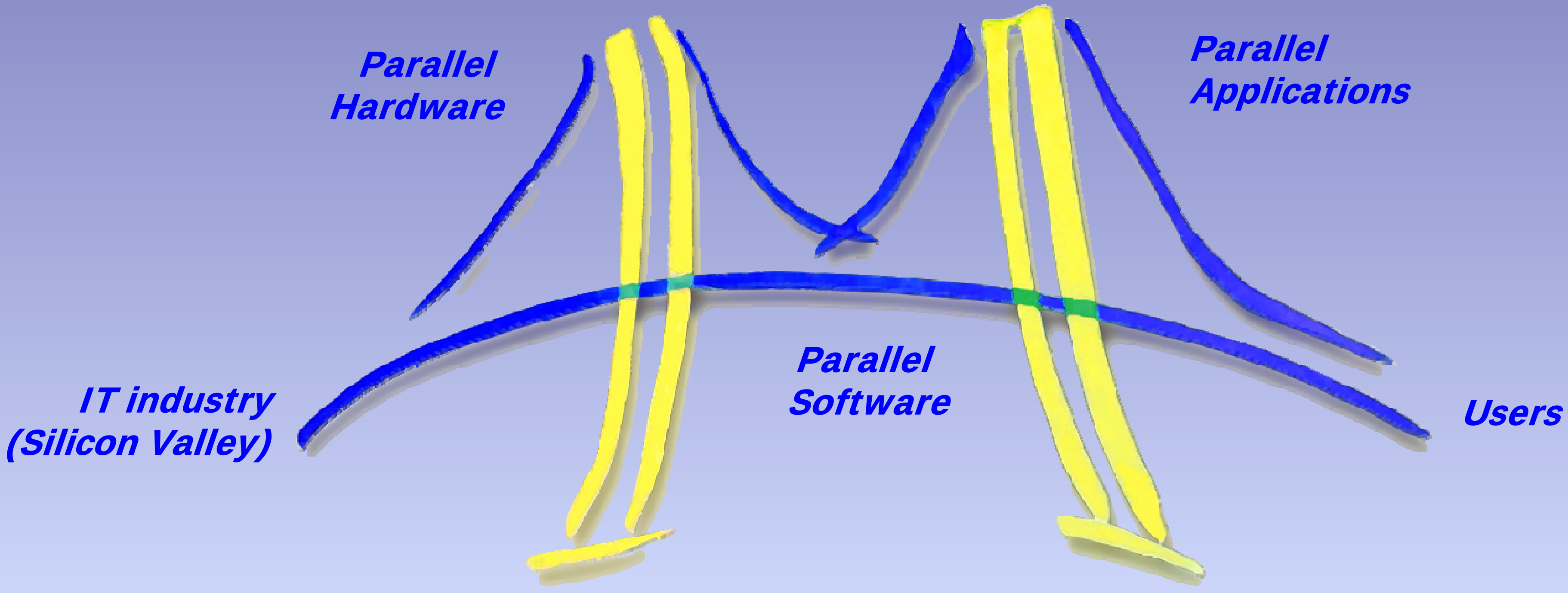


Enforcing Textual Alignment of Collectives using Dynamic Checks

Amir Kamil and Katherine Yelick
UC Berkeley Parallel Computing Laboratory

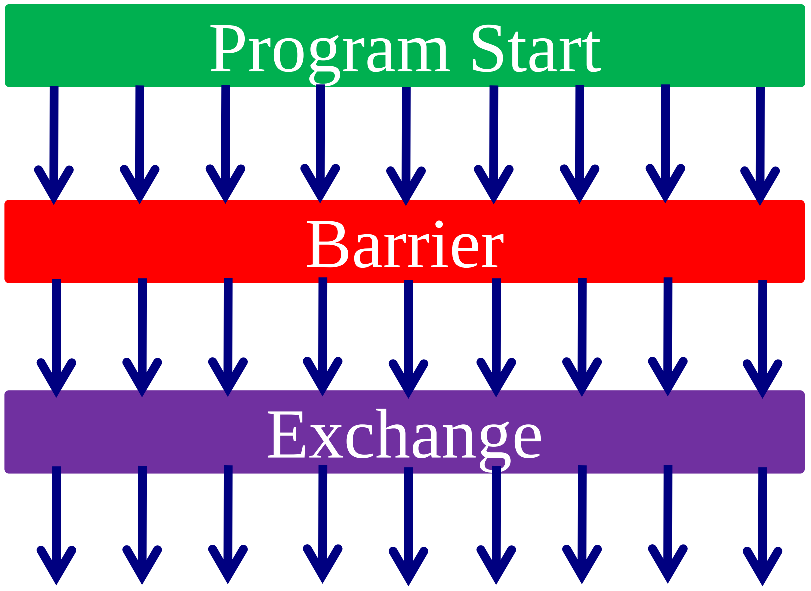
October 10, 2009

Overview

- Synchronization analysis is critical to many analyses and optimizations
  - Race detection, lock elimination, memory model enforcement
- *Collective operations* used by many parallel programs for communication/synchronization
  - Operations that are executed collectively by all or a subset of all threads
- Alignment restrictions on collectives affect programmability and analyzability
- Dynamic alignment checking increases both

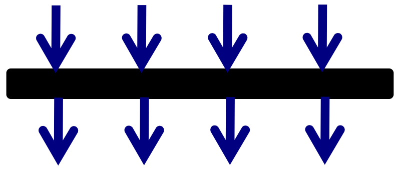
Titanium Language

- Titanium is a high performance dialect of Java
- *Partitioned global address space* memory model
- Uses *single-program, multiple data* model of parallelism

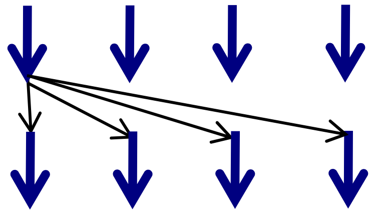


Collective Operations

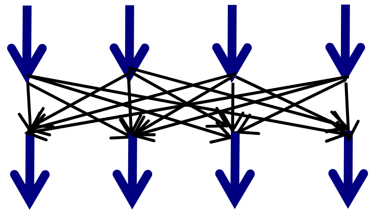
- Collectives used for synchronization and synchronous communication
- *Barrier*: all threads must reach it before any can proceed



- *Broadcast*: one to all communication



- *Exchange*: all to all communication



Collective Alignment

- Many parallel languages make no attempt to ensure that collectives line up
  - Example code that is legal but will 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

- Titanium has *textual collectives*: all threads must execute the same *textual* sequence of collectives
  - Stronger guarantee than structural correctness – this example is illegal:

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

- Language semantics statically guarantee textual alignment

