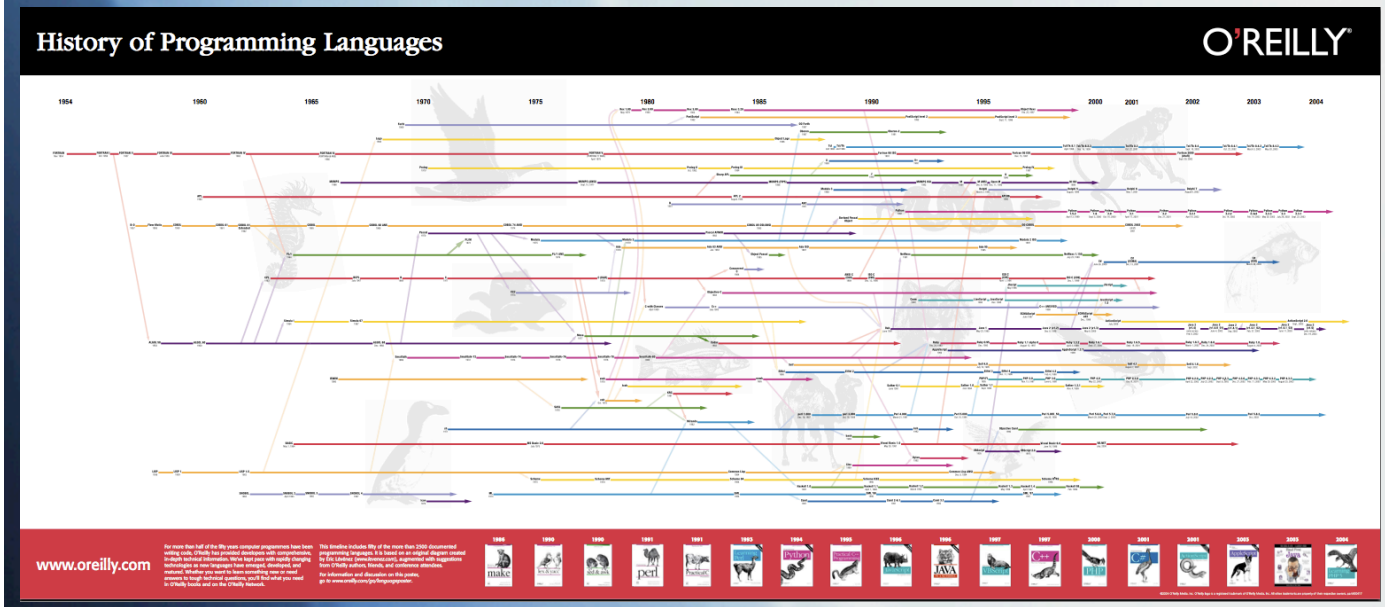
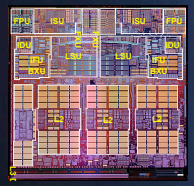
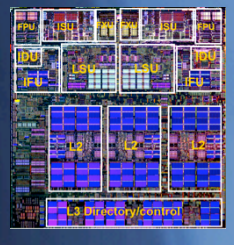
RICE George R. Brown School of Engineering
Computer Science

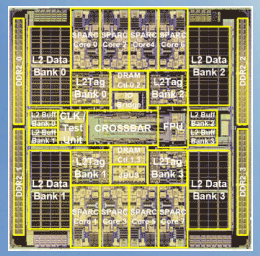
Multicore Programming Models and their Implementation Challenges

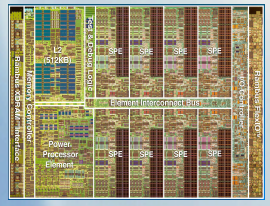


Vivek Sarkar
Rice University
vsarkar@rice.edu

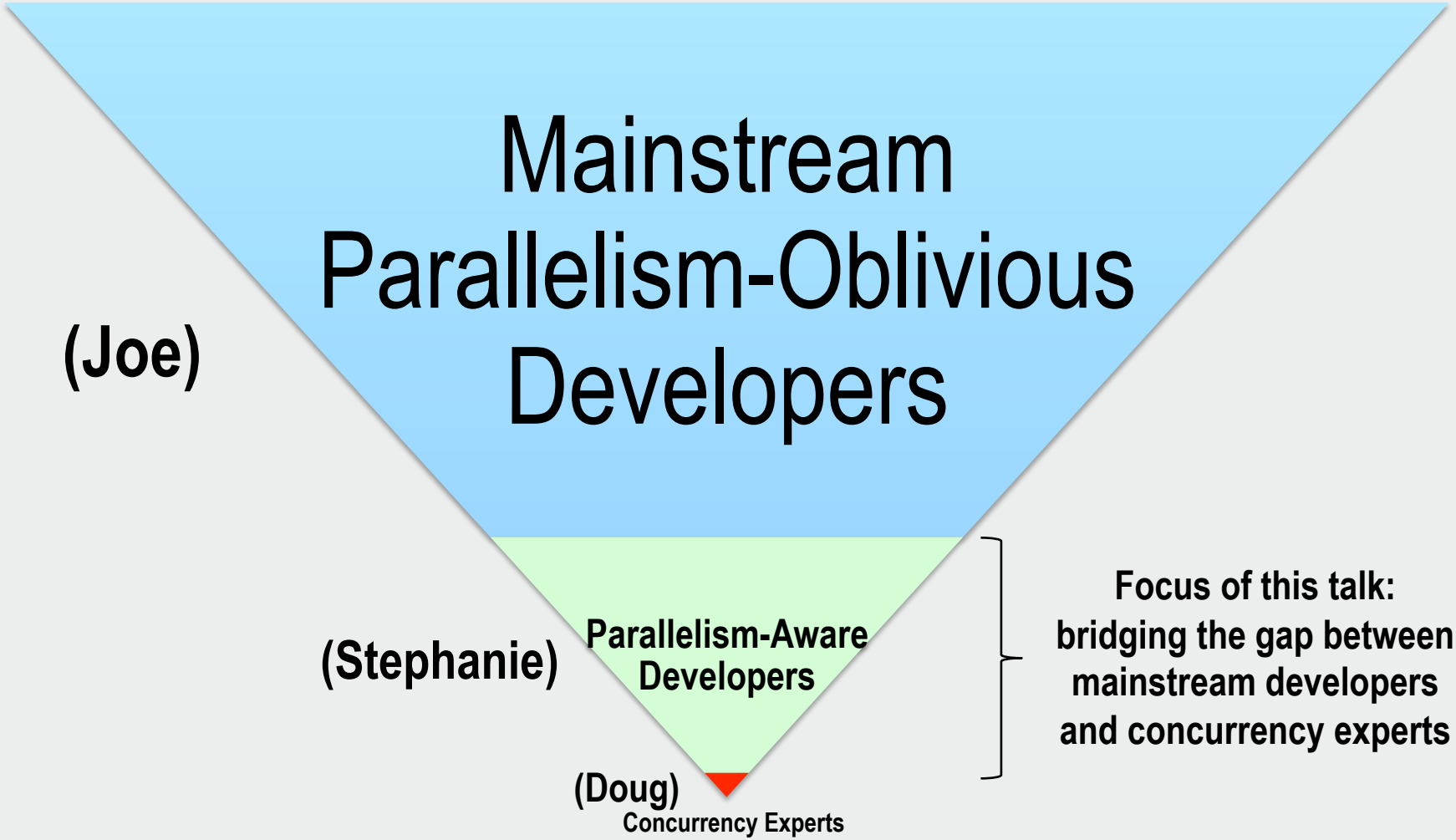








Inverted Pyramid of Parallel Programming Skills



Outline

- A Sample of Modern Multicore Programming Models
- X10 + Habanero Execution Model and Implementation Challenges
- Rice Habanero Multicore Software research project

Comparison of Multicore Programming Models along Selected Dimensions

	Dynamic Parallelism	Locality Control	Mutual Exclusion	Collective & Point-to-point Synchronization	Data Parallelism
Cilk	Spawn, sync	None	Locks	None	None
Java Concurrency	Executors, Task Queues	None	Locks, monitors, atomic classes	Synchronizers	Concurrent collections
Intel TBB	Generic algs, tasks	None	Locks, atomic classes	None	Concurrent containers
.Net Parallel Extensions	Generic algs, tasks	None	Locks, monitors	Futures	PLINQ
OpenMP	SPMD (v2.5), Tasks (v3.0)	None	Locks, critical, atomic	Barriers	None
CUDA	None	Device, grid, block, threads	None	Barriers	SIMD
Intel Concurrent Collections	Tagged prescription of steps	None	None	Tagged put & get operations on Item Collections	None
X10 + *Habanero extensions* (builds on Java Concurrency)	Async, finish	Places	Atomic blocks, Java atomic classes	Future, force, clocks, *phasers, delayed async*	SIMD/MIMD array operations, Java concurrent collections

Dynamic Parallelism in Cilk

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Identifies a function as a ***Cilk procedure***, capable of being spawned in parallel.

The named ***child*** Cilk procedure can execute in parallel with the ***parent*** caller.

Control cannot pass this point until all spawned children have returned.

Cilk Language

- Cilk is a faithful extension of C
  - if Cilk keywords are elided → C program semantics
    - Except for inlets …
- Limitations on software reuse and composability
  - —spawn keyword can only be applied to a cilk function
  - —spawn keyword cannot be used in a C function
  - —cilk function cannot be called with normal C call conventions
    - must be called with a spawn & waited for by a sync
  - Fully strict – parent must wait for child for terminate

Callbacks with Cilk Inlets

```
int max, ix = -1;
inlet void update ( int val, int index ) {
  if (idx == -1 || val > max) {
    ix = index; max = val;
  }
}
⋮
for (i=0; i<1000000; i++) {
  update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

1. The non-**spawn** args to **update()** are evaluated.
2. The Cilk procedure **foo(i)** is spawned.
3. Control passes to the next statement.
4. When **foo(i)** returns, **update()** is invoked.

Summary of Cilk constructs

- spawn – create child task
- sync – await termination of child tasks
- SYNCHED – check if child tasks have terminated
- inlet – callback computation upon termination of child task
- Advanced features (for “Doug”)
  - Cilk_lockvar
  - Cilk_fence()
  - abort

java.util.concurrent

- General purpose toolkit for developing concurrent applications
  - No more “reinventing the wheel”!
- Goals: “Something for Everyone!”
  - Make some problems trivial to solve by everyone (“Joe”)
    - Develop thread-safe classes, such as servlets, built on concurrent building blocks like ConcurrentHashMap
  - Make some problems easier to solve by concurrent programmers (“Stephanie”)
    - Develop concurrent applications using thread pools, barriers, latches, and blocking queues
  - Make some problems possible to solve by concurrency experts (“Doug”)
    - Develop custom locking classes, lock-free algorithms

Overview of `j.u.c`

- **Executors**
  - `Executor`
  - `ExecutorService`
  - `ScheduledExecutorService`
  - `Callable`
  - `Future`
  - `ScheduledFuture`
  - `Delayed`
  - `CompletionService`
  - `ThreadPoolExecutor`
  - `ScheduledThreadPoolExecutor`
  - `AbstractExecutorService`
  - `Executors`
  - `FutureTask`
  - `ExecutorCompletionService`
- **Queues**
  - `BlockingQueue`
  - `ConcurrentLinkedQueue`
  - `LinkedBlockingQueue`
  - `ArrayBlockingQueue`
  - `SynchronousQueue`
  - `PriorityBlockingQueue`
  - `DelayQueue`
- **Concurrent Collections**
  - `ConcurrentMap`
  - `ConcurrentHashMap`
  - `CopyOnWriteArray{List,Set}`
- **Synchronizers**
  - `CountDownLatch`
  - `Semaphore`
  - `Exchanger`
  - `CyclicBarrier`
- **Locks:** `java.util.concurrent.locks`
  - `Lock`
  - `Condition`
  - `ReadWriteLock`
  - `AbstractQueuedSynchronizer`
  - `LockSupport`
  - `ReentrantLock`
  - `ReentrantReadWriteLock`
- **Atomics:** `java.util.concurrent.atomic`
  - `Atomic[Type]`
  - `Atomic[Type]Array`
  - `Atomic[Type]FieldUpdater`
  - `Atomic{Markable,Stampable}Reference`

Intel® Threading Building Blocks Components

(Stephanie)

(Joe)

Generic Parallel Algorithms
- parallel_for
- parallel_reduce
- pipeline
- parallel_sort
- parallel_while
- parallel_scan

Concurrent Containers
- concurrent_hash_map
- concurrent_queue
- concurrent_vector

Task scheduler

(Doug)

Low-Level Synchronization Primitives
- atomic
- mutex
- spin_mutex
- queuing_mutex
- spin_rw_mutex
- queuing_rw_mutex

Memory Allocation
- cache_aligned_allocator
- scalable_allocator

Timing
- tick_count

Serial Example

