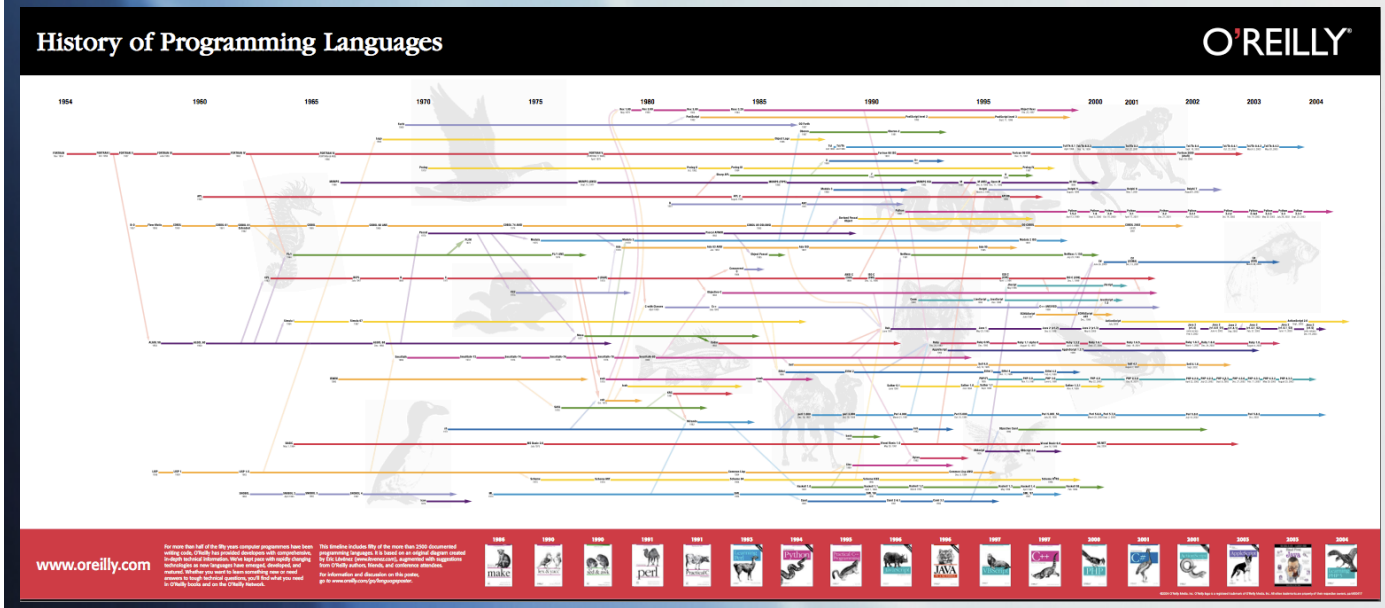
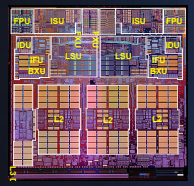
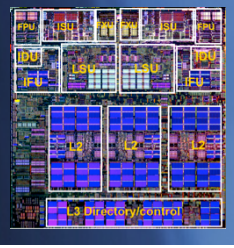
RICE George R. Brown School of Engineering
Computer Science

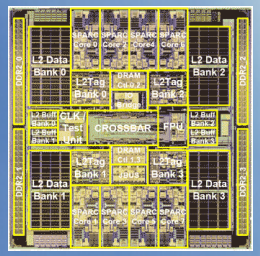
Habanero Multicore Software Research Project
http://habanero.rice.edu

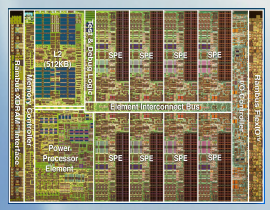


Vivek Sarkar
Rice University
vsarkar@rice.edu

June 18, 2010






Habanero Team

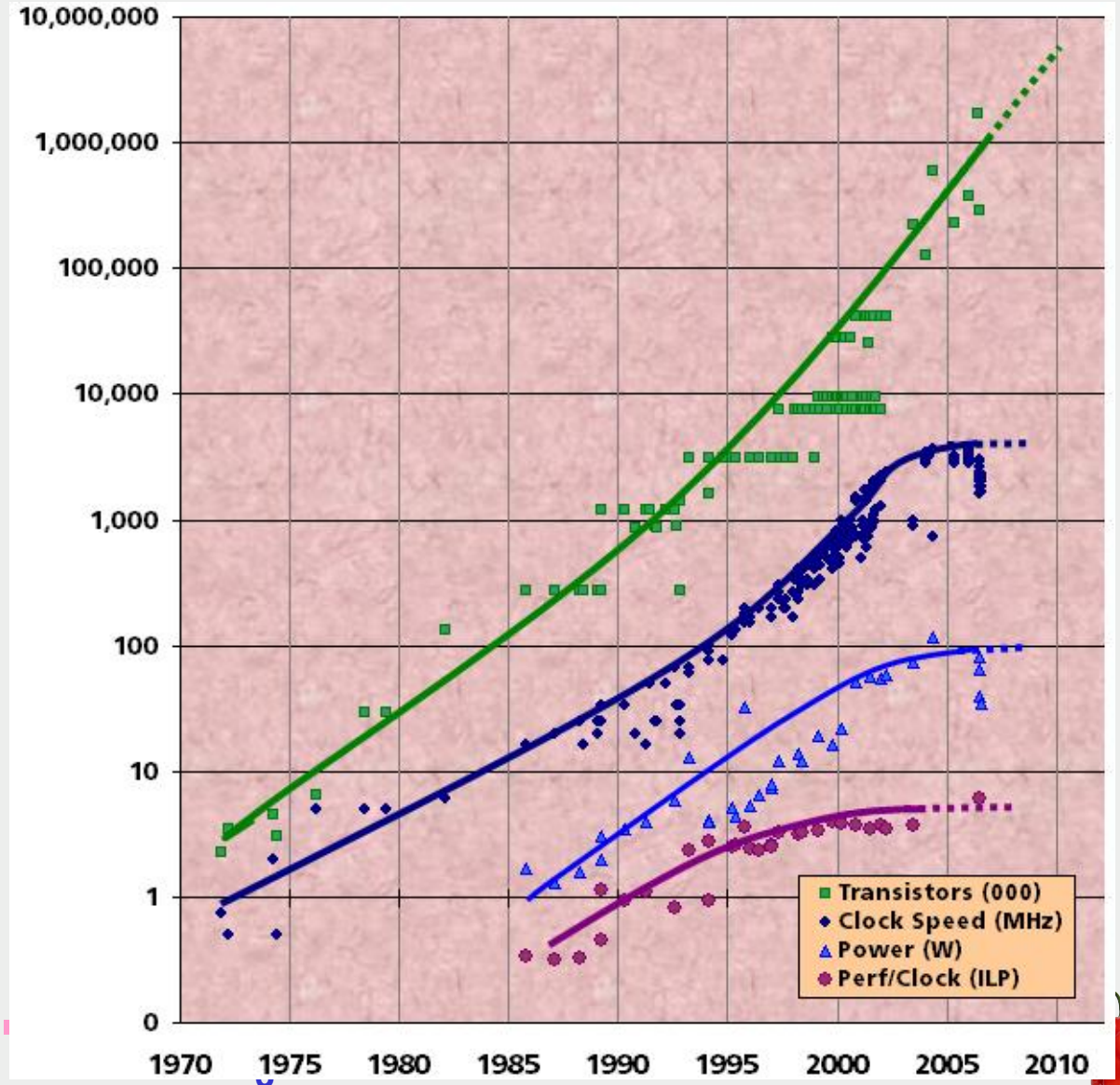
- Faculty
  - Vivek Sarkar
- Senior Research Scientist
  - Michael Burke
- Research Scientists
  - Zoran Budimlić, Philippe Charles
- Research Programmer
  - Vincent Cavé
- Postdocs
  - Rajkishore Barik, Jun Shirako, Yonghong Yan, Jisheng Zhao
- PhD Students
  - Sanjay Chatterjee, Yi Guo, Shams Imam,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

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
- Hardware cannot automatically extract parallelism for multicore

➔ ***Multicore parallelism must be exposed to and managed by software***

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



RICE

Extreme Scale Systems

- Characteristics of Extreme Scale systems anticipated in 2015 – 2020
  - *Massively multi-core (~ 100's of cores/chip)*
  - *Performance driven by parallelism, constrained by energy*
  - *Subject to frequent faults and failures*
- Three Classes of Extreme Scale Systems

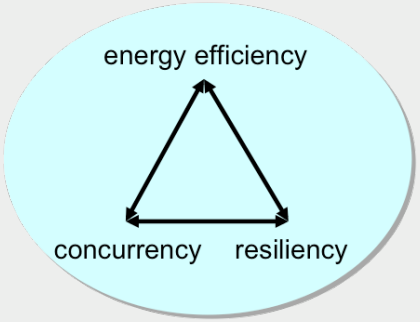
Terascale Embedded/Commodity
single chassis, 100's of Watts,
O(10^3) concurrency

Petascale Departmental
2 - 4 racks, 100's of KW,
O(10^6) concurrency

ExaScale Data Center
Max of ~500 racks, 10's of MW,
O(10^9) concurrency

Key Challenges

- ***Energy Efficiency***
- ***Concurrency***
- ***Resiliency***



RICE

Examples of Past Successful Execution Models

Execution Model	Device trend	Arch. trend	System s/w trend
Von Neumann	SSI devices	Scalar instrs.	Scalar compilers
Vector Parallelism	MSI devices	Vector instrs.	Vectorizing compilers
Shared-memory Parallelism	VLSI microprocessors	Cache coherence	Multithreaded OS and runtime
Bulk-synchronous Parallelism	VLSI microprocessors	Interconnects	Message-passing libraries (MPI)
??	Homogeneous & heterogeneous multicore processors	50B transistors/ chip w/ power constraints	Lightweight asynchronous tasks and data transfers

Source: ExaScale Computing Software Study --- Software Challenges in Extreme Scale Systems (Short paper: June 2009, Full report: September 2009)

Rice Habanero Multicore Software Project: Enabling Technologies for Extreme Scale

Parallel Applications

Portable execution model

1) Lightweight asynchronous tasks and data transfers
   - *async, finish, asyncMemcpy*

2) Locality control for task and data distribution
   - *hierarchical place tree*

3) Mutual exclusion
   - *ownership-based isolation*

4) Collective, point-to-point, stream synchronization
   - *phasers*

Habanero Programming Languages

Habanero Static Compiler & Parallel Intermediate Representation

Habanero Runtime & Dynamic Compiler

Two-level programming model

Declarative Coordination Language for Domain Experts,

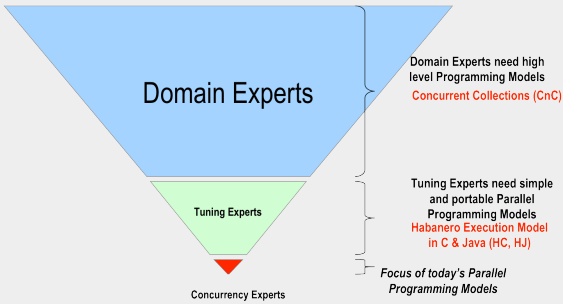
CnC (Intel Concurrent Collections)

+

Task-Parallel Languages for Tuning Experts,

Habanero-Java (from X10 v1.5)

and Habanero-C



Extreme Scale Platforms

Projects under way in the Habanero Group (http://habanero.rice.edu)

- NSF Expeditions Center for Domain-Specific Computing (CDSC)
  - Collaboration with UCLA, UCSB, OSU, http://www.cdsc.ucla.edu
- Habanero Concurrent Collections (CnC)
  - http://habanero.rice.edu/cnc (includes link to download)
  - Collaboration with Intel, UCLA (derived from Intel CnC)
- Habanero Java (HJ)
  - http://habanero.rice.edu/hj (includes link to download)
  - Collaboration with IBM, MIT (HJ derived from IBM X10 v1.5)
- Habanero C/C++ (HC)
  - Collaboration with U. Delaware
- DARPA-funded Platform Aware Compilation Environment (PACE)
  - Collaboration with OSU, Stanford, ETI, TI
- SRC FCRP Multiscale Systems Center (MuSyC)
  - Collaboration with Berkeley et al, http://www.musyc.org

Scope of this talk

NSF Expeditions Center for Domain-Specific Computing (CDSC)

A diversified & highly accomplished team: 8 in CS&E; 1 in EE; 2 in medical school; 1 in applied math


Aberle


Baraniuk


Bui


Chang


Cheng


Cong (Director)

	UCLA	Rice	UCSB	Ohio State
Domain-specific modeling	Bui, Reinman, Potkonjak	**Sarkar,** Baraniuk		Sadayappan
CHP creation	Chang, Cong, **Reinman**		Cheng	
CHP mapping	Cong, Palsberg, Potkonjak	**Sarkar**	Cheng	Sadayappan
Application modeling	Aberle, **Bui,** Vese	Baraniuk		
Experimental systems	All (led by Cong & Bui)	All	All	All