Control Flow Restrictions

- A statement *may have global effects* if it or any substatement is a collective operation or calls a method declared as global
- Textual alignment requires the following
  - If any branch of a conditional may have global effects, then all threads must take the same branch
  - If the body/test of a loop may have global effects, then all threads must execute the same number of iterations

Single-Valued Expressions

- A *single-valued* expression has *coherent* values on all threads when evaluated
  - Example: `Ti.numProcs() > 1`
- All threads guaranteed to take the same branch of a conditional guarded by a single-valued expression
  - Only such conditionals may have collectives

```
if (Ti.numProcs() > 1)
  Ti.barrier(); // multiple threads
else
  ; // only one thread
```

THE `single` TYPE SYSTEM

- Titanium's **`single`** type system determines which expressions are single-valued
- Basic rule is that single-valued expressions may only be computed from other single-valued expressions
- Literals and certain constant expressions (e.g. **`Ti.numProcs()`**) are single-valued
- Rules for method calls, objects/fields, and exceptions can be very complicated

Drawbacks to **`single`**

- Type system is cumbersome and difficult to understand
  - Bugs found as recently as 2006!
- Annotations put a burden on programmer
- Type system still has problems
  - Unchecked casts to **`single`**
  - **`single`** has incomplete meaning when applied to array-based container
    - e.g. an object of type **`String single`** only has the same length on all threads, but not necessarily the same contents
- No natural way to extend it to allow collectives on thread subsets

