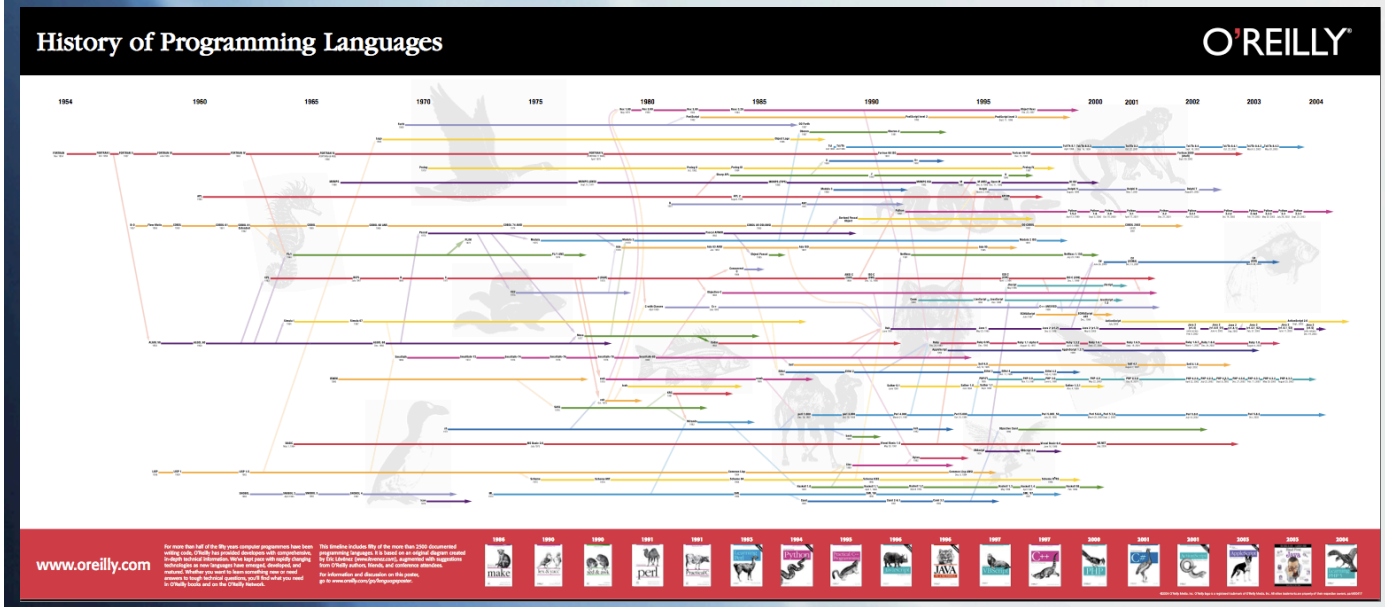
RICE George R. Brown School of Engineering
Computer Science

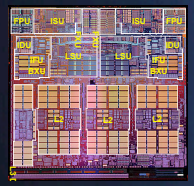
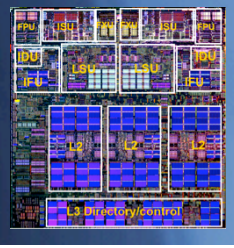
Virtual Machine and Intermediate Language Challenges for Parallelism

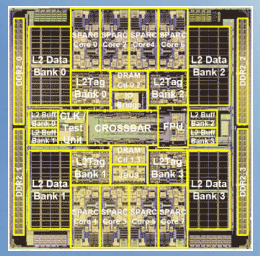


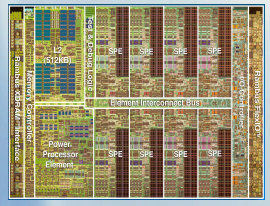
Vivek Sarkar
Rice University

vsarkar@rice.edu

Oct 25, 2009

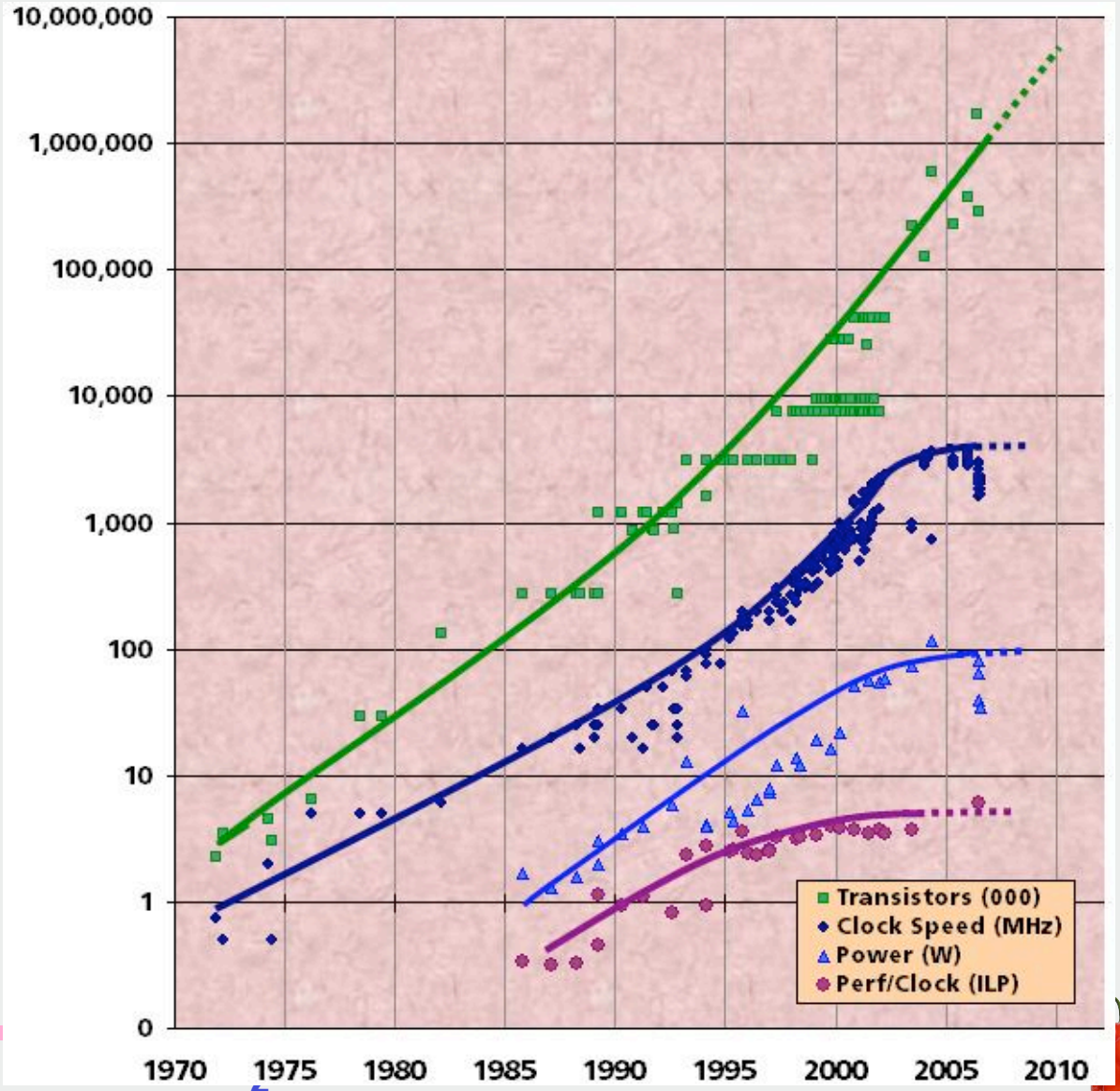








The Multicore Revolution: why Concurrency has become critical for Mainstream Computing

- Chip density is continuing to increase ~2x every 2 years
  - Clock speed is not
  - Number of processor cores is doubling instead
- There is little or no hidden parallelism (ILP) to be found
- ***Parallelism must be exposed to and managed by software***



Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)

RICE

Virtual Machines and Intermediate Languages

- VM's and IL's operate at a higher level of abstraction than machine code
- However, most mainstream VM's and IL's express parallelism using low-level primitives such as threads and locks
- Further, most parallel primitives are just represented as library calls
  - Exception: MonitorEnter, MonitorExit

Rice Habanero Multicore Software Project (habanero.rice.edu)

Parallel Applications

(Seismic analysis, Medical imaging, Finite Element Methods, …)

Challenge: Develop new programming technologies and pedagogical foundations for portable parallelism on future multicore hardware