Palsberg


Potkonjak


Reinman

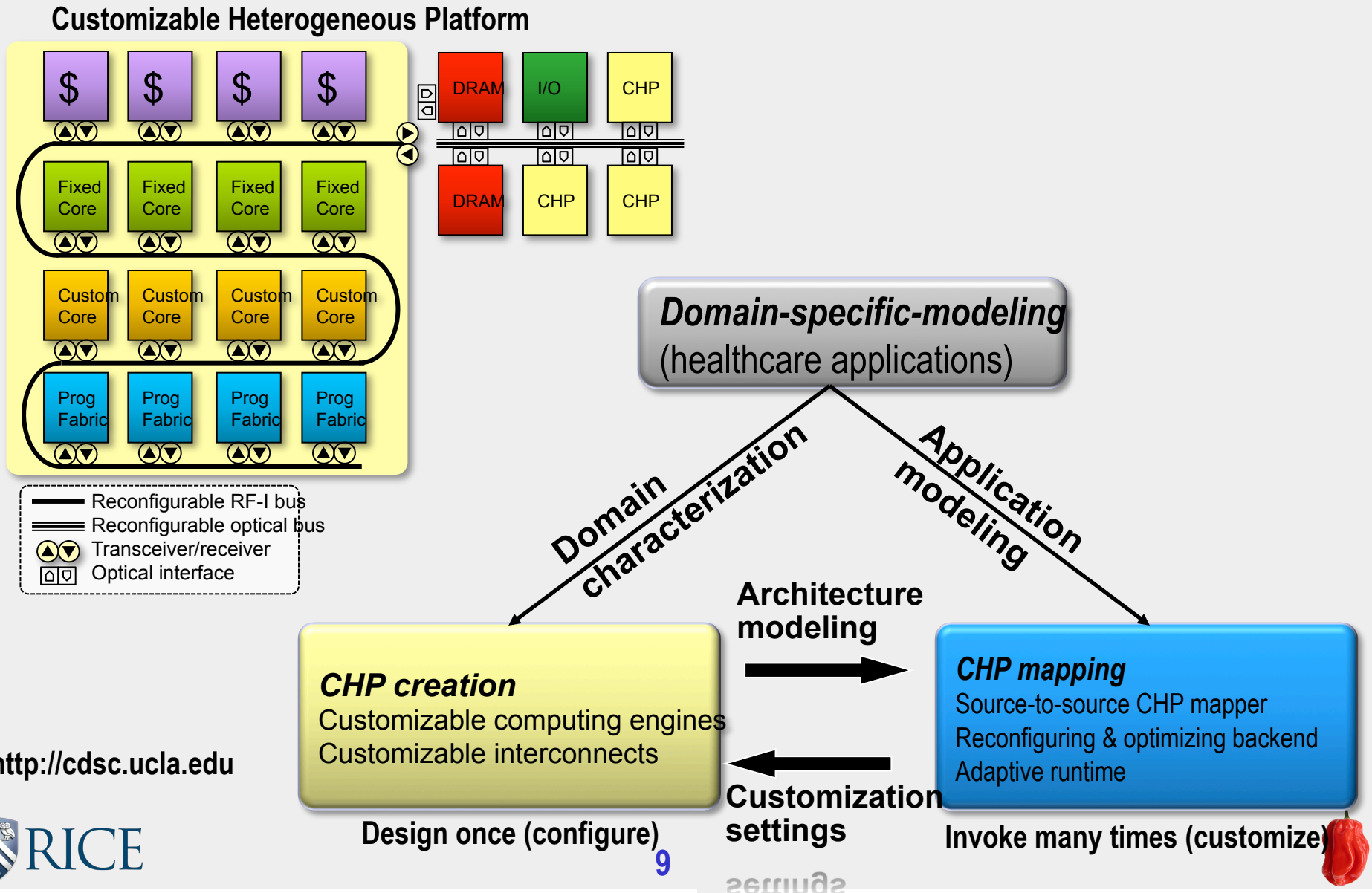

Sadayappan


Sarkar (Associate Dir)


Vese

RICE

Center for Domain-Specific Computing: Overview of Proposed Research



http://cdsc.ucla.edu

RICE

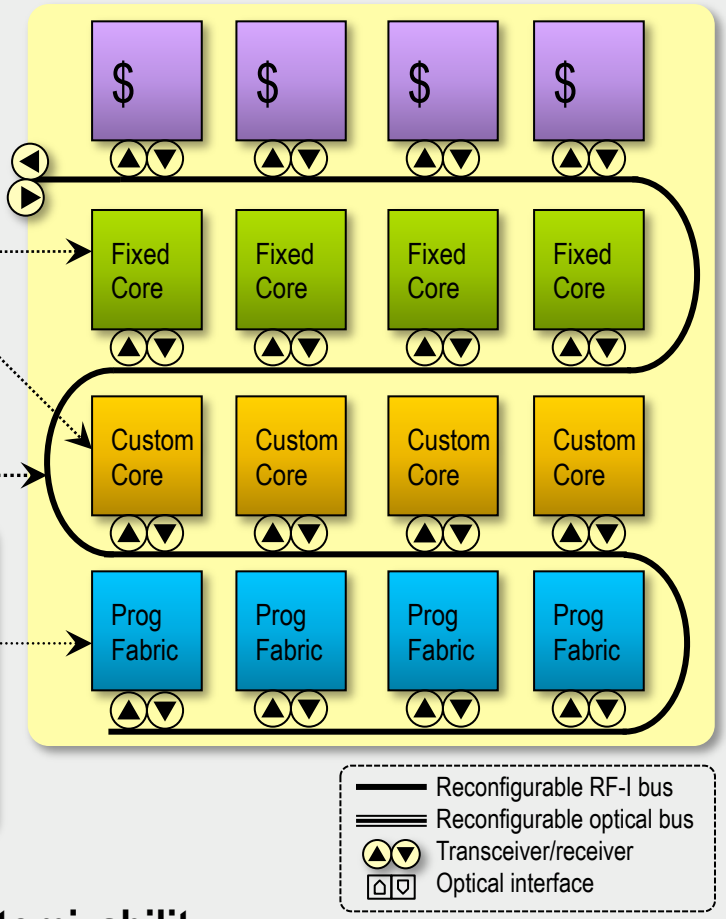
CHP Creation – Design Space Exploration

Core parameters
- Frequency & voltage
- Datapath bit width
- Instruction window size
- Issue width
- Cache size & configuration
- Register file organization
- # of thread contexts
- …

NoC parameters
- Interconnect topology
- # of virtual channels
- Routing policy
- Link bandwidth
- Router pipeline depth
- Number of RF-I enabled routers
- RF-I channel and bandwidth allocation
- …

Custom instructions & accelerators
- Amount of programmable fabric
- Shared vs. private accelerators
- Custom instruction selection
- Choice of accelerators
- …

Customizable Heterogeneous Platform (CHP)



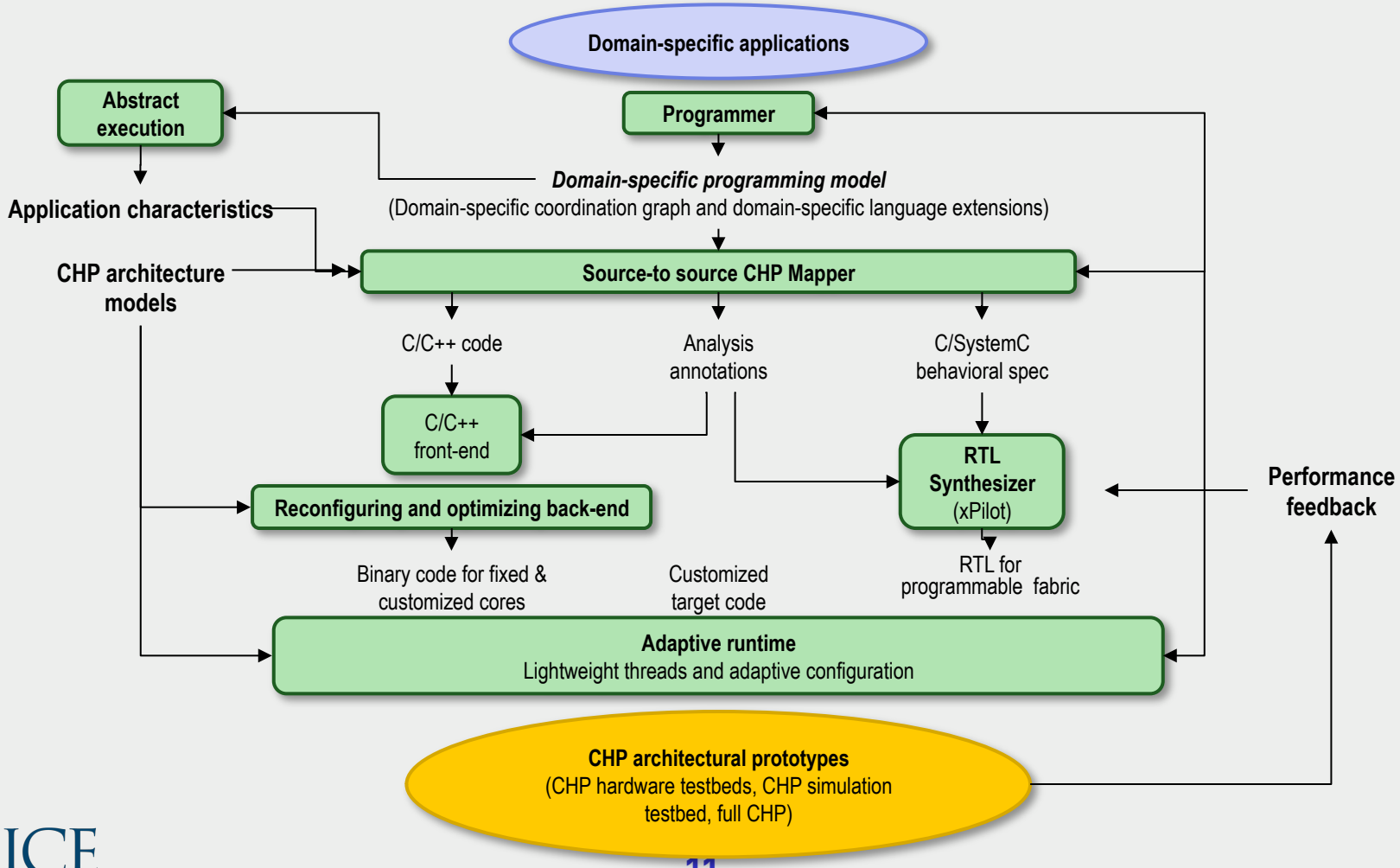
Key questions: **Optimal trade-off between efficiency & customizability**
Which options to fix at CHP creation? Which to be set by CHP mapper?

RICE

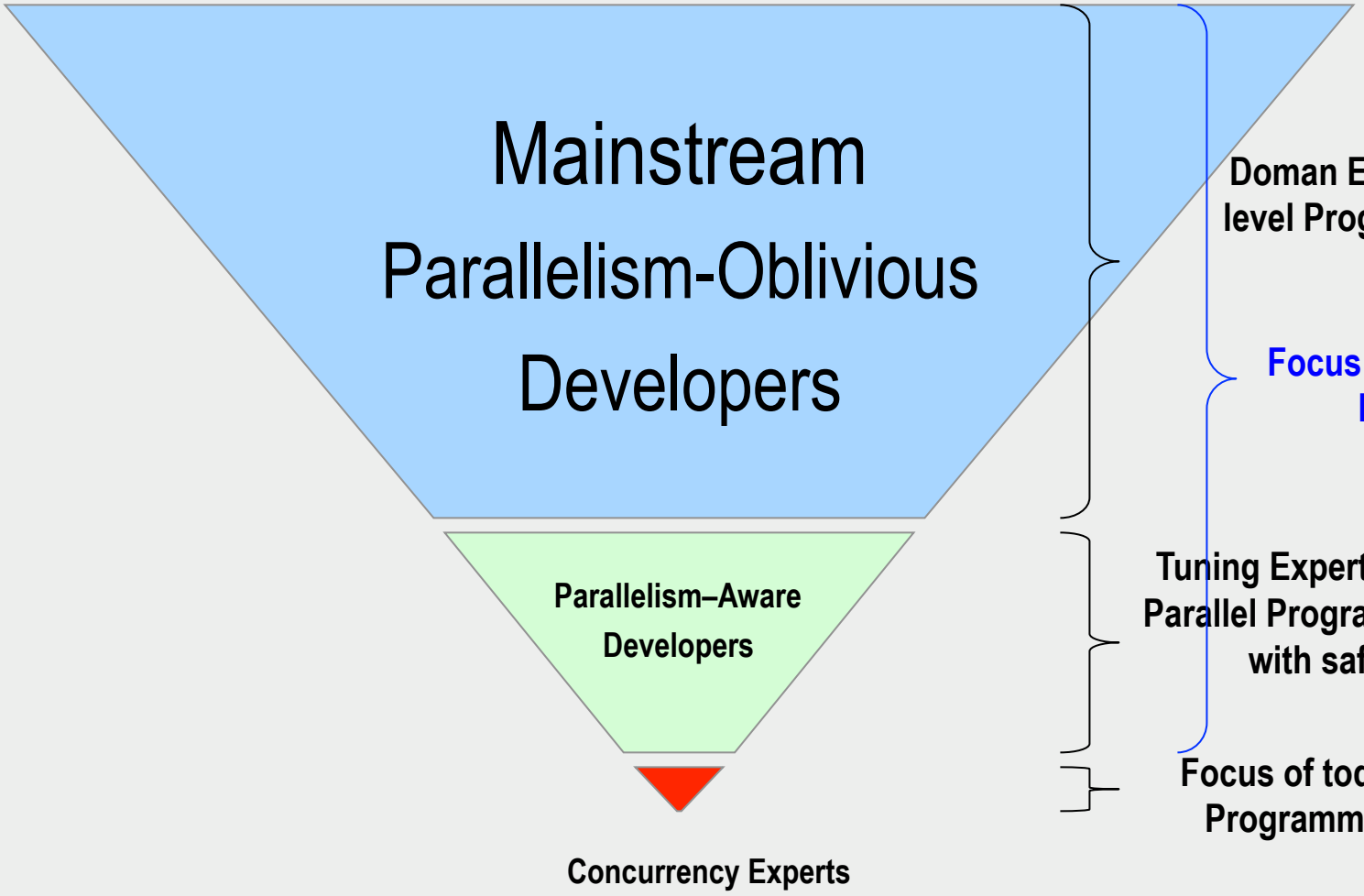
CHP Mapping – Compilation and Runtime Software Systems for Customization

