EECS
Electrical Engineering and Computer Sciences

BERKELEY PAR LAB

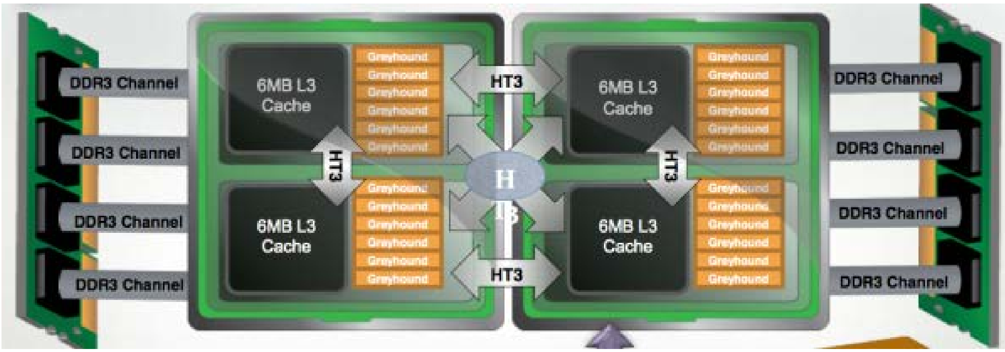
PARALLEL COMPUTING LABORATORY

Single Program, Multiple Data Programming for Hierarchical Computations

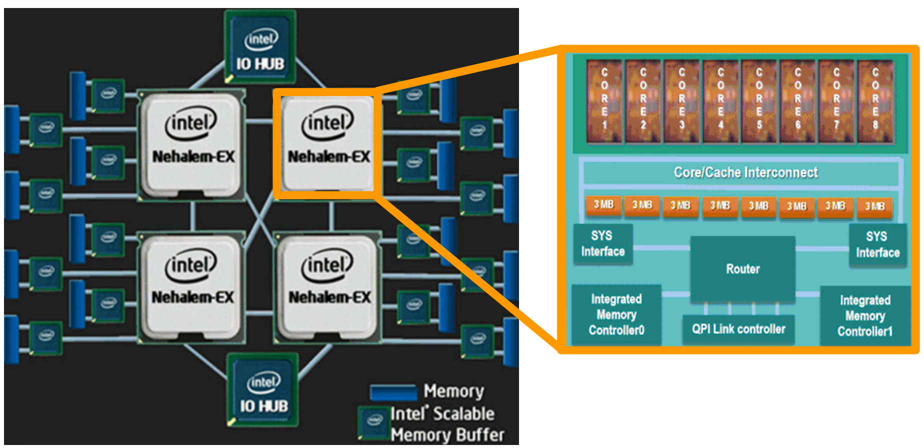
Amir Kamil
Dissertation Talk
Advisor: Katherine Yelick
May 8, 2012

Hierarchical Machines

- Parallel machines have hierarchical structure



Dual Socket AMD MagnyCours

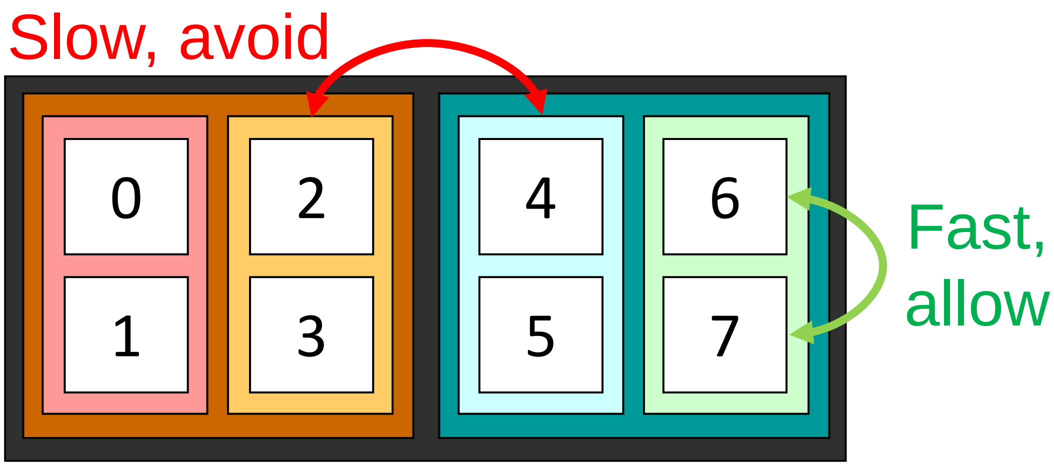


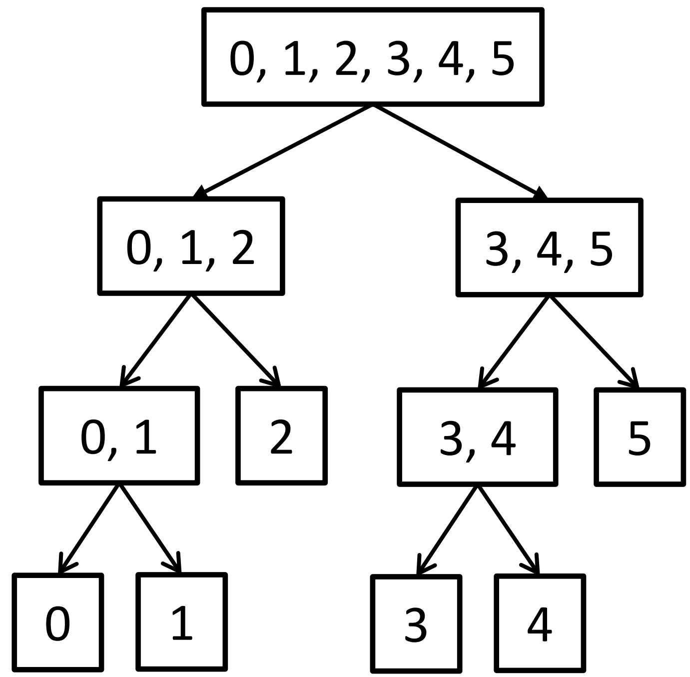
Quad Socket Intel Nehalem EX

- 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

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)

Thesis Statement

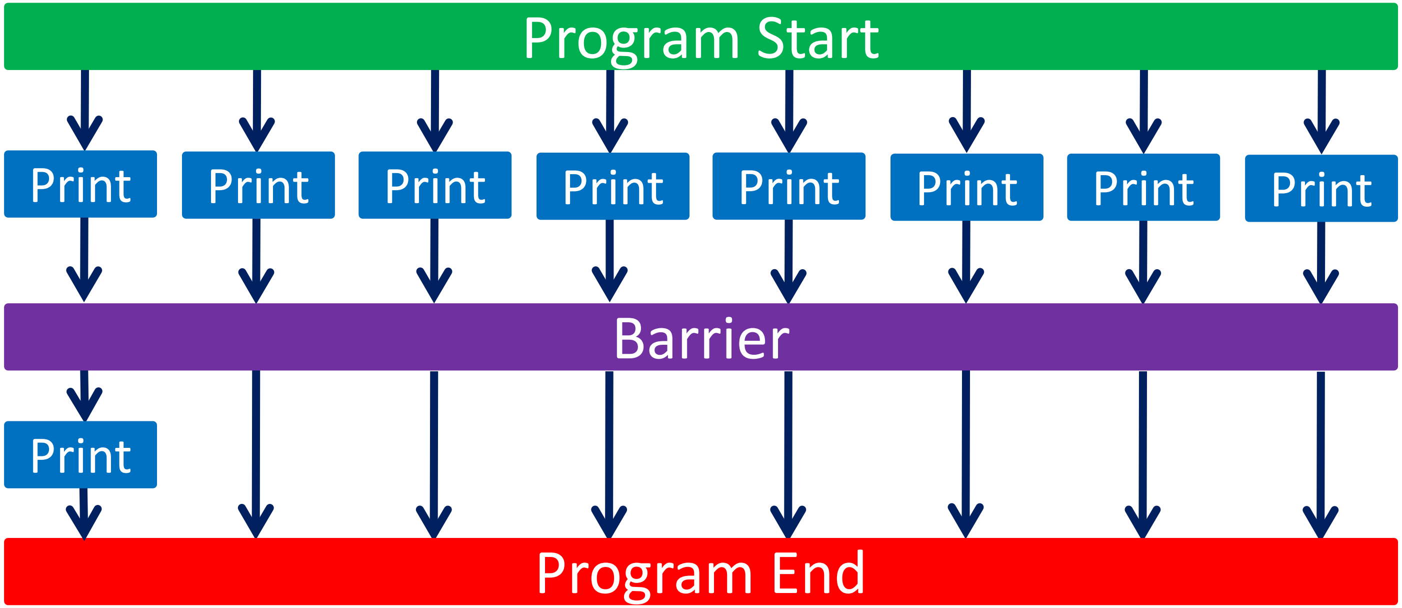
Hierarchical constructs can productively and efficiently express hierarchical algorithms and exploit the hierarchical structure of parallel machines.