Alignment Checking Schemes

- Comparison of different alignment schemes

	Programmer burden	Restrictions on program structure	Early error detection	Accuracy/ precision	Compiler/ runtime complexity	Performance reduction	Subset support
Type system	**High**	**High**	**High**	**High**	**Medium**	**No**	**No**
Static analysis	**Low**	**Medium**	**High**	**Medium**	**High**	**No**	**Yes**
Dynamic checks	**Low**	**High**	**Medium**	**High**	**Low**	**Yes**	**Yes**
No checking	**Low**	**Low**	**None**	**None**	**None**	**No**	**Yes**

Dynamic Enforcement

- A dynamic enforcement scheme can reduce programmer burden but still provide accurate results for analysis and optimization
- Basic idea:
  - Track control flow on all threads
  - Prior to performing a collective, check that preceding control flow matches on all threads
- Compiler instruments source code to perform tracking and checking

Alignment Tracking

- Track conditionals and loops at runtime
  - Only statements that may have global effects need be tracked
- Execution history is saved on each thread
  - Only in debugging mode to generate better error messages
- A running hash is computed for each thread that summarizes its execution

Tracking Example

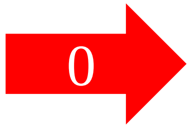
0, 1

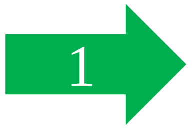
```
5 if (Ti.thisProc() == 0)
6   globalMethod();
7 else
8   globalMethod();
9 Ti.barrier();
```

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

* Entries prior to line 5

Tracking Example





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

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

* Entries prior to line 5

Tracking Example

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

0, 1

Thread	Hash	Execution History
0	0x307ea68d	...*, (5, then), ...**
1	0xfe4a0bc3	...*, (5, else), ...**

* Entries prior to line 5   ** Entries from call to **`globalMethod()`**

Alignment Checking

- Thread alignment is compared prior to each global operation
- Hash value is broadcast from thread 0
  - If any thread's value differs, error generated
- In debugging mode, saved history used to determine location in code where control flow diverged
  - Saved history can be erased at each collective, since preceding control flow must match for collective to execute

Checking Example

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

0, 1

Thread	Hash	Hash from 0	Execution History
0	0x307ea68d		...*, (5, then), ...**
1	0xfe4a0bc3		...*, (5, else), ...**

* Entries prior to line 5 ** Entries from call to **`globalMethod()`**

Checking Example

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

0, 1 → 9

Thread	Hash	Hash from 0	Execution History
0	0x307ea68d	0x307ea68d	...*, (5, then), ...**
1	0xfe4a0bc3	0x307ea68d	...*, (5, else), ...**

* Entries prior to line 5 ** Entries from call to **`globalMethod()`**

Checking Example

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

0, 1 → 9

Threa d	Hash	Hash from 0	Execution History
0	0x307 ERROR	ea68d	…*, (5, then), …**
1	0xfe4a0bc3	0x307ea68d	…*, (5, else), …**

* Entries prior to line 5 ** Entries from call to **`globalMethod()`**

Checking Example

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

0, 1

MISALIGNMENT

ERROR

Thread	Hash	Hash from	
0	0x307	ea68d	...*, (5, then), ...**
1	0xfe4a0bc3	0x307ea68d	...*, (5, else), ...**

* Entries prior to line 5 ** Entries from call to `globalMethod()`

Example

```
5 if (Ti.thisProc() == 0) // misaligned
6   globalMethod();
7 else
8   globalMethod();
9 Ti.barrier(); // failure
```