Goals: Efficient mapping of domain-specific specification to customizable hardware

– Adapt the CHP to a given application for drastic performance/power efficiency improvement

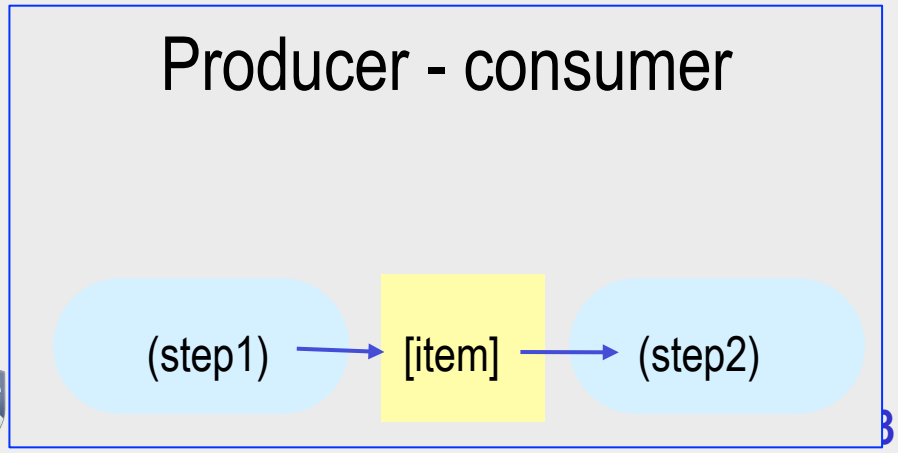


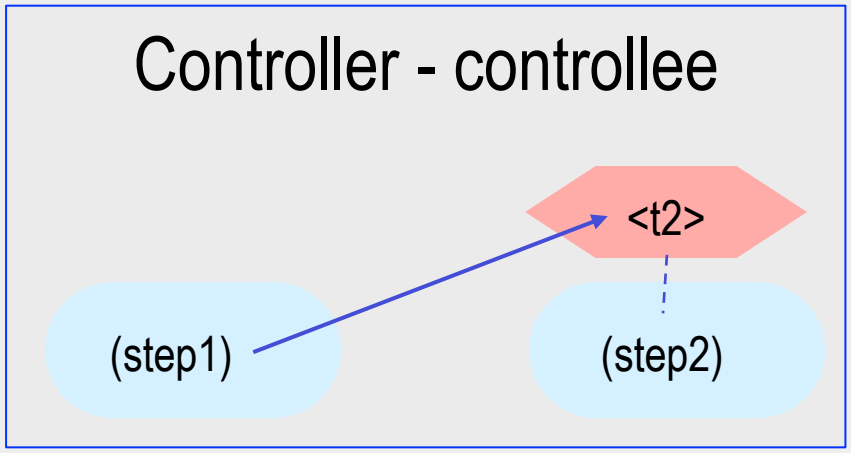
Parallel Software Challenge & Inverted Pyramid of Parallel Programming Skills



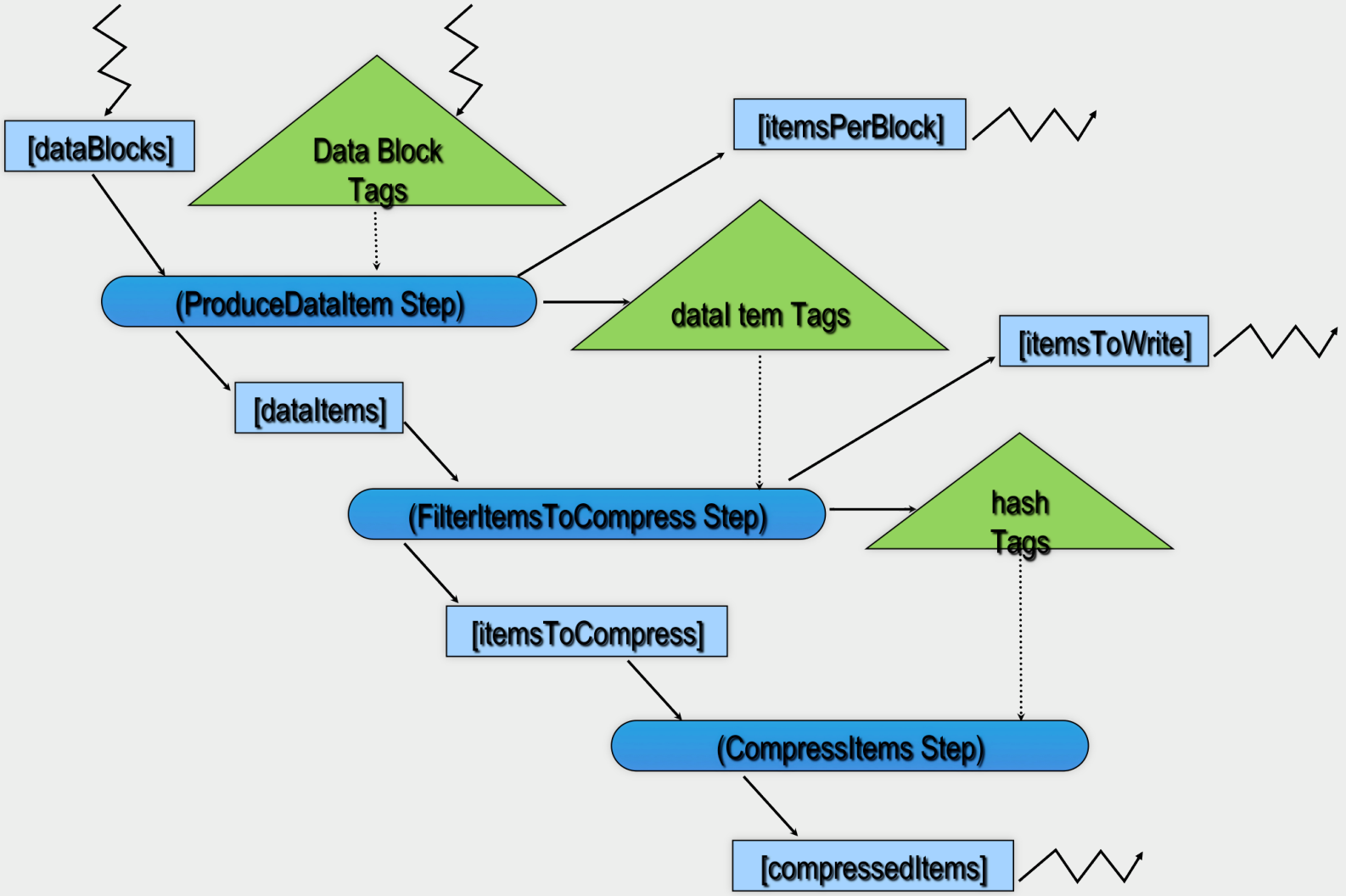
The Concurrent Collections (CnC) model for Domain Experts --- Coarse-Grained Dynamic Dataflow Execution

- Introduce a coordination language that specifies only the semantic ordering constraints in a parallel program, and no more
  - Producer-consumer ordering constraints (data dependence)
  - Controller-controllee ordering (control dependence)
- Result is:
  - Deterministic
  - Race-free
  - Fault-tolerant

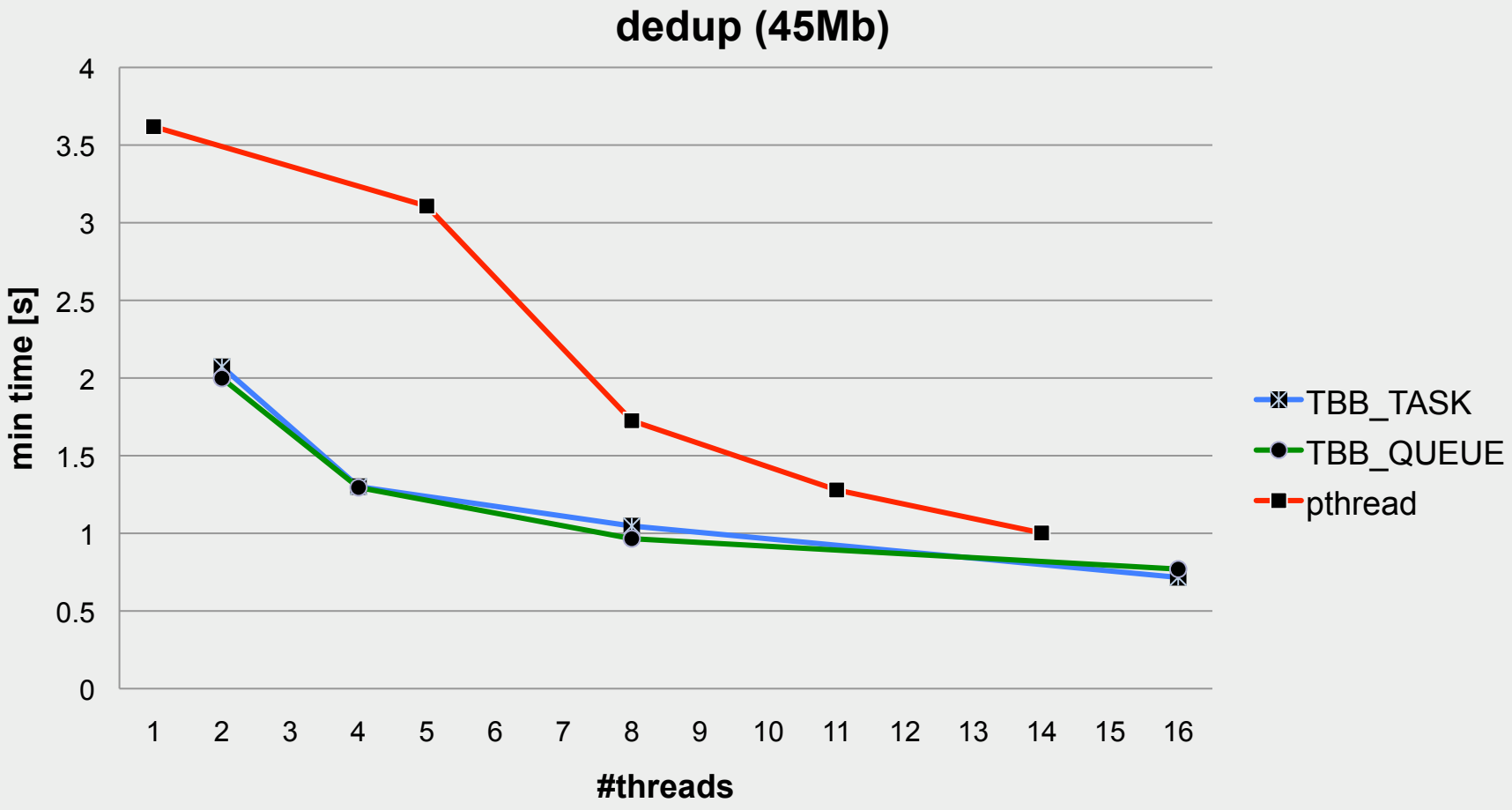




CnC representation of PARSEC benchmark, dedup



Execution times for pthreads and CnC-TBB implementations of the PARSEC dedup benchmark on a 16-core Intel Caneland SMP (TBB_TASK, TBB_QUEUE are scheduling options)



