and count relative to current team
  - Collective operations (barrier, broadcast) execute over current team
- Existing code executes as if its team is the entire world

Outline

- Language Extensions
- Case Study: Sorting
- Conclusions and Future Work

Conclusions

- Hierarchical language extensions simplify job of programmer
  - Can organize application around machine characteristics
  - Easier to specify algorithmic hierarchy
  - Seamless code composition
  - Better productivity, performance with team collectives
    - See poster for details
- Language extensions are safe to use
  - Safety provided by lexical scoping and a straightforward extension of LCPC’09 work

Future Work

- Machine structure
  - Use *hwloc* library to determine machine structure
  - Better representation of machine structure and thread mapping
  - Can we help programmers in mapping application hierarchies to machine structures?
  - How to handle heterogeneity?
- Extend compiler analyses to handle new language constructs

This slide intentionally left blank.