FOCUS OF THIS TALK --- Habanero execution model:

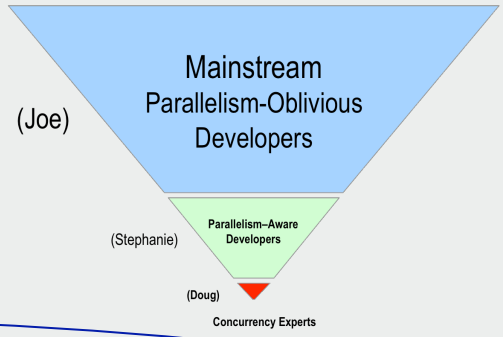
1) Lightweight dynamic task creation & termination (*async*, *finish)*
2) Collective and point-to-point synchronization (*phasers)*
3) Mutual exclusion (*isolated)*
4) Locality control --- task and data distributions *(hierarchical place tree)*

Habanero Programming Languages

Habanero Static Compiler & Parallel Intermediate Representation

Habanero Runtime & Dynamic Compiler

Two-level programming model
Implicitly Parallel Coordination Language for Joe, CnC (Intel Concurrent Collections)
+
Explicitly Parallel Programming Languages for Stephanie, Habanero-Java (from X10 v1.5) and Habanero-C

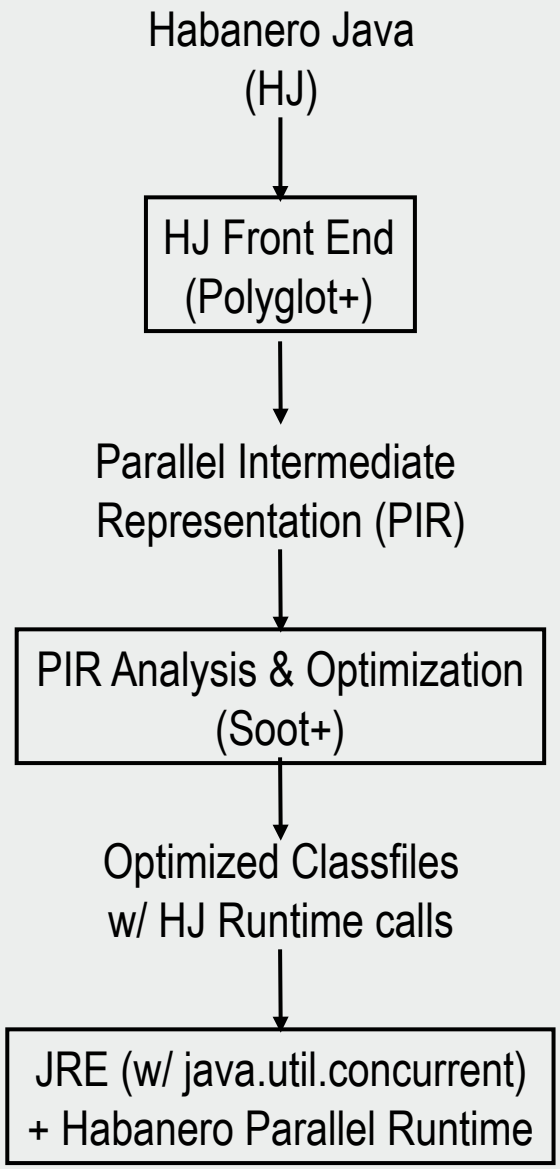


Multicore Platforms

(Cell, Clearspeed, Cyclops, GeForce, Niagara, Opteron, Power, Xeon, …)

RICE

Habanero-Java (HJ) Compiler



HJ is based on X10 v1.5 from IBM

- Extension of Java 1.4
  - Java 5 & 6 language features (generics, metadata, etc.) are currently not supported in HJ front-end
  - However, Java 5 & 6 libraries can be called --- just don't call a method that performs a blocking operation ☺
- HJ extensions are focused on parallelism and are orthogonal to the foundational sequential language
  - Habanero-C implementation in progress
  - We are looking for a suitable dynamic language that runs on a JVM as our next candidate --- suggestions?
- X10 reference: "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", OOPSLA 2005, Onward! Track.

RICE

Habanero Java (HJ) Execution Model: Portable Parallelism in Four Dimensions

1. Lightweight dynamic task creation & termination
   - *async*, *finish*, *future*, *force* (from X10)
2. Collective and point-to-point synchronization
   - *phasers* (extension of X10's clocks)
3. Mutual exclusion and isolation
   - *isolated* (extension of transactions & X10's atomic)
4. Locality control --- task and data distributions
   - *hierarchical place tree* (extension of X10's places)

HJ's Async and Finish Statements for Task Creation and Termination

async S

- Creates a new child task that executes statement S
- Parent immediately moves on to statement following the async
- If S refers to a local variable from an enclosing statement, that variable must be declared as final
- Child task cannot be aborted or cancelled
- Analogous to pthread_create()

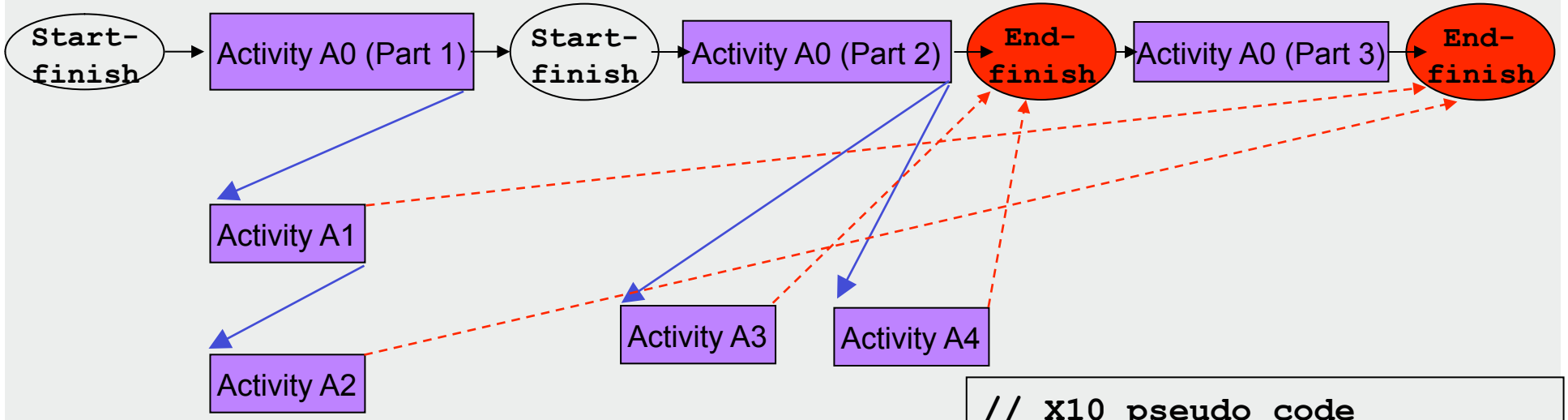
finish S

- Execute S, but wait until *all* (transitively) spawned asyncs in S's scope have terminated.
- Implicit finish between start and end of main program
- Analogous to pthread_join(), but applied to all descendant tasks

Rooted exception model

- Trap all exceptions thrown by spawned activities
- Throw an (aggregate) exception if any spawned async terminates abruptly

Async-Finish Computation Graph



Continue edge

Spawn edge

Dependence edge

```
// X10 pseudo code
finish { // start-finish
  Activity A0 (Part 1);
  async {A1; async A2;}
  finish { // start-finish
       Activity A0 (Part 2);
       async A3;
       async A4;
  } // end-finish
  Activity A0 (Part 3);
  // end-finish
}
```

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

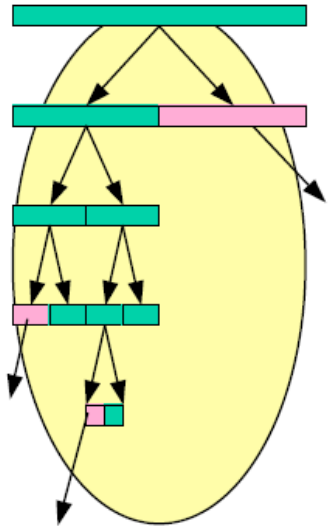
Example of Using Async-Finish to create a Parallel Loop