- Demonstration in Titanium language, a single program, multiple data (SPMD) dialect of Java

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.");
}
```



SPMD vs. Data Parallelism

- SPMD has *local view* execution model
  - Fixed set of threads, each of which is explicitly assigned work

```
int start = numPerProc * Ti.thisProc();
int end = start + numPerProc - 1;
foreach (i in [start:end])
  C[i] = A[i] + B[i];
```

- Data parallelism is *global view*
  - Single logical thread of control
  - Compiler responsible for distributing work across computational units

```
forall (i in C.domain())
  C[i] = A[i] + B[i];
```

Global Operations

- Data parallelism allows even simpler expression of global operations

```
forall (i in C.domain())
  C[i] = A[i] + B[i];
```

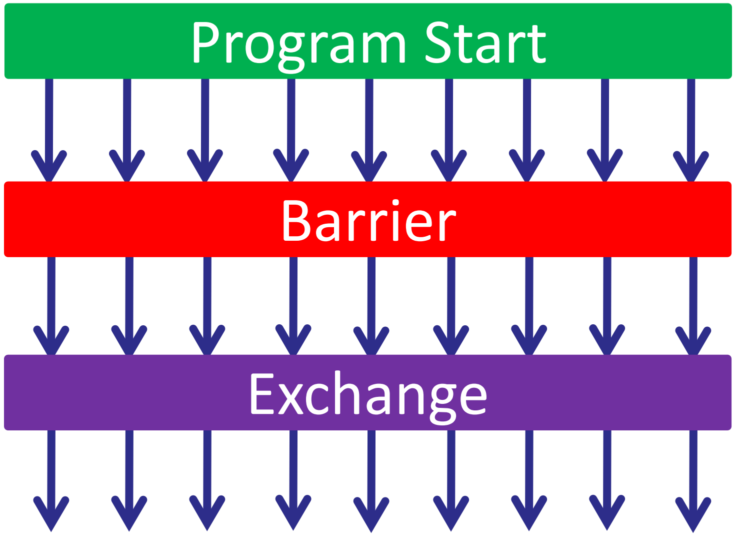
↓

```
C = A + B;
```

- Similar global operations can be built in SPMD using *collective operations*

Collective Operations

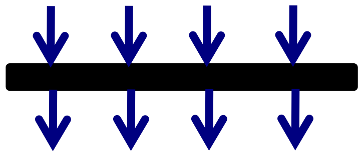
- Threads synchronize using global *collective operations*



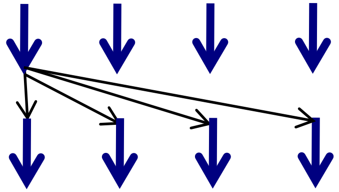
- Collective operations also used for global communication
- Collectives allow easier program analysis

Collective Examples

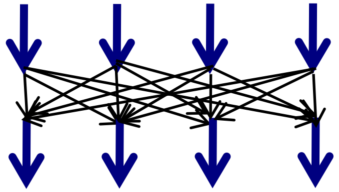
- *Barrier*: all threads must reach it before any can proceed



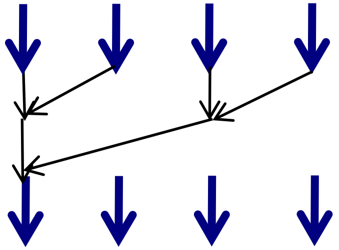
- *Broadcast*: explicit one to all communication



- *Exchange*: explicit all to all communication



- *Reduce*: explicit all to one communication



EECS Electrical Engineering and Computer Sciences

Algorithm Example: Merge Sort

BERKELEY PAR LAB

- Task parallel

```
int[] mergeSort(int[] data) {
  int len = data.length;
  if (len < threshold)
    return sequentialSort(data);
  d1 = fork mergeSort(data[0:len/2-1]);
  d2 = mergeSort(data[len/2:len-1]);
  join d1;
  return merge(d1, d2);
}
```

- Cannot fork threads in SPMD
  - Must rewrite to execute over fixed set of threads

Algorithm Example: Merge Sort

- SPMD

```
int[] mergeSort(int[] data, int[] ids) {
  int len = data.length;
  int threads = ids.length;
  if (threads == 1) return sequentialSort(data);
  if (myId in ids[0:threads/2-1])
    d1 = mergeSort(data[0:len/2-1],
                   ids[0:threads/2-1]);
  else
    d2 = mergeSort(data[len/2:len-1],
                   ids[threads/2:threads-1]);
  barrier(ids);
  if (myId == ids[0]) return merge(d1, d2);
}
```

Algorithm Example: Merge Sort

- SPMD

```
int[] mergeSort(int[] data, int[] ids) {
  int len = data.length;
  int threads = ids.length;
  if (threads == 1) return sequentialSort(data);
  if (myId in ids[0:threads/2-1])
    d1 = mergeSort(data[0:len/2-1],
                   ids[0:threads/2-1]);
  else
    d2 = mergeSort(data[len/2:len-1],
                   ids[threads/2:threads-1]);
  barrier(ids);
  if (myId == ids[0]) return merge(d1, d2);
}
```

Team

Algorithm Example: Merge Sort

- SPMD

```
int[] mergeSort(int[] data, int[] ids) {
  int len = data.length;
  int threads = ids.length;
  if (threads == 1) return sequentialSort(data);
  if (myId in ids[0:threads/2-1])
    d1 = mergeSort(data[0:len/2-1],
                   ids[0:threads/2-1]);
  else
    d2 = mergeSort(data[len/2:len-1],
                   ids[threads/2:threads-1]);
  barrier(ids);
  if (myId == ids[0]) return merge(d1, d2);
}
```

Team

Team Collective

Thread Teams

- Thread *teams* are basic units of cooperation
  - Groups of threads that cooperatively execute code
  - Collective operations over teams
- Other languages have teams
  - MPI communicators, UPC teams
- However, those teams are flat
  - Do not match hierarchical structure of algorithms, machines
  - Misuse of teams can result in deadlock