```
static void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i!=n; ++i )
        Foo(a[i]);
}
```

Will parallelize by dividing iteration space of i into chunks

Parallel Version

TBB extensions in red

```
class ApplyFoo {
    float *const my_a;
public:
    ApplyFoo( float *a ) : my_a(a) {}
    void operator()( const blocked_range<size_t>& range ) const {
        float *a = my_a;
        for( int i= range.begin(); i!=range.end(); ++i )
            Foo(a[i]);
    }
};

void ParallelApplyFoo(float a[], size_t n ) {
    parallel_for( blocked_range<int>( 0, n ),
                  ApplyFoo(a),
                  auto_partitioner() );
}
```

Task

Pattern

Iteration space

Automatic grain size

Microsoft Parallel LINQ (PLINQ)

- Declarative data parallelism via LINQ-to-Objects
  - PLINQ supports all LINQ operators
    - Select, Where, Join, GroupBy, Sum, etc.
  - Activated with the AsParallel extension method:
    - var q = from x in data where p(x) orderby k(x) select f(x);
      var q = from x in data.AsParallel() where p(x) orderby k(x) select f(x);
  - Works for any IEnumerable<T>

- Query syntax enables runtime to auto-parallelize
  - Automatic way to generate more Tasks, like Parallel
  - Graph analysis determines how to do it
  - Classic data parallelism: partitioning + pipelining

PLINQ “Gotchas”

- Ordering not guaranteed
  - int[] data = new int[] { 0, 1, 2 };
    var q = from x in data.AsParallel(QueryOptions.PreserveOrdering)
    select x * 2;
    int[] scaled = q.ToArray(); // == { 0, 2, 4 }?
- Exceptions are aggregated
  - object[] data = new object[] { "foo", null, null };
    var q = from x in data.AsParallel() select o.ToString();
  - NullRefExceptions on data[1], data[2], or both?
  - PLINQ will always aggregate under AggregateException
- Side effects and mutability are serious issues
  - Most queries do not use side effects… but:
  - var q = from x in data.AsParallel() select x.f++;
  - OK if all elements in data are unique, else race condition

Intel Concurrent Collections (CnC)

Domain Expert: (person)
Only domain knowledge
No tuning knowledge

"Joe"

The application problem

Decomposition into Serial Steps
Decomposition constraints required by the application

Concurrent Collections Program

Tuning Expert: (person, runtime, compiler)
No domain knowledge
Only tuning knowledge

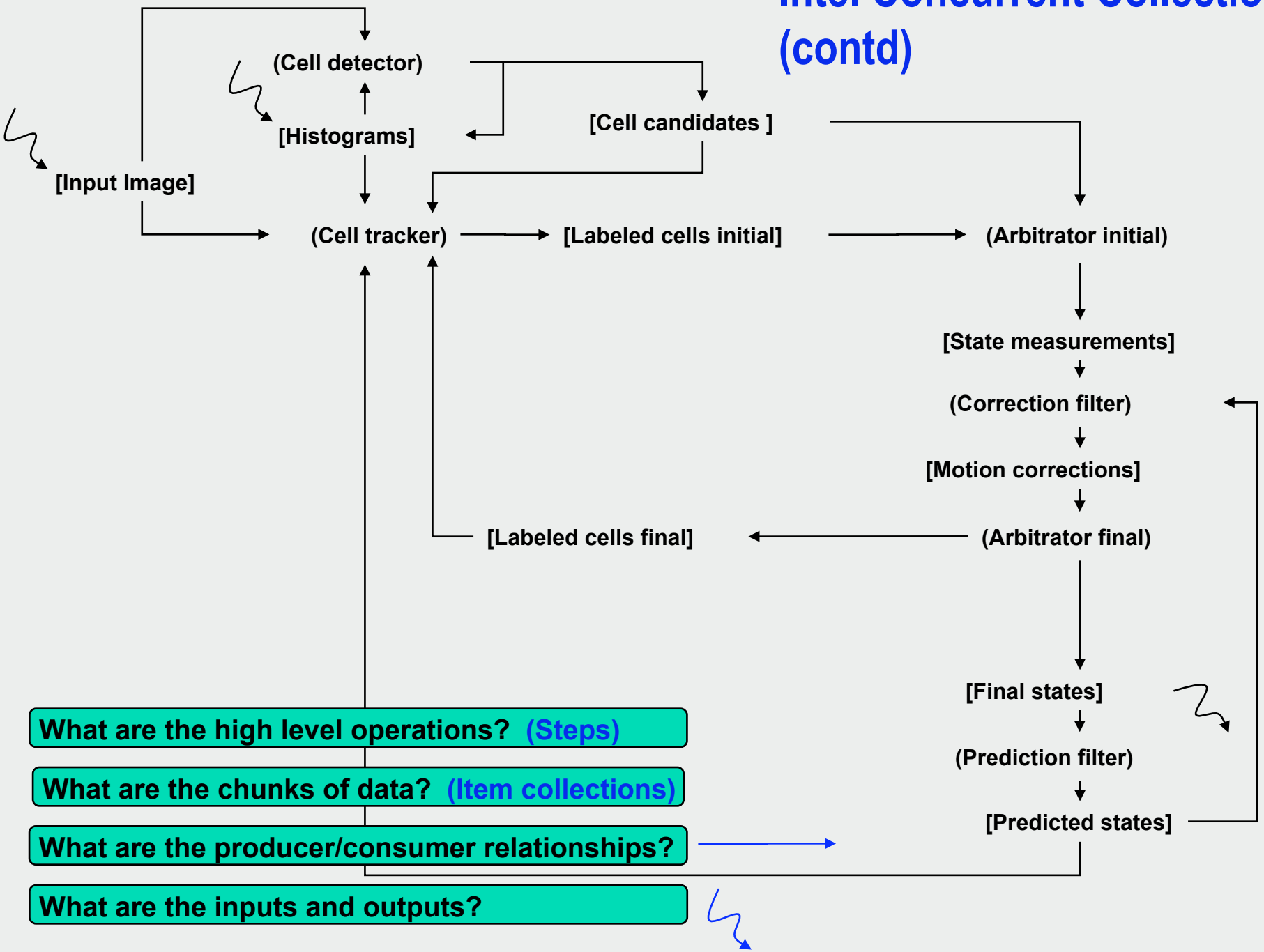
"Stephanie"

Architecture
Actual parallelism
Locality
Overhead
Load balancing
Distribution among processors
Scheduling within a processor

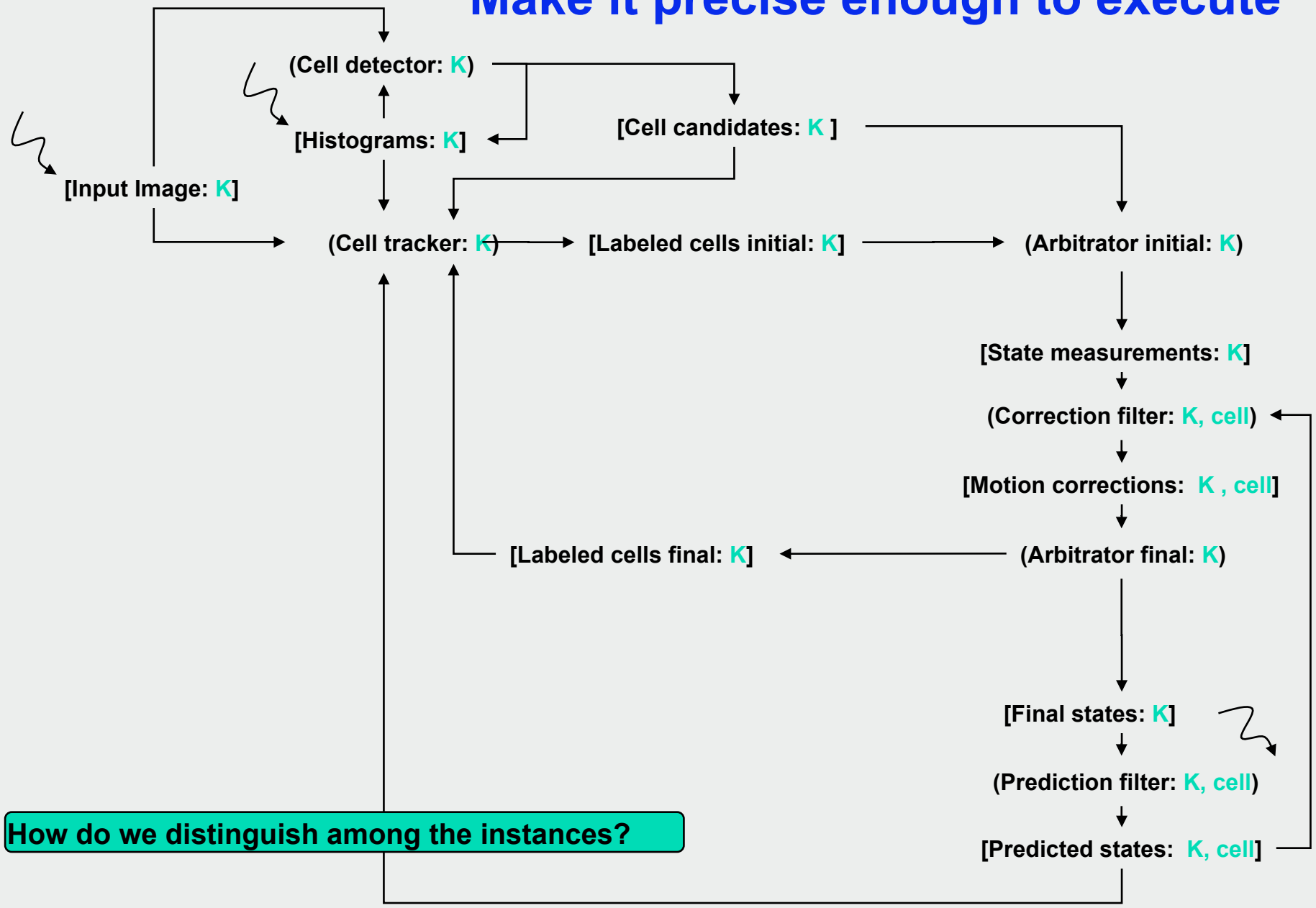
Mapping to target platform

Source: Kathleen Knobe
http://softwarecommunity.intel.com/articles/eng/3862.htm

Intel Concurrent Collections (contd)

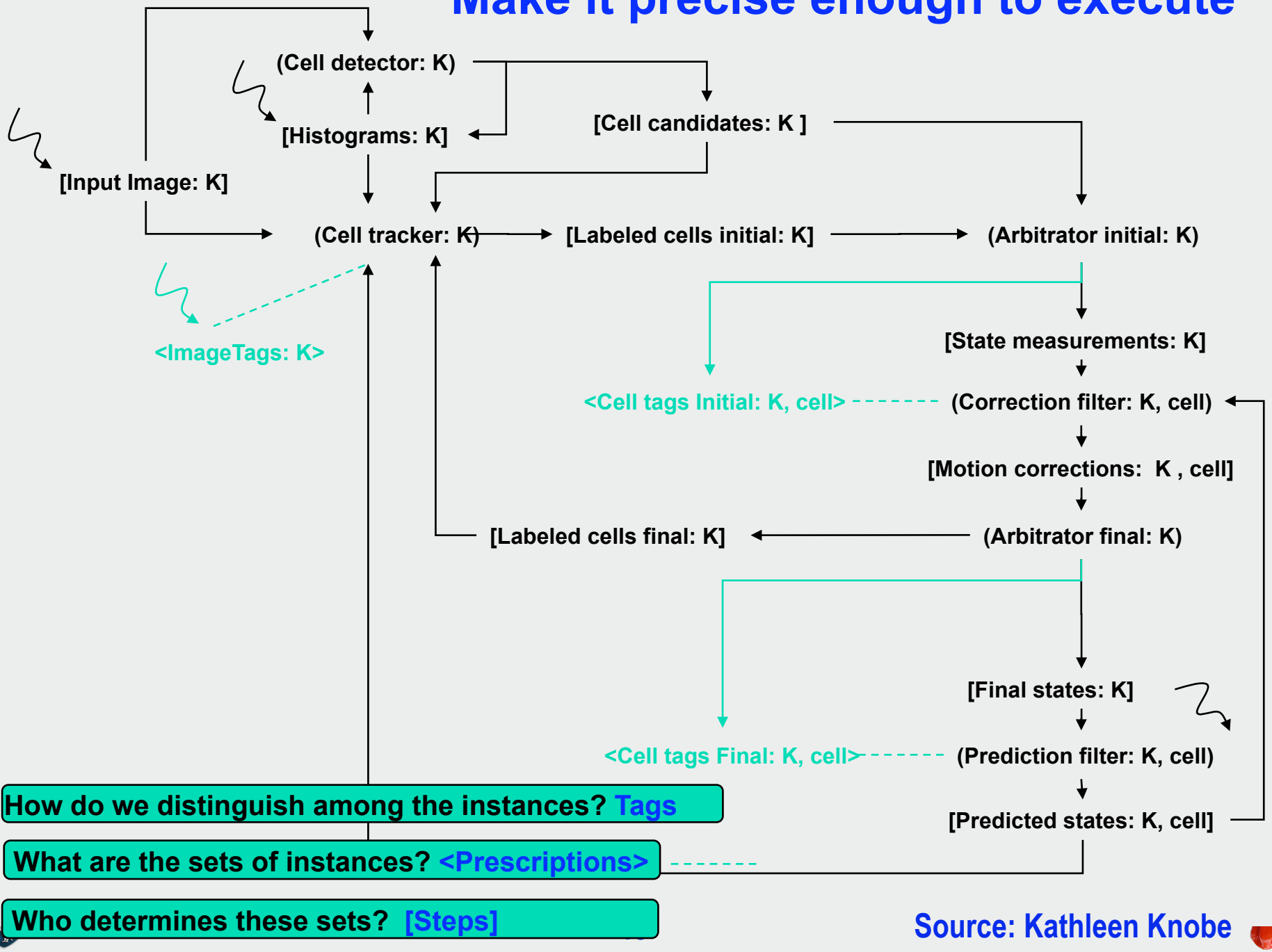


Make it precise enough to execute



Source: Kathleen Knobe

Make it precise enough to execute



Source: Kathleen Knobe

Domain Expert's view of Concurrent Collections

- No thinking about parallelism
  - Only domain knowledge
- No overwriting
    - Collections on single assignment
- No arbitrary serialization
    - only constraints on ordering via tagged puts and gets
- Steps are functional
    - No side-effects
- Result is:
    - Deterministic
    - Race-free

Comparison of Multicore Programming Models along Selected Dimensions

	Dynamic Parallelism	Locality Control	Mutual Exclusion	Collective & Point-to-point Synchronization	Data Parallelism
Cilk	Spawn, sync	None	Locks	None	None
Java Concurrency	Executors, Task Queues	None	Locks, monitors, atomic classes	Synchronizers	Concurrent collections
Intel TBB	Generic algs, tasks	None	Locks, atomic classes	None	Concurrent containers
.Net Parallel Extensions	Generic algs, tasks	None	Locks, monitors	Futures	PLINQ
OpenMP	SPMD (v2.5), Tasks (v3.0)	None	Locks, critical, atomic	Barriers	None
CUDA	None	Device, grid, block, threads	None	Barriers	SIMD
Intel Concurrent Collections	Tagged prescription of steps	None	None	Tagged put & get operations on Item Collections	None
X10 + *Habanero extensions* (builds on Java Concurrency)	Async, finish	Places	Atomic blocks, Java atomic classes	Future, force, clocks, *phasers, delayed async*	SIMD/MIMD array operations, Java concurrent collections

Outline

- A Sample of Modern Multicore Programming Models
- X10 + Habanero Execution Model and Implementation Challenges
- Rice Habanero Multicore Software research project

X10 Acknowledgments

- X10 Core Team (IBM)
  - Ganesh Bikshandi, Sreedhar Kodali, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Pradeep Varma, Sayantan Sur, Olivier Tardieu, Krishna Venkat, Tong Wen, Jose Castanos, Ankur Narang, Komondoor Raghavan
- X10 Tools
  - Philippe Charles, Robert Fuhrer
- Emeritus
  - Kemal Ebcioglu, Christian Grothoff, Vincent Cave, Lex Spoon, Christoph von Praun, Rajkishore Barik, Chris Donawa, Allan Kielstra
- Research colleagues
  - Vivek Sarkar, Rice U
  - Satish Chandra,Guojing Cong
  - Ras Bodik, Guang Gao, Radha Jagadeesan, Jens Palsberg, Rodric Rabbah, Jan Vitek
  - Vinod Tipparaju, Jarek Nieplocha (PNNL)
  - Kathy Yelick, Dan Bonachea (Berkeley)
  - Several others at IBM

Recent Publications

1. “Type Inference for Locality Analysis of Distributed Data Structures”, PPoPP 2008.
2. “Deadlock-free scheduling of X10 Computations with bounded resources”, SPAA 2007
3. “A Theory of Memory Models”, PPoPP 2007.
4. “May-Happen-in-Parallel Analysis of X10 Programs”, PPoPP 2007.
5. “An annotation and compiler plug-in system for X10”, IBM Technical Report, Feb 2007.
6. “Experiences with an SMP Implementation for X10 based on the Java Concurrency Utilities” Workshop on Programming Models for Ubiquitous Parallelism (PMUP), September 2006.
7. "An Experiment in Measuring the Productivity of Three Parallel Programming Languages”, P-PHEC workshop, February 2006.
8. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", OOPSLA conference, October 2005.
9. "Concurrent Clustered Programming", CONCUR conference, August 2005.
10. "X10: an Experimental Language for High Productivity Programming of Scalable Systems", P-PHEC workshop, February 2005.

Tutorials

- TiC 2006, PACT 2006, OOPSLA 2006, PPoPP 2007, SC 2007
- Graduate course on X10 at U Pisa (07/07)
- Graduate course at Waseda U (Tokyo, 04/08)

X10 + Habanero Execution Model: Portable Parallelism in Four Dimensions

1. Lightweight dynamic task creation & termination
   - *async*, *finish*, (from X10)
2. Locality control --- task and data distributions
   - *places* (from X10)
3. Mutual exclusion
   - *atomic* (from X10 & related work on transactions)
4. Collective and point-to-point synchronization
   - *phasers* (from Habanero --- extension of X10 *clocks*)

Async and Finish

Stmt ::= **async** *Stmt*

async S

- Creates a new child activity that executes statement S
- Returns immediately
- S may reference final variables in enclosing blocks
- Activities cannot be named
- Activity cannot be aborted or cancelled

Stmt ::= **finish** *Stmt*

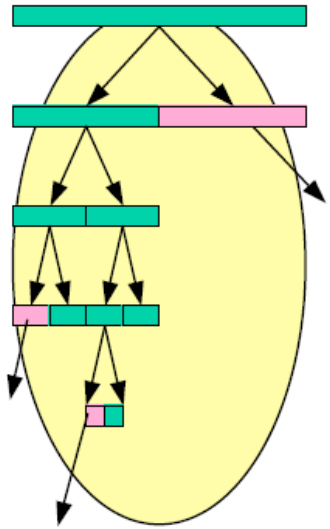
finish S

- Execute S, but wait until all (transitively) spawned asyncs have terminated.
- Rooted exception model
  - Trap all exceptions thrown by spawned activities.
  - Throw an (aggregate) exception if any spawned async terminates abruptly.
- implicit finish between start and end of main program

Recursive Parallelism with Finish and Async