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


Another Example: Using Async-Finish to create a Recursive Parallel Tree Iterator

```
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

Async Expressions (Futures) for Task Creation and Termination

async<T> Expr-Block

- Creates a new child task that executes Expr-Block (which contains a return statement with a return value of type T)
- Async expression immediately returns a handle of type future<T>to parent task, which moves on to operation following the async
- Values of type future<T> can only be assigned to final variables (deadlock avoidance)
- If Expr-Block refers to a local variable from an enclosing statement, that variable must be declared as final

Expr.force()

- Evaluates Expr, and blocks if Expr's value is unavailable
- Expr must be of type future<T>
- Return value from Expr.force() will then be T
- Analogous to pthread_join() for a single task
- Exception is propagated when the future is forced

Simple Example of HJ Future and Force

- Modified version of example from "Java Concurrency Utilities in Practice", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial

```
interface Pic      { byte[] getImage(); }
interface Renderer { Pic render(byte[] raw); }

class App { // sample usage
  void app(final byte[] raw) throws ... {
    final Renderer r = …;
    final future<Pic> p = async<Pic> {return r.render(raw);}
    doSomethingElse();
    display(p.force()); // wait if not yet ready
  }
  // ...
}
```


Simple Example of Java FutureTask library

- Original example from “Java Concurrency Utilities in Practice”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial

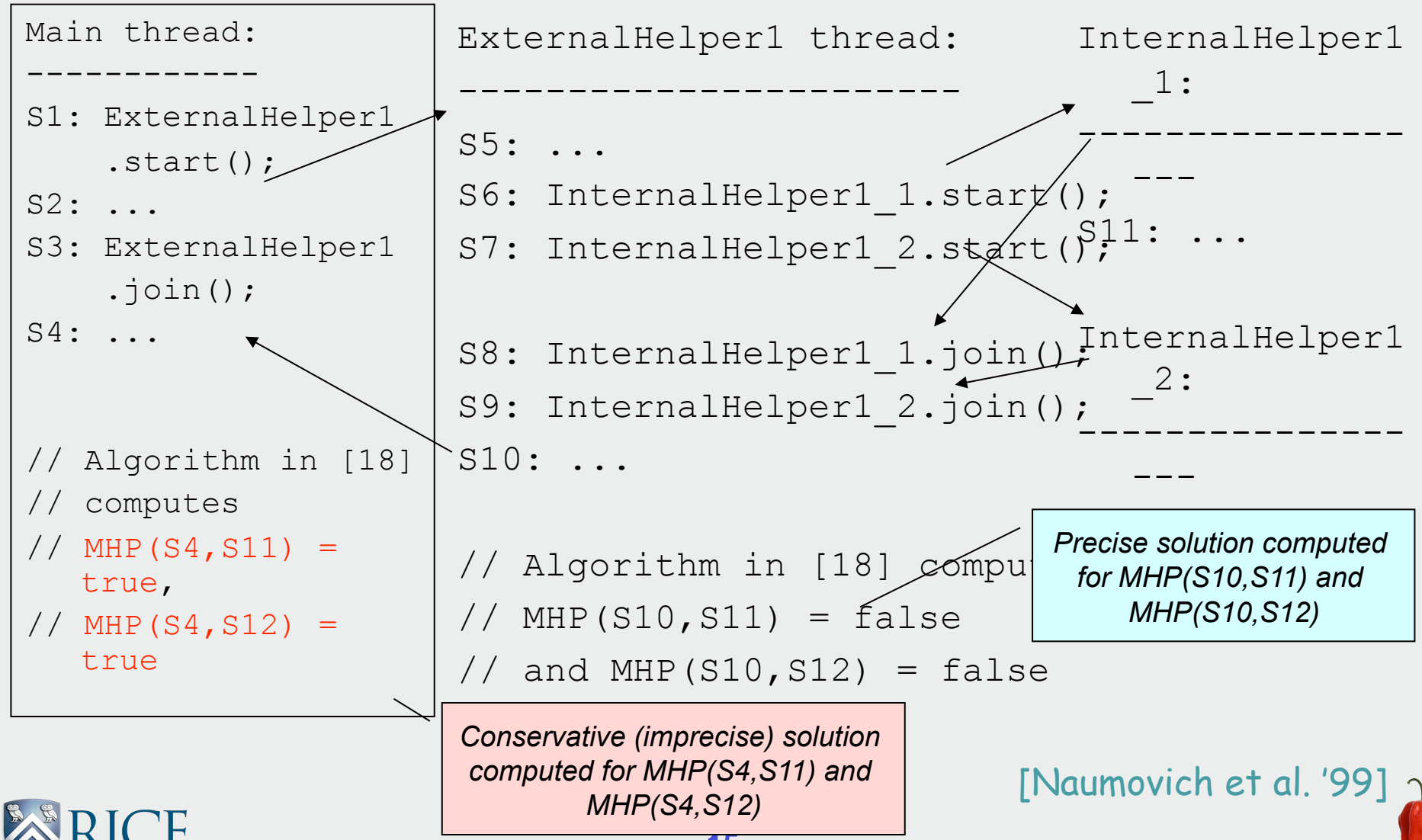
```
interface Pic      { byte[] getImage(); }
interface Renderer { Pic render(byte[] raw); }

class App { // sample usage
    void app(final byte[] raw) throws ... {
        final Renderer r = …;
        FutureTask<Pic> p = new FutureTask<Pic>(
            new Callable<Pic>() {
               Pic call() {
                  return r.render(raw);
               }
            });
        new Thread(p).start();
        doSomethingElse();
        display(p.get()); // wait if not yet ready
    }
    // ...
}
```


Concurrency obfuscated in bytecode for app() method

```
0:  new     #2; //class RendererA
3:  dup
4:  invokespecial   #3; //Method RendererA."<init>":()V
7:  astore_2
8:  new     #4; //class java/util/concurrent/FutureTask
11: dup
12: new     #5; //class App$1
15: dup
16: aload_0
17: aload_2
18: aload_1
19: invokespecial   #6; //Method App$1."<init>":
    (LApp;LRenderer;[B)V
22: invokespecial   #7; //Method java/util/concurrent/
    FutureTask."<init>":(Ljava/util/concurrent/
    Callable;)V
25: astore_3
26: new     #8; //class java/lang/Thread
29: dup
30: aload_3
31: invokespecial   #9; //Method java/lang/
    Thread."<init>":(Ljava/lang/Runnable;)V
34: invokevirtual   #10; //Method java/lang/
    Thread.start:()V
37: aload_0
38: invokevirtual   #11; //Method
    doSomethingElse:()V
41: aload_0
42: aload_3
43: invokevirtual   #12; //Method java/util/
    concurrent/FutureTask.get:()Ljava/lang/Object;
46: checkcast       #13; //class Pic
49: invokevirtual   #14; //Method display:(LPic;)V
52: goto    57
55: astore  4
57: return
```


Limitations of Library IL Interfaces for Thread Creation and Termination e.g., May-Happen-in-Parallel Analysis

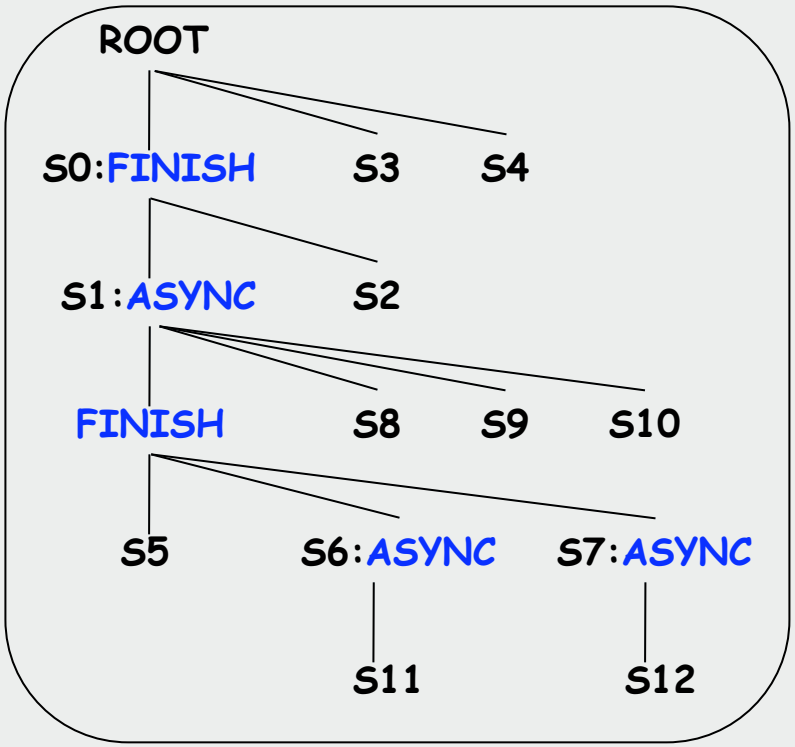


[Naumovich et al. '99]

MHP Analysis using HJ Program Structure Tree