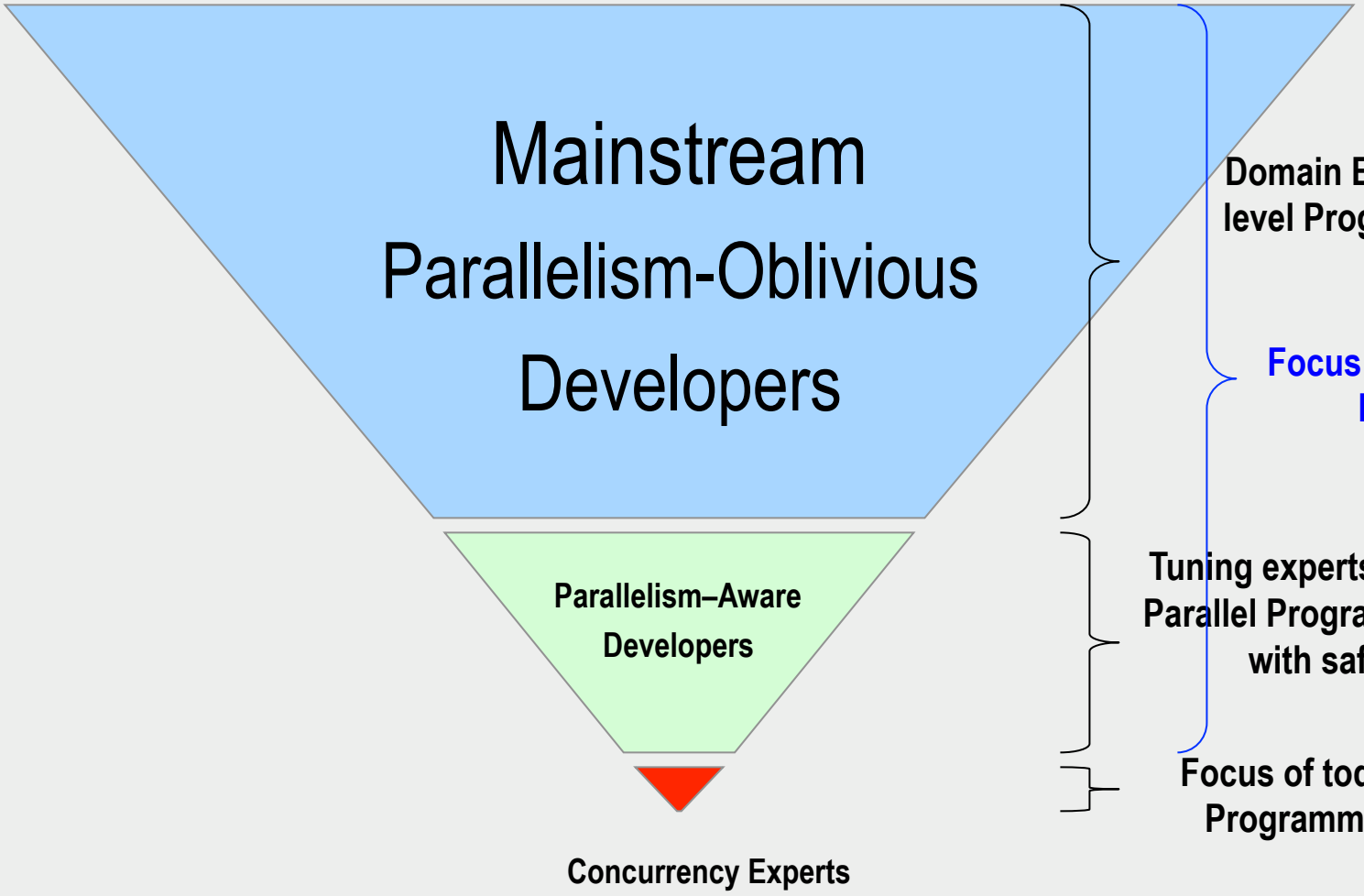
"The Concurrent Collections Programming Model", Z.Budimlic et al, submitted for publication

RICE

Comparing CnC with Microsoft Dryad Execution Engine

- Overview of Dryad
  - General-purpose distributed execution engine for coarse-grain data-parallel applications.
  - Combines sequential “vertices” with directed communication “channels” to form an acyclic dataflow graph.
  - Support for distributed systems --- channels are implemented TCP, files, or shared memory pipes as appropriate.
  - Dryad graph is specified by an embedded language (in C++) using a combination of operator overloading and API calls.
- Differences with CnC
  - CnC supports cyclic graphs with first-class tagged controller-controllee relationships and tagged item collections.
  - Current CnC implementations are focused on multicore rather than distributed systems (no central manager, no fault tolerance)

Parallel Software Challenge & Inverted Pyramid of Parallel Programming Skills



The Habanero-Java & Habanero-C models for Tuning Experts

- Tuning experts need to map & tune domain experts' CnC model (graph + steps) onto parallel systems
  - Exploit parallelism across and within steps
  - Optimize Locality, Data Movement, Load balancing, Scheduling, ..

- Habanero Approach: support a portable abstract execution model that supports high performance with high productivity
  1. *Lightweight dynamic task creation & termination*
  2. *Locality control --- task and data distributions*
  3. *Mutual exclusion and isolation*
  4. *Collective and point-to-point synchronization*

Deadlock freedom guarantee for these four constructs

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 v1.0	None	Device, grid, block, threads	None	Barriers	SIMD
Intel Concurrent Collections	Tagged prescription of steps	None	None	Tagged put & get operations on Item Collections	Map-Reduce (in progress)
Habanero-Java (builds on Java Concurrency)	Async, finish	Places	Isolated blocks, Java atomic classes	Phasers, delayed async	SIMD/MIMD array operations, Java concurrent collections

Habanero Model

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*

Async and Finish (from X10 v1.5)

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

async S

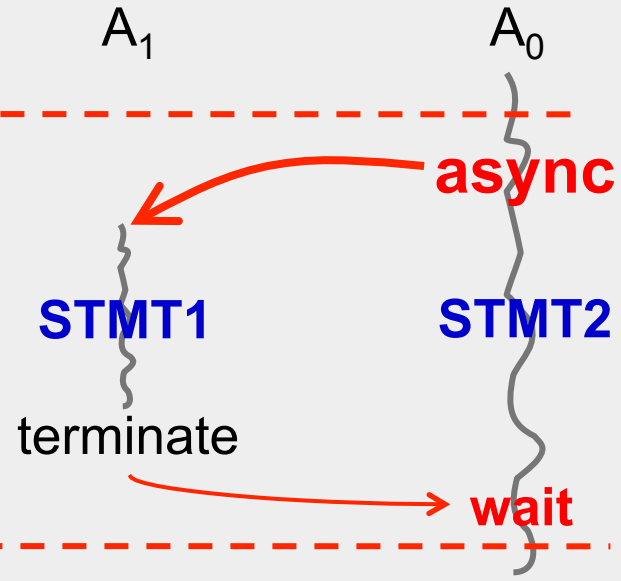
- Creates a new child activity that executes statement S
- Returns immediately

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

finish S

- Execute S, but wait until all (transitively) spawned asyncs have terminated.
- Implicit finish between start and end of main program

```
//A0 (Parent)
finish {    //Begin finish
  async {
    STMT1;  //A1 (Child)
  }
  STMT2;    //A0
}             //End finish
```



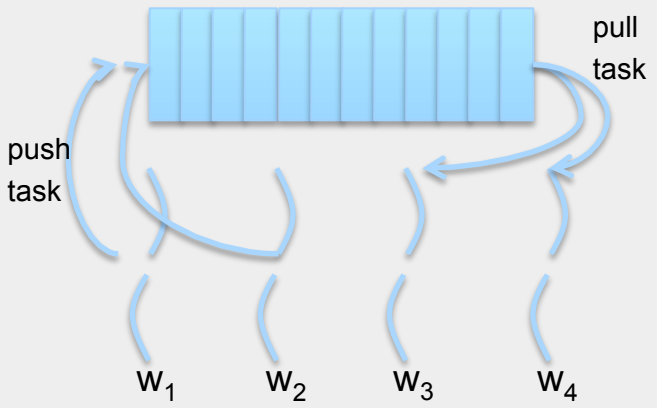
Scheduling Algorithms for Async-Finish Task Parallelism

Work-Sharing (Java ThreadPoolExecutor, OpenMP, …)

- Busy worker pushes task at one end of global deque
- Access to global deque needs to be synchronized: scalability bottleneck

Work-Stealing (Cilk, TBB, Java ForkJoin, …)

- One deque per worker (better scalability)
- Idle worker steals tasks from busy workers
- Two scheduling policies of interest
  - **Work-first policy:** worker executes child task eagerly and leaves continuation to be stolen
  - **Help-first policy:** worker pushes child task to be stolen (asks for help) and executes continuation

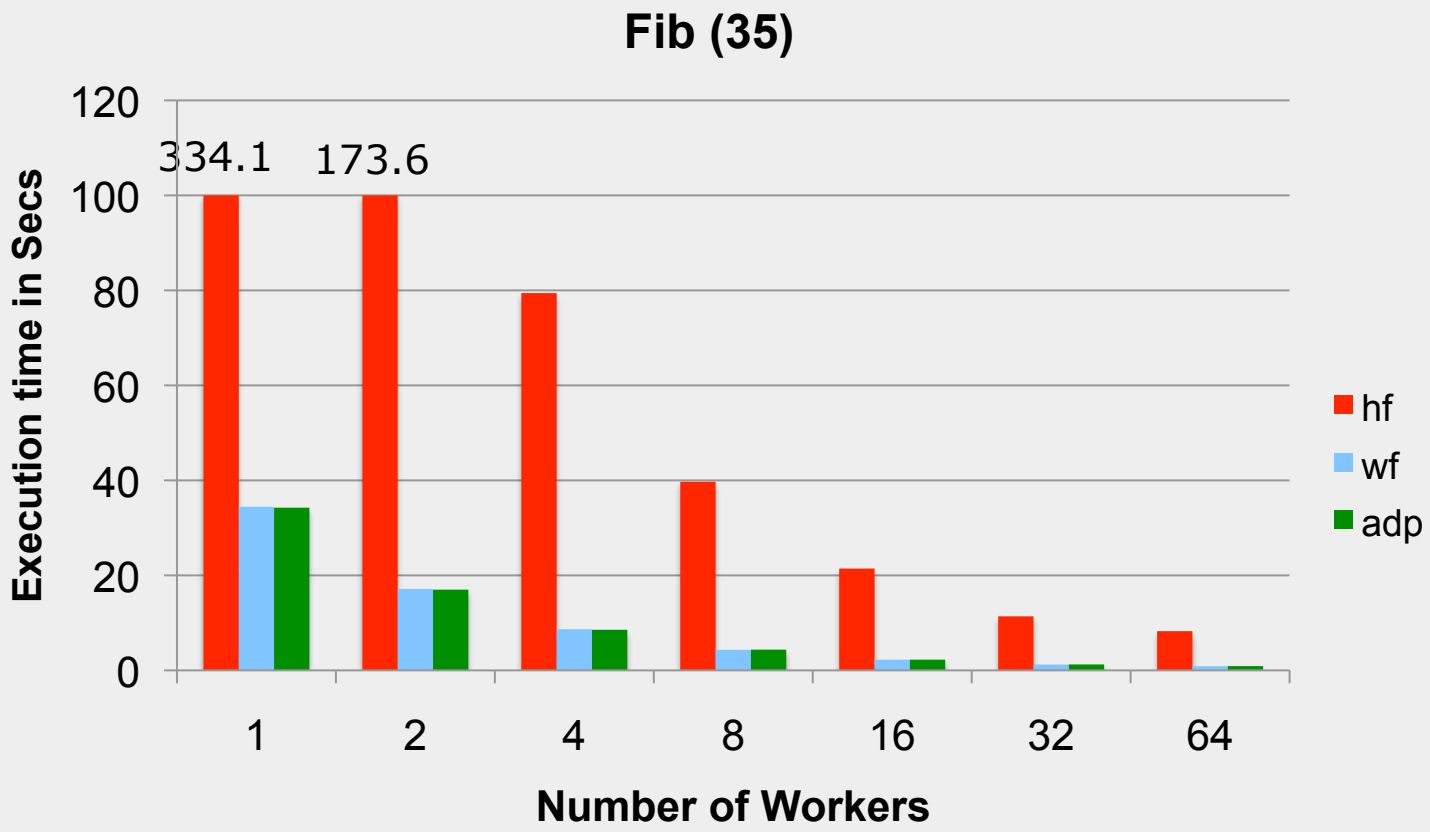


work-sharing

"Work-First and Help-First Scheduling Policies for Terminally Strict Parallel Programs", Y.Guo, R.Barik, R.Raman, V.Sarkar, IPDPS 2009.

"SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Systems", Y.Guo, J.Zhao, V.Cave, V.Sarkar, IPDPS 2010.

Experimental Results for Fib(35) microbenchmark on a Niagara 2 system with 1 – 64 workers (hf = help-first, wf = work-first, adp = adaptive)



The adaptive policy matches the performance of the better policy (work-first)

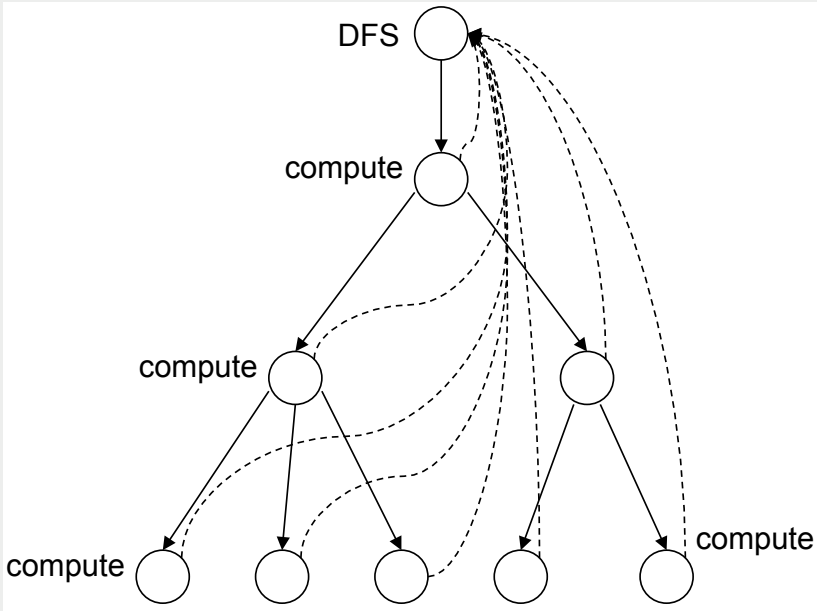
Experimental Results for Java ForkJoin(1024) microbenchmark on a Niagara 2 system with 1 – 64 workers (hf = help-first, wf = work-first, adp = adaptive)

The adaptive policy matches the performance of the better policy (help-first).

This experiment also underscores the importance of choosing the right number of workers for a given application and platform.

