RICE
George R. Brown
School of Engineering
Computer Science

Software Challenges for Extreme Scale Systems

Vivek Sarkar

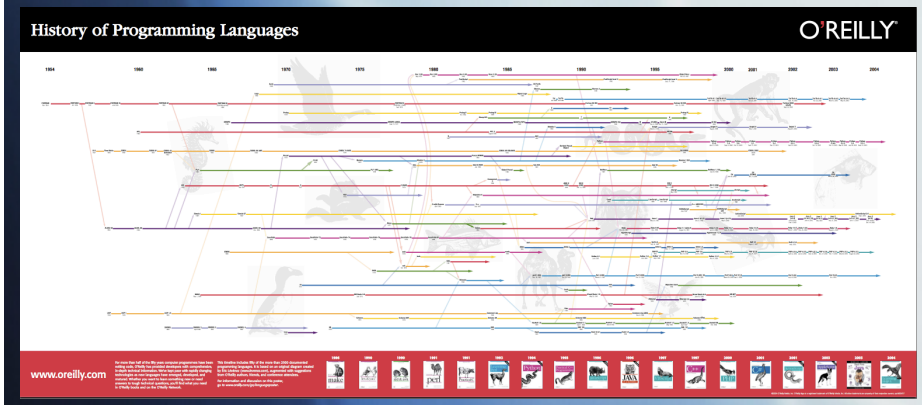
E.D. Butcher Chair in Engineering

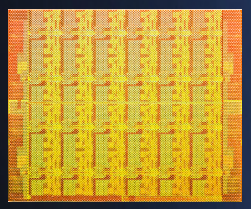
Professor of Computer Science

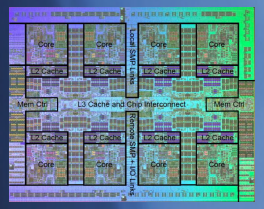
Professor of Elec. & Comp. Engineering

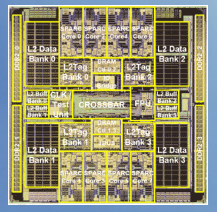
Rice University

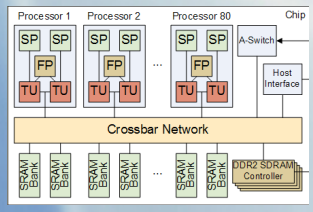
vsarkar@rice.edu

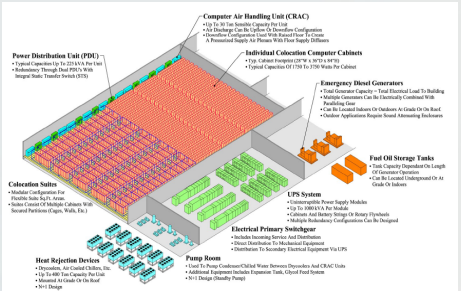












Outline

1 Exascale Software Study

2 Habanero Multicore Software Research project

RICE

Acknowledgments

- Exascale Software study conducted over a series of seven meetings held from June 2008 to February 2009
  - Summary in short paper (SciDAC Review, Jan 2010)
  - Details in full report (Sep 2009)

- Disclaimer
  - 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.

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
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***

"Software Challenges in Extreme Scale Systems". V. Sarkar, W. Harrod, A.E. Snavely. SciDAC Review Special Issue on Advanced Computing: The Roadmap to Exascale, pp. 60-65, January 2010.

RICE

Energy Efficiency Challenge in Extreme Scale Systems

- Goal: deliver 1000x increase in computational capability with modest (< 10x) increase in power requirements
  - Use of increased concurrency with lower clock frequency
  - Data movement is a major contributor to energy efficiency
  - Computation becomes relatively cheap (“free”)
- Software implications
  - Redesign software stack to enable expression and optimization of data movement and other high-energy operations
  - New viewpoint: high-performance correlated with low-energy
    - Software will need to compensate for reduction/removal of high-energy features from hardware such as out-of-order execution, branch prediction, caches, etc.
    - Software will need to cope with non-uniformities across cores arising from thermal & power management

Concurrency Challenge for Applications on Extreme Scale Systems

- Expose at least 10,000-fold more concurrency relative to current applications, subject to the following constraints
  - 10x – 100x lower bytes/ops ratios than current systems
  - Serial bottlenecks arising from software stack (Amdahl's Law)
  - Additional concurrency for latency hiding (Little's Law)
  - Data movement constraints implied by energy constraints
- Software implications
  - Exploit strong scaling algorithms in applications
  - Redesign software stack to reduce serial bottlenecks
  - Redesign software stack to enable more efficient data movement
  - Redesign communication, scheduling, and synchronization runtime systems to be *asynchronous* at all levels

What is Asynchrony?