```
Team t1 = new Team(0:7);
Team t2 = new Team(0:3);
if (myId == 0) barrier(t1);
else barrier(t2);
```

Structured Teams

- Structured, hierarchical teams are the solution
  - Expressive: match structure of algorithms, machines
  - Safe: eliminate many sources of deadlock
  - Analyzable: enable simple program analysis
  - Efficient: allow users to take advantage of machine structure, resulting in performance gains

Related Work

- Languages that incorporate machine hierarchy
  - Sequoia: hierarchical task structure
  - HTA, Chapel: hierarchically defined data structures
  - HPT, Fortress: hierarchical locales (memory/execution spaces)
- Mixed and nested task/data parallelism a form of control hierarchy
  - MPI+OpenMP, NESL
- None of the above is SPMD

Why SPMD?

- SPMD simplifies parallel programming by imposing structure on programs
  - Forces programmer to think about parallelism, locality of data
  - Fixed set of threads – exact degree of parallelism exposed
  - Threads execute same code – reduces need to keep track of which thread executes what
  - Simple implementation
  - Provides good performance
- Simple program analysis
- Large-scale machines almost exclusively programmed using SPMD

EECS
Electrical Engineering and Computer Sciences

Contributions

BERKELEY PAR LAB

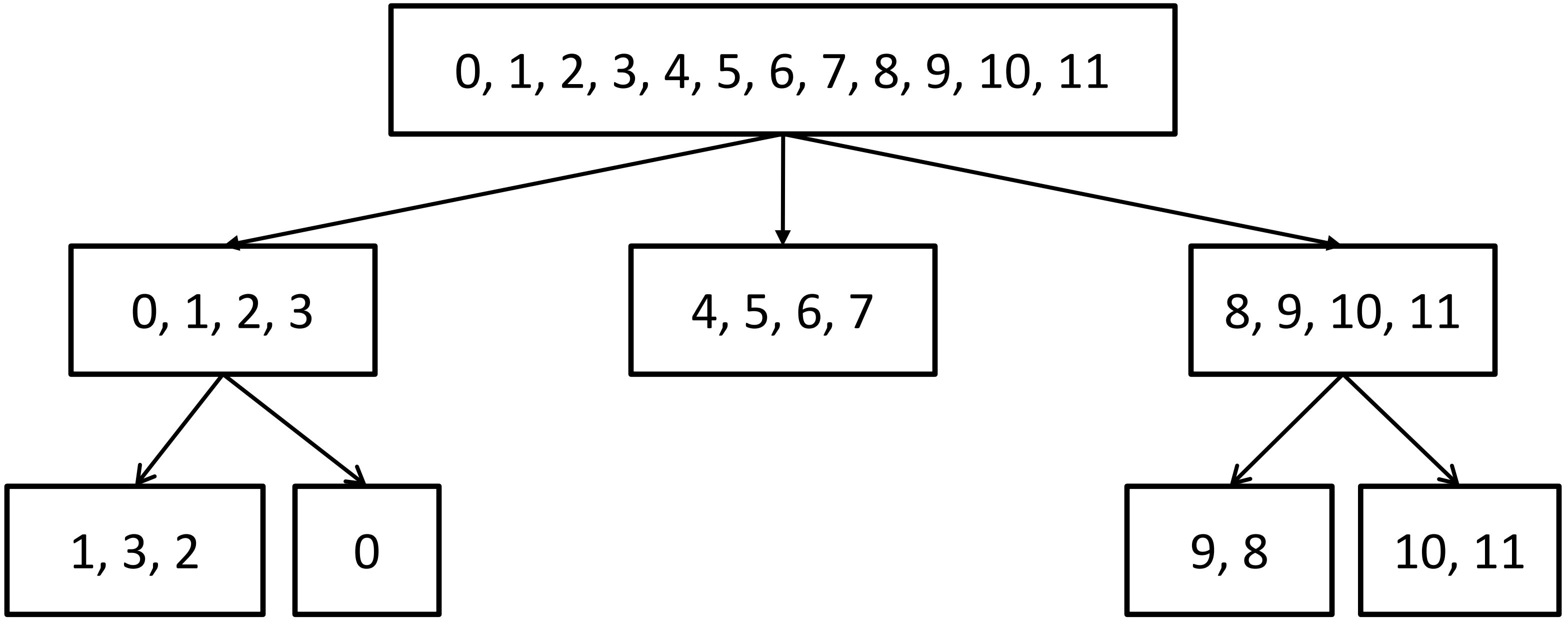
- New language constructs to express hierarchical computation
  - Algorithmic and machine-dependent hierarchy
  - Improve productivity and performance
- Dynamic alignment of collectives
  - Improve safety and debugging of explicitly parallel programs
- Program analysis
  - Hierarchical pointer analysis
  - Concurrency analysis for textually aligned SPMD

Outline

- Language Extensions
- Alignment of Collectives
- Pointer Analysis
- Application Case Studies
- Conclusions

Team Data Structure

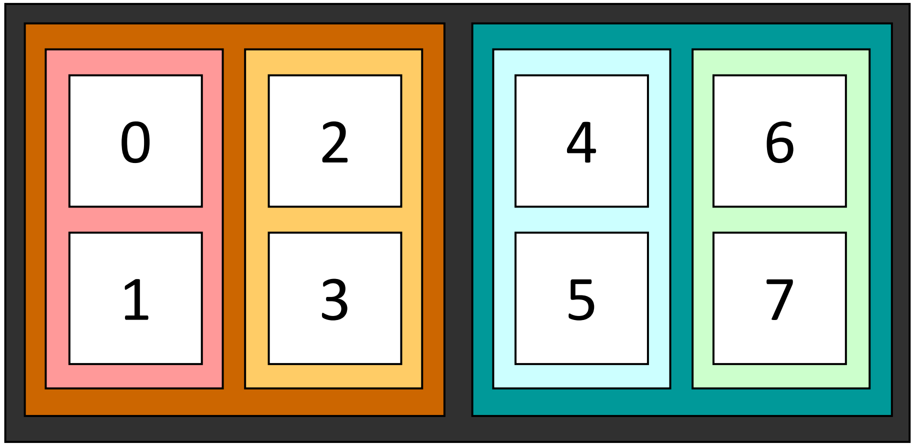
- Threads comprise teams in tree-like structure
  - Allow arbitrary hierarchies (e.g. unbalanced trees)
- First-class object to allow easy creation and manipulation
  - Library functions provided to create regular structures

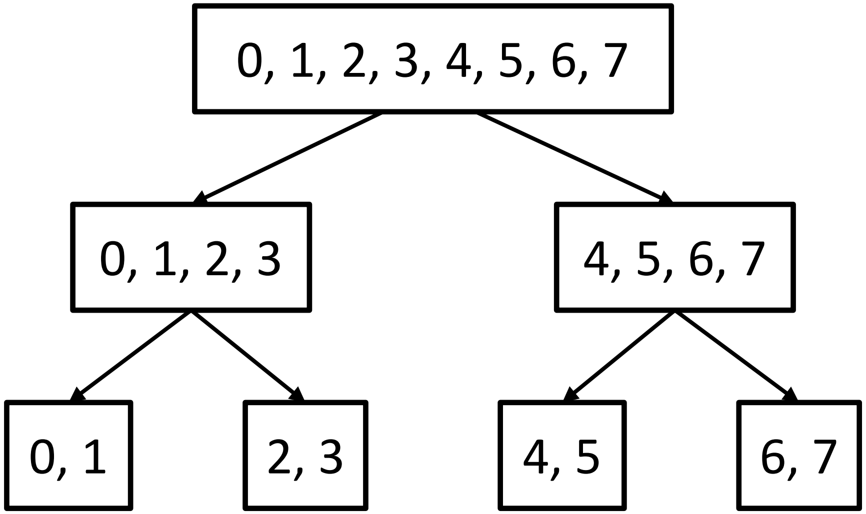


Machine Structure

- Provide mechanism for querying machine structure and thread mapping at runtime

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





EECS Electrical Engineering and Computer Sciences

Language Constructs

BERKELEY PAR LAB

- 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

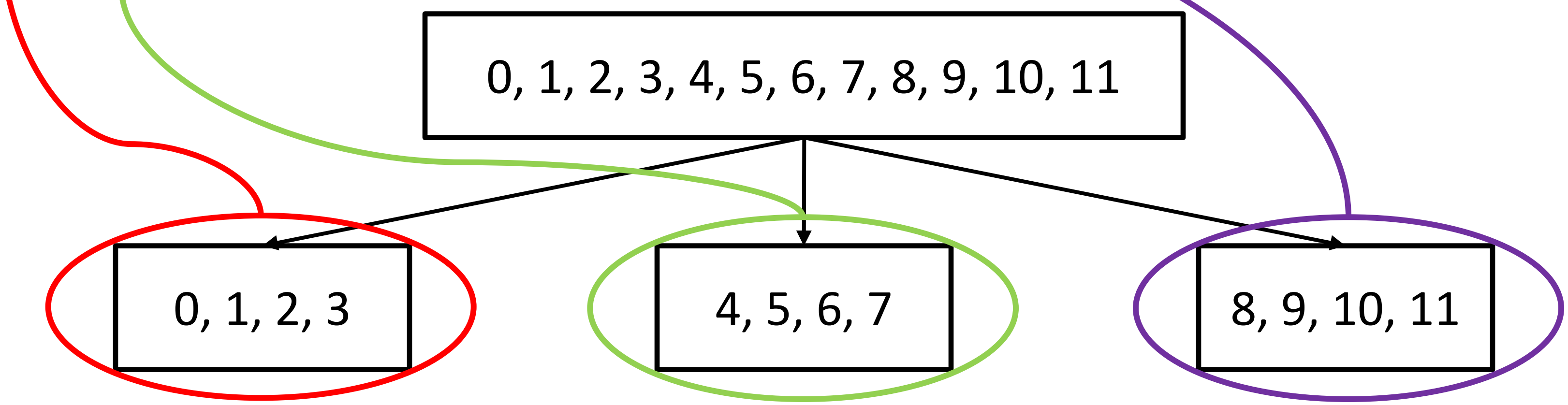
```
teamsplit(T) {
  row_reduce();
}
```

- Lexical scope prevents some types of deadlock
  - Execution team determined by enclosing construct

Partition Semantics

- Different subteams of **T** execute each of the branches

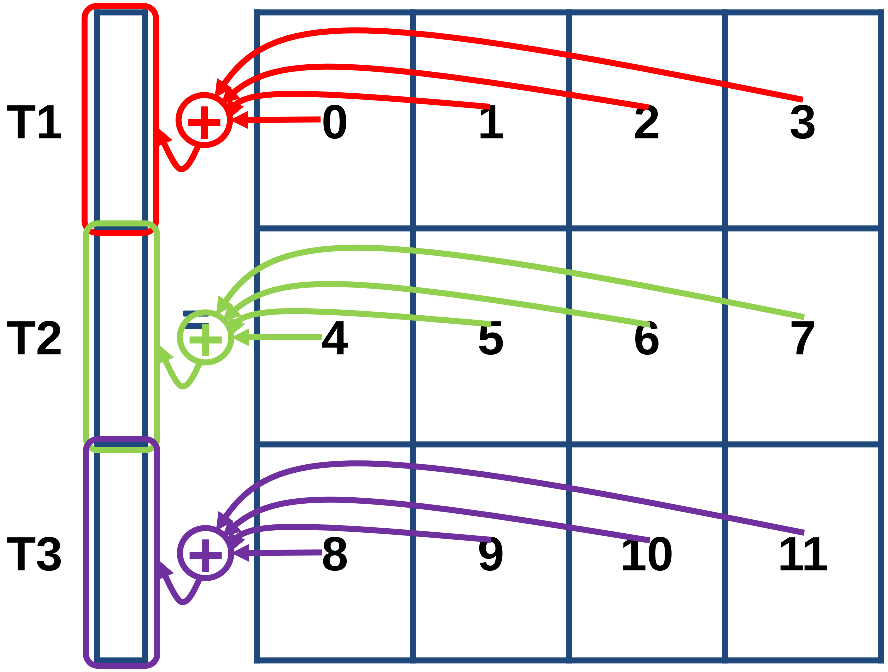
```
partition(T) {
   { model_fluid(); }
   { model_muscles(); }
   { model_electrical(); }
}
```



Teamsplit Semantics

- Each subteam of **`rowTeam`** executes the reduction on its own

```
teamsplit(rowTeam) {
  Reduce.add(mtmp, myResults0, rpivot);
}
```



Multiple Hierarchy Levels

- Constructs can be nested

```
teamsplit(T) {
  teamsplit(T.myChildTeam()) {
    level1_work();
  }
  level2_work();
}
```

- Program can use multiple teams

```
teamsplit(columnTeam) {
  myOut.vbroadcast(cpivot);
}
teamsplit(rowTeam) {
  Reduce.add(mtmp, myResults0, rpivot);
}
```

Outline

- Language Extensions
- Alignment of Collectives
- Pointer Analysis
- Application Case Studies
- Conclusions

Collective Alignment

- Many parallel languages make no attempt to ensure that collectives line up
  - Example code that will compile but deadlock:

```
if (Ti.thisProc() % 2 == 0)
   Ti.barrier(); // even ID threads
else
   ; // odd ID threads
int i = broadcast Ti.thisProc() from 0;
```

Textual Collective Alignment

- In *textual alignment,* all threads must execute the same *textual* sequence of collectives
- In addition, all threads must agree on control flow decisions that may result in a collective
  - Following is illegal:

```
  if (Ti.thisProc() % 2 == 0)
    myBarrier(); // even ID threads
  else
    myBarrier(); // odd ID threads
...
static void myBarrier() {
  Ti.barrier();
}
```

Benefits of Textual Alignment

- Textual alignment prevents deadlock due to misaligned collectives
- Easy to reason about, analyze
  - Concurrency analysis paper in LCPC'05
- Most applications only use textually aligned collectives

Alignment Checking Schemes

- Different schemes can be used to enforce textual alignment

	Programmer burden	Restrictions on program structure	Early error detection	Accuracy/ Precision	Performance reduction	Team support
Type system	High	High	High	High	No	No
Static inference	Low	Medium	High	Low	No	Yes
Dynamic checks	Low	High	Medium	High	Yes	Yes
No checking	None	None	No	None	No	Yes

Dynamic Enforcement

- A dynamic enforcement scheme can reduce programmer burden but still provide safety and accurate results for analysis and optimization
- Basic idea:
  - Track control flow on all threads
  - Check that preceding control flow matches when:
    - Performing a team collective
    - Changing team contexts
- Compiler instruments source code to perform tracking and checking

Tracking Example

0, 1 →

```
5 if (Ti.thisProc() == 0)
6   Ti.barrier();
7 else
8   Ti.barrier();
```

Thread	Hash	Execution History
0	0x0dc7637a	…*
1	0x0dc7637a	…*

* Entries prior to line 5

Tracking Example

```
5 if (Ti.thisProc() == 0)
6    Ti.barrier();
7 else
8    Ti.barrier();
```

0 → line 6; 1 → line 8

Control flow decision noted, hash updated

Thread	Hash	Execution History
0	0x7e8a6fa0	…*, (5, then)
1	0x2027593c	…*, (5, else)

* Entries prior to line 5

Checking Example

```
5 if (Ti.thisProc() == 0)
6     Ti.barrier();
7 else
8     Ti.barrier();
```

Hash broadcast from thread 0

Thread	Hash	Hash from 0	Execution History
0	0x7e8a6fa0		…*, (5, then)
1	0x2027593c		…*, (5, else)

* Entries prior to line 5

Checking Example


```
5 if (Ti.thisProc() == 0)
6    Ti.barrier();
7 else
8    Ti.barrier();
```

Hash from 0 compared with local hash

Thread	Hash	Hash from 0	Execution History
0	0x7e8a6fa0	0x7e8a6fa0	…*, (5, then)
1	0x2027593c	0x7e8a6fa0	…*, (5, else)

* Entries prior to line 5

EECS
Electrical Engineering and Computer Sciences

Checking Example

BERKELEY PAR LAB