```
void refine(final int n, final int l, final int nmax) {
        left = new Tree(this,2.0*l);
        right =new Tree(this, 2.0*l+1);
        final nullable Tree ll = left, rr=right;
        if (n < (nmax-1)) {
                async {ll.refine(n+1,2*l,nmax);}
                async { rr.refine(n+1,2*l+1,nmax);}
        }
        if (n < nmax) data = null;

. . .
// Main program
. . .
finish refine(root, 1, nmax);
```



From "What's in it for the Users? Looking Toward the HPCS Languages and Beyond", D. Bernholdt, W.R. Elwasif, Robert J. Harrison, PGAS 2006

Loop Parallelism with Finish and Async

```
int iters = 0; delta = epsilon+1;
while ( delta > epsilon ) {
  finish {
    for ( jj = 1 ; jj <= n ; jj++ ) {
      final int j = jj;
      async { // for-async can be replaced by foreach
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
        diff[j] = Math.abs(newA[j]-oldA[j]);
      } // async
    } // for
  } // finish
  delta = diff.sum(); iters++;
  temp = newA; newA = oldA; oldA = temp;
}
System.out.println("Iterations: " + iters);
```

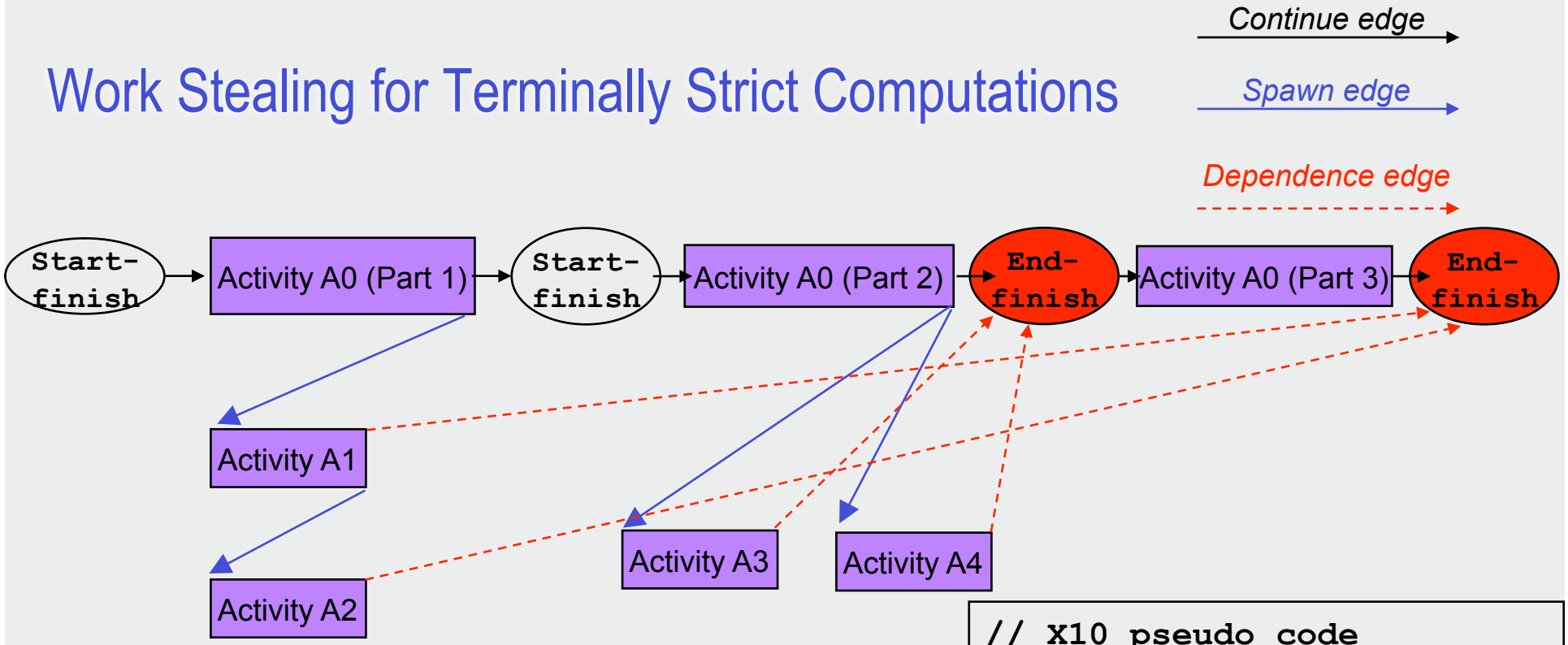
Loop Parallelism with Foreach

```
int iters = 0; delta = epsilon+1;
while ( delta > epsilon ) {
  finish {
    foreach ( point[j] : [1:n] ) {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
        diff[j] = Math.abs(newA[j]-oldA[j]);
    } // foreach
  } // finish
  delta = diff.sum(); iters++;
  temp = newA; newA = oldA; oldA = temp;
}
System.out.println("Iterations: " + iters);
```

Implementation Challenges for Finish & Async

- Extend space-efficient scalable work-stealing schedulers to support terminally strict finish-async programs
  - [PPoPP 2007, ICPP 2008]
- Reduce footprint impact of inflated blocked activities
  - Delayed asyncs (work in progress)

Work Stealing for Terminally Strict Computations



"Deadlock-Free Scheduling of X10 Computations with Bounded Resources", S.Agarwal et al, SPAA 2007.

Theorem 2.6: A work-stealing execution of a *terminally strict* multithreaded computation with finish & async constructs on P processor uses at most S_1*P space in its dequeue's, where S_1 is the maximum stack depth in a sequential execution of the program.