- Error message (debugging mode)

  ti.lang.Alignment.AlignmentError: collective alignment failed on processor 1 at foo.java:9:8

  last location: else branch at foo.java:5:12

  last location on processor 0: then branch at foo.java:5:12

  previous location: none

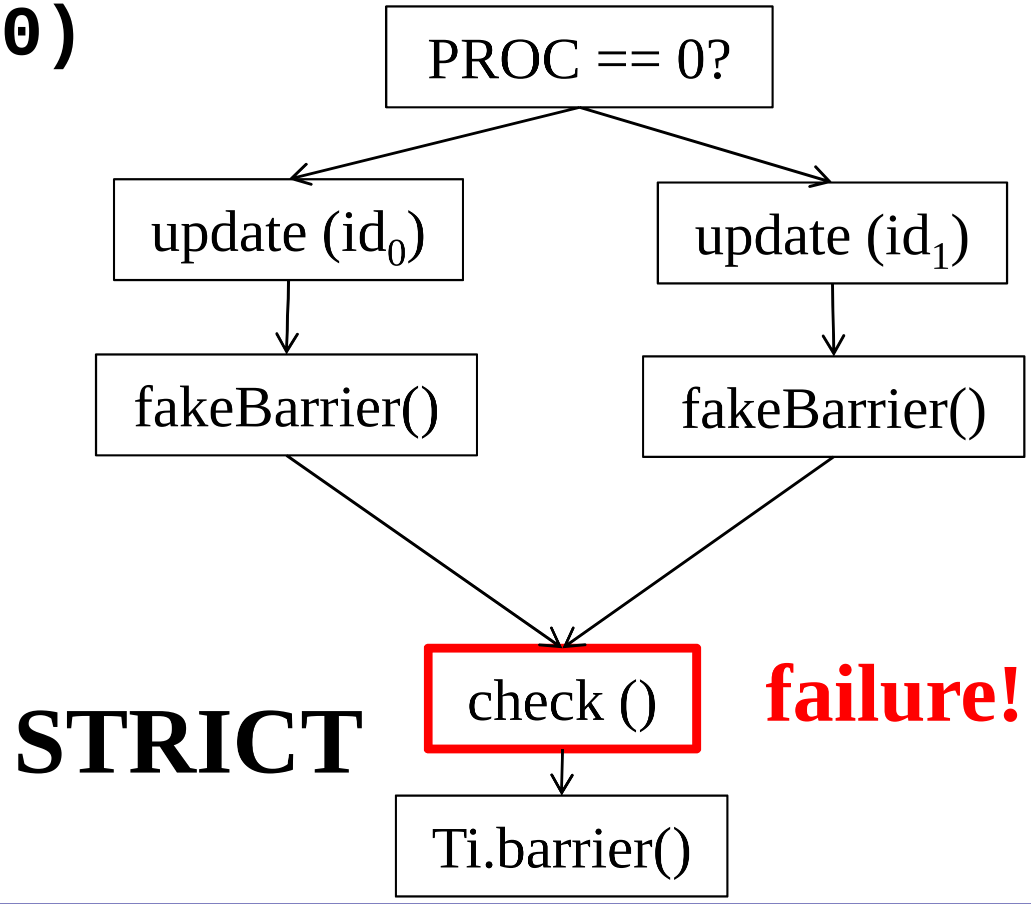
Strict vs. Weak Alignment

- Strict alignment guarantees alignment of statements that statically *may have* global effects
- Weak alignment only guarantees alignment of statements that dynamically execute collective operations
  - Under weak alignment, if a tracked statement does not actually execute a collective, then it must be erased from the hash and execution history after completing

Strict vs. Weak Example

- The following fails under strict alignment but succeeds under weak alignment:

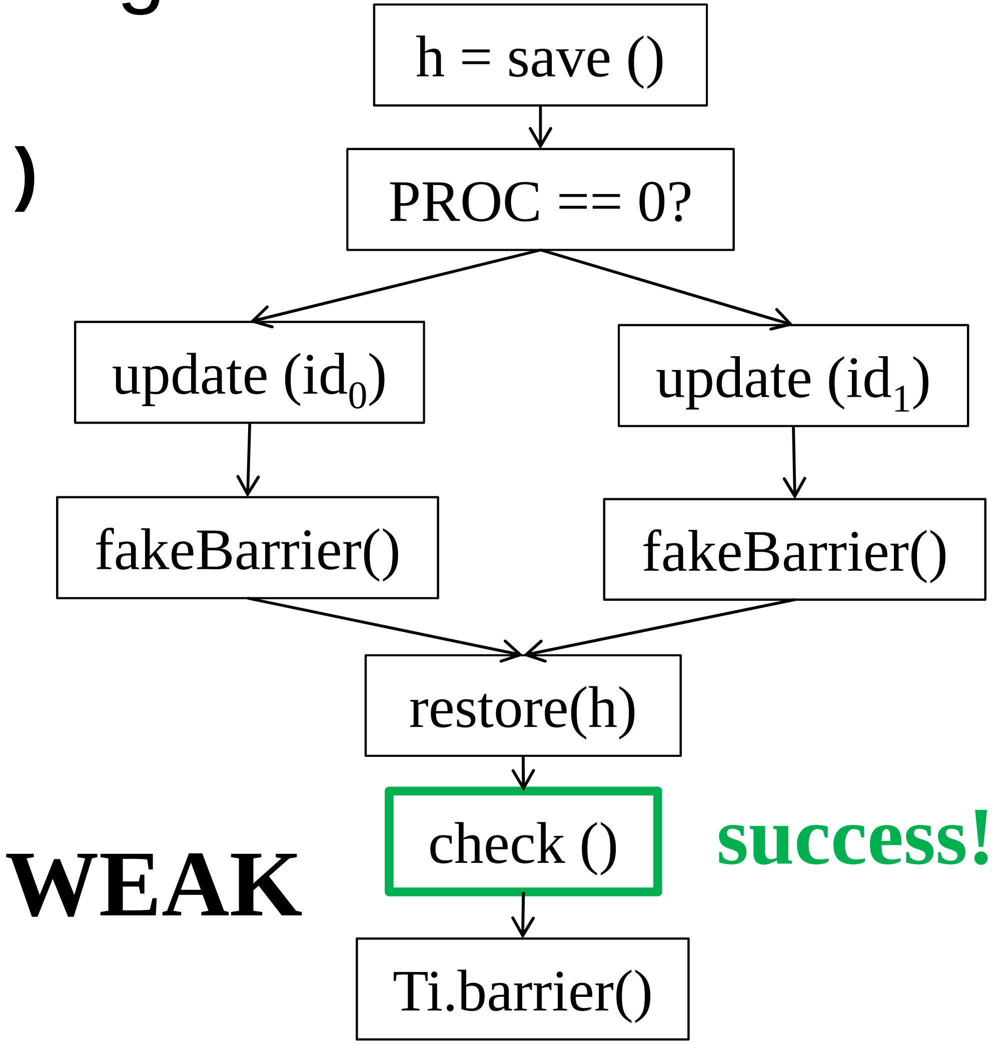
```
if (Ti.thisProc() == 0)
  fakeBarrier();
else
  fakeBarrier();
Ti.barrier();
```



Strict vs. Weak Example

- The following fails under strict alignment but succeeds under weak alignment:

```
if (Ti.thisProc() == 0)
  fakeBarrier();
else
  fakeBarrier();
Ti.barrier();
```



Strict or Weak Alignment?

- Advantages of strict alignment
  - Strict alignment more closely matches current semantics
  - Easier to reason about whether or not an operation may have global effects than if it actually executes a collective
  - Strict alignment performs fewer operations since it does not save and restore the hash
- Advantages of weak alignment
  - Accepts code that is rejected under weak alignment