- Removal of ordering constraints, wherever they may occur
  - Barrier synchronization → Point-to-point synchronization
  - Forall loops → Asynchronous tasks
  - Blocking communication → Nonblocking communication
  - Coarse-grain locking → Fine-grain locking
  - Conference calls → Email or svn updates
  - . . .
- Asynchrony enables parallelism
  - Tolerates variability at all levels

Thomas Sterling's Four Horsemen (SLOC)

- Starvation

Programmer-system interface

- Latency
- Overhead
- Contention

Figure source: http://www.tribulation.com/images/horsemen.jpg

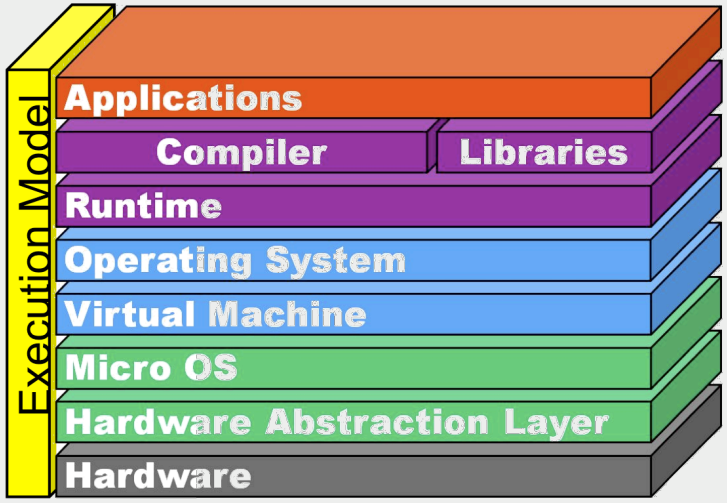
Contract: programmer provides abundant parallelism and asynchrony to avoid starvation, and implementation addresses latency, overhead, contention

Resiliency Challenge

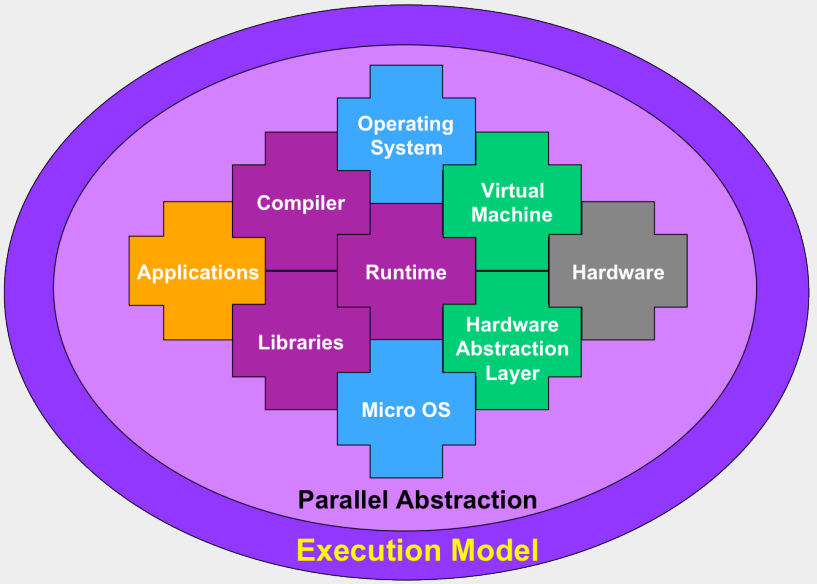
- MTBF in Extreme Scale systems will decrease and extra computing capacity will be lost due to increases in failures, checkpoints and recoveries
- Implications
  - New failure containment algorithms & recovery models
    - Current approaches incur increased overhead with strong scaling
  - New compiler technologies to support resilience
  - Reliability-aware power management
  - Increased tolerance for soft errors (which are correlated with lower voltages for energy efficiency)
  - New programming model constructs with functional/idempotent tasks that enable scalable checkpoint & recovery
- "System Resilience at Extreme Scale", white paper by M. Elnozahy et al

Focus Areas in Exascale Software Study

1. *Developing Applications* for