Example of Escaping Asyncs: Parallel Depth-First Search Spanning Tree

```
class V  {
  V [] neighbors;
  V parent;
  . . .
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent = n;
    return parent == n;
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
    }
  } // compute
  void DFS() {
     parent = this; finish compute();
  } // DFS
} // class V
. . . root.DFS(); . . .
```

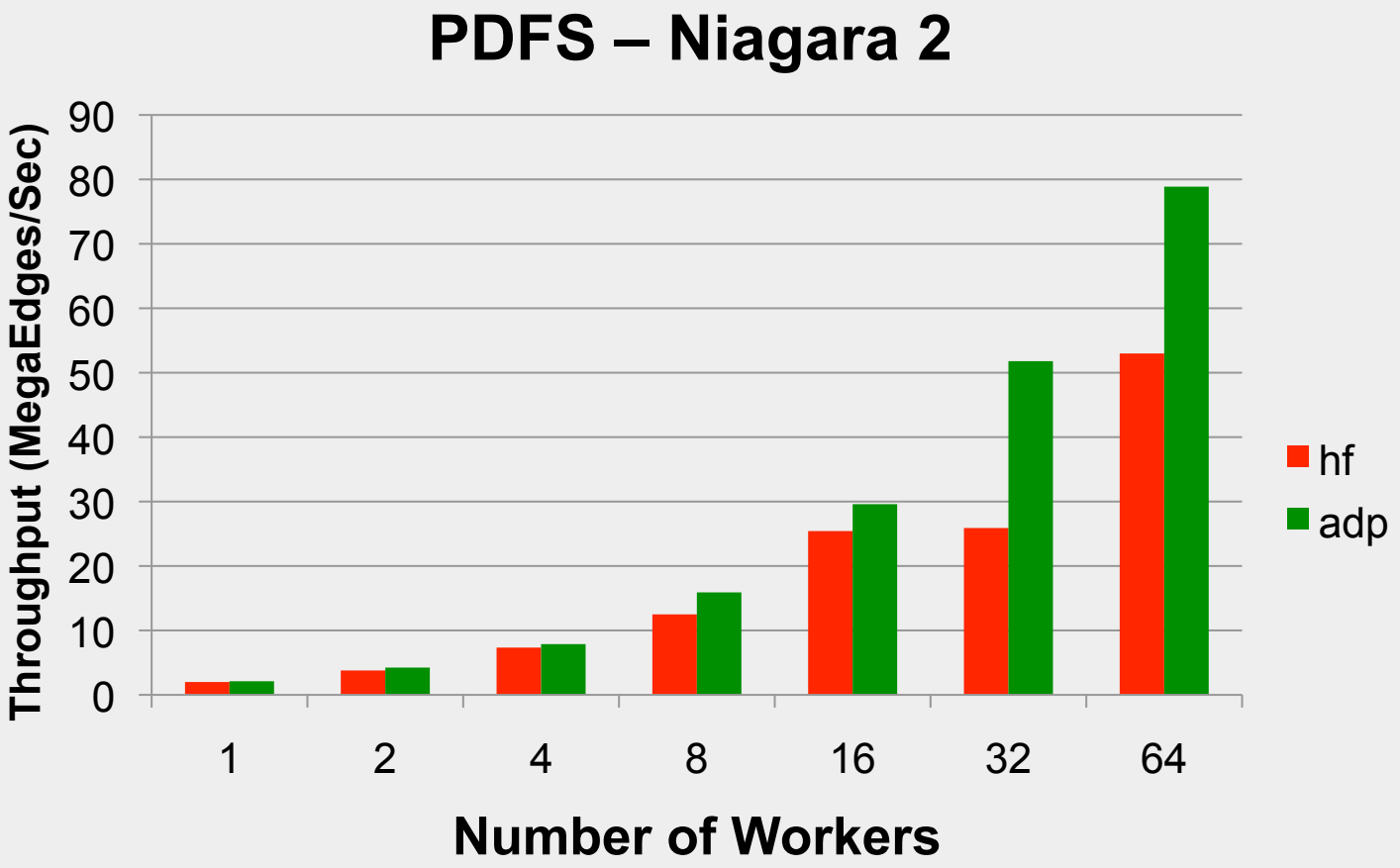


Async edge

Finish edge

Question: What is the best policy for this async?

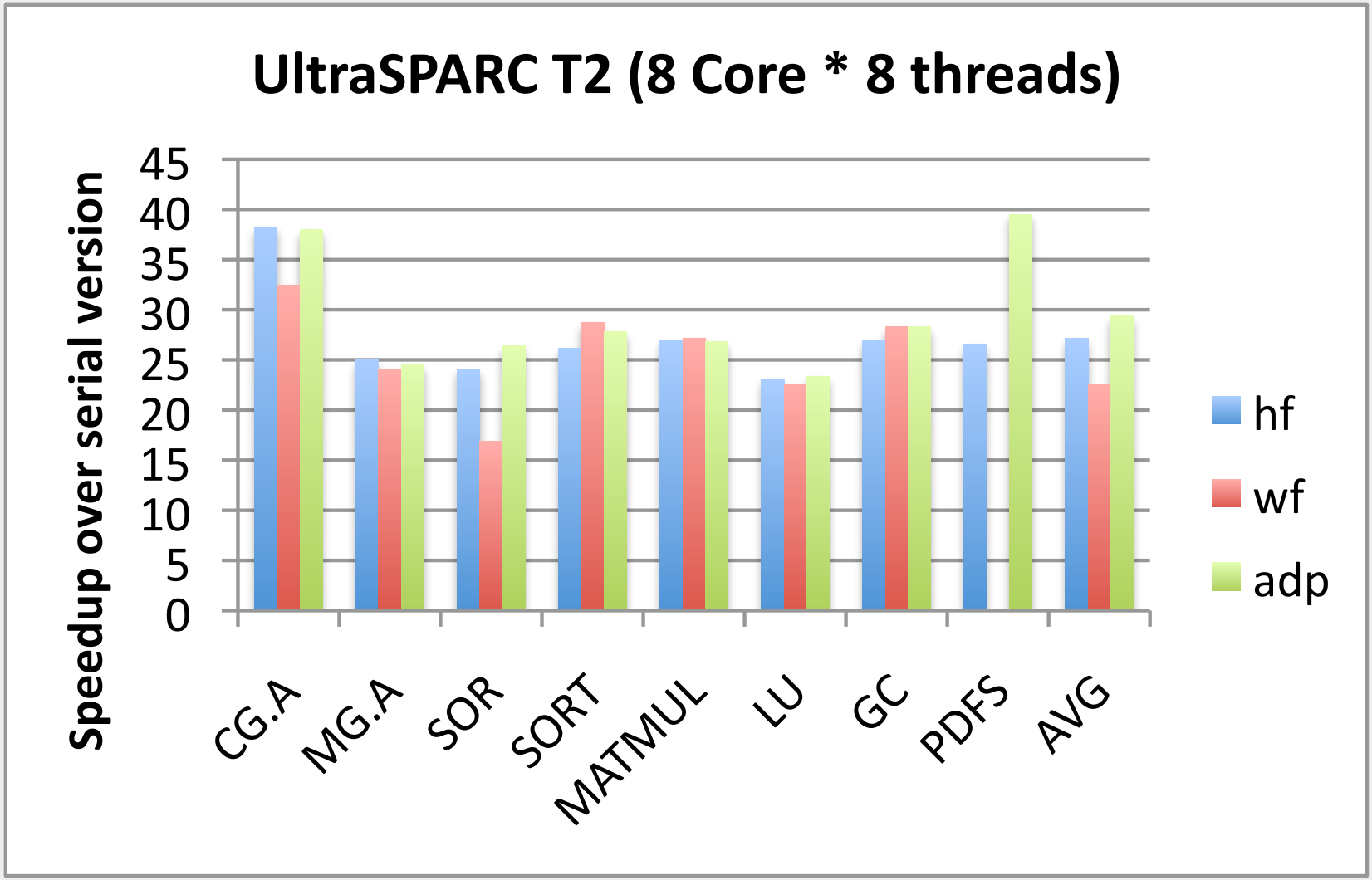
PDFS on a Torus Graph with 4M vertices



Work-first overflows the stack

Adaptive performs better than help-first

Experimental Results for Regular & Irregular Benchmarks (hf = help-first, wf = work-first, adp = adaptive)

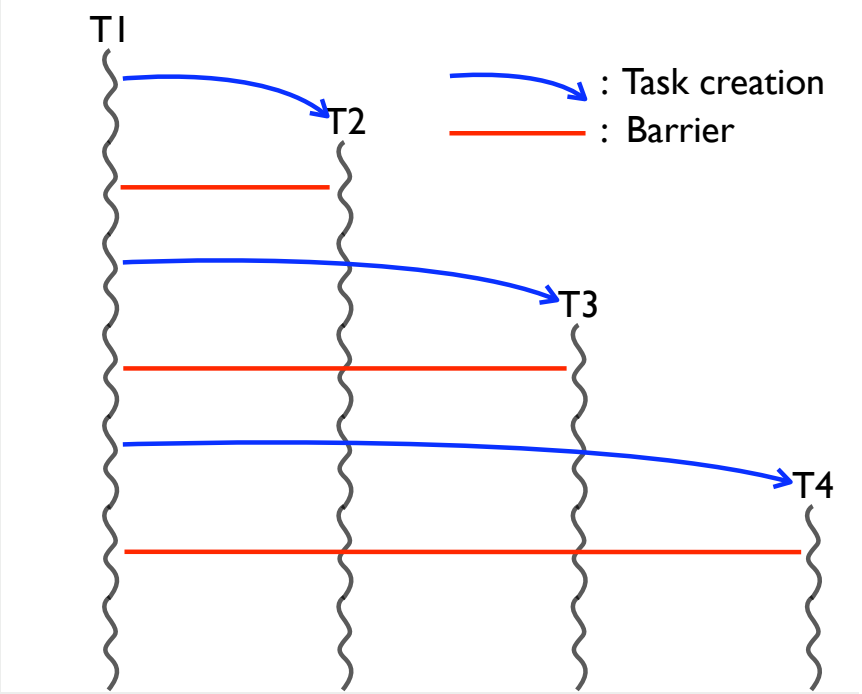


Habanero Execution Model: Portable Parallelism in Four Dimensions

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*

Phasers

- Designed to handle multiple communication patterns
  - Collective Barriers
  - Point-to-point synchronizations
- Supporting dynamic parallelism
  - # tasks can be varied dynamically
- Deadlock freedom
  - Absence of explicit wait operations
- Accumulation
  - Reductions (sum, prod, min, …) combined with synchronizations
- Streaming parallelism
  - As extensions of accumulation to support buffered streams
- Technology transfer
  - Subset of phaser functionality incorporated in Java 7



java.util.concurrent.Phaser library in Java 7

- Implementation of subset of phaser functionality by Doug Lea in Java Concurrency library

Date: Mon, 07 Jul 2008 13:19:01 -0400
From: Doug Lea
Subject: [concurrency-interest] Phasers (were: TaskBarriers)
To: concurrency-interest@cs.oswego.edu

The flexible barrier functionality that was previously restricted to ForkJoinTasks (in class forkjoin.TaskBarrier) is being redone as class Phaser (targeted for j.u.c, not j.u.c.forkjoin), that can be applied in all kinds of tasks. For a snapshot of API, see
http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/Phaser.html

Comments and suggestions are very welcome as always. The API is likely to change a bit as we scope out further uses, and also, hopefully, stumble upon some better method names.

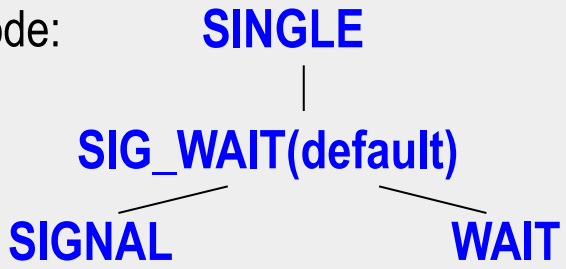
Among its capabilities is allowing the number of parties in a barrier to vary dynamically, which CyclicBarrier doesn't and can't support, but people regularly ask for.

