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
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

Slow, avoid

Fast, allow
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)
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

```java
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
- Print

Barrier

- Print

Program End
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];
    ```
Data parallelism allows even simpler expression of global operations

\[
\text{forall } (i \text{ in } C.\text{domain}()) \quad C[i] = A[i] + B[i];
\]

\[
C = A + B;
\]

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
Algorithm Example: Merge Sort

- 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**

```java
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**

```java
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**

```java
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);
}
```
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

```java
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
Contributions

- 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();
```
Thread teams may execute distinct tasks

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

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

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

Lexical scope prevents some types of deadlock
  - Execution team determined by enclosing construct
Different subteams of $T$ execute each of the branches

```cpp
partition(T) {
    { model_fluid(); }
    { model_muscles(); }
    { model_electrical(); }
}
```
Teamsplit Semantics

- Each subteam of rowTeam executes the reduction on its own

```java
teamsplit(rowTeam) {
    Reduce.add(mtmp, myResults0, rpivot);
}
```
Multiple Hierarchy Levels

- Constructs can be nested

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

- Program can use multiple teams

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

- Language Extensions
- Alignment of Collectives
- Pointer Analysis
- Application Case Studies
- Conclusions
Many parallel languages make no attempt to ensure that collectives line up

- Example code that will compile but deadlock:
  ```c
  if (Ti.thisProc() % 2 == 0)
    Ti.barrier(); // even ID threads
  else
    ; // odd ID threads
  int i = broadcast Ti.thisProc() from 0;
  ```
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
Different schemes can be used to enforce textual alignment

<table>
<thead>
<tr>
<th>Type system</th>
<th>Programmer burden</th>
<th>Restrictions on program structure</th>
<th>Early error detection</th>
<th>Accuracy/Precision</th>
<th>Performance reduction</th>
<th>Team support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static inference</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic checks</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>No checking</td>
<td>None</td>
<td>None</td>
<td>No</td>
<td>None</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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
5  if (Ti.thisProc() == 0)
6  Ti.barrier();
7  else
8  Ti.barrier();

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0dc7637a</td>
<td>...,*</td>
</tr>
<tr>
<td>1</td>
<td>0x0dc7637a</td>
<td>...,*</td>
</tr>
</tbody>
</table>

* Entries prior to line 5
5  if (Ti.thisProc() == 0)
6    Ti.barrier();
7  else
8    Ti.barrier();

Control flow decision noted, hash updated

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8a6fa0</td>
<td>...*, (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td>...*, (5, else)</td>
</tr>
</tbody>
</table>

* Entries prior to line 5
Checking Example

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

Hash broadcast from thread 0

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8a6fa0</td>
<td></td>
<td>...*, (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td></td>
<td>...*, (5, else)</td>
</tr>
</tbody>
</table>

* 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

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8a6fa0</td>
<td>0x7e8a6fa0</td>
<td>…*, (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td>0x7e8a6fa0</td>
<td>…*, (5, else)</td>
</tr>
</tbody>
</table>

* 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

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from 0</th>
<th>Execution History</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8</td>
<td>ERROR</td>
<td>a6fa0</td>
</tr>
<tr>
<td></td>
<td>ERROR</td>
<td></td>
<td>...*, (5, then)</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td>0x7e8a6fa0</td>
<td>...*, (5, else)</td>
</tr>
</tbody>
</table>

* Entries prior to line 5
Checking Example

5 if (Ti.thisProc() == 0)

6 Ti.barrier();

7 else

8 Ti.barrier();

Meaningful error generated

<table>
<thead>
<tr>
<th>Thread</th>
<th>Hash</th>
<th>Hash from</th>
<th>MISALIGNMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x7e8</td>
<td>a6fa0</td>
<td><em>(5, then)</em> **</td>
</tr>
<tr>
<td>1</td>
<td>0x2027593c</td>
<td>0x7e8a6fa0</td>
<td><em>(5, else)</em> **</td>
</tr>
</tbody>
</table>

* 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Static or Dynamic</th>
<th>Debugging Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>static (baseline)</td>
<td>Static</td>
<td>N/A</td>
</tr>
<tr>
<td>strict</td>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>strict/debug</td>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>weak</td>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>weak/debug</td>
<td>Dynamic</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Overhead of Dynamic Alignment is Minimal

Cluster Applications Time

<table>
<thead>
<tr>
<th>Processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time Relative to Static

Good
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 (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