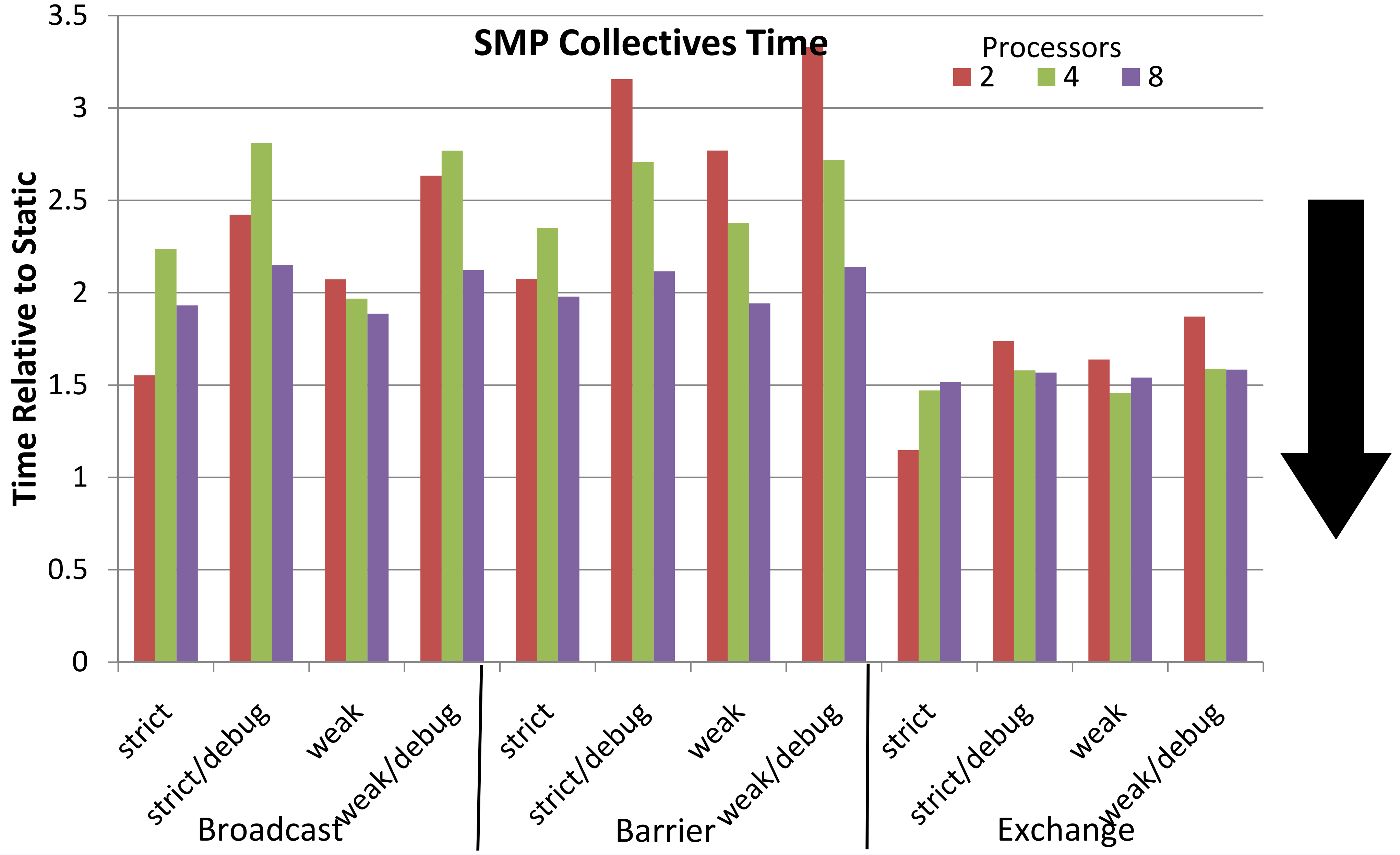
Evaluation

- Performance tested on two machines
  - Eight-core (2x4) Intel Xeon E5435 2.33GHz SMP
  - Cluster of dual-processor 2.2GHz Opterons with InfiniBand interconnect
- Three primitive collective operations tested
  - Broadcast: one-to-all communication
  - Barrier: threads wait until all have reached it
  - Exchange: all-to-all communication
- Three NAS Parallel Benchmarks tested
  - Conjugate gradient (CG)
  - Fourier transform (FT)
  - Multigrid (MG)