The nice new class name is due to Vivek Sarkar. For a preview of some likely follow-ons (mainly, new kinds of FJ tasks that can register in various modes for Phasers, partially in support of analogous X10 functionality), see the paper by Vivek and others:
http://www.cs.rice.edu/~vsarkar/PDF/SPSS08-phasers.pdf

-Doug

Phaser Operations in Habanero Java (habanero.rice.edu/hj)

- **Phaser allocation**
  - **Phaser ph = new Phaser(mode);**
    - Phaser **ph** is allocated with **registration mode**
    - Mode:



Registration mode defines capability

There is a lattice ordering of capabilities

- **Activity registration**
  - **async phased (ph_1<$mode_1$>, ph_2<$mode_2$>, …) <stmt>**
    - Spawned activity is registered with **ph_1** in **$mode_1$**, **ph_2** in **$mode_2$**, …
    - child activity's capabilities must be subset of parent's
- **Synchronization**
  - **Next;**
    - Advance each phaser that activity is registered on to its next phase
    - Semantics depends on registration mode

next / signal / wait operations

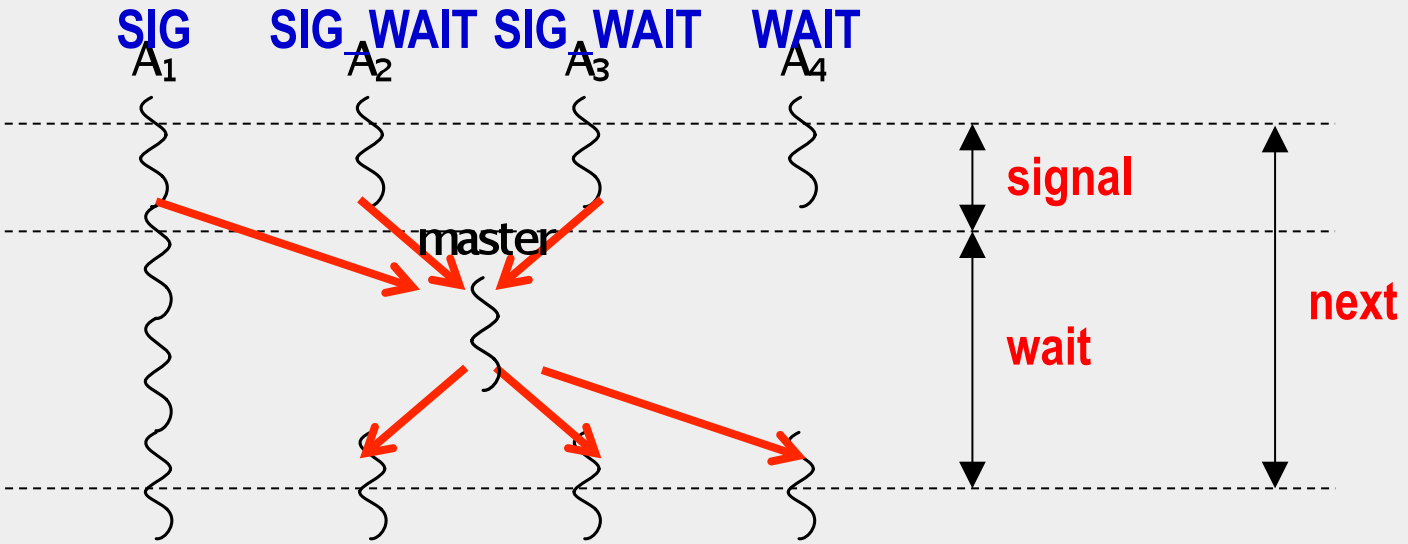
next = { **Notify "I reached next" = signal (or ph.signal())**
Wait for others to notify = wait

Semantics of next depends on registration mode

SIG_WAIT: **next = signal + wait**

SIG: **next = signal** (Don't wait for any activity)

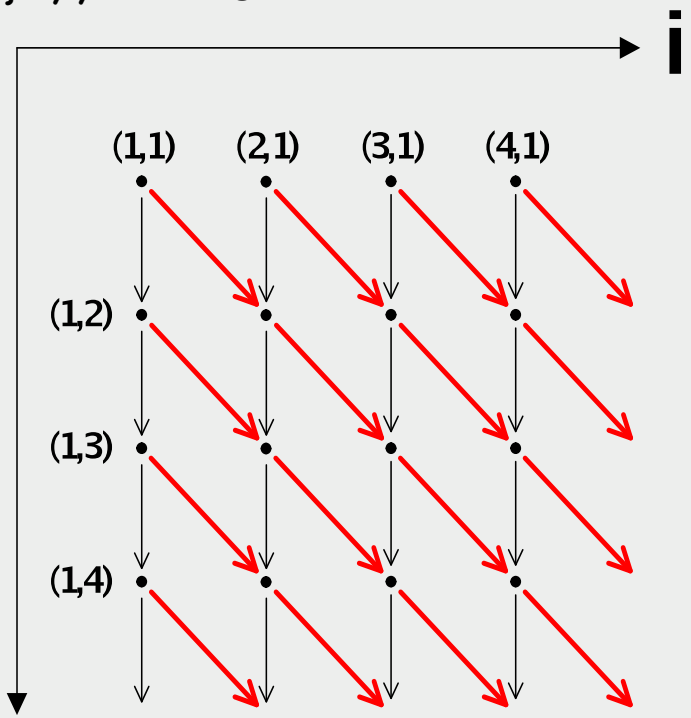
WAIT: **next = wait** (Don't disturb any activity)

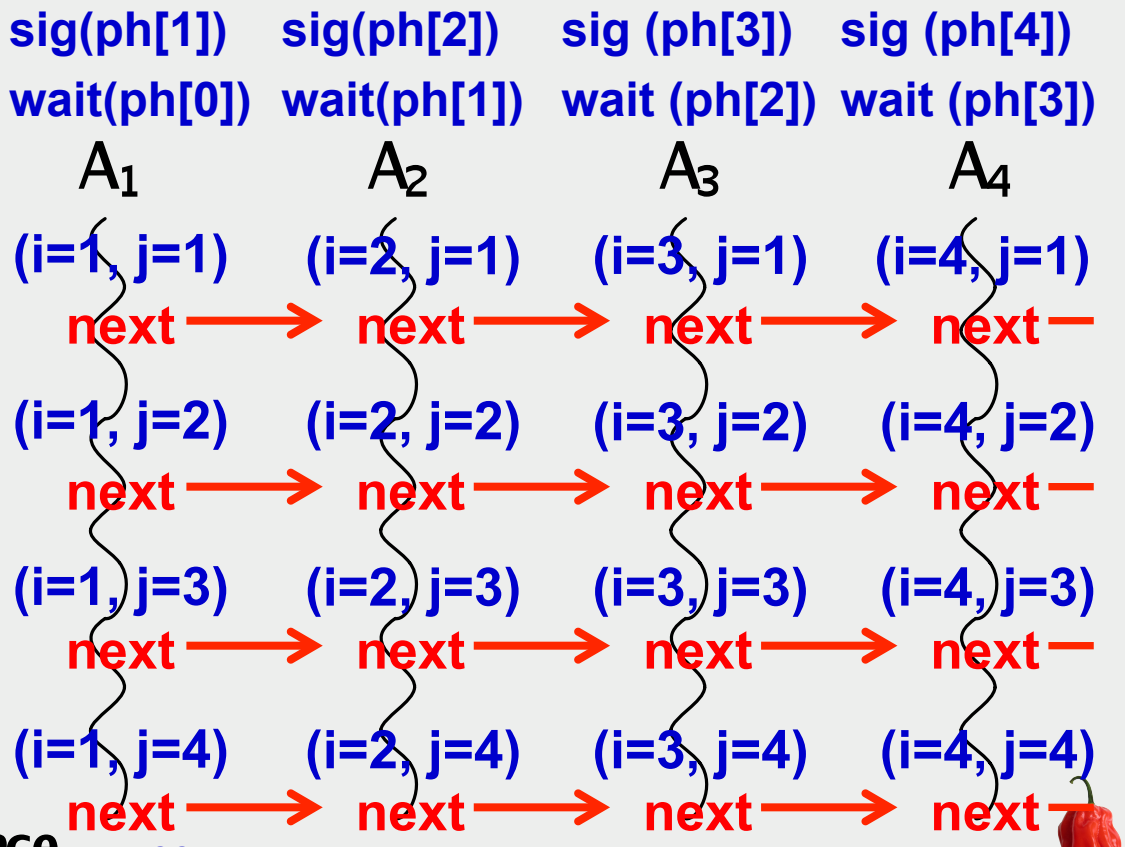


A master activity receives all signals and broadcasts a barrier completion

Example of Pipeline Synchronization with Phasers

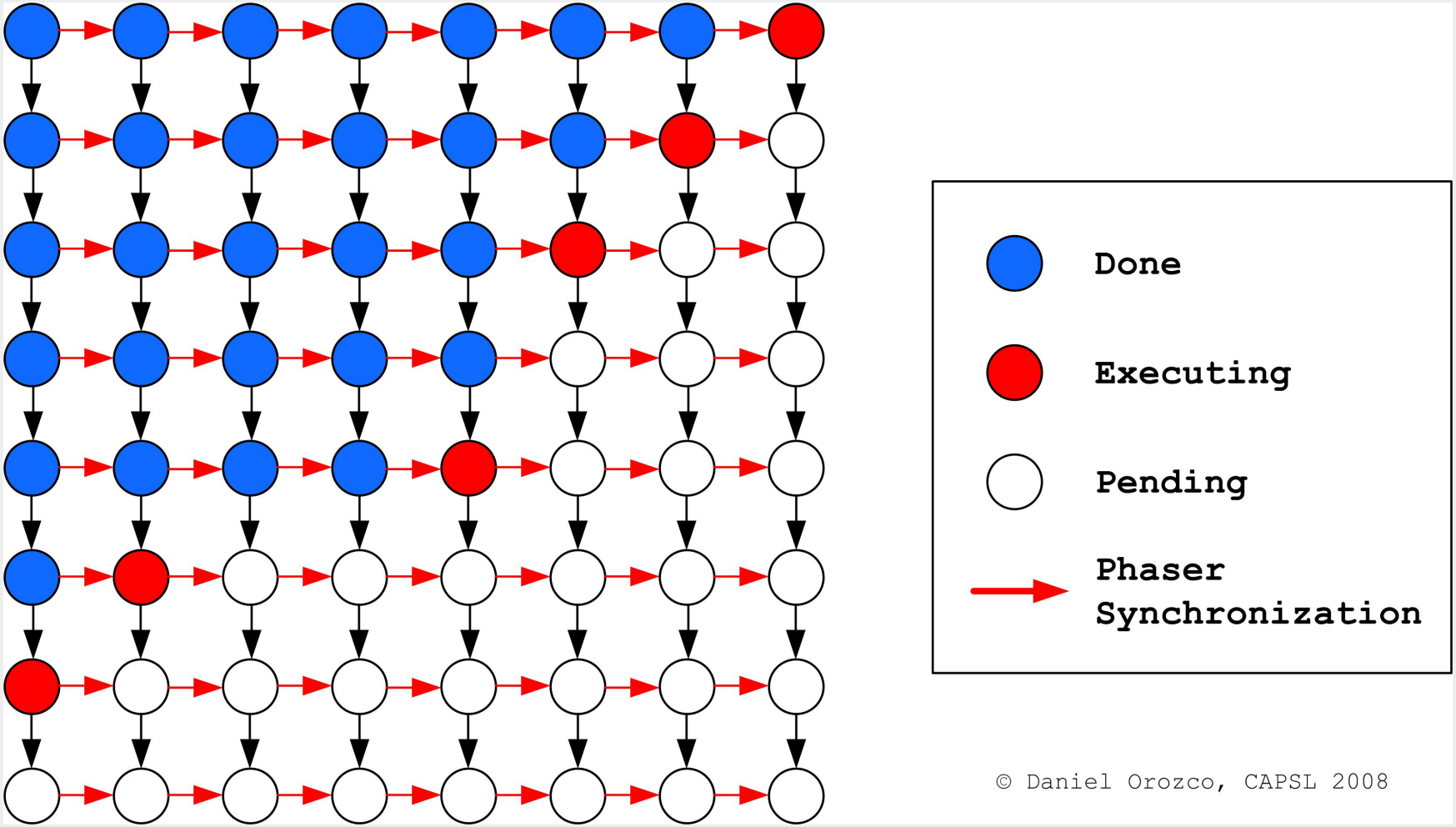
```
finish {
  phaser [] ph = new phaser[m+1];
  for (int i = 1; i < m; i++)
    async phased (ph[i]<SIG>, ph[i-1]<WAIT>){
      for (int j = 1; j < n; j++) {
        a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
        next;
      } // for
} // finish
```





RICE: Loop carried dependence

Example of Point-to-Point Synchronization with Phasers (contd)



Asynchronous Reductions with Phaser Accumulators (Example)

```
phaser ph = new phaser(signalWait);
accumulator a = new accumulator(ph, accumulator.SUM, int.class);
accumulator b = new accumulator(ph, accumulator.MIN, double.class);
```

Allocation: Specify operator and type of accumulator

```
foreach (point [i] : [0:n-1]) phased (ph<signalWait>) {
   int iv = 2*i + j;
   double dv = -1.5*i + j;
   a.send(iv); b.send(dv);
   // Do other work before next

   next;

   int sum = a.result().intValue();
   double min = b.result().doubleValue();
   …
}
```

send: Send a value to accumulator

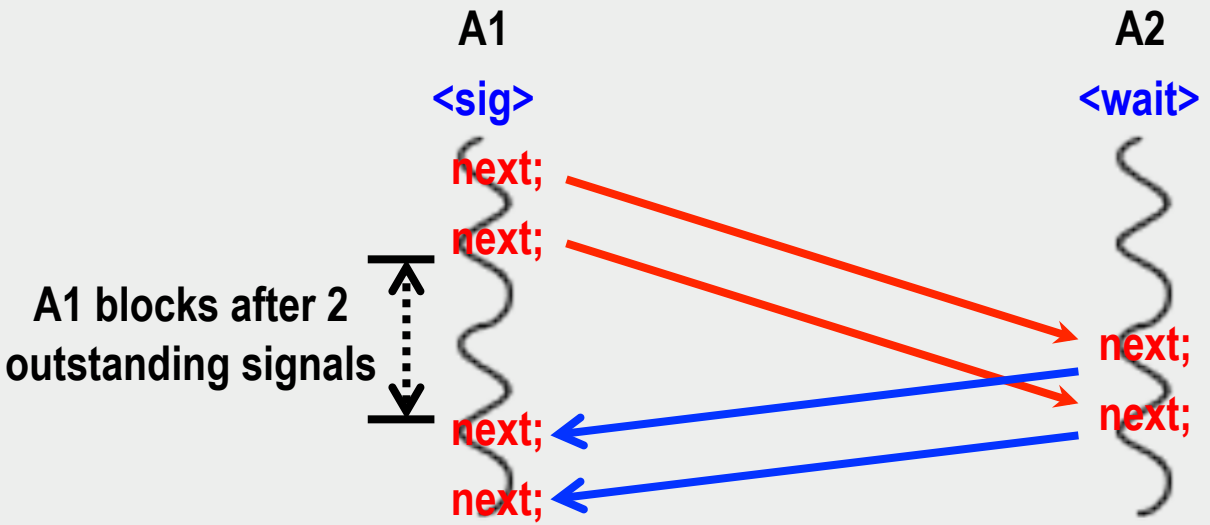
next: Barrier operation; advance the phase

result: Get the result from previous phase (no race condition)