```
5 if (Ti.thisProc() == 0)
6   Ti.barrier();
7 else
8   Ti.barrier();
```

0 → line 6

1 → line 8

Hash from 0 compared with local hash

Thread	Hash	Hash from 0	Execution History
0	0x7e8 ERROR	a6fa0	…*, (5, then)
1	0x2027593c	0x7e8a6fa0	…*, (5, else)

* Entries prior to line 5

Checking Example


```
  5 if (Ti.thisProc() == 0)
0 6   Ti.barrier();
  7 else
1 8   Ti.barrier();
```

Meaningful error generated

Thread	Hash	Hash fron	MISALIGNMENT
0	0x7e8 ERROR	;a6fa0	…*, (5, then), …**
1	0x2027593c	0x7e8a6fa0	…*, (5, else), …**

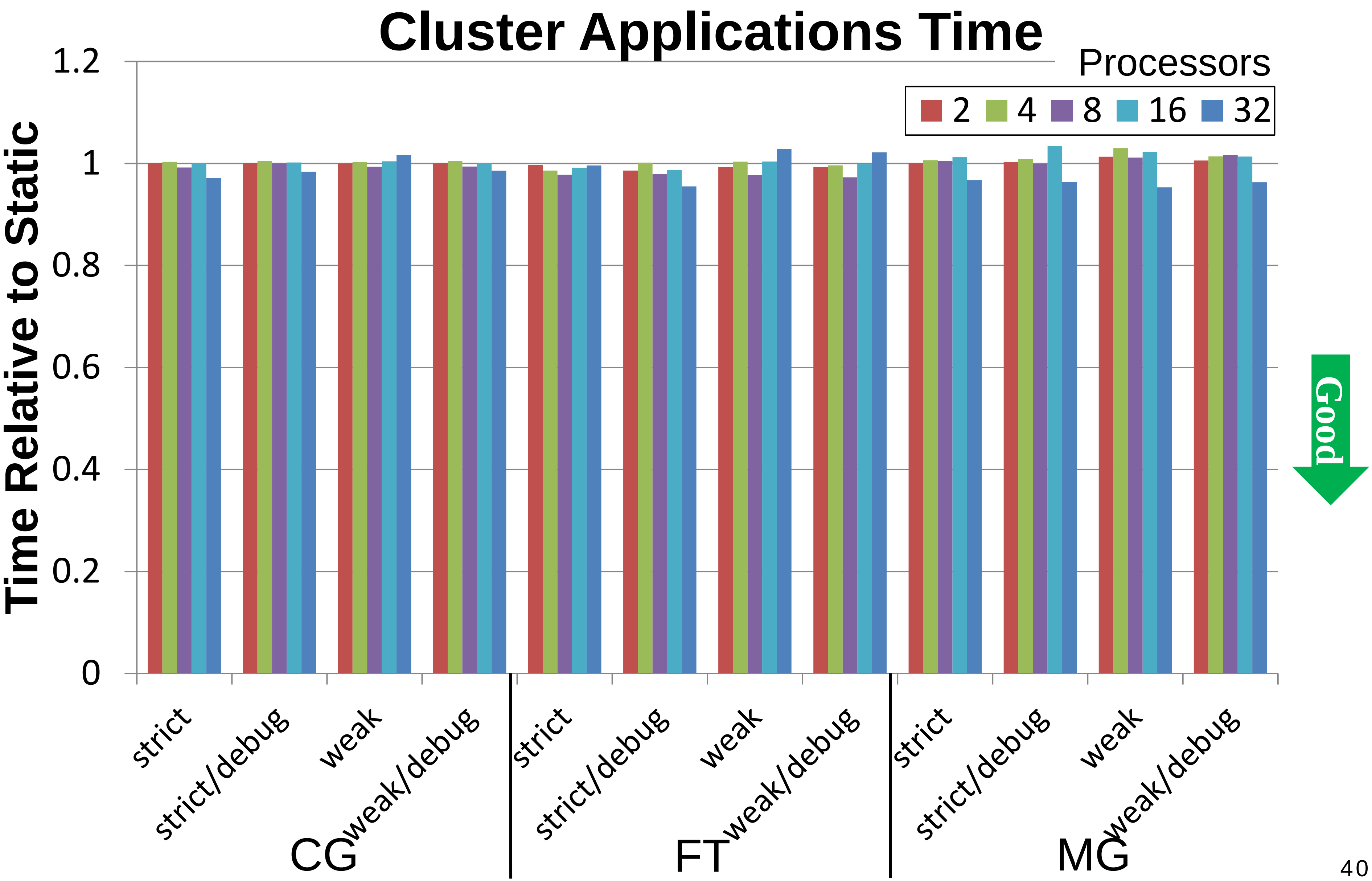
* Entries prior to line 5

Evaluation

- Performance tested on cluster of dual-processor 2.2GHz Opterons with InfiniBand interconnect
- Three NAS Parallel Benchmarks tested
  - Conjugate gradient (CG)
  - Fourier transform (FT)
  - Multigrid (MG)
- Enforcement variants

Name	Static or Dynamic	Debugging Information
static (baseline)	Static	N/A
strict	Dynamic	No
strict/debug	Dynamic	Yes
weak	Dynamic	No
weak/debug	Dynamic	Yes

Overhead of Dynamic Alignment is Minimal



Summary

- Dynamic checking removes annotation burden from programmers, works with teams
- Minimal performance impact on applications
- Dynamic checking can be applied to languages without strong type systems (e.g. UPC)

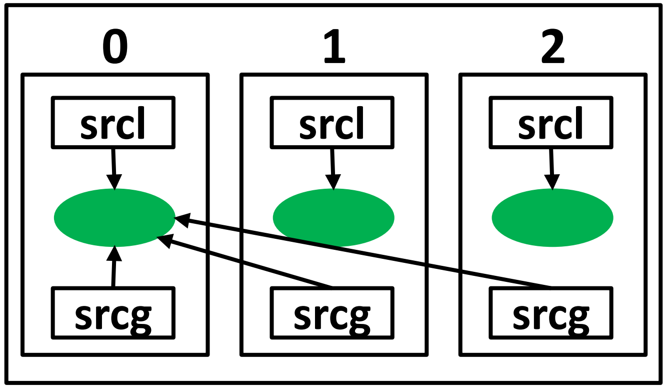
Outline

- Language Extensions
- Alignment of Collectives
- Pointer Analysis
- Application Case Studies
- Conclusions

Partitioned Global Address Space

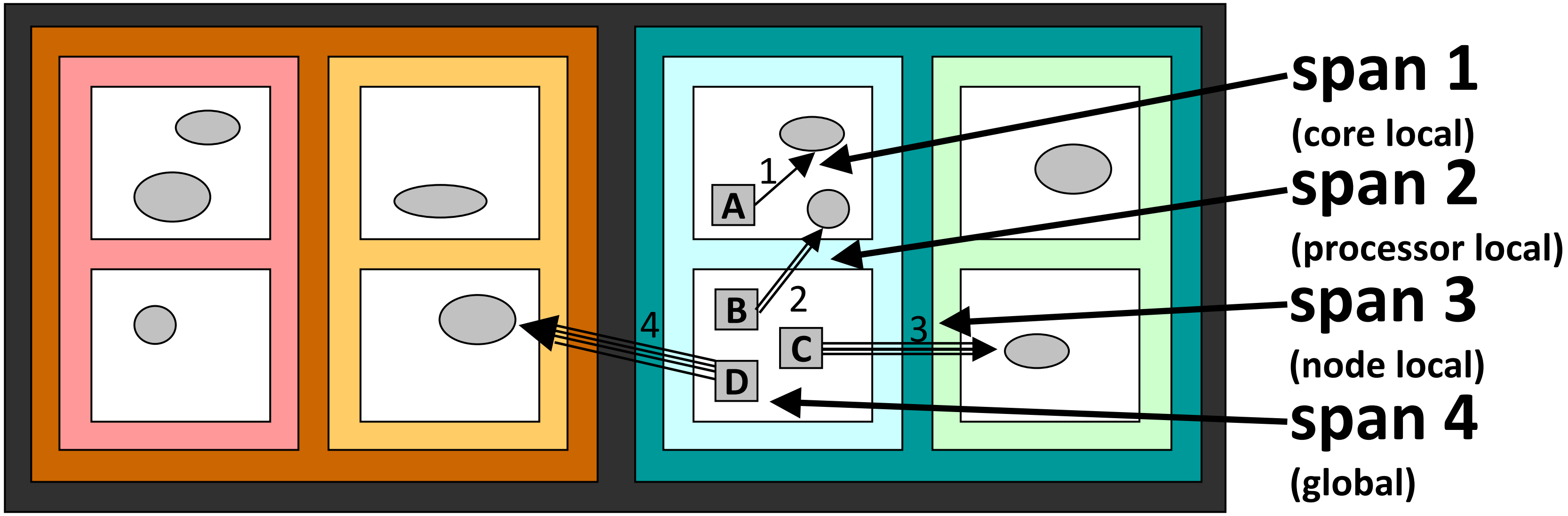
- Partitioned global address space (PGAS) abstraction provides illusion of shared memory on non-shared memory machines