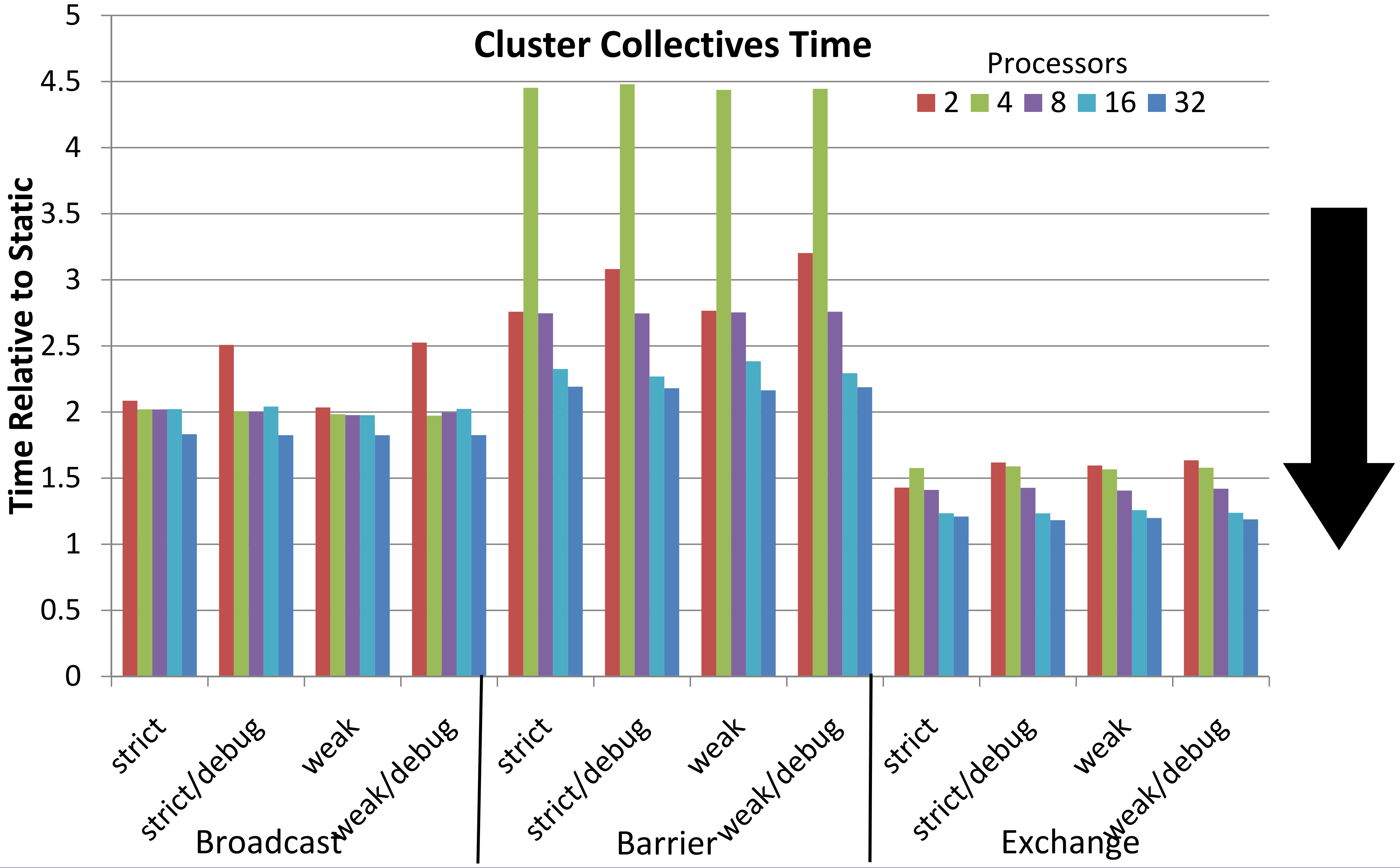
Enforcement Variants

- Five enforcement variants tested
  - static: use **`single`** type system
  - strict: strict dynamic alignment
  - strict/debug: strict dynamic alignment with alignment history saved
  - weak: strict dynamic alignment
  - strict/debug: weak dynamic alignment with alignment history saved