```java
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
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**
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

Diagram:

- $\top = \text{global}$
- $(t_2, 1) = (t_7, 1)$
- $(t_2, 2)$
- $(t_7, 2)$
- $\bot = \text{none}$

- $T = \text{global}$
- $(t_m, 1)$
- thread local
Hierarchical Pointer Analysis

- Pointer analysis possible over hierarchical teams
  - Allocation sites $\rightarrow$ abstract locations (alocs)
  - Variables $\rightarrow$ 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

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

<table>
<thead>
<tr>
<th>Alocs</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>(1, thread local)</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>
Communication produces version of source abstract locations with greater span

- Collective takes into account team over which it is executed

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

<table>
<thead>
<tr>
<th>Alocs</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to Sets</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>(1, thread local)</td>
</tr>
<tr>
<td>b</td>
<td>(1, (t2, 1))</td>
</tr>
</tbody>
</table>
Evaluation

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Line Count</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>amr</td>
<td>7581</td>
<td>Adaptive mesh refinement suite</td>
</tr>
<tr>
<td>gas</td>
<td>8841</td>
<td>Hyperbolic solver for a gas dynamics problem</td>
</tr>
<tr>
<td>cg</td>
<td>1595</td>
<td>NAS conjugate gradient benchmark</td>
</tr>
<tr>
<td>ft</td>
<td>1192</td>
<td>NAS Fourier transform benchmark</td>
</tr>
<tr>
<td>mg</td>
<td>1952</td>
<td>NAS multigrid benchmark</td>
</tr>
</tbody>
</table>
Running Time

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA1</td>
<td>Single-level pointer analysis</td>
</tr>
<tr>
<td>PA2</td>
<td>Two-level pointer analysis (thread-local and global)</td>
</tr>
<tr>
<td>PA3</td>
<td>Three-level pointer analysis</td>
</tr>
</tbody>
</table>
Low Overhead for Hierarchy

**Pointer Analysis Running Time**

- **Benchmark**
  - amr
  - gas
  - cg
  - ft
  - mg

- **Time (s)**
  - PA1
  - PA2
  - PA3

- **Results**
  - Good
Race Detection

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>concur</td>
<td>Concurrency analysis plus constraint-based data sharing analysis and type-based alias analysis</td>
</tr>
<tr>
<td>concur+PA1</td>
<td>Concurrency analysis plus single-level pointer analysis</td>
</tr>
<tr>
<td>concur+PA3</td>
<td>Concurrency analysis plus three-level pointer analysis</td>
</tr>
</tbody>
</table>
More Precise Results

Static Race Detection

Possible Races (Log Scale)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>concur</th>
<th>concur+PA1</th>
<th>concur+PA3</th>
</tr>
</thead>
<tbody>
<tr>
<td>amr</td>
<td>11493</td>
<td>505</td>
<td>12</td>
</tr>
<tr>
<td>gas</td>
<td>3067</td>
<td>45</td>
<td>5</td>
</tr>
<tr>
<td>cg</td>
<td>1474</td>
<td>25</td>
<td>12</td>
</tr>
<tr>
<td>ft</td>
<td>755</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>mg</td>
<td>4393</td>
<td>20</td>
<td>3</td>
</tr>
</tbody>
</table>

Good
Outline

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

- Number of threads approximately halved in each iteration
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
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);
        }
    }
}
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
  - Sampling to compute splitters
  - Redistribution
  - Local sort
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
Code for v0.1

```java
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!
And it works!

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

- pure (distribution time)
- mixed (distribution time)
- pure (sort time)
- mixed (sort time)
Optimized Distributed Sort (Cray XT4)
(10,000,000 elements/core, 10,000 samples/core)

- Pure (distribution time)
- Mixed (distribution time)
- Pure (sort time)
- Mixed (sort time)

Time (s)

Nodes (4 cores/node)

- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128
- 256
- 512

Good
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
Both row and column teams needed

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

NAS CG Class B/D Time (Cray XE6)

- original/B
- original/D
- row/B
- row/D
- column/B
- column/D

Time (s) vs. Cores

- Good
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.