- Pointers can reference local or remote data
  - Location of data can be reflected in type system
  - Runtime handles any required communication

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



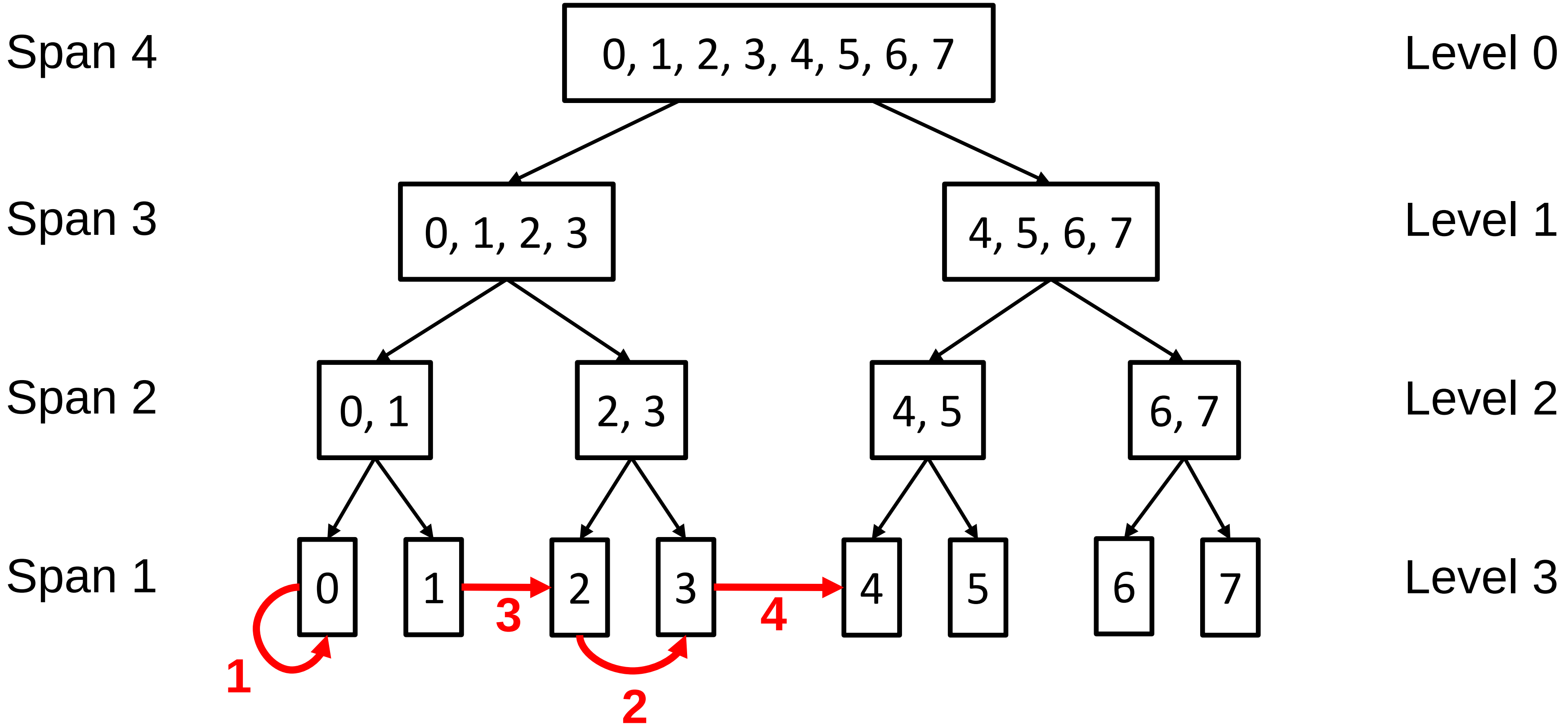
Hierarchical Memory

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



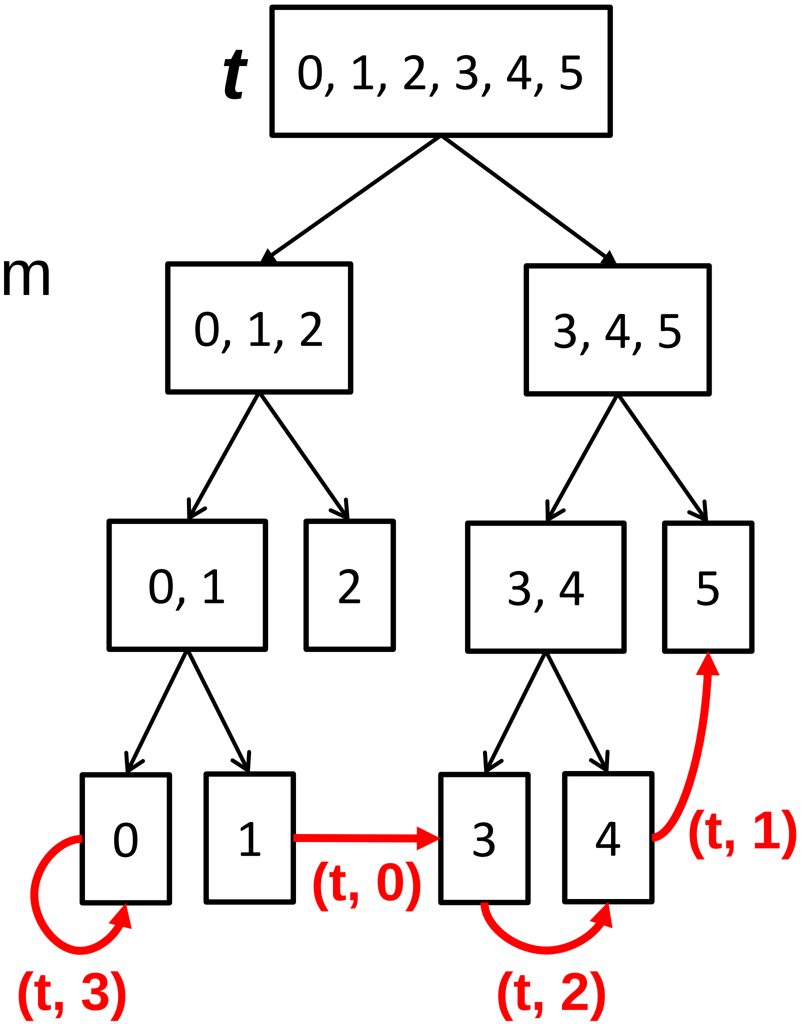
Pointer Span and Machine Team

- Span of pointer related to level of least common ancestor of the source thread and the potential targets in the machine hierarchy
  - *span = # of levels - target level*



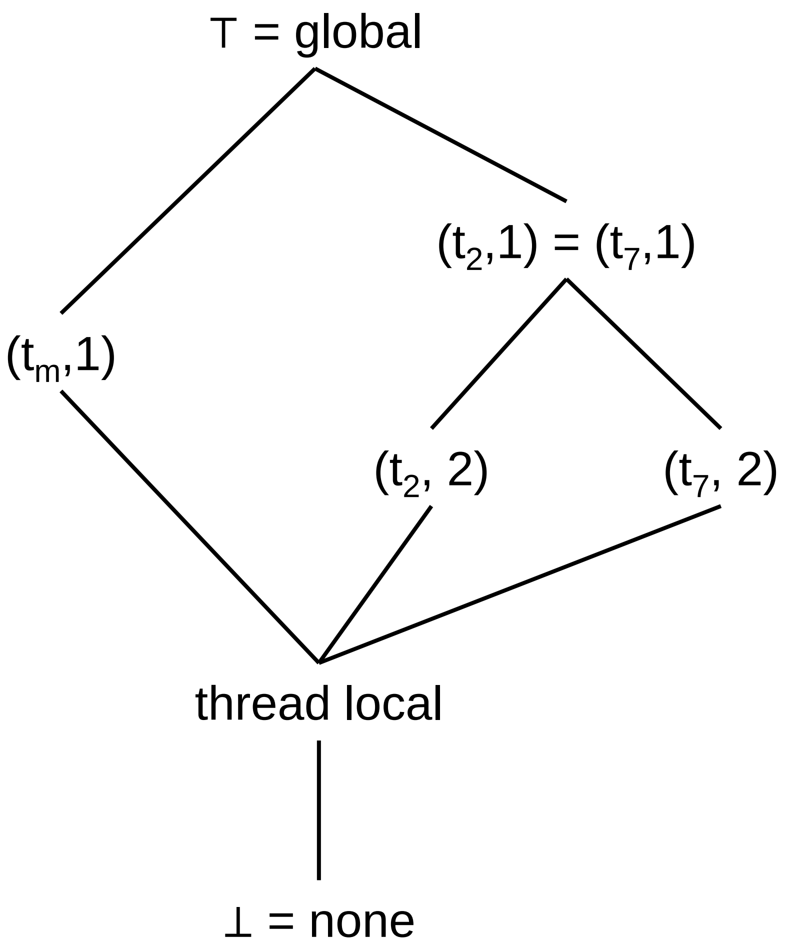
Pointers and Arbitrary Teams

- Pointer span can be generalized to handle arbitrary teams
  - "Span" of pointer is now the combination of a specific team hierarchy and a level in that hierarchy



Pointers and Multiple Teams

- Relationship between teams can be represented as a lattice
- Span of a pointer is an element of the lattice
- Pointer analysis can determine span of pointers



Hierarchical Pointer Analysis

- Pointer analysis possible over hierarchical teams
  - Allocation sites → *abstract locations (alocs)*
  - Variables → points-to sets of alocs
- Abstract locations have span (e.g. thread local, global)
- SPMD model simplifies analysis
  - Allows effects of an operation on all threads to be simultaneously computed
  - Results are the same for all threads

Pointer Analysis: Allocation

- Allocation creates new thread local abstract location
  - Result of allocation must reside in local memory