```
// X10 pseudo code
main(){ // implicit finish
  Activity A0 (Part 1);
  async {A1; async A2;}
  try {
    finish {
      Activity A0 (Part 2);
      async A3;
      async A4;
    }
  catch (…) { … }
  Activity A0 (Part 3);
}
```

Delayed Async's

Stmt ::= **async await**(*Cond*) *Stmt*

- Creates a new child activity for statement Stmt
- Returns immediately
- New activity can only start execution of Stmt when Cond becomes true

➔ Activity does not get inflated into an active task before Cond becomes true

Example

- Consider the case when Stmt contains a blocking operation such as a CnC get operation e.g.,

```
S1;
C.get(…);  // May block
S2;
```

- "async Stmt" can be translated to a delayed async as follows:

```
async {
        S1;
        async await(C.containsTag(…))
            S2;
}
```

Mapping of Concurrent Collections to X10 + Habanero

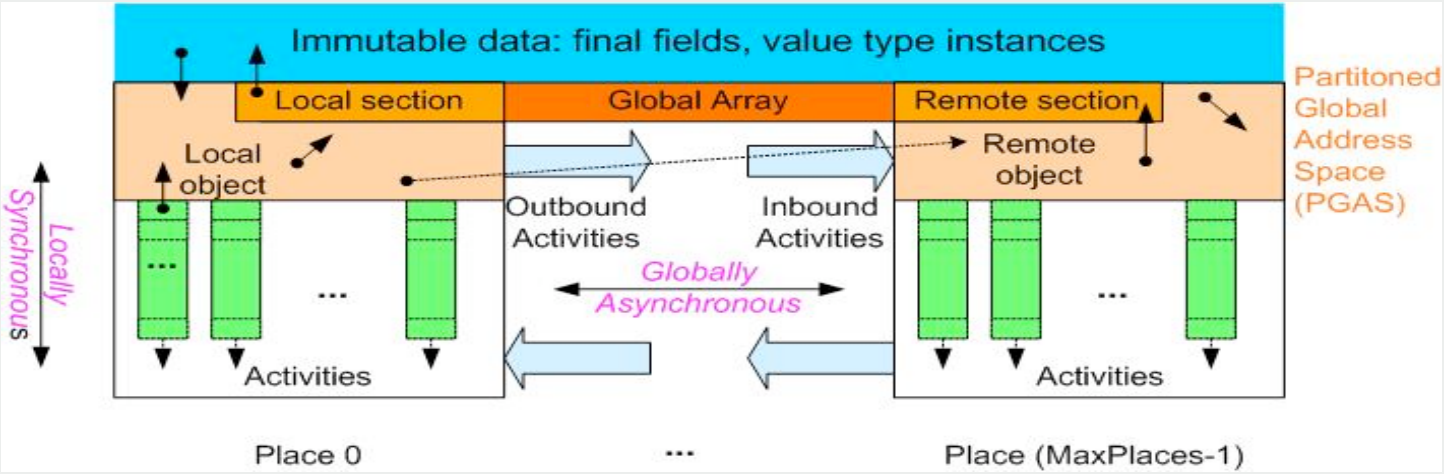
CnC Construct	Translation to X10 + Habanero
Tag	Point object
Prescription	Async or Delayed async
Item Collection	java.util.concurrent.ConcurrentHashMap
put() on Item Collection	Nonblocking put() on ConcurrentHashMap
get() on Item Collection	Blocking or nonblocking get on ConcurrentHashMap

"Multicore Implementations of the Concurrent Collections Programming Model", Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, Leo Treggiari (work in progress)

X10 + Habanero Execution Model: Portable Parallelism in Four Dimensions

1. Lightweight dynamic task creation & termination
   - *async*, *finish,* (from X10)
2. Locality control --- task and data distributions
   - *places* (from X10)
3. Mutual exclusion
   - *atomic* (from X10 & related work on transactions)
4. Collective and point-to-point synchronization
   - *phasers* (from Habanero --- extension of X10 *clocks*)

2. Task and data distributions with Places



Storage classes:

- Activity-local
- Place-local
- Partitioned global
- Immutable

- Dynamic parallelism with a *Partitioned Global Address Space*
- *Places* encapsulate binding of activities and globally addressable mutable data
  - Number of places currently fixed at launch time
- Each *datum* has a designated place specified by its distribution
- Each *async* has a designated place specified by its distribution --- subsumes threads, structured parallelism, messaging, DMA transfers, etc.
- *Immutable* data (value types, value arrays) is place-independent and offers opportunity for functional-style parallelism
- Type system for places --- "Type Inference for Locality Analysis of Distributed Data Structures", S.Chandra et al, PPoPP 2008.

Extension of Async with Places

Examples

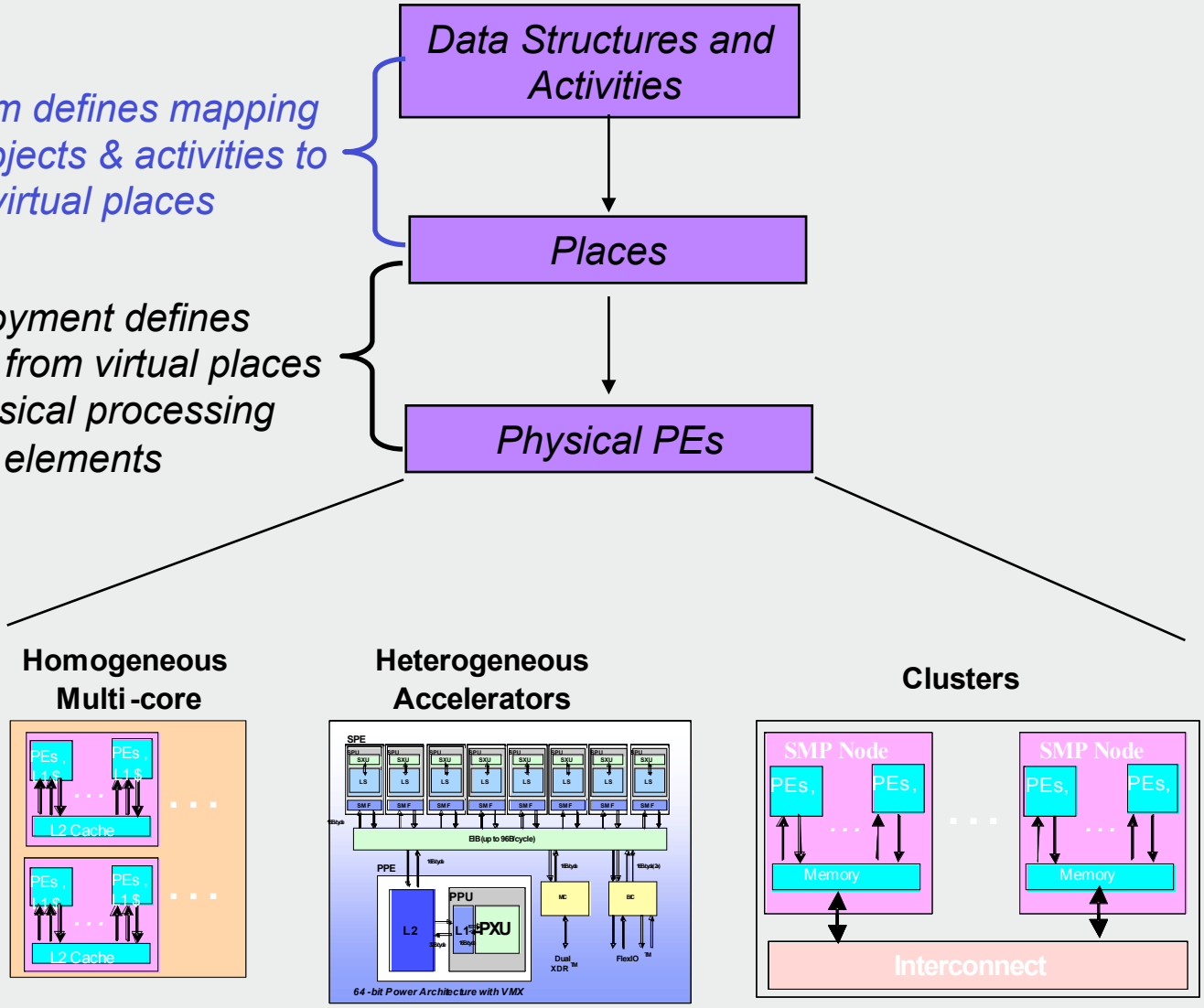
```
1) finish { // Inter-place parallelism
       final int x = … , y = … ;
       async (a) a.foo(x); // Execute at a's place
       async (b.distribution[i])
           b[i].bar(y); // Execute at b[i]'s place
   }