Streaming Phasers

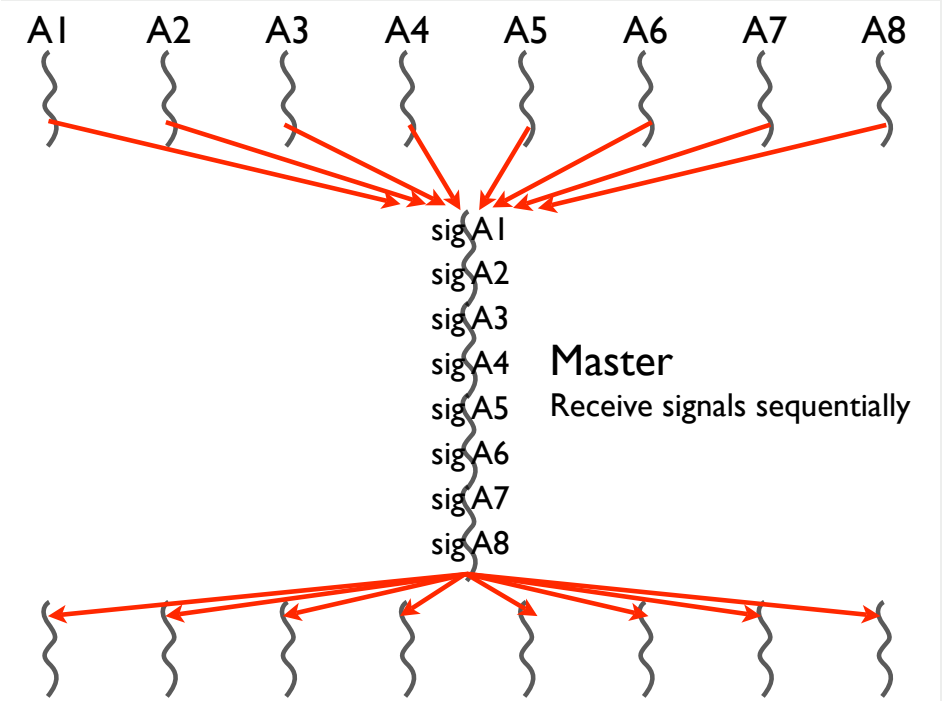
- **Bounded Phasers:** Limits maximum phase difference between producers and consumers
- **Accumulator Extension:** Non-barrier mode with bounded phasers

```
phaser ph = new phaser(<SIG_WAIT>, 2 /*Bound size*/);
async phased (ph<SIG>) { next; next; ... /*A1*/ }
async phased (ph<WAIT>) { next; next; ... /*A2*/ }
```