```
S0: finish {
    S1: async {
    finish {
                    S5: ...
                    S6: async S11
                    S7: async S12
                    ...
    }
    S8: ...
    S9: ...
    S10: ...
    }
    S2: ...
}
S3: ...
S4: ...
```

Program Structure Tree



MHP(S4, S11) = false
MHP(S4, S12) = false

"May-happen-in-parallel analysis of X10 programs", S.Agarwal et al. PPoPP 2007.

Habanero Parallel Intermediate Representation (PIR)

- Habanero PIR includes explicit operators for
  1. Async
  2. Finish
  3. Force
  4. Forall
  5. Foreach (like forall without a finish)

     (Forall and foreach operators are later lowered to finish-async)
- Similar extensions will be needed for virtual machine intermediate languages

Habanero Java (HJ) Execution Model: Portable Parallelism in Four Dimensions

1. Lightweight dynamic task creation & termination
   - *async*, *finish, future, force* (from X10)
2. Collective and point-to-point synchronization
   - *phasers* (extension of X10's clocks)
3. Mutual exclusion and isolation
   - *isolated* (extension of transactions & X10's atomic)
4. Locality control --- task and data distributions
   - *hierarchical place tree* (extension of X10's places)

Past Limitations in Collective and Point-to-point Synchronization (Examples)

- **OpenMP**
  - Support for barriers
  - No support for producer-consumer (point-to-point) synchronization and dynamic parallelism
- **Java 5 Concurrency**
  - CyclicBarrier JUC library, wait-notify synchronization
  - Possibility of deadlock, no dynamic parallelism
- **Intel Thread Building Blocks**
  - Support for generic parallel algorithms, concurrent containers, and low-level synchronization
  - No support for barrier or producer-consumer synchronization

Overview of Phasers

- The phaser construct addresses past limitations by supporting
  - General m→n synchronization patterns including barriers and point-to-point
  - Multiple modes for a task to register on a phaser (SIG, WAIT, SIG_WAIT)
  - Dynamic parallelism (set of registered tasks can vary dynamically)
  - A integrated "next" statement by a task to advance each phaser that it is registered on according to its registration mode
  - Single-instance statements that are executed once during phase transitions
  - A signal statement for "split-phase barriers"
  - Guaranteed deadlock avoidance

Overview of Phasers (contd)

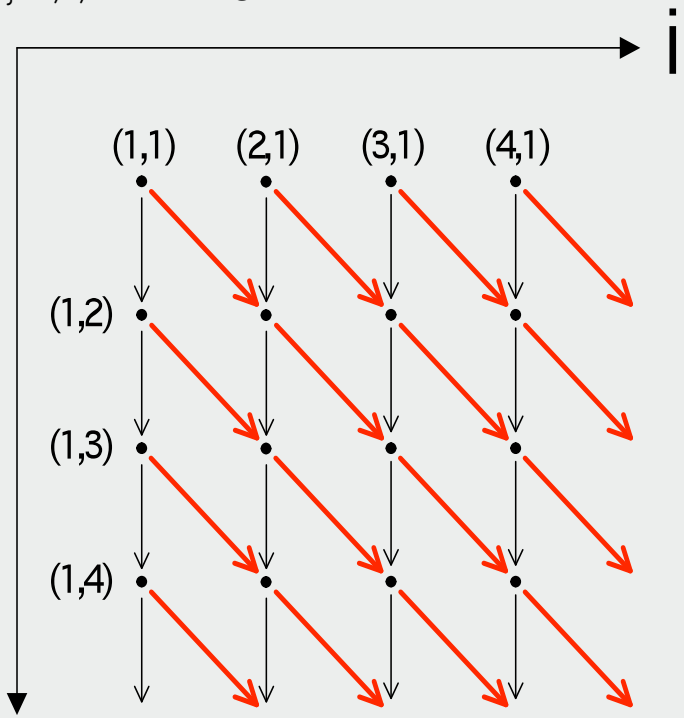
- Amenable to efficient implementation
  - Prototype implementation in Rice Habanero project
  - Implementation of a variant of phasers by Doug Lea in new JSR 166 Phaser library
- References
  - "Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization", J.Shirako, D.Peixotto, V.Sarkar, W.Scherer, ICS 2008
  - "Phaser Accumulators: a New Reduction Construct for Dynamic Parallelism", J.Shirako, D.Peixotto, V.Sarkar, W.Scherer, IPDPS 2009

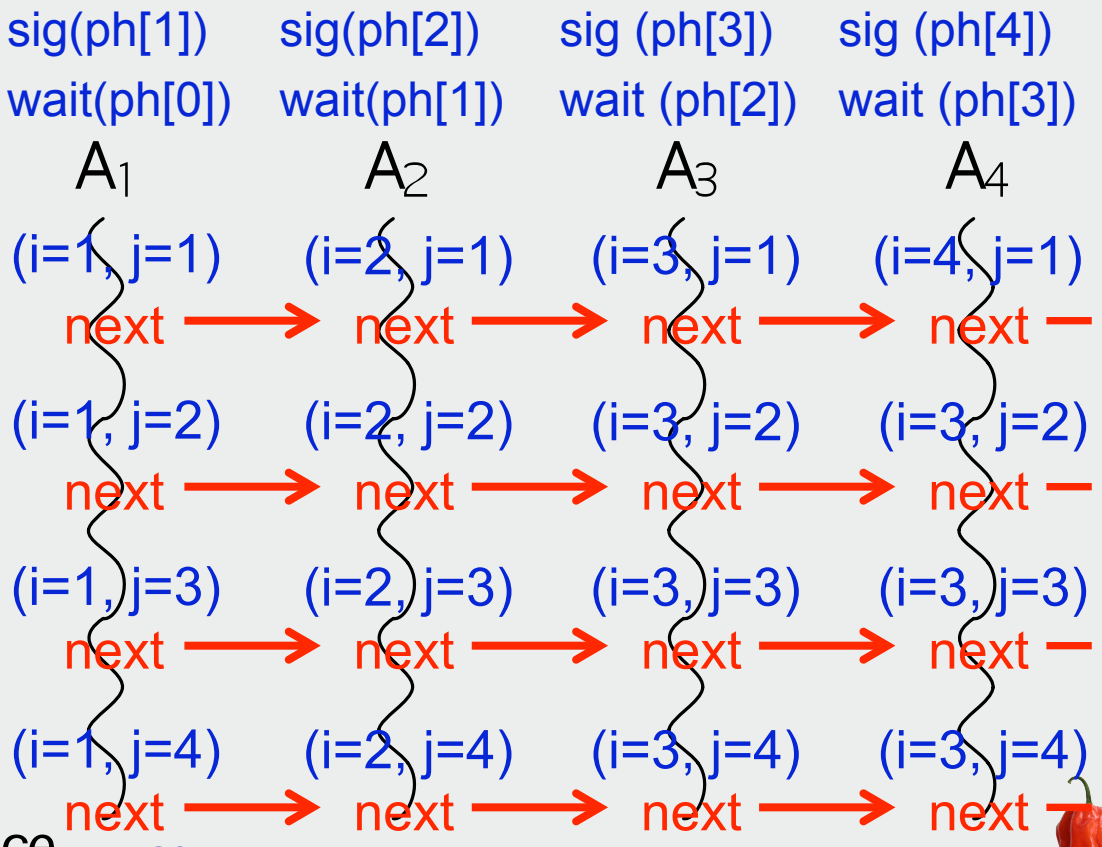
Example: 1D SOR w/ Split-phase Barrier and Single Statement