2) // Implicit and explicit versions of remote fetch-and-op
   a) a.x = foo(a.x, b.y) ;
   b) async (b) {
         final double v = b.y; // Can be any value type
         async (a) atomic a.x = foo(a.x, v);
      }
```

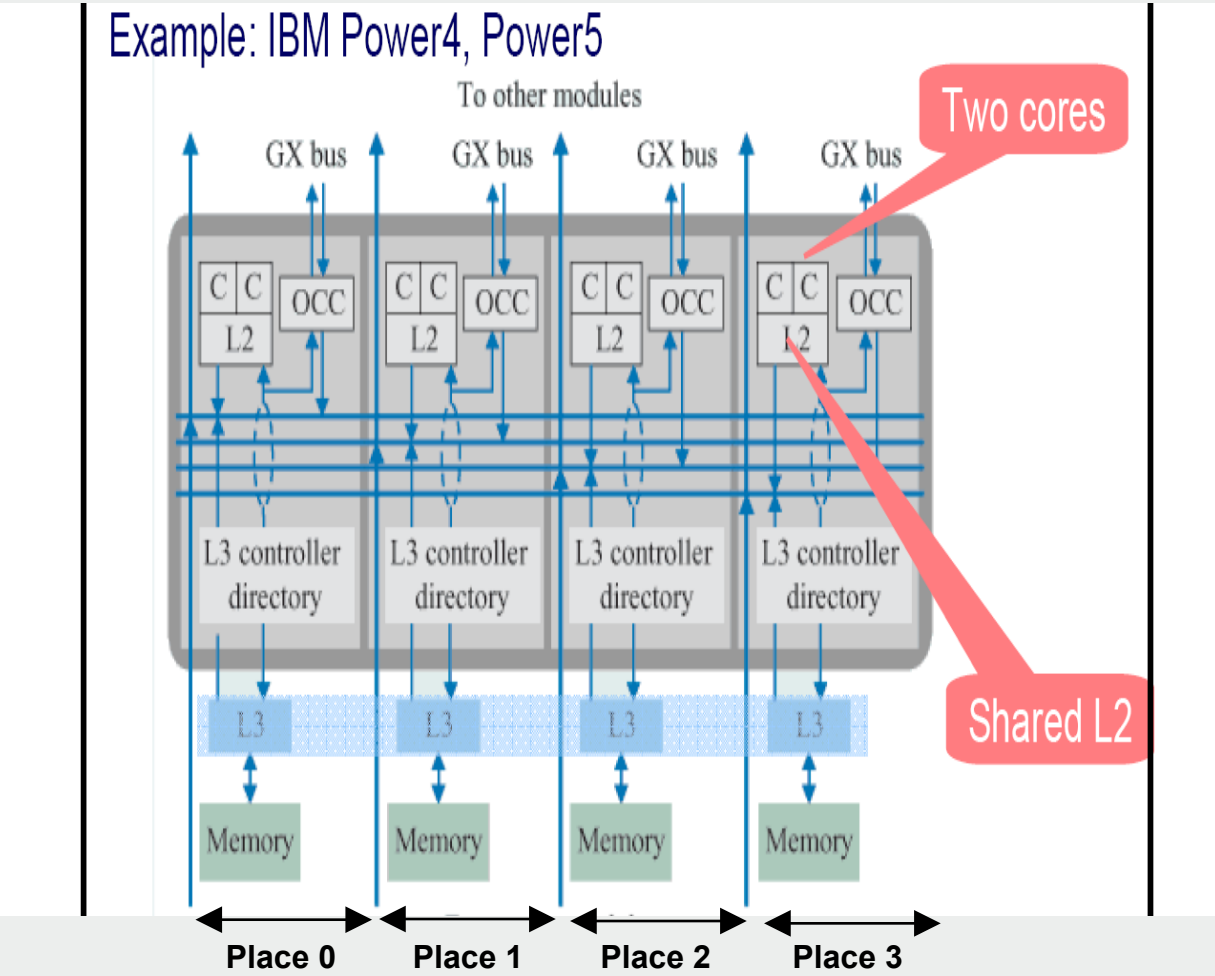
Portable Parallelism via Place Deployments

Program defines mapping from objects & activities to virtual places

Deployment defines mapping from virtual places to physical processing elements



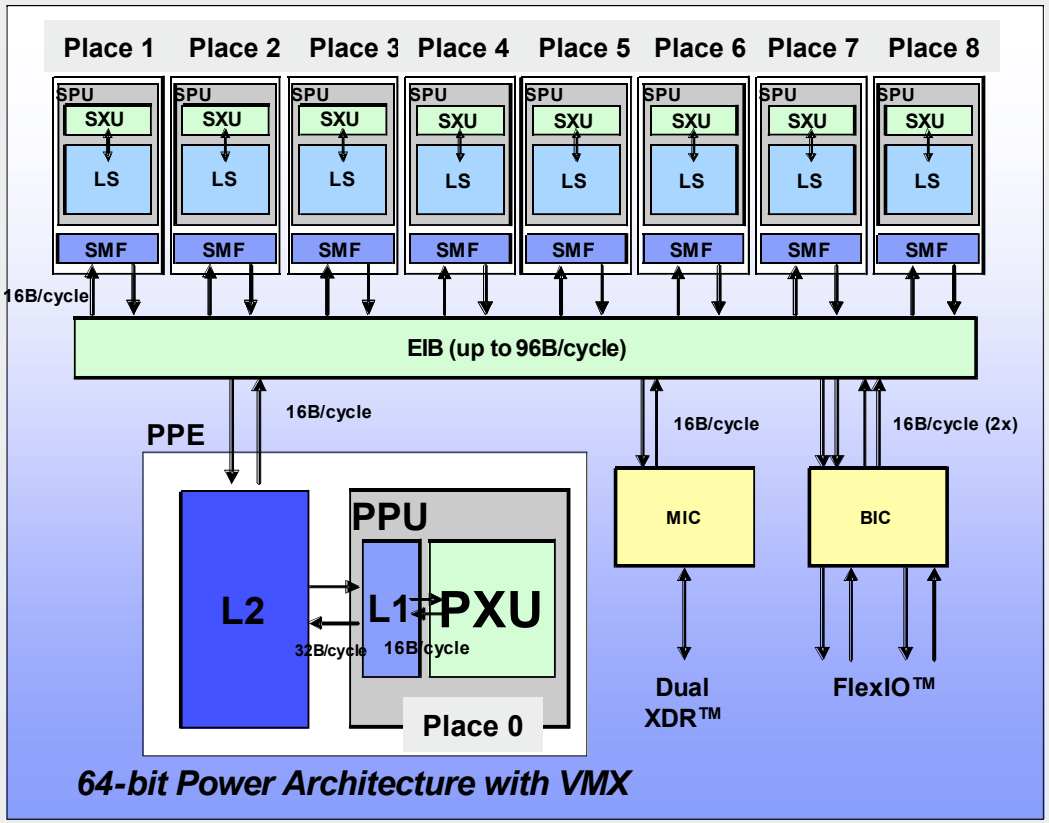
Place Deployment on a Multicore SMP



- Basic Approach -- partition heap into multiple place-local heaps
- Each object is allocated in a designated place
- Each activity is created (and pinned) at a designated place
- Allow an activity to synchronously access data at remote places outside of atomic sections

➔ Thus, places serve as affinity hints for intra-SMP locality

Possible Place Deployment for Cell



- Basic Approach:
  - map 9 places on to PPE + eight SPEs
  - Use finish & async's as high-level representation of DMAs
- Implementation Challenges:
  - Weak PPE
  - SIMDization critical for performance
  - Lack of hardware support for coherence
  - Limited memory on SPE's
  - Different ISA's for PPE and SPE.

X10 + Habanero Execution Model: Portable Parallelism in Four Dimensions

1. Lightweight dynamic task creation & termination
   - *async*, *finish,* (from X10)
2. Locality control --- task and data distributions
   - *places* (from X10)
3. Mutual exclusion
   - *atomic* (from X10 & related work on transactions)
4. Collective and point-to-point synchronization
   - *phasers* (from Habanero --- extension of X10 *clocks*)

Overview of Phasers

- **Extension of X10 clocks**
- **Designed to handle multiple communication patterns**
  - Collective Barrier
  - Point-to-point synchronization
- **Phase ordering property**
- **Deadlock freedom in absence of explicit wait operations**
- **Dynamic parallelism**
  - # activities synchronized on phaser can vary dynamically
- **Amenable to efficient implementation**
  - Lightweight local-spin multicore implementation in Habanero project
  - Phasers recently incorporated by Doug Lea into JSR-166y for possible inclusion in Java 7 (google "phasers java")

- **"Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization", J.Shirako, D.Peixotto, V.Sarkar, W.Scherer, ICS 2008**

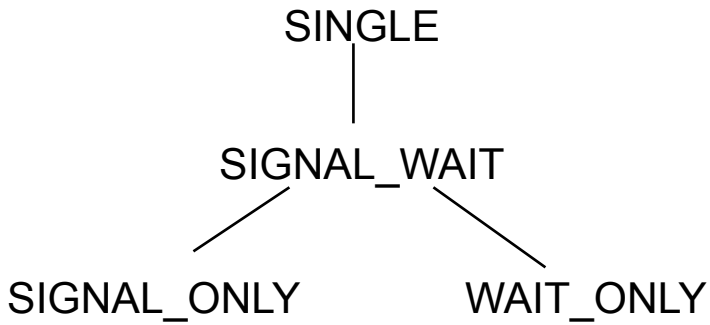