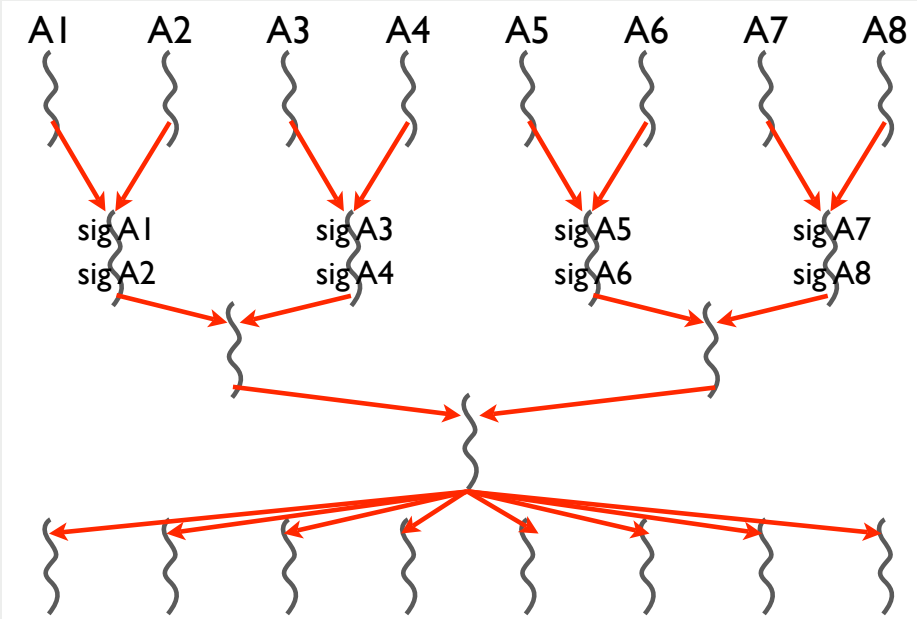


Hierarchical Implementations of Phasers with Sub-Master Tree

Single Level

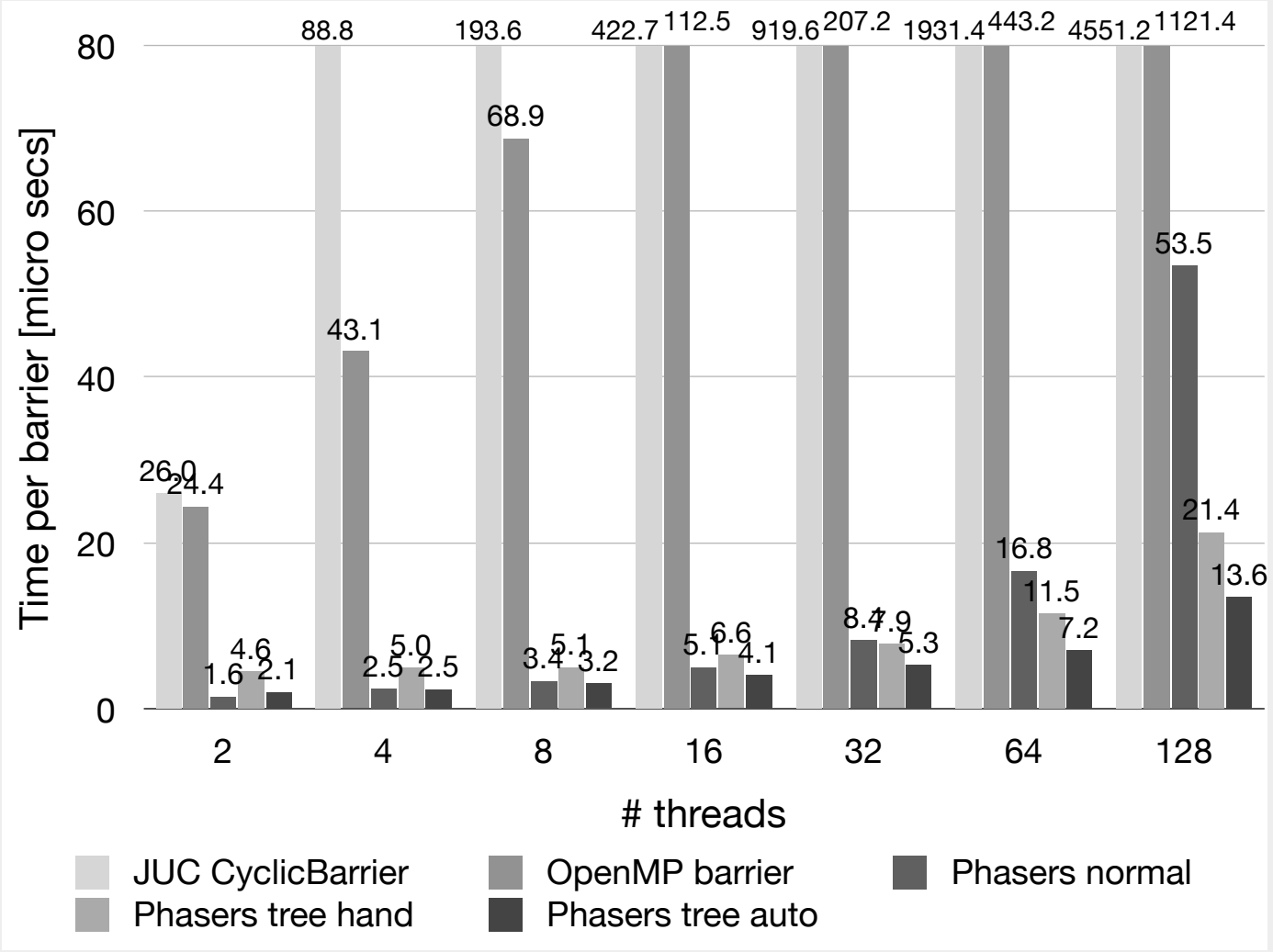


Hierarchical

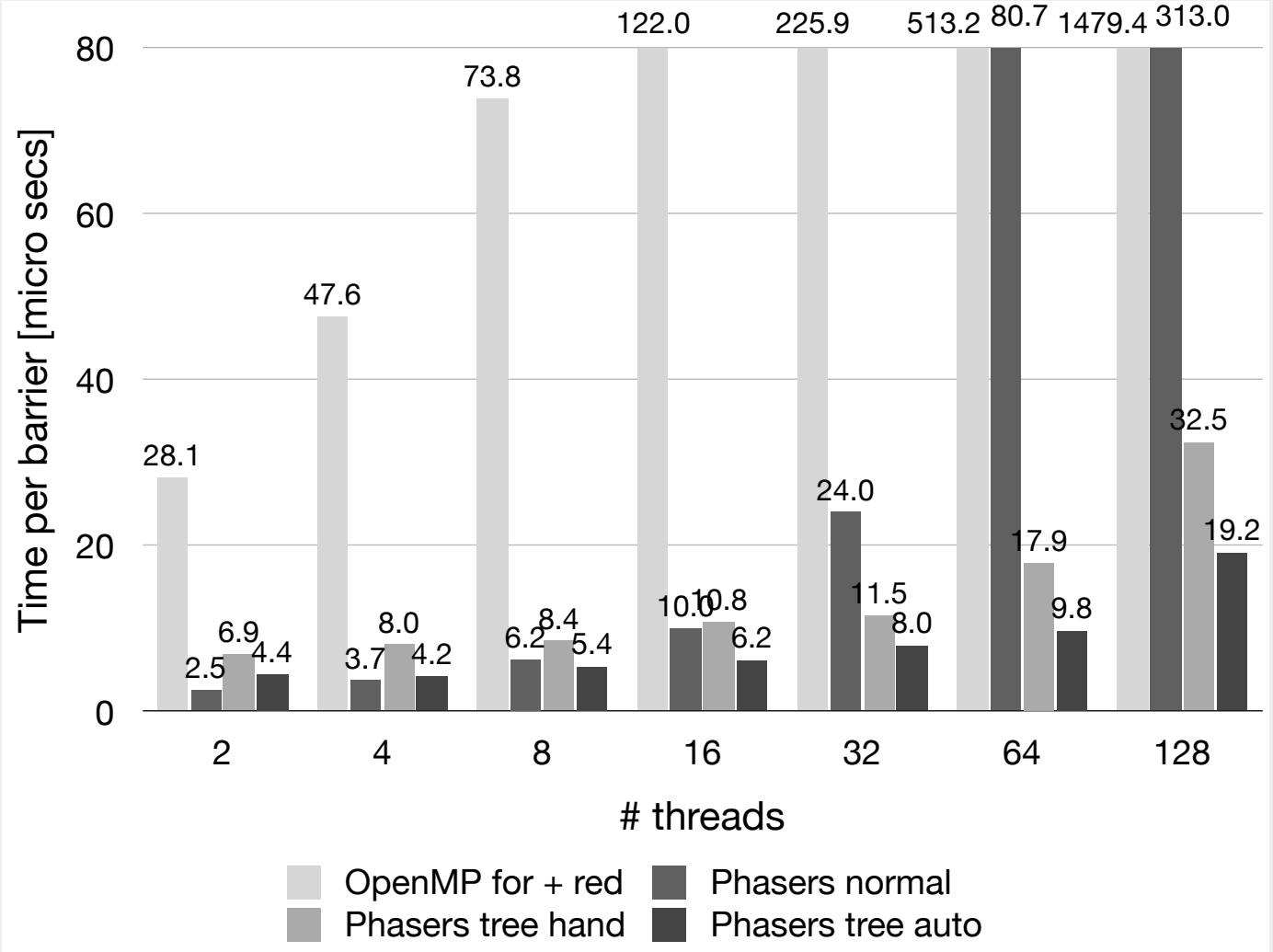


Sub-masters in the same tier receive signals in parallel

Barrier Performance with EPCC Syncbench on 128-thread Niagara 2

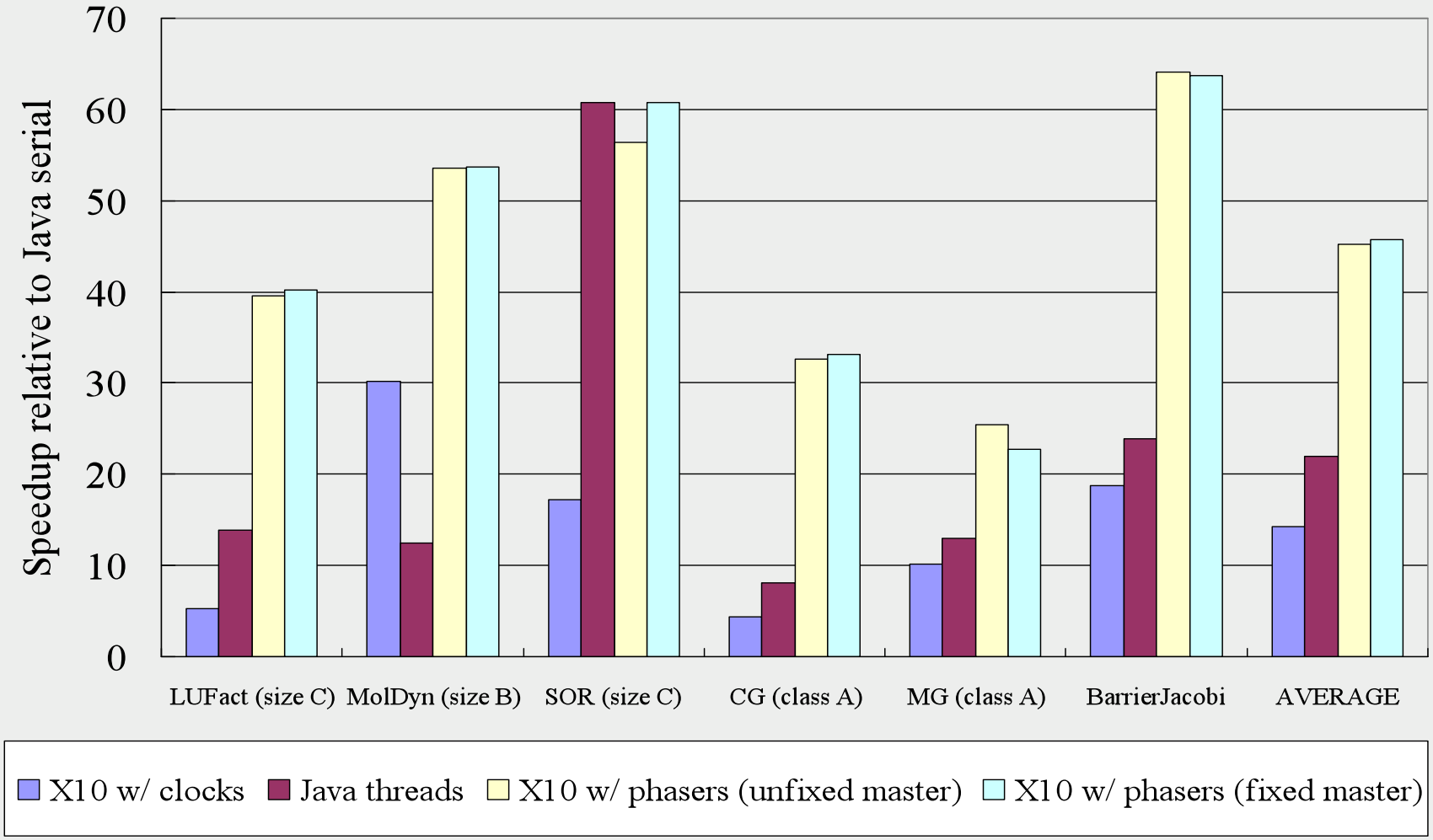


Barrier and reduction with EPCC Syncbench on 128-thread Niagara 2



Speedup on 64-way Power5+ SMP with Single Master

Java Grande Benchmarks & NAS Parallel Benchmarks



Average speedup with phasers (fixed master)

3.19x faster than X10 clocks, **2.08x** faster than Java threads (using CyclicBarrier)

RICE

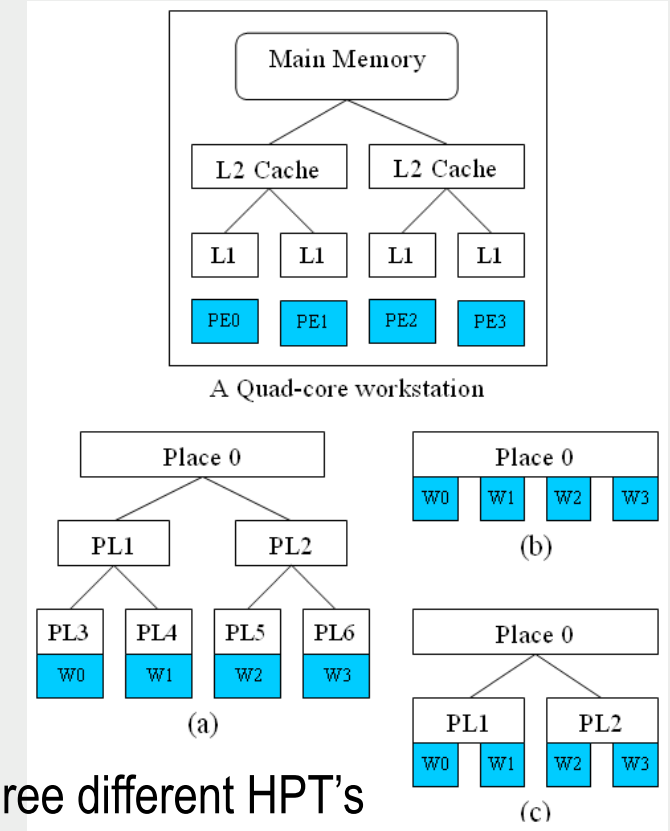
Habanero Execution Model: Portable Parallelism in Four Dimensions

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, PGAS
  - Hierarchical memory model with static parallelism e.g., Sequoia
- HPT 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
- Places can be used for CPUs and accelerators
- 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



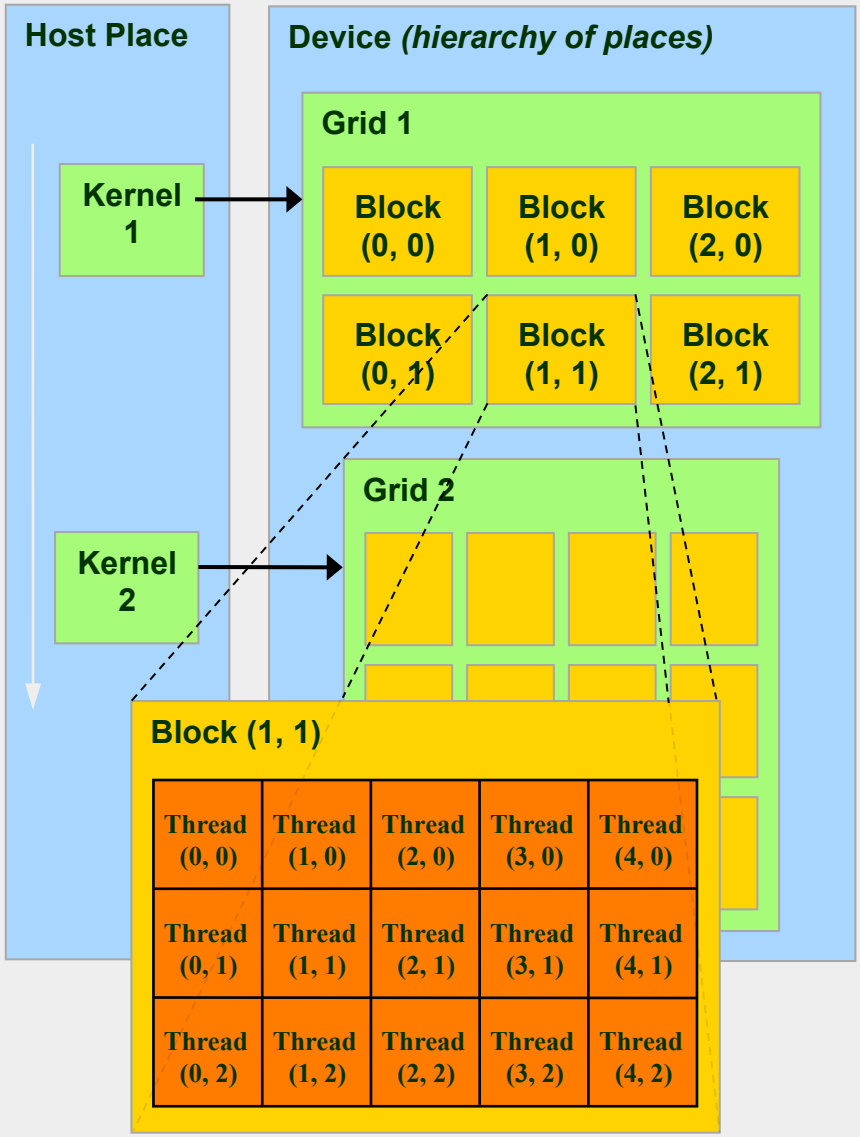
Three different HPT's for a quad-core processor

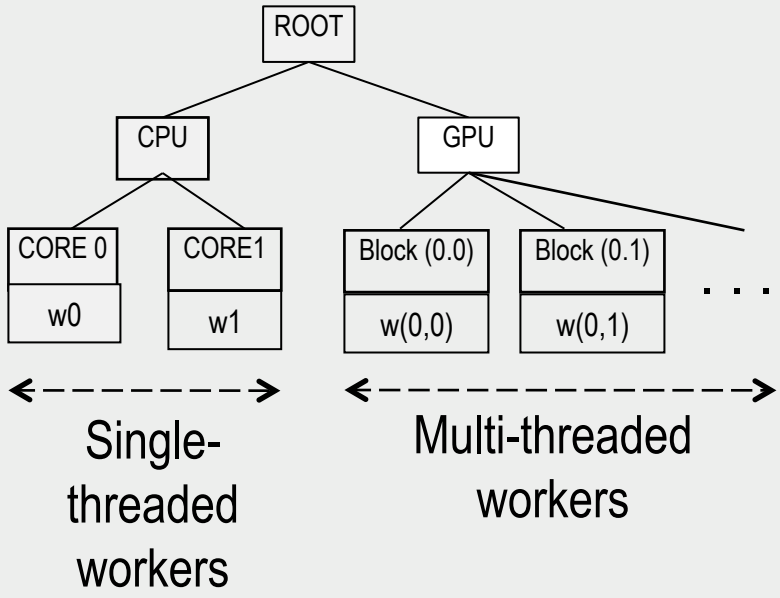
Examples of Using Places to Co-locate Computation and Data

```
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) isolated a.x = foo(a.x, v);
      }
```

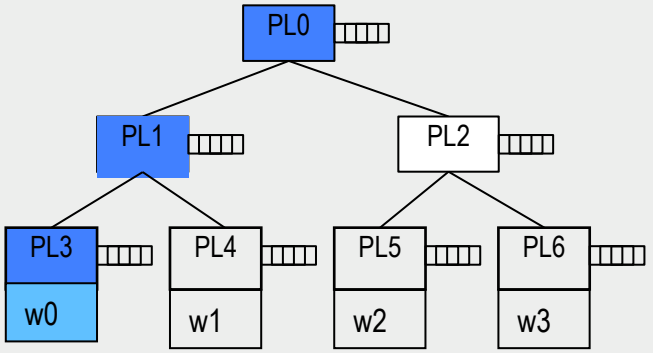

Hierarchical Place Tree for CPU + GPU





Locality-aware Scheduling using the HPT

- 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., use to model hierarchical shared memories
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., use to model inter-processor DMA (direct memory access) with *finish* for termination

Testing the Robustness of an Execution Model (Recap)

- Can it support multiple programming models of interest?
  - *Experiences with Habanero-Java, Habanero-C, CnC*
- Can it be mapped to a wide range of target hardware of interest?
  - *Experiences with a wide range of mainstream multicore CPUs, GPUs, and experimental processors like Cyclops*
- Does it emphasize management of critical system resources and goals?
  - e.g., locality, energy, concurrency, resiliency
  - *Hierarchical place tree emphasizes co-location of data and computation, async tasks and phasers emphasize asynchrony*

Summary

- Extreme Scale systems projected for 2015 – 2020 will need fundamental changes in Execution Model to address Concurrency and Energy Challenges
- System software will need to be co-designed across multiple levels and with hardware for effective management of locality and parallelism in Extreme Scale systems
- This talk presented early experiences with the Habanero execution model, and key primitives that could be useful in a portable execution model for extreme scale multicore processors
- Future work: implementation of Habanero execution model on integrated heterogeneous multicore systems

Acknowledgments

- Rice Habanero Multicore Software Research project
  - http://habanero.rice.edu
  - Sponsors and Donors: AMD, DARPA, IBM, Intel, Microsoft, NSF, NVIDIA, Sun
- Center for Domain-Specific Computing (CDSC)
  - UCLA, Rice University, Ohio State University, UCSB (http://www.cdsc.ucla.edu)
  - NSF Expeditions in Computing program
- Platform-Aware Compilation Environment (PACE) project
  - DARPA Architecture-Aware Compilation Environment (AACE) program
- Exascale Software study held from June 2008 to February 2009
  - Summary in short paper (SciDAC conference, June 2009), details in full report (Sep 2009)
- X10 project (http://x10-lang.org)
- Intel Concurrent Collections project (http://whatif.intel.com)
- Disclaimers
  - This work was supported in part by the National Science Foundation under the Expeditions in Computing and HECURA programs. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.
  - This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under AFRL contract FA8650-07-C-7724. The views, opinions, and/or findings contained in this presentation are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.