```
final int n_per_thread = n/t; ph = new phaser(SINGLE);
foreach phased (point [index] : [0:t-1]) {
  int start = index*n_per_thread + 1;
  for (int i=0; i<iterations; i++) {
    // Update local boundary values
    int j=start;              myNew[j]=(myVal[j-1] + myVal[j+1])/2.0;
    j=start+n_per_thread-1; myNew[j]=(myVal[j-1] + myVal[j+1])/2.0;
    signal; // Split-phase barrier
    // Update local interior values
    for (int j=start+1; j<start+n_per_thread-1; j++)
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
    next single {
     System.out.println(myVal[n/4] + " " + myVal[n/2]);
    } // Barrier + single statement
  } // for
} // foreach
```


Example of Pipeline Parallelism with Phasers

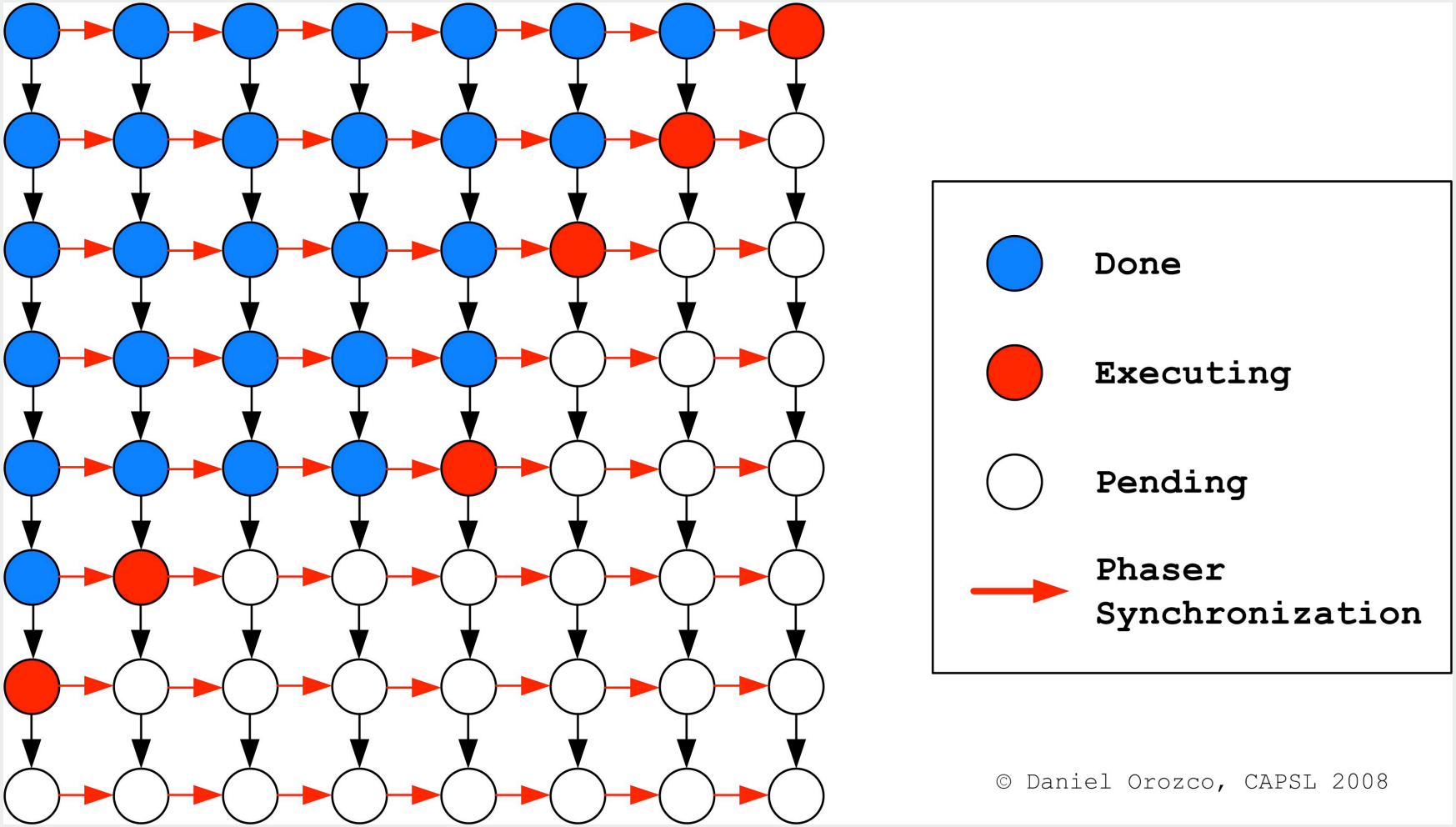
```
finish {
  phaser [] ph = new phaser[m+1];
  for (int i = 1; i < m; i++)
    async phased (ph[i]<SIG>, ph[i-1]<WAIT>){
      for (int j = 1; j < n; j++) {
        a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
        next;
      } // for
} // finish
```





RICE: Loop carried dependence

Example of Pipeline Parallelism with Phasers (contd)



Phasers can also be used for computations on irregular synchronization structures e.g., unstructured meshes

Habanero Parallel Intermediate Representation (PIR)

- Habanero PIR includes explicit operators for
  - next
  - signal
  - start and end of single statement
- Similar extensions will be needed for virtual machine intermediate languages

Habanero Execution Model: Portable Parallelism in Four Dimensions

1. Lightweight dynamic task creation & termination
   - *async*, *finish*
2. Collective and point-to-point synchronization
   - *phasers*
3. Mutual exclusion and isolation
   - *isolated*
4. Locality control --- task and data distributions
   - *places*

HJ isolated statement

isolated <body>

- Only one task can execute an isolated statement at a time
  - ➔<body> is executed in isolation of other instances of <body>
  - ➔Guarantees mutual exclusion between any pair of isolated statement instances
  - ➔*Isolation consistency memory model* (weak atomicity)
    - ➔no guarantee on interactions with non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested; use of isolated for inner statements is redundant
- Isolated statements must not contain any other parallel statement e.g., forall, async, finish, force, next

Scalar Replacement Scenarios for non-volative variables

Can the read in Line 4 reuse the value written in Line 2?

```
1: final A a = new A()
2: a.f = …
3: async { … }
4: … = a.f
```

Case 1

```
1: final A a = new A ()
2: a.f = …
3: async { if (...) a.f = F (a.f) }
4: … = a.f
```

Case 2

```
1: final A a = new A ()
2: a.f = …
3: finish async { a.f = …}
4: … = a.f
```

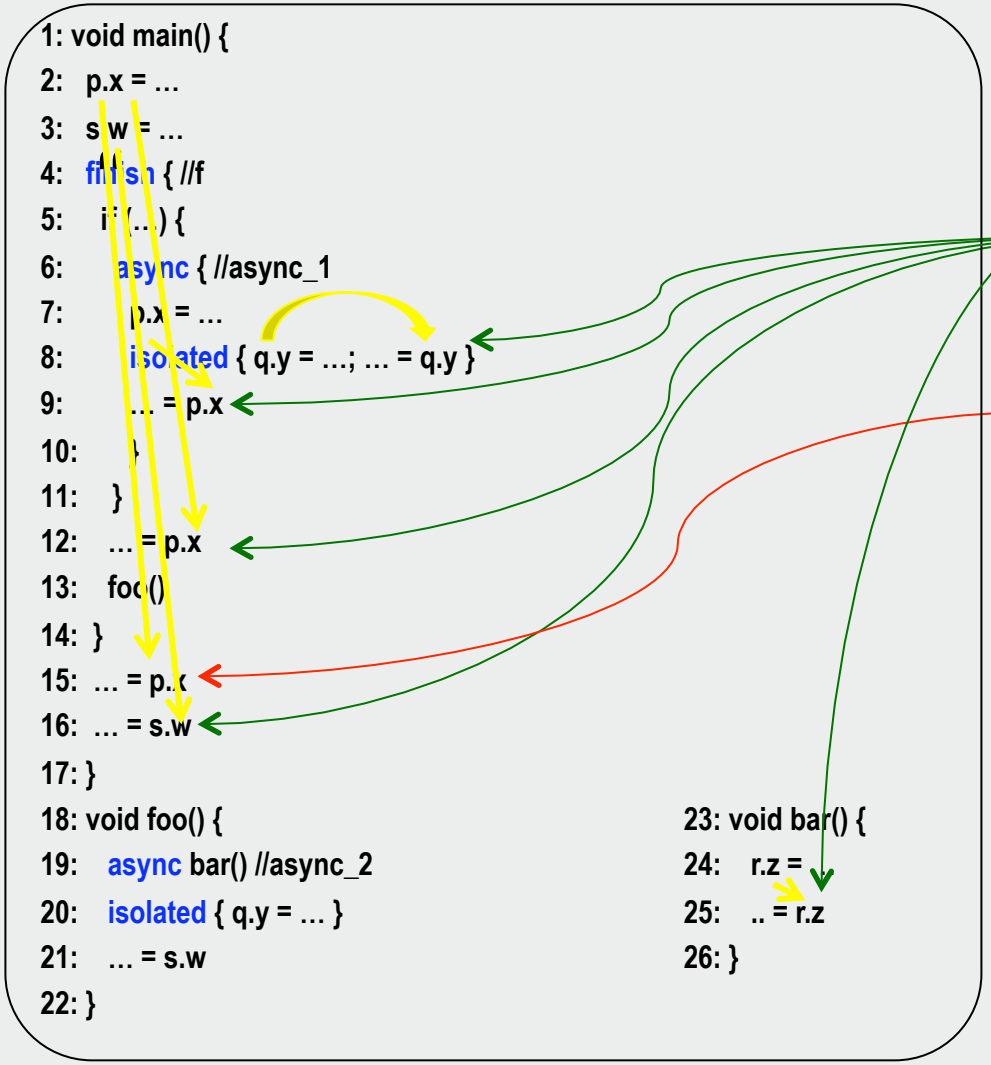
Case 3