Collective and Point-to-point Synchronization with Phasers

phaser ph = new phaser(MODE);

- **Allocate a phaser, register current activity with it according to MODE. Phase 0 of ph starts.**
- **MODE can be SIGNAL_ONLY, WAIT_ONLY, SIGNAL_WAIT (default) or SINGLE**
- ***Finish Scope rule:* phaser ph cannot be used outside the scope of its immediately enclosing finish operation**

async phased (MODE1(ph1), MODE2(ph2), …) S

- **Spawn S as an asynchronous (parallel) activity that is registered on phasers ph1, ph2, … according to MODE1, MODE2, …**
- ***Capability rule:* parent activity can only transmit phaser capabilities to child activity that are a subset of the parent's capabilities, according to the lattice:**



Phaser Primitives (contd)

next;

- **Advance *each* phaser that activity is registered on to its next phase; semantics depends on registration mode**

next <stmt> // next-with-single statement

- **Execute next operation with single instance of <stmt> during phase transition**
- **All activities executing a next-with-single statement must execute the same static statement**

ph.signal();

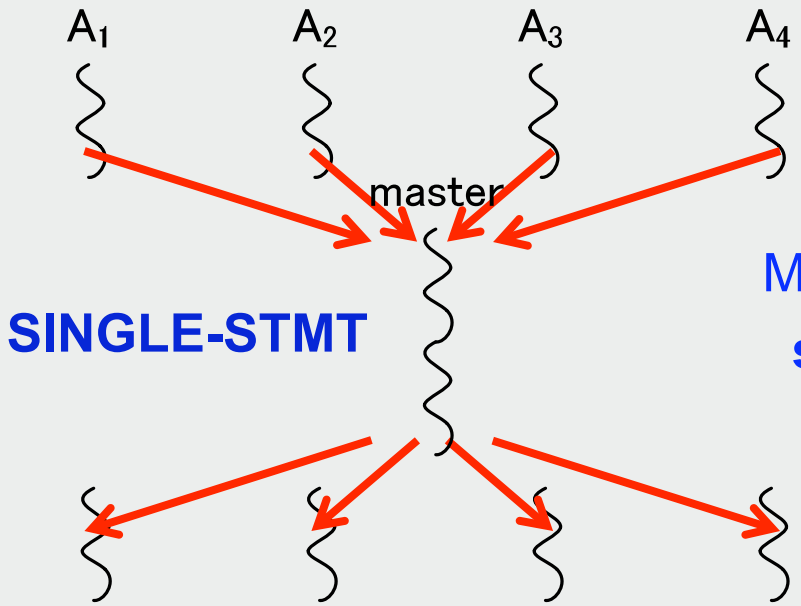
- **Nonblocking operation that signals completion of work by current activity for this phase of phaser ph**
- **Error if activity does not have signal capability**

signal;

- **Perform ph.signal() on each phaser that activity is registered on with a signal capability**

next-with-single statement

- **Single activity region executed during phase transition**
  - **next { SINGLE-STMT }**
    - Available for SINGLE mode
    - **SINGLE-STMT is executed by one (master) activity**
      - Like “barrier action” in JUC’s CyclicBarrier
    - All activities executing a next-with-single statement must execute the same static statement

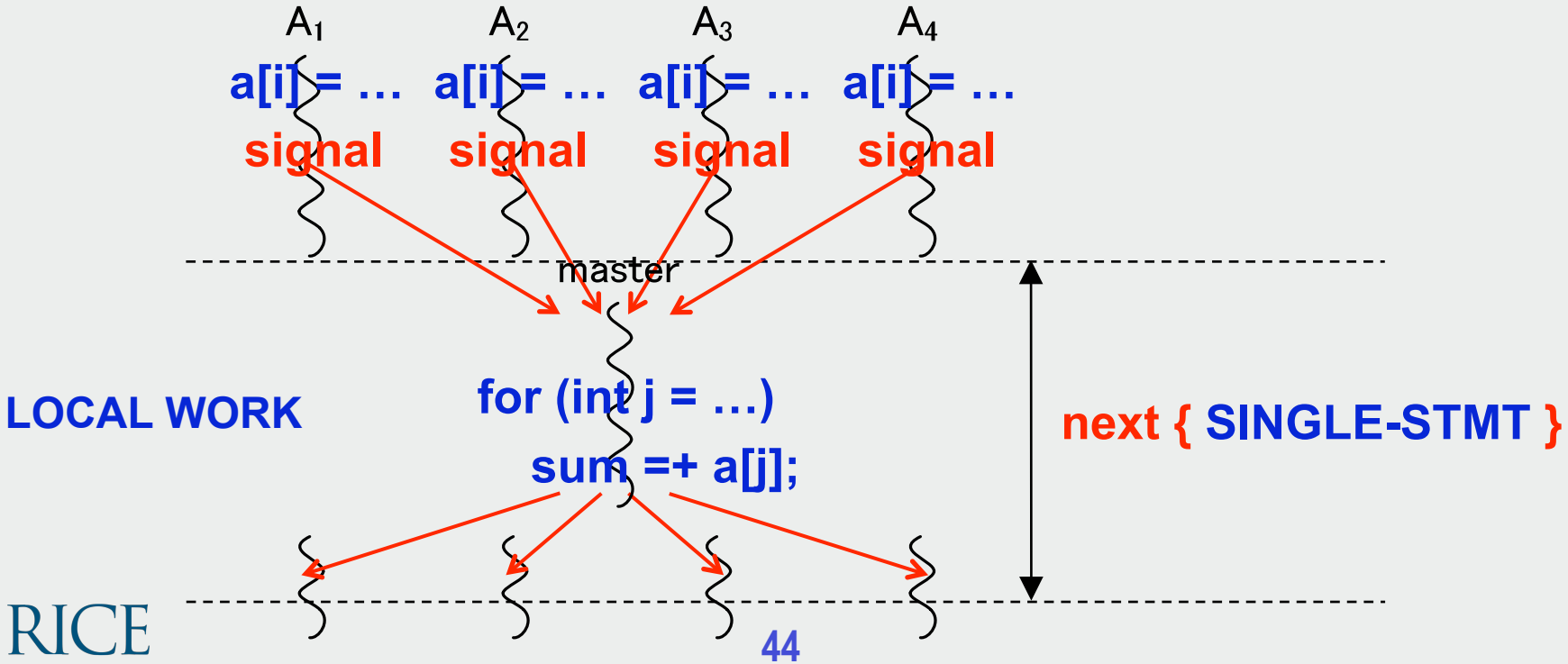


Master is selected with following priority

sig-wait-next > sig-wait > wait-only

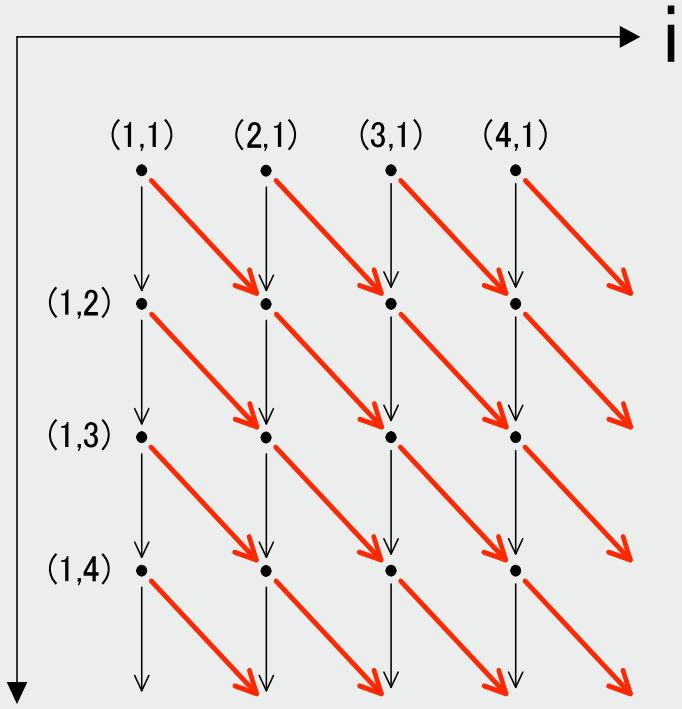
Example of next-with-single statement & Split-phase Barrier

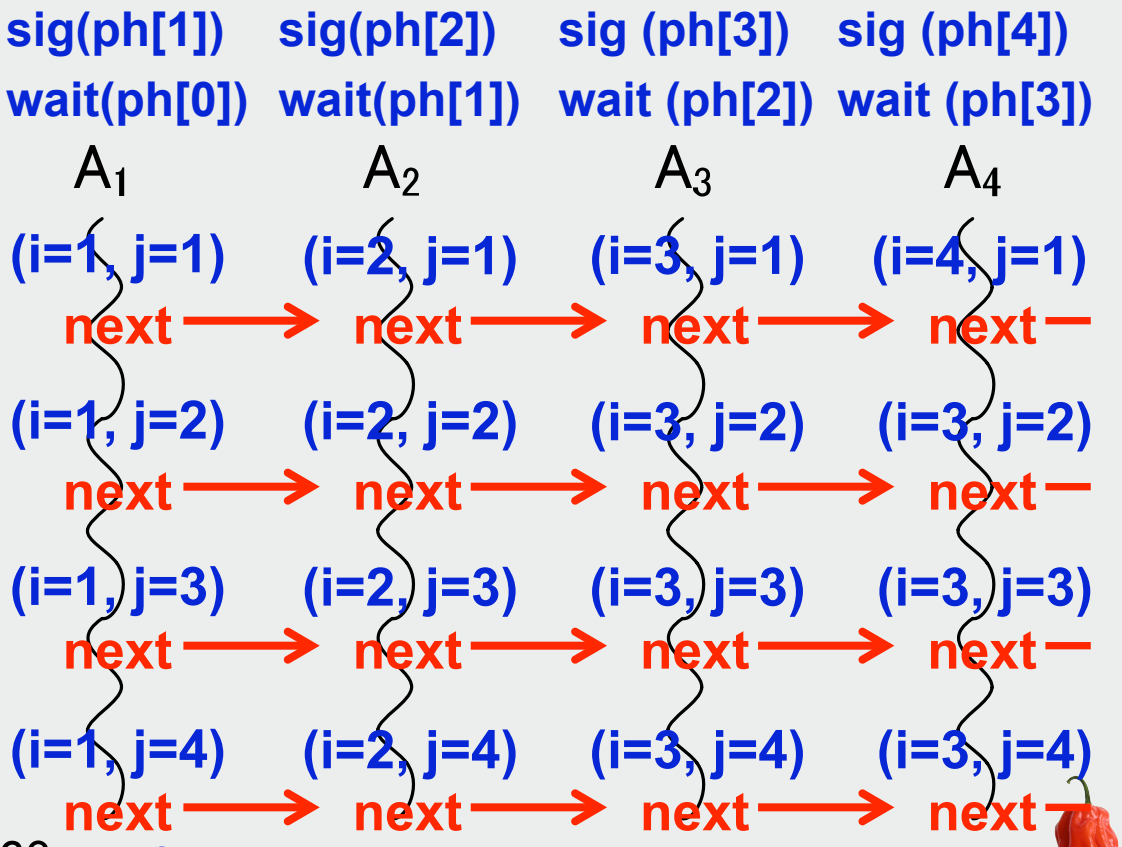
```
finish {
  phaser ph = new phaser(SINGLE);
  for (int i = 0; i < n; i++)
    async phased( ph<SINGLE> ) {
      a[i] = foo(…);
      signal;
      b[i] = F(a[i]); // LOCAL WORK …
      next { for(int j=0; j<n; j++)sum += a[j]; }
    } // async
} // finish
```



Example of Pipeline Parallelism with Phasers