```
  static void bar() {
L1: Object b, a = new Object();
    teamsplit(t2) {
      b = broadcast a from 0;
    }
  }
```

Alocs	1

Points-to Sets	
a	(1, thread local)
b	

Pointer Analysis: Communication

- Communication produces version of source abstract locations with greater span
  - Collective takes into account team over which it is executed

```
   static void bar() {
L1: Object b, a = new Object();
    teamsplit(t2) {
      b = broadcast a from 0;
    }
  }
```

Alocs	1

Points-to Sets	
a	(1, thread local)
b	(1, (t_2, 1))

Evaluation

- Pointer analysis implemented for 3-level machine hierarchy
- Evaluated on five application benchmarks

Benchmark	Line Count	Description
amr	7581	Adaptive mesh refinement suite
gas	8841	Hyperbolic solver for a gas dynamics problem
cg	1595	NAS conjugate gradient benchmark
ft	1192	NAS Fourier transform benchmark
mg	1952	NAS multigrid benchmark

EECS
Electrical Engineering and Computer Sciences

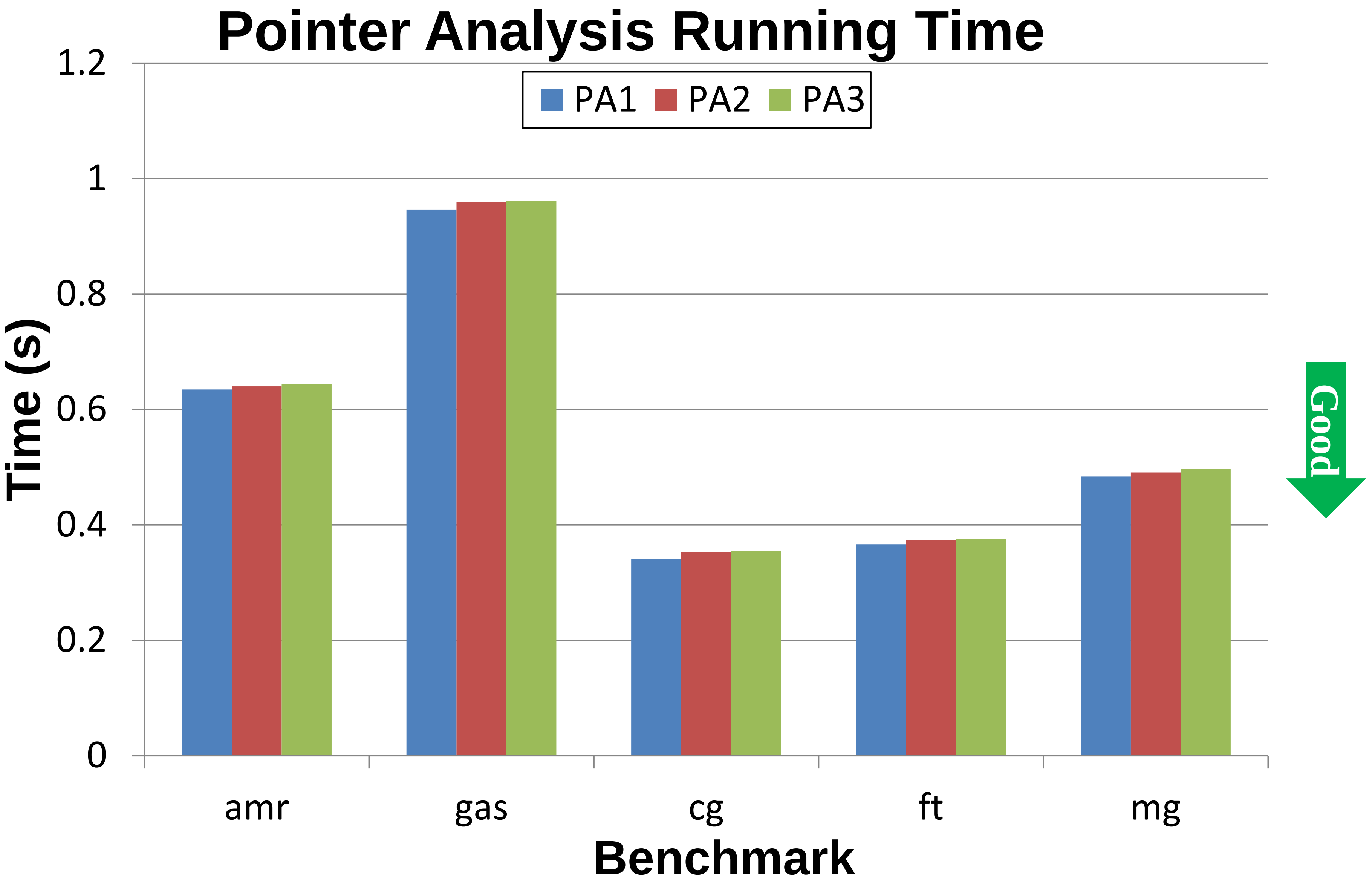
Running Time

BERKELEY PAR LAB

- Determine cost of introducing hierarchy into pointer analysis
- Tests run on 2.93GHz Core i7 with 8GB RAM
- Three analysis variants compared

Name	Description
PA1	Single-level pointer analysis
PA2	Two-level pointer analysis (thread-local and global)
PA3	Three-level pointer analysis

Low Overhead for Hierarchy

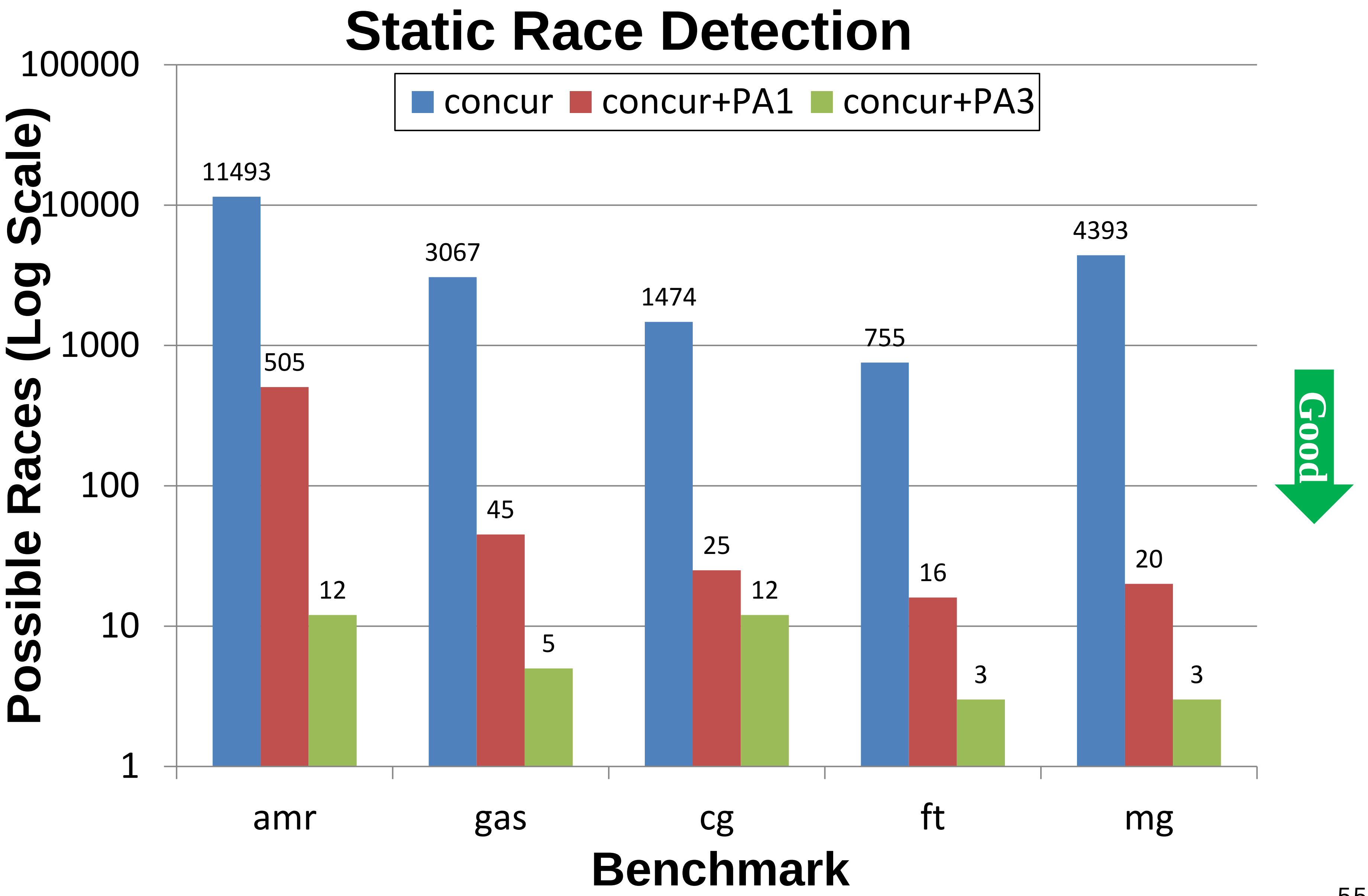


Race Detection

- Pointer analysis used with concurrency analysis to detect potential races at compile-time
- Three analyses compared

Name	Description
concur	Concurrency analysis plus constraint-based data sharing analysis and type-based alias analysis