```
1: final A a = new A ()
2: a.f = …
3: async { isolated if (…) a.x++ }
4: … = a.f
```

Case 4

Scalar replacement is legal for cases 1, 2, 4 in Isolation Consistency Memory model

Scalar Replacement for Load Elimination Example



Can be replaced by a scalar

Cannot be replaced by a scalar

Effectiveness of scalar replacement algorithm is critically dependent on high-level PIR primitives for parallelism

"Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs". Rajkishore Barik, Vivek Sarkar. PACT 2009.

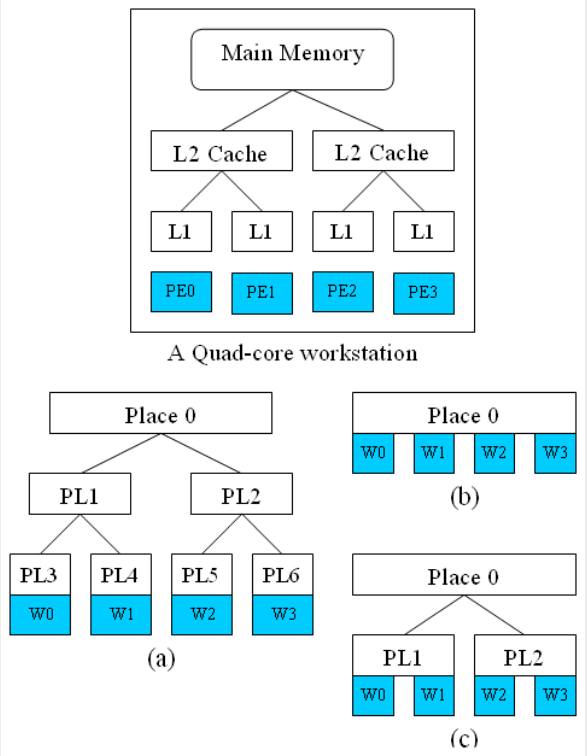
Habanero Execution Model: Portable Parallelism in Four Dimensions

1. Lightweight dynamic task creation & termination
   - *async*, *finish*
2. Collective and point-to-point synchronization
   - *phasers*
3. Mutual exclusion and isolation
   - *isolated*
4. Locality control --- task and data distributions
   - *places*

Hierarchical Place Trees (HPT)

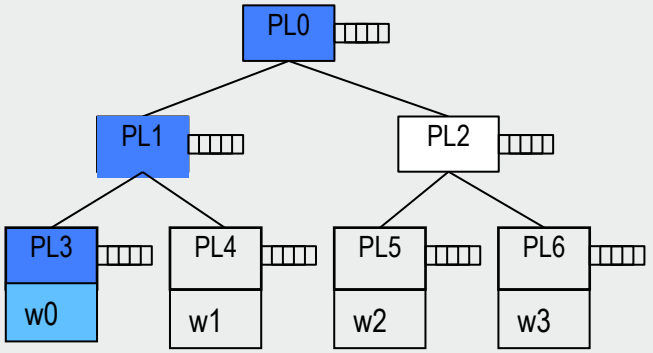
- Past approaches
  - Flat single-level partition e.g., HPF, X10
  - Hierarchical memory model with static parallelism e.g., Sequoia
- Our approach
  - Hierarchical memory + Dynamic parallelism
- Place denotes memory hierarchy level
  - Cache, SDRAM, device memory, …
- Leaf places include worker threads
  - e.g., W0, W1, W2, W3
- Multiple HPT configurations
  - For same hardware and programs
  - Trade-off between locality and load-balance

"Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement", Y.Yan et al, LCPC 2009



HPT locality-aware scheduling

- Workers attached to leaf places
  - Bind to hardware core
- Each place has a queue
  - async <*pl*> <*stmt*>: push task onto *pl*'s queue



- A worker executes tasks from ancestor places from bottom-up
  - W0 executes tasks from PL3, PL1, PL0
- Tasks in a place queue can be executed by all workers in the place's subtree
  - Task in PL2 can be executed by workers W2 or W3

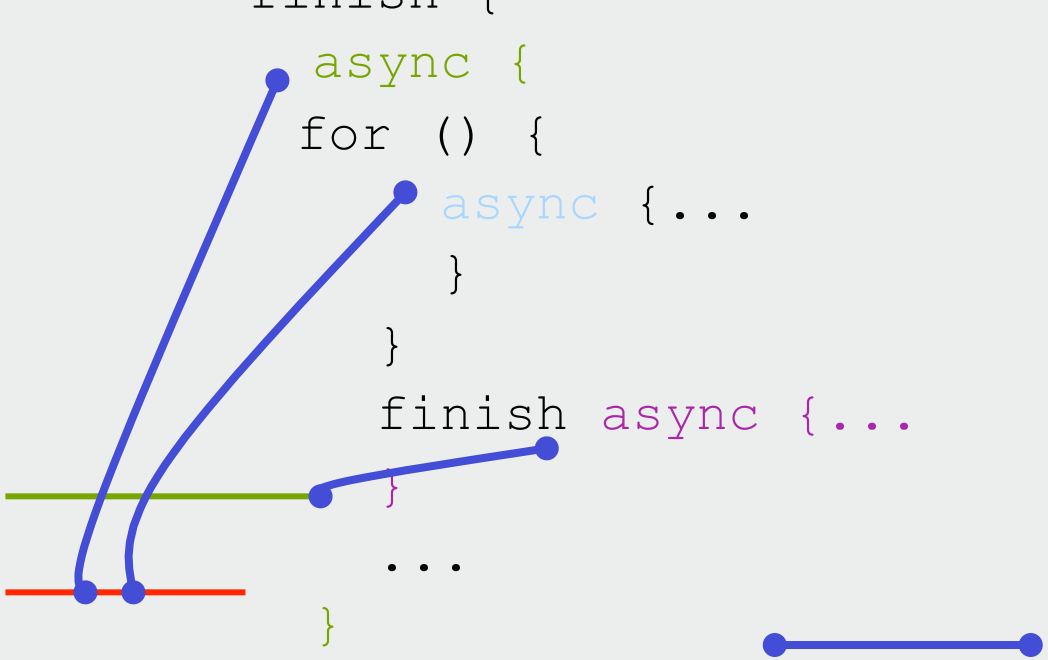
Data transfers in HPT

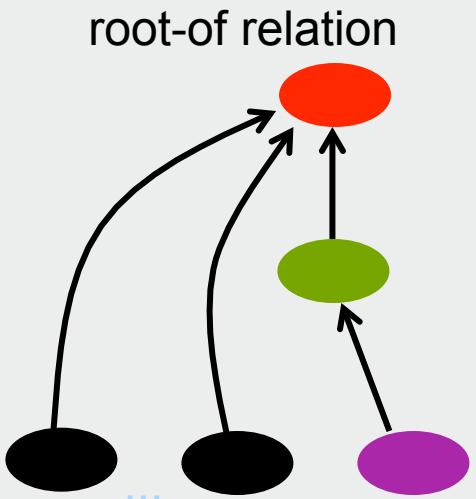
Three data transfer interfaces:

1. *Implicit data transfer through data distribution*
   - Data can be distributed (e.g., block/cyclic) at each level of hierarchical place tree
   - e.g., used to model cache-based memory hierarchies
2. *Explicit data transfer using synchronous copy-in / copy-out*
   - Syntax: **async** [<*pl*>] **IN** (…) **OUT** (…) **INOUT** (…) <*stmt*>
   - e.g., used to model memory-to-memory transfers for accelerators such as GPGPUs
3. *Explicit data transfer using asynchronous memory copy*
   - Syntax: asyncMemcpy(dest, src);
   - e.g., used to model inter-processor DMA (direct memory access)

Exception model (from X10 v1.5)