```
finish {
  phaser [] ph = new phaser[m+1];
  for (int i = 1; i < m; i++)
    async phased (ph[i]<SIG>, ph[i-1]<WAIT>){
      for (int j = 1; j < n; j++) {
        a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
        next;
      } // for
} // finish
```





RICE: Loop carried dependence

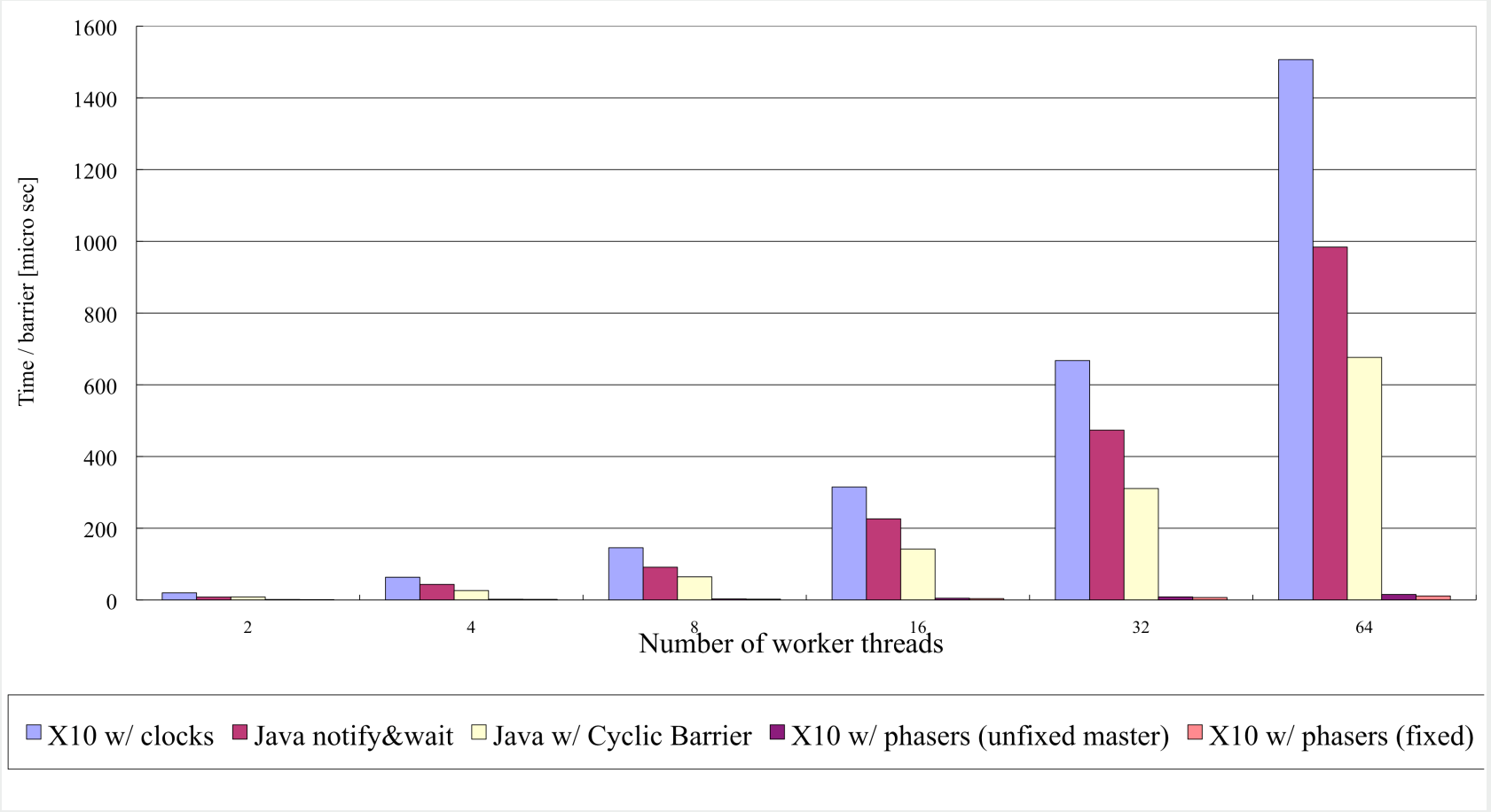
Implementation Challenges for Phasers

- Efficient performance (especially in context of JVM managed runtime)
- Support for dynamic parallelism
- Support for single statements
- Support for split-phase barriers
- Extension to reductions & collectives (in progress)

Performance Results for Phasers

- **Benchmarks**
  - BarrierBench microbenchmark
  - Java Grande Forum Benchmarks
    - LUFact, MolDyn, SOR
  - NAS Parallel Benchmarks
    - CG, MG
  - java.util.concurrent
    - BarrierJaccobi
- **Implementations used for comparison**
  - X10 w/ clocks (unoptimized reference implementation)
  - Java threads (hand-optimized synchronization)
  - X10 (Habanero) w/ Phasers (Fixed vs. Unfixed master)
  - OpenMP (only for BarrierBench microbenchmark)

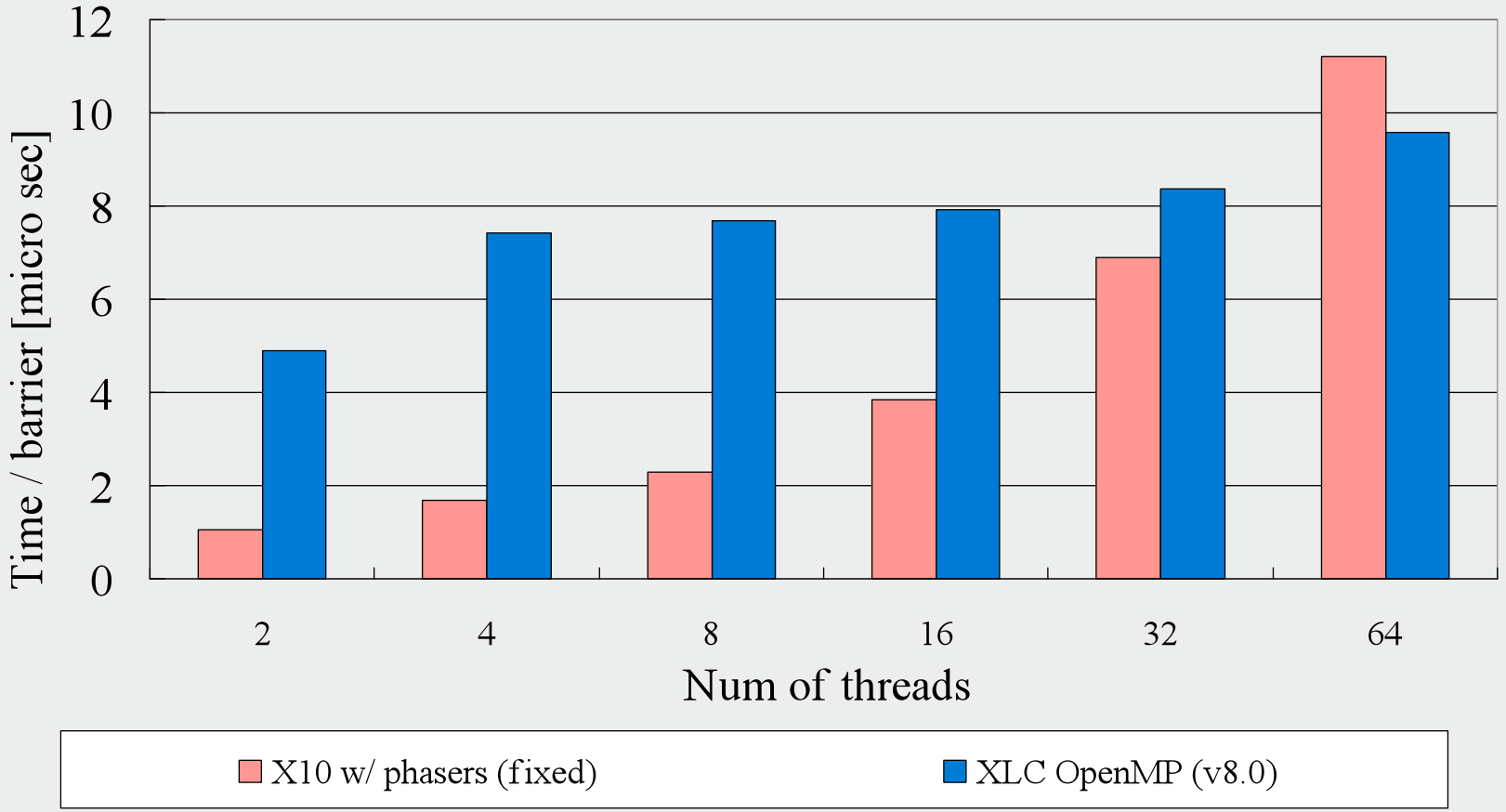
Barrier Overhead on 64-way Power5+ SMP: EPCC Java BarrierBench Microbenchmark



Barrier overhead compared to Phaser (fixed master) w/ 64 threads

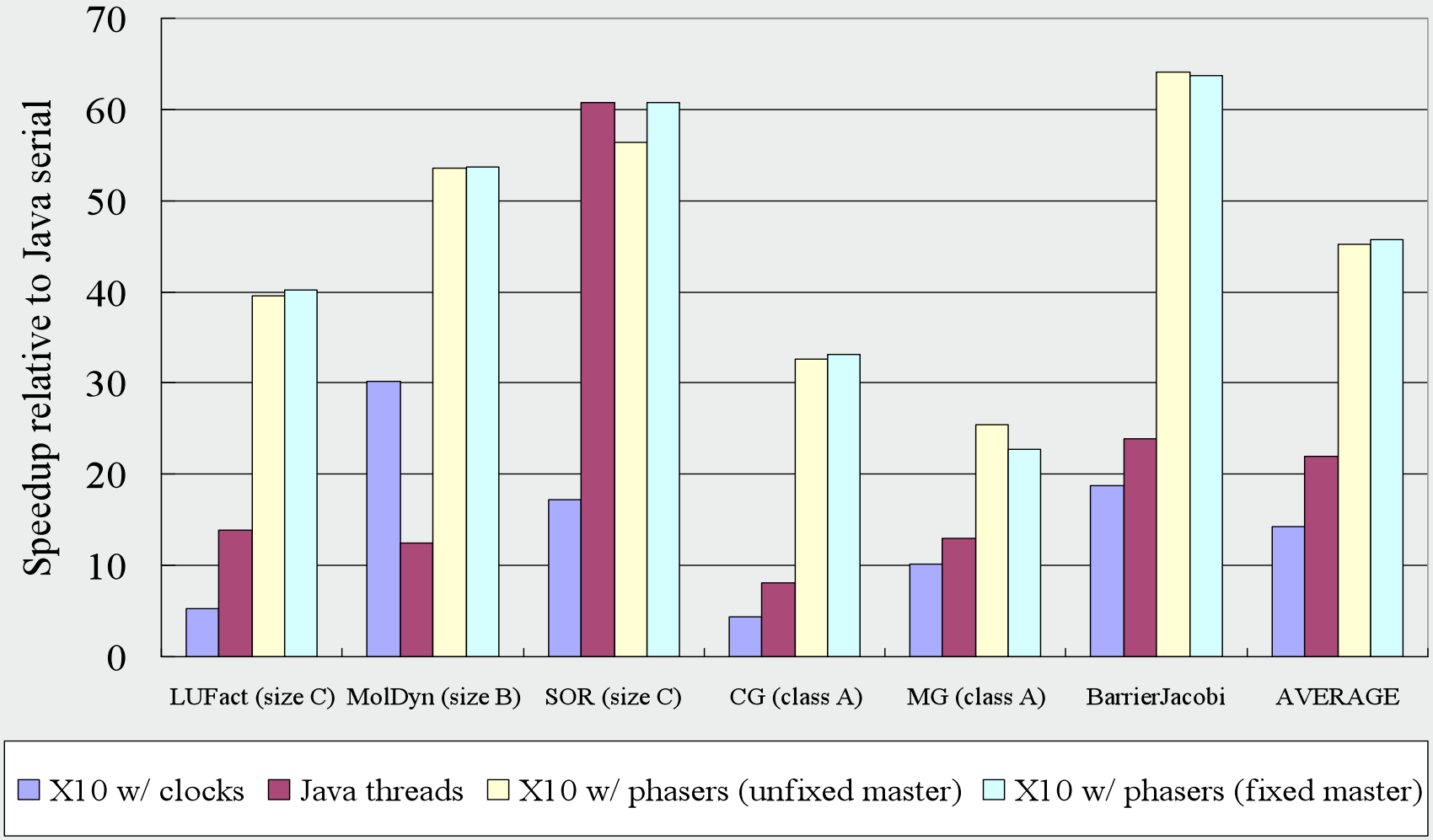
X10 clocks: **134.5x**, Java notify&wait: **87.8x,** JUC cyclic barrier: **60.4x**

Barrier Overhead on 64-way Power5+ SMP: EPCC Barrier Microbenchmark (contd)



- **Barrier overhead compared Phaser (fixed master) to OpenMP**
  - 2 to 32 threads: **Phaser's overhead is lower**
  - 64 threads; **OpenMP's overhead is lower**

Speedup on 64-way Power5+ SMP:
Java Grande Benchmarks & NAS Parallel Benchmarks



Average speedup with phasers (fixed master)

3.19x faster than X10 clocks, **2.08x** faster than Java threads

Outline

- A Sample of Modern Multicore Programming Models
- X10 + Habanero Execution Model and Implementation Challenges
- Rice Habanero Multicore Software research project

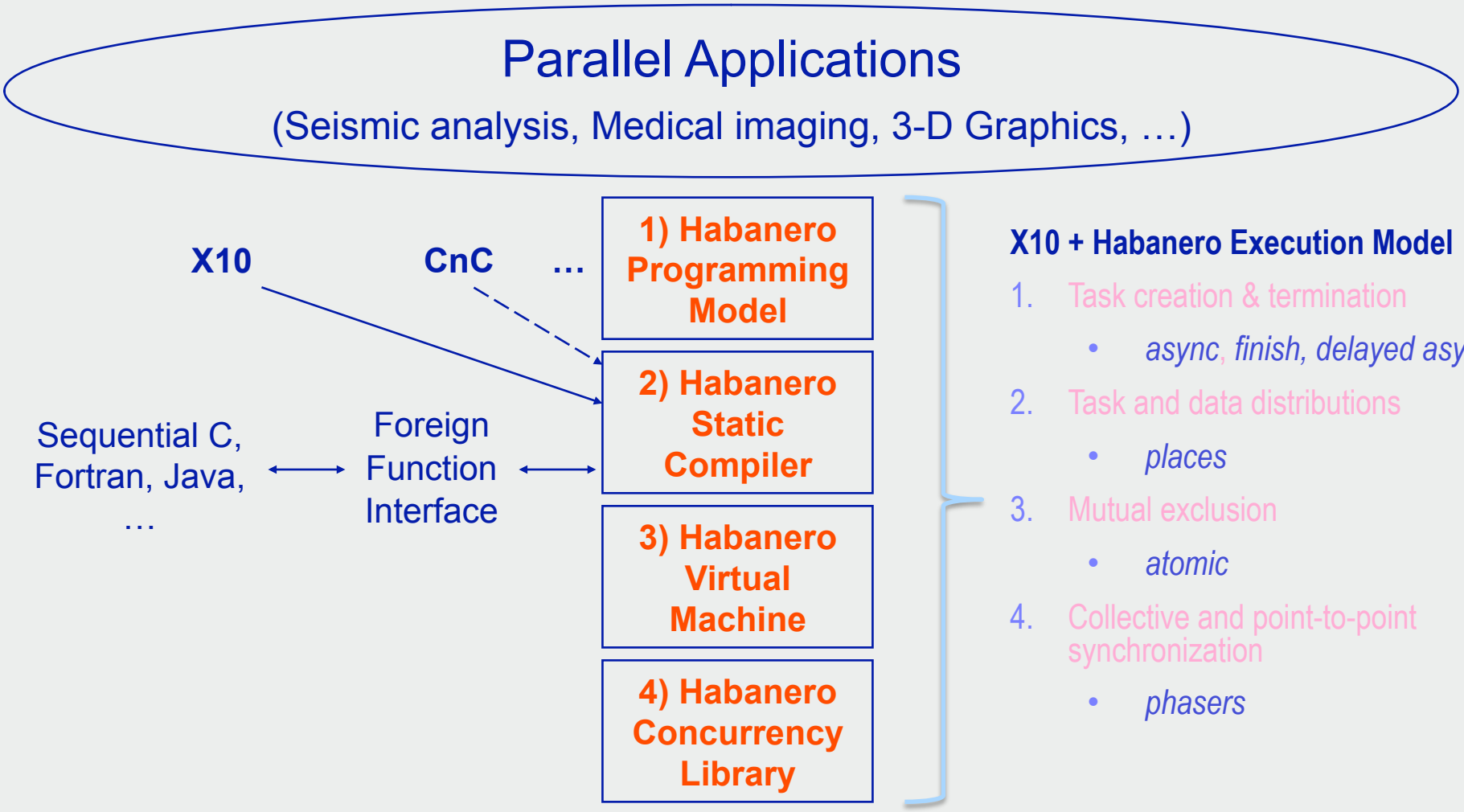
Habanero Acknowledgments

- Faculty
  - Vivek Sarkar, Bill Scherer
- Research Scientists
  - Zoran Budimlic, Chuck Koelbel
- Postdocs
  - Jun Shirako, Yonghong Yan
- Graduate Students
  - Rajkishore Barik, Yi Guo, Mack Joyner, David Peixotto, Raghavan Raman,
- Other collaborators at Rice
  - Keith Cooper, Tim Harvey, John Mellor-Crummey, Krishna Palem, Linda Torczon, Anna Youssefi, Rui Zhang

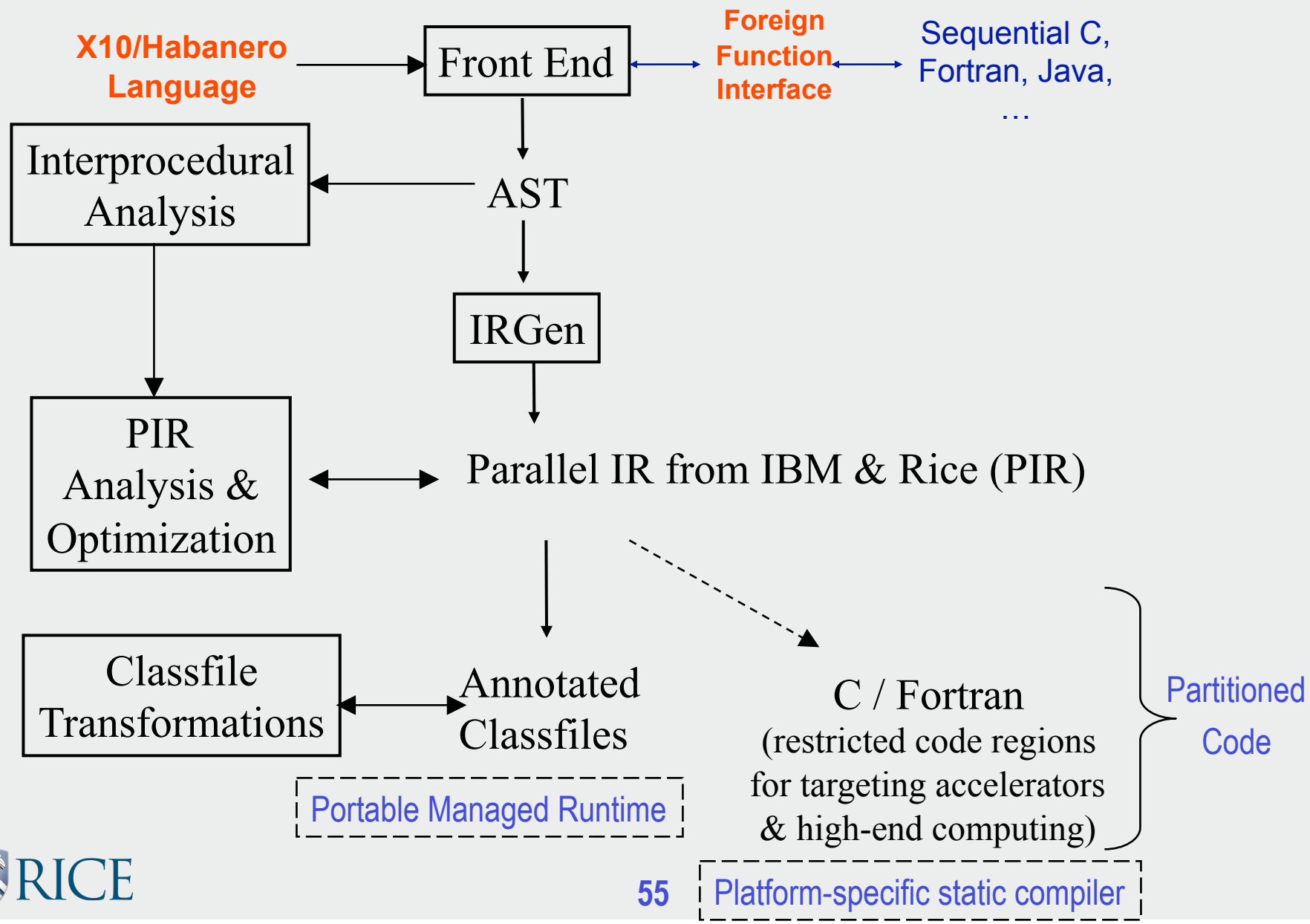
Significant Experience at Rice to tackle Multicore Software Implementation challenges

- **Languages**
  - Co-Array Fortran, X10 --- explicit parallelism
  - Sisal, Fortran 90, HPF, Matlab, LabView --- implicit parallelism
- **Static Optimizing Compilers**
  - JaMake (Java optimization)
  - D System, PTRAN, ASTI (Fortran optimization, automatic vectorization, automatic parallelization, locality & communication optimizations)