concur+PA1	Concurrency analysis plus single-level pointer analysis
concur+PA3	Concurrency analysis plus three-level pointer analysis

More Precise Results



Outline

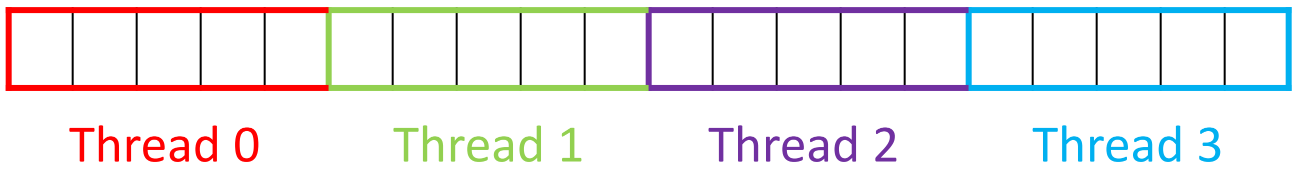
- Language Extensions
- Alignment of Collectives
- Pointer Analysis
- Application Case Studies
- Conclusions

Sorting

- Distributed sorting application using new hierarchical constructs
- Three pieces: sequential, shared memory, and distributed
  - Sequential: quick sort 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

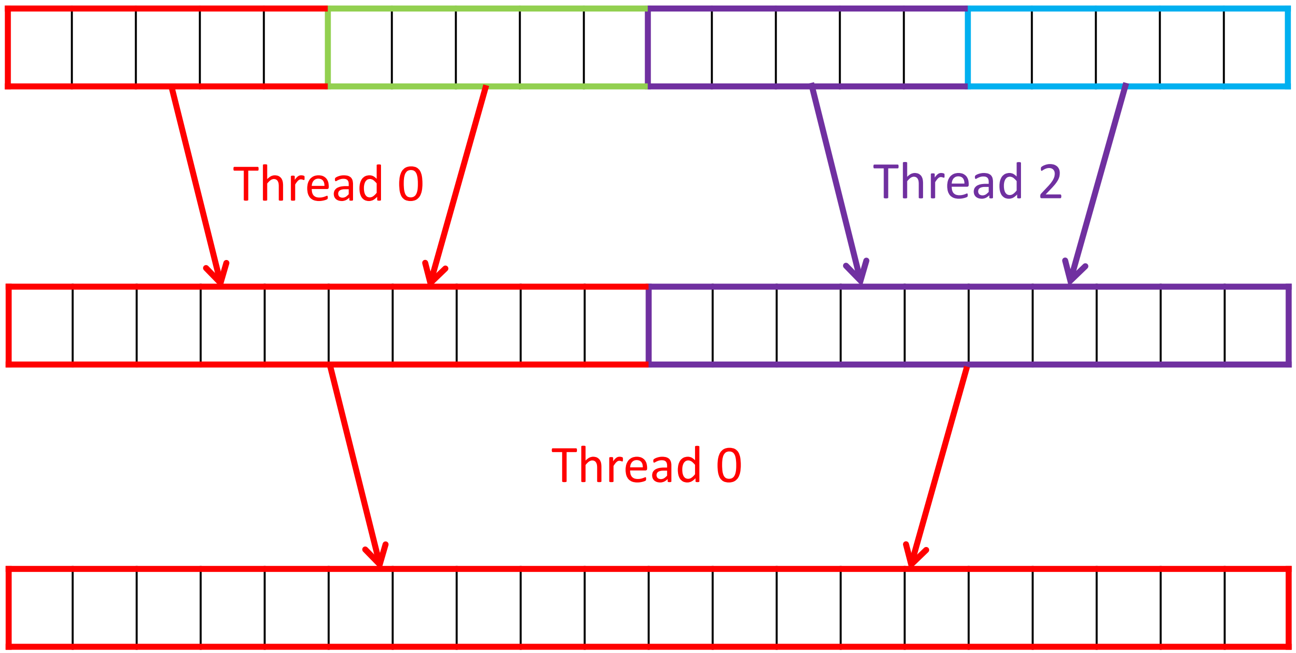
- Divide elements equally among threads



- Each thread calls sequential sort to process its elements

Shared Memory Merge

- Merge in parallel



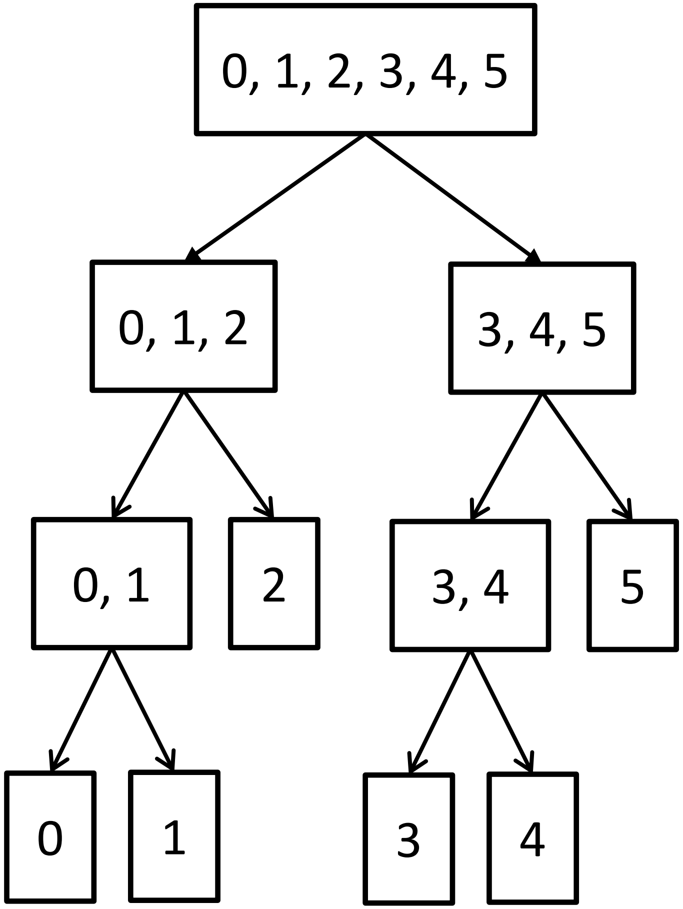
- Number of threads approximately halved in each iteration

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



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(t.myChildTeam());
    }
    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 Summary

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

Distributed Sort

- Existing unoptimized sample sort written 12 years ago by Kar Ming Tang
- Algorithm

Thread 0	Thread 1	Thread 2	Thread 3
49 3 88 4	94 79 0 59	98 3 45 89	78 77 30 31

  - Sampling to compute splitters

49 3 **88** 4	94 79 0 **59**	98 3 45 89	78 77 **30** 31

  - Redistribution

3 4 0 3	49 45 30 31	79 59 78 77	88 94 98 89

  - Local sort

0 3 3 4	30 31 45 49	59 77 78 79	88 89 94 98

CLUMPS Sort v0.1

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

EECS
Electrical Engineering and Computer Sciences

CLUMPS Sort v0.1 Code

BERKELEY PAR LAB