```
public void main (String[] args) {
  ...
  finish {
    async {
    for () {
        async {...
          }
      }
      finish async {...
      }
      ...
    }
  } // finish
}
```





exception flow along root-of relation

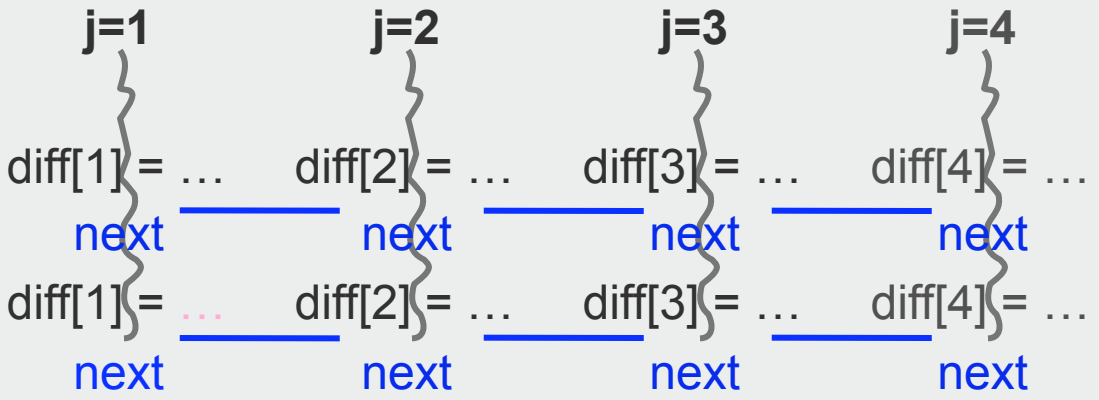
Exceptions that escape an async are propagated to the nearest enclosing finish block

hj.lang.MultipleExceptions (from X10 v1.5)

```
int result = 0;
try {
   finish {
     ateach (point [i]:dist.factory.unique()) {
       throw new Exception ("Exception from "+here.id)
     }
     result = 42;
   } // finish
} catch (hj.lang.MultipleExceptions me) {
   System.out.print(me);
}
assert (result == 42); // always true
```


Problem: how to chunk foreach loops with "next" statements?

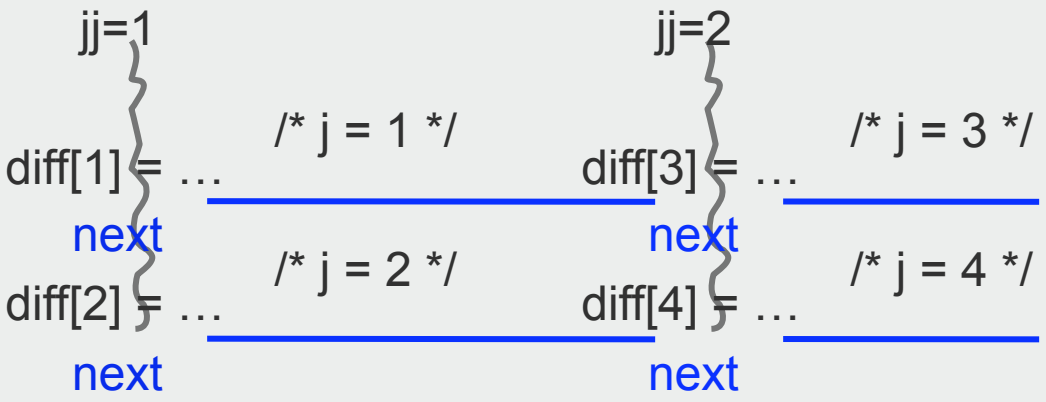
```
delta = epsilon+1; iters = 0;
phaser ph = new phaser(SINGLE);
foreach ( point[j] : [1:n] ) phased(ph<SINGLE>) {
    while ( delta > epsilon ) {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
        diff[j] = Math.abs(newA[j]-oldA[j]);
        next { // barrier with single statement
            delta = diff.sum(); iters++;
            temp = newA; newA = oldA; oldA = temp;
        }
    }
}
```



- **Synchronization with fine-grained dynamic parallelism**
  - Incurs significant overhead if n is larger than # hardware threads
  - "Chunking parallel loops in the presence of synchronization", J.Shirako et al, ICS 2009.

Incorrect Loop Chunking (Semantics of code is changed)

```
delta = epsilon+1; iters = 0;
phaser ph = new phaser(SINGLE);
foreach ( point[jj] : [1:n:S] ) phased(ph<SINGLE>) {
    for (int j = jj ; j <= min(jj+S-1,n) ; j++) {
        while ( delta > epsilon ) {
            newA[j] = (oldA[j-1]+oldA[j+1])/2.0 ;
            diff[j] = Math.abs(newA[j]-oldA[j]);
            next { // barrier with single statement
               delta = diff.sum(); iters++;
               temp = newA; newA = oldA; oldA = temp;
            }
        }
    }
}
```



- **Goal: Correct chunking of parallel loops with**
  - Barrier and point-to-point synchronization in loop body
  - Potential exceptions in loop body

Overview of Parallel Loop Chunking

- **Step 1:**

  Apply strip mining to the parallel loop

- **Step 2:**

  Isolate *next* synchronization statements

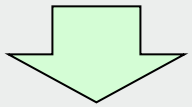
  - **Legal combinations of Loop transformations**

- **Step 3:**

  Serialize inner parallel loops

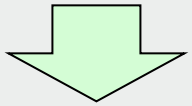
* i-foreach represents “inner foreach” that is used only during transformation

```
phaser ph = new phaser();
foreach (point i:[1:10000]) phased {
  STMT1
  next;
  STMT2
}
```



Step 1

```
phaser ph = new phaser();
foreach (point g:[1:10]) phased {
  i-foreach (point i:…) phased {
    STMT1
    next;
    STMT2
  }
}
```



Step 2

```
phaser ph = new phaser();
foreach (point g:[1:10]) phased {
  i-foreach (point i:…) phased {
    STMT1
  }
  next;
  i-foreach (point i:…) phased {
    STMT2
  }
}
```


Extensions for Exceptions

- **Semantics for exceptions in chunked loops**
  - Original iterations: Parallel
  - Chunked iterations: Serialized
    - Must perform all iterations and collect all exceptions

```
// Original
phaser ph = new phaser();
foreach (point i:R) phased {
  STMT1 //May throw exception
  …
```

```
// After loop chunking
phaser ph = new phaser();
foreach (point g:lg(R)) phased {
  for (point i : le(R, g))
    STMT1 //May throw exception
  …
```

- **Modified transformation rules for i-foreach**
  - e.g., **Loop distribution**

```
i-foreach (p: ie(R,g)) phased {
   S1;   S2;
}
```

Exception-free

```
i-foreach (p: ie(R,g)) phased
   S1;
i-foreach (p: ie(R,g)) phased
   S2;
```

Handling exceptions