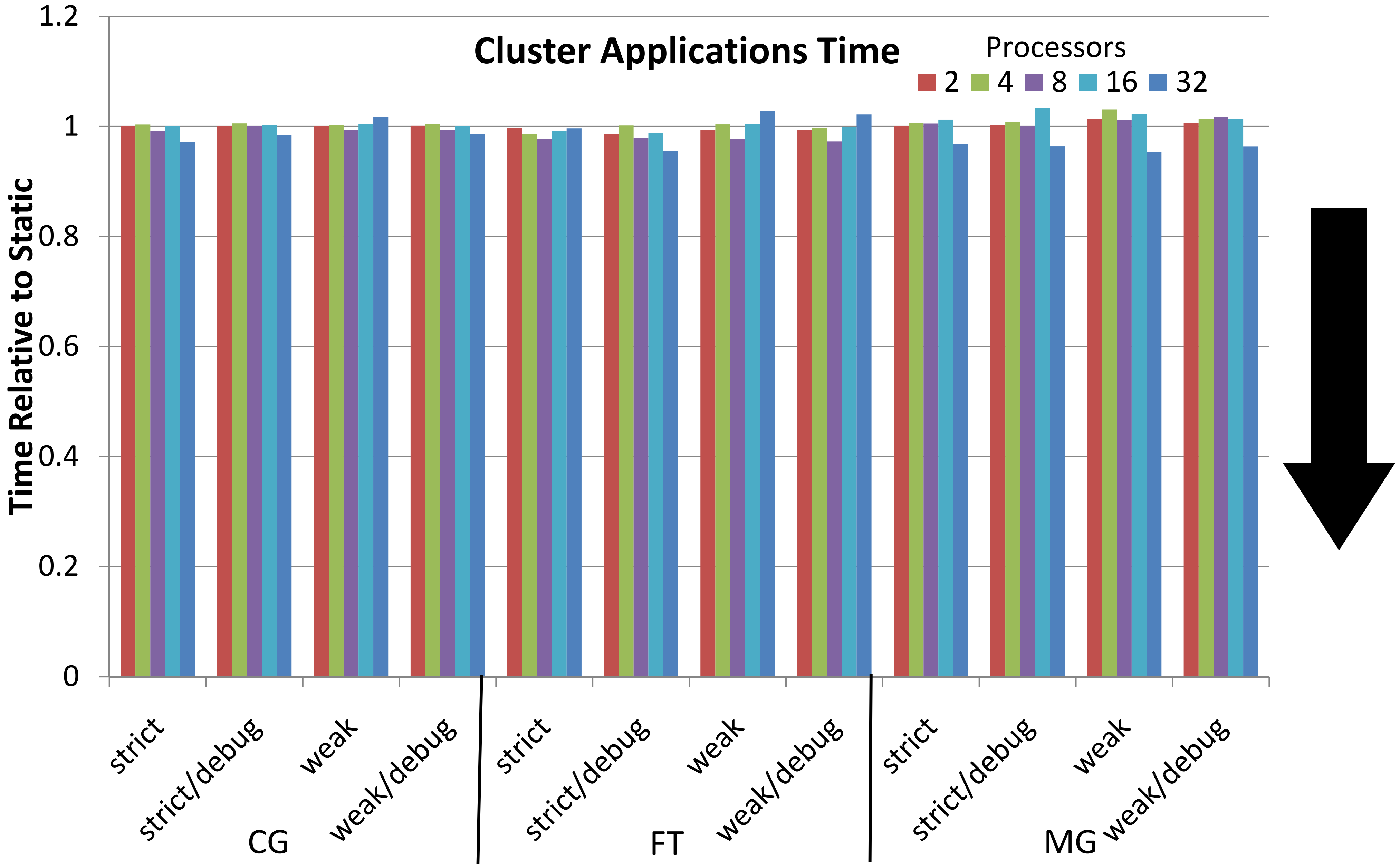
SMP Collectives Results



Cluster Collectives Results



Cluster Application Results



Performance Escape Hatches

- Effects on application performance can be potentially reduced in multiple different ways
  - None used right now
- Optimizations
  - Remove redundant checks
- Hybrid static/dynamic analysis
  - Use static inference and then only check those collectives that are not inferred to be aligned
- Turn off checking in production runs
  - Use checking only when debugging an application

Conclusions

- Dynamic checking removes annotation burden from programmers
- Minimal performance impact on applications
  - Most applications avoid spending time in collectives
  - Applications that do spend a lot of time in collectives don't scale anyway
- Multiple strategies to further reduce overhead
- Dynamic checking can be applied to languages without strong type systems (e.g. UPC)