- Code for v0.1

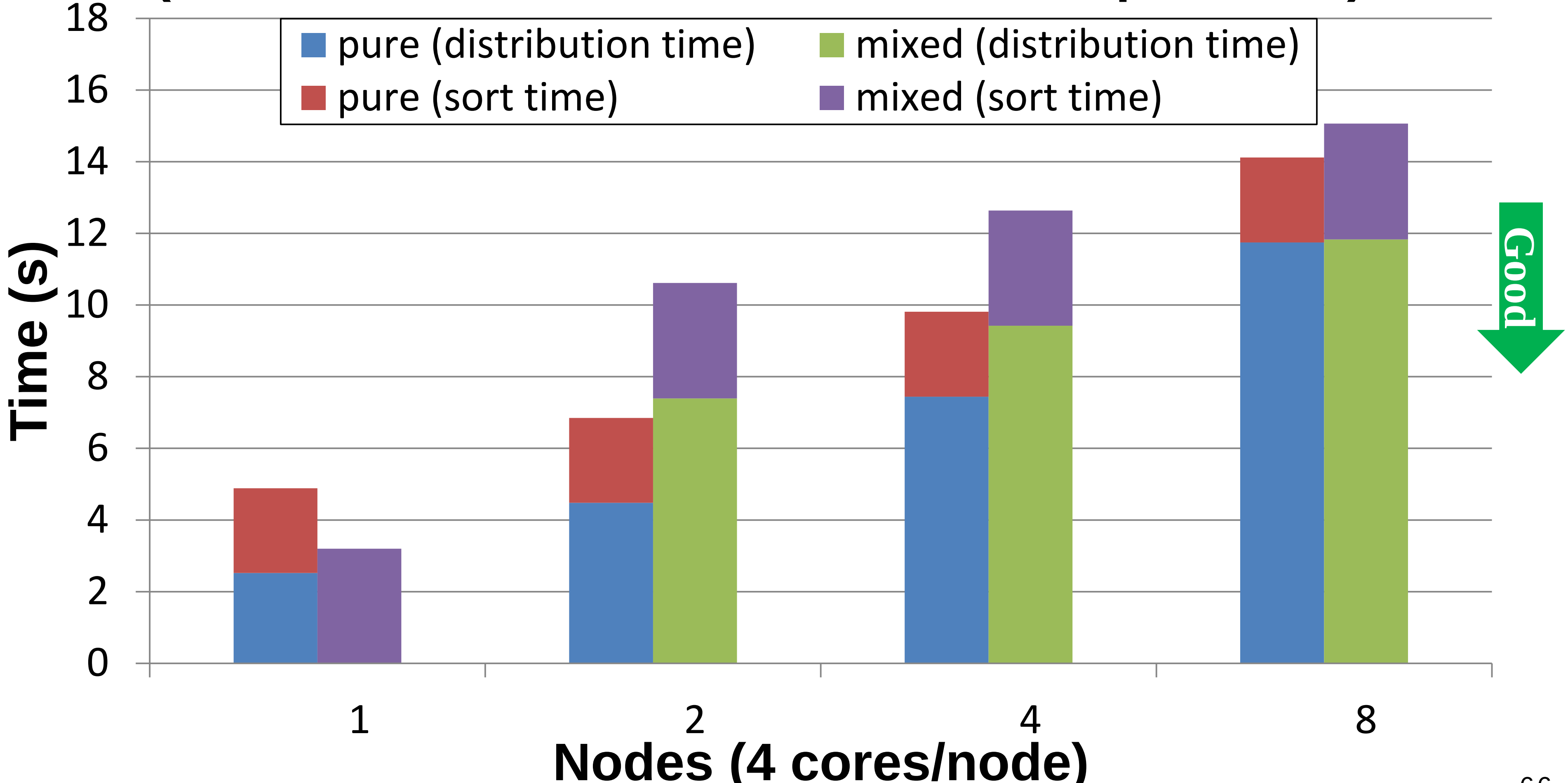
```
Team team = Ti.defaultTeam();
team.initialize(false);
Team smplTeam = team.makeTransposeTeam();
smplTeam.initialize(false);
partition(smplTeam) {
  { sampleSort(); }
}
teamsplit(team) {
  keys = SMPSort.parallelSort(keys);
}
```

- 10 lines of code, 5 minutes to solution!

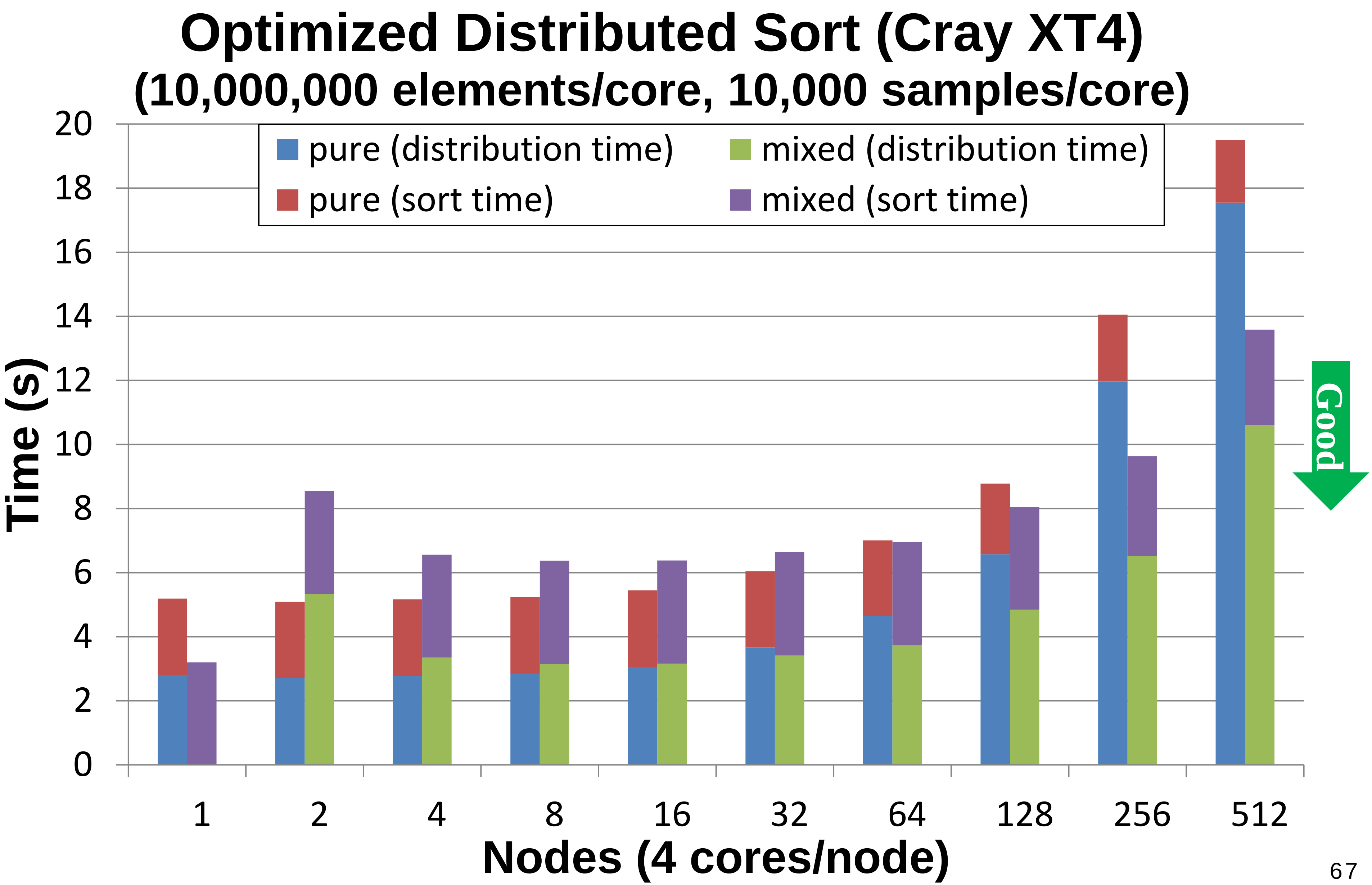
CLUMPS Sort v0.1 Results

- And it works!

Initial Distributed Sort (Cray XT4)
(10,000,000 elements/core, 10,000 samples/core)

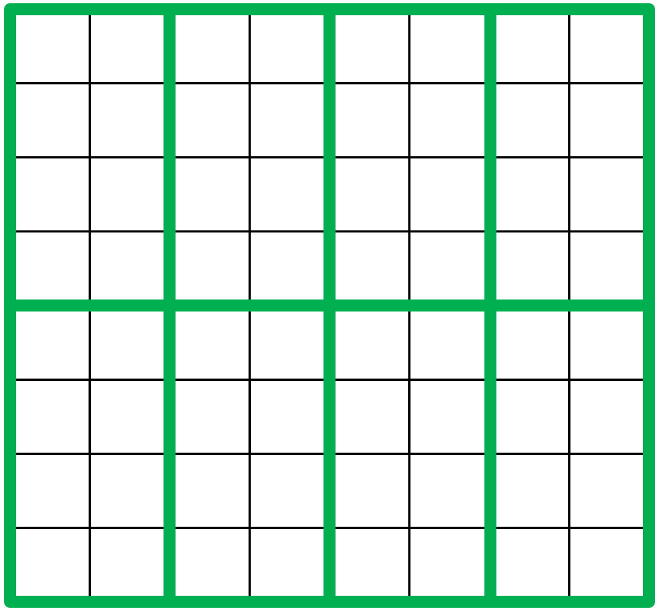


Optimized CLUMPS Sort



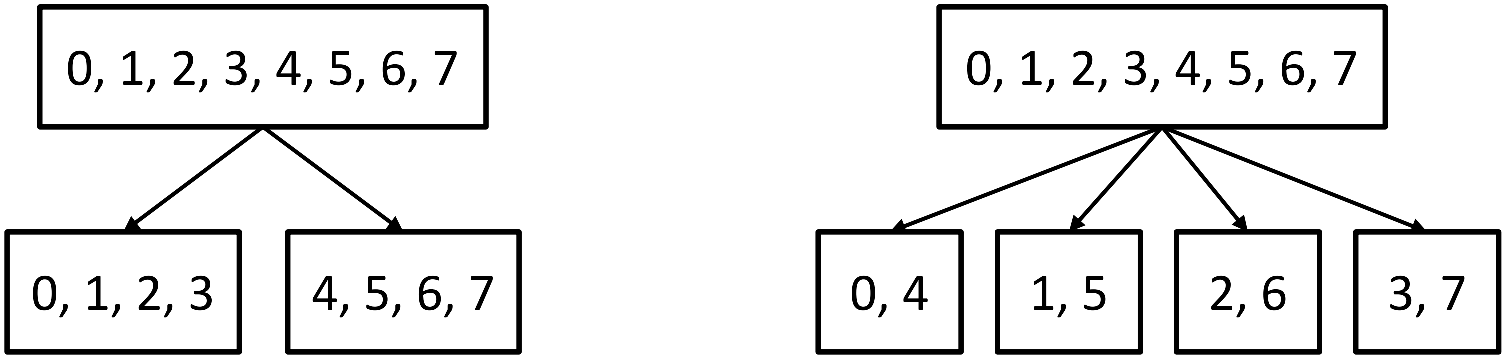
Conjugate Gradient

- NAS conjugate gradient (CG) application written and optimized by Kaushik Datta
- Includes parallel sparse matrix-vector multiplies
  - Randomly generated matrix has no special structure
  - Divided in both row and column dimensions
  - Reductions over row threads
  - Broadcasts over column threads
- Without teams, Kaushik had to hand-roll collectives



Row and Team Hierarchies

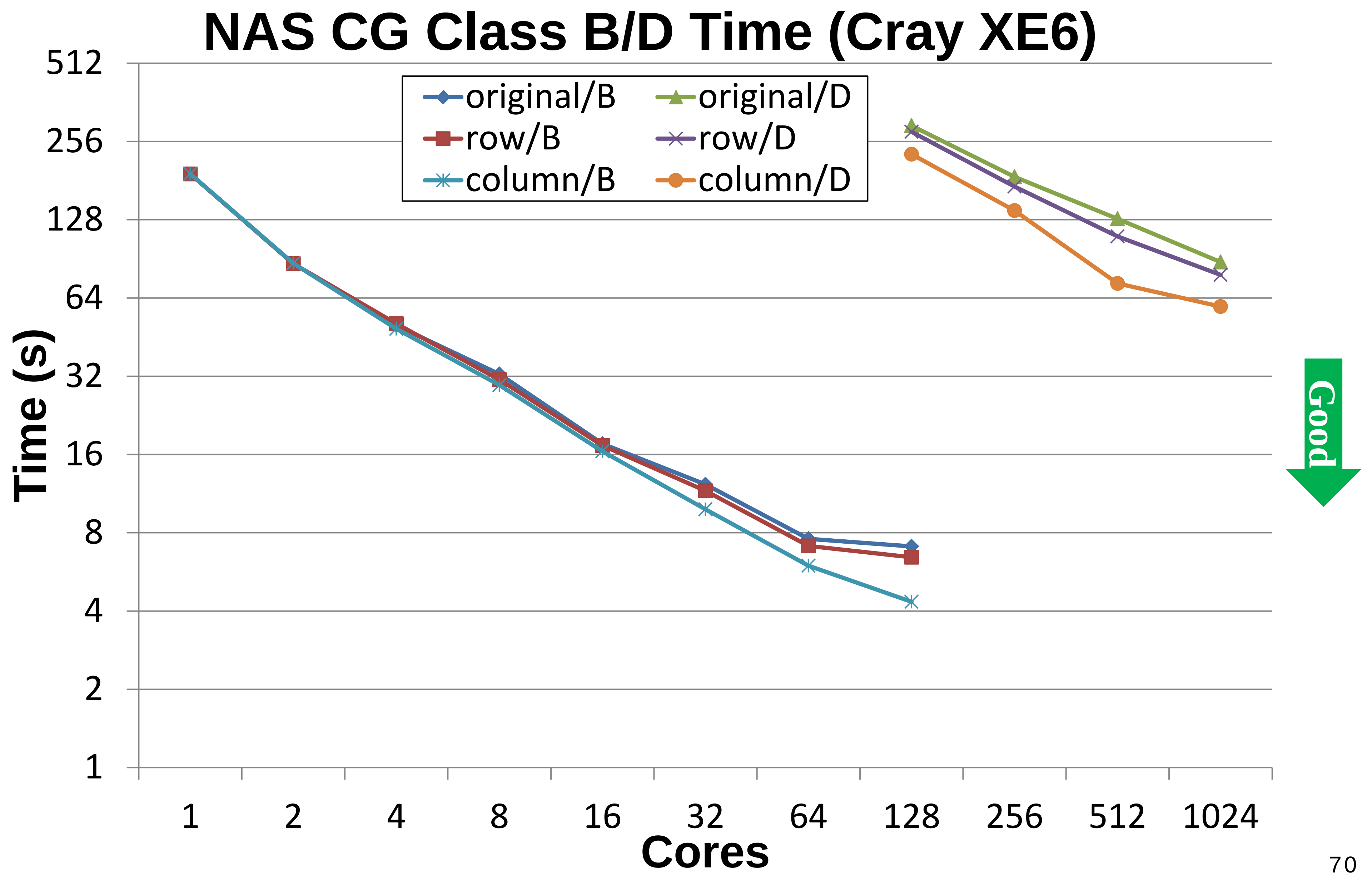
- Both row and column teams needed



- Team code for reductions and broadcasts

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

CG Running Time



Outline

- Language Extensions
- Alignment of Collectives
- Pointer Analysis
- Application Case Studies
- Conclusions

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
- Language extensions are safe to use and easy to analyze
  - Safety provided by lexical scoping and dynamic alignment checking
  - Simple pointer analysis that takes into account machine and algorithmic hierarchy

This slide intentionally left blank.