```
boolean exFlag[] = new boolean [R.size()];
i-foreach (p: ie(R,g)) phased
   try { S1; }
   catch (Exception e) {
     exFlag[p] = true;
     pushException(e);
   }
i-foreach (p: ie(R,g)) phased
  if (!exFlag[p]) try { S2; } catch …
```


Other Topics

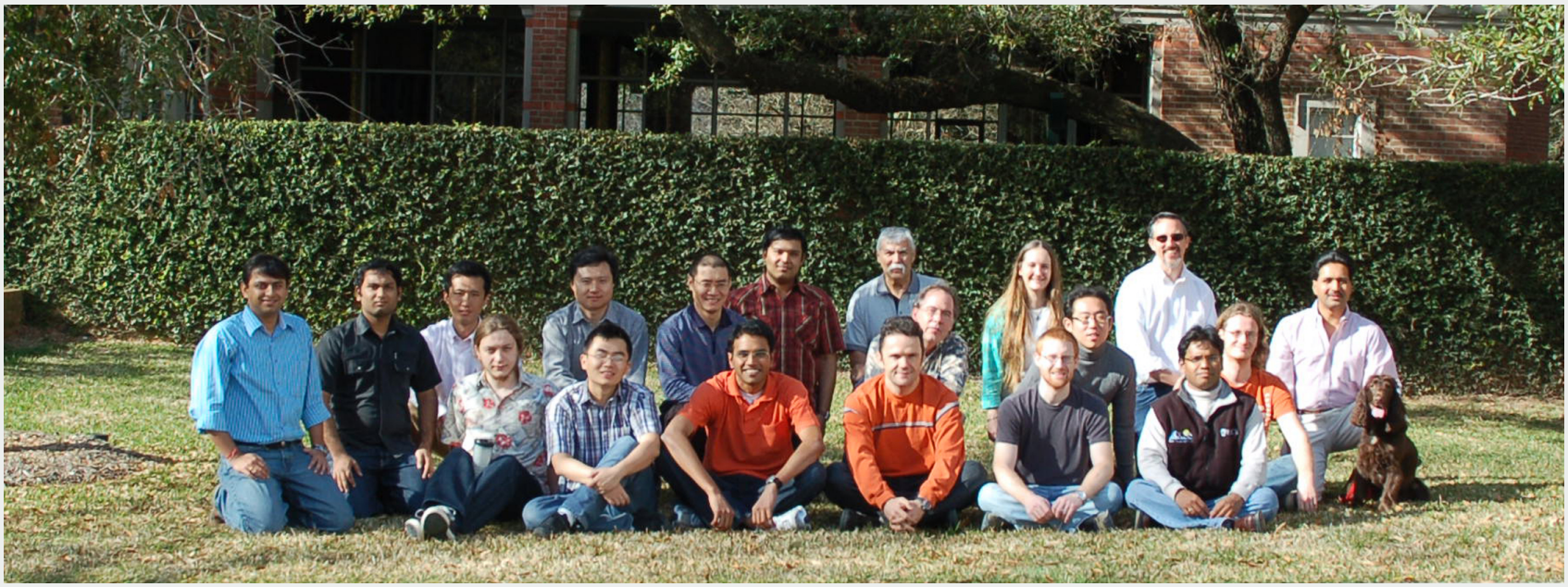
- Lower-level VMIL primitives for parallelism
  - CAS, fence, …
- VM support for parallelism
  - Concurrent garbage collection, concurrent JIT compilation, …
- For more details on the the Habanero project, come to the Habanero Poster at the OOPSLA Poster Session on Monday (Oct 26), 5:30pm–7:30pm, Fantasia Ballroom

Habanero Team

- Faculty
  - Vivek Sarkar, Bill Scherer
- Senior Research Scientist
  - Michael Burke
- Research Scientists
  - Zoran Budimlić, Philippe Charles, Chuck Koelbel
- Research Programmer
  - Vincent Cavé
- Postdocs
  - Jun Shirako, Yonghong Yan, Jisheng Zhao
- PhD Students
  - Rajkishore Barik, Sanjay Chatterjee, Yi Guo, Shams Imam, David Peixotto, Raghavan Raman, Dragoș Sbîrlea, Kamal Sharma, Sağnak Taşırlar
- Undergraduate Students
  - Max Grossman, Jarred Payne
- Other collaborators at Rice
  - Laksono Adhianto, Rich Baraniuk, Keith Cooper, Tim Harvey, John Mellor-Crummey, Krishna Palem, Mona Sheikh, Alina Simion, Walid Taha, Linda Torczon, Anna Youssefi, Rui Zhang, Ryan Zhang, Fengmei Zhao, Lin Zhong, …

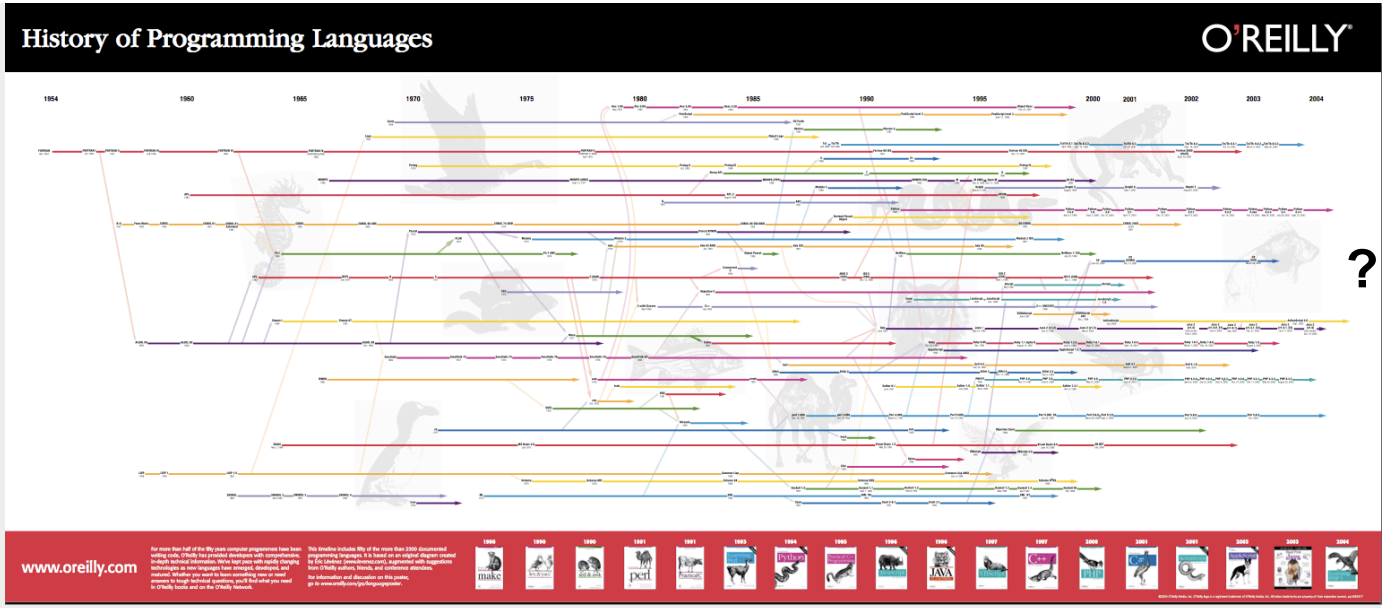
RICE

Habanero Team Pictures



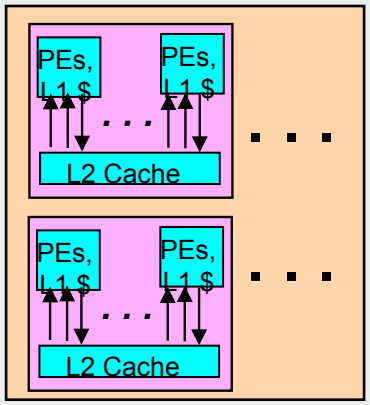
Send email to Vivek Sarkar (vsarkar@rice.edu) if you are interested in a PhD or postdoc position in the Habanero project, or in collaborating with us!

Conclusion

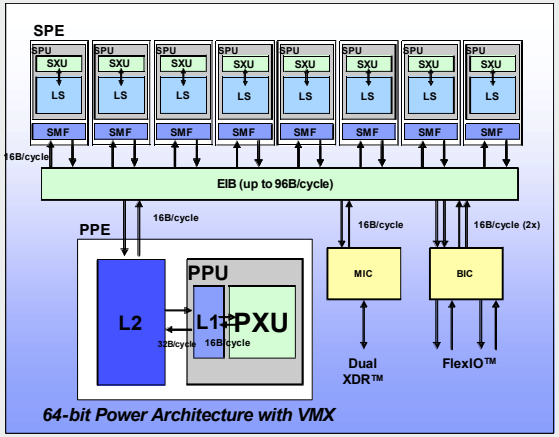


?

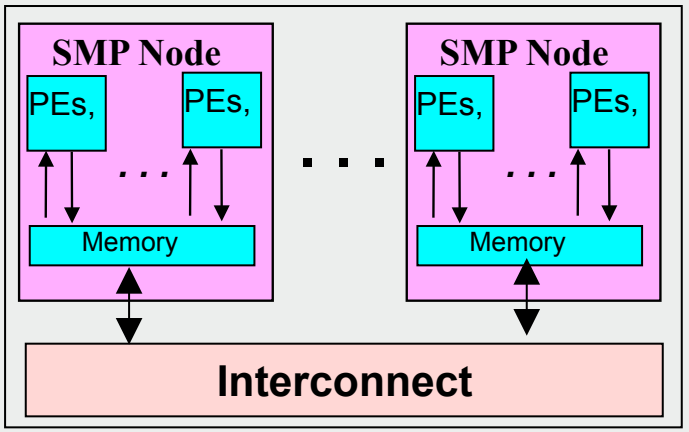
Homogeneous Multi-core



Heterogeneous Accelerators



High Performance Clusters



Advances in virtual machine intermediate languages are essential to address the implementation challenges of multicore programming