  - Massively Scalar, Trimaran (Optimizing back ends)
- **Dynamic Optimizing Compilers & Runtimes**
  - Jikes Research Virtual Machine for Java
  - Java Concurrency Utilities
- **Parallel Tools**
  - HPCToolkit
- **Applications Expertise**
  - Computer Graphics and Visualization
  - Compressive Sensing
  - Seismic Analysis
  - DoE/SciDAC applications

Habanero Project Overview (habanero.rice.edu)



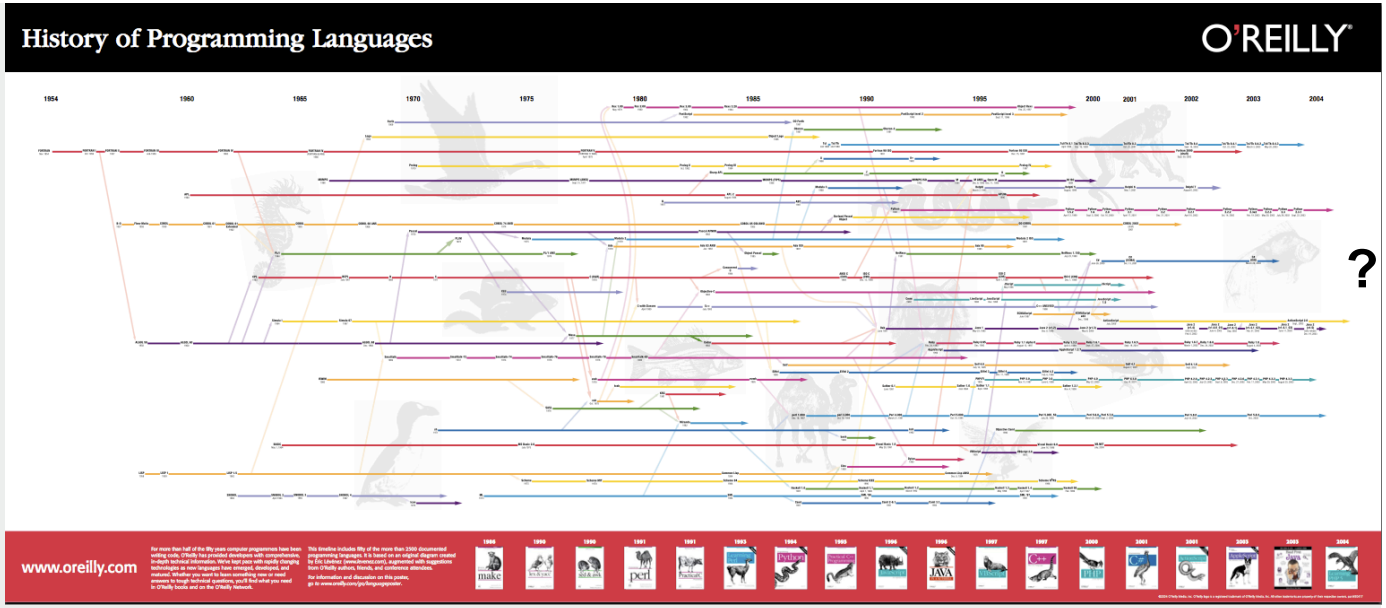
2) Habanero Static Parallelizing & Optimizing Compiler



Three Levels in PIR

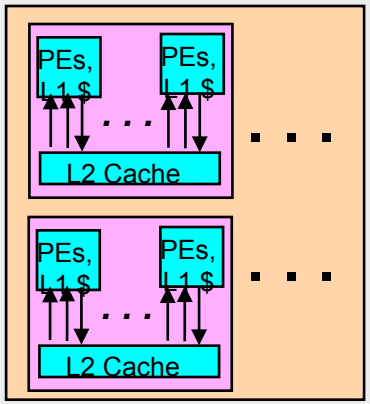
- Level 1 is a *hierarchical parallel IR* with parallel constructs expressed as new nested regions
  - Example optimization: foreach chunking with phaser operations
- Level 2 is a *flattened parallel IR* with new operators for parallel constructs
  - Example optimization: load elimination
- Level 3 is a *sequential IR with runtime calls* for concurrency constructs
  - Example optimization: optimize frame operations in runtime calls for work-stealing scheduling

Conclusion

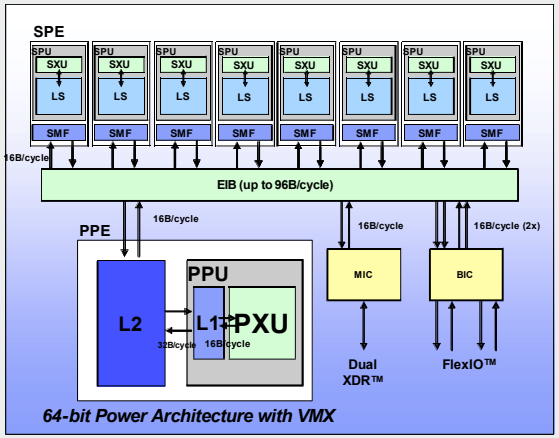


?

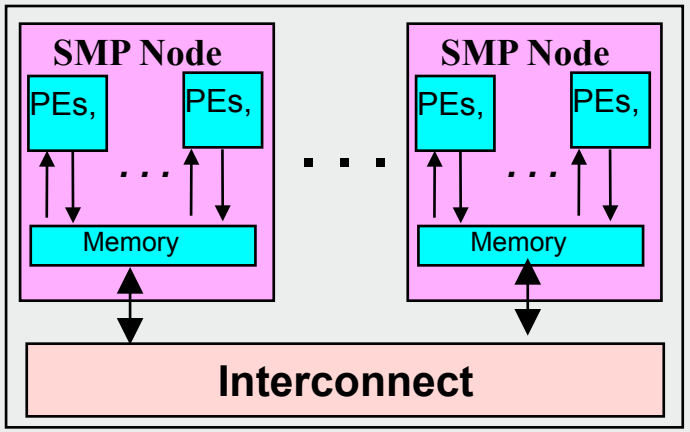
Homogeneous Multi-core



Heterogeneous Accelerators



High Performance Clusters



Advances in parallel languages, compilers, and runtimes are necessary to address the implementation challenges of multicore programming

RICE

Acknowledgments

- Charles Leiserson (Cilk slides)
- Doug Lea (j.u.c slides)
- Arch Robison (TBB slides)
- Joe Duffy, Igor Ostrovsky (TPL & PLINQ slides)
- Kathleen Knobe (Intel Concurrent Collection slides)
- IBM X10 team
- Rice Habanero team

Reminder: PPoPP 2009 deadline is fast approaching …

Welcome to PPoPP 2009

14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming

February 14-18, 2009, Raleigh, North Carolina (co-located with HPCA-15)

Conference Information

The ACM SIGPLAN 2009 Symposium on Principles and Practice of Parallel Programming (PPoPP 2009) will be held in Raleigh, NC, February 14-18, 2009. PPoPP is a forum for leading work on all aspects of parallel programming, including foundational results, techniques, tools, and practical experience. In the context of the symposium, "parallel programming" is construed to encompass work on concurrent, multithreaded, multicore, accelerated, multiprocessor, and tightly-clustered systems. Given the rise of multicore processors, PPoPP is particularly interested in work that seeks to transition parallel programming into the computing mainstream.

PPoPP'08 will be co-located with HPCA-15.

Important Dates:

Abstract Submission:	August 11, 2008 (11:59pm Eastern Daylight Time)
Full Paper Submission:	August 18, 2008 (11:59pm Eastern Daylight Time)
Poster Submission:	August 18, 2008 (11:59pm Eastern Daylight Time; no separate abstract required)
Rebuttal Period:	October 1-3, 2008
Notification of Acceptance:	October 17, 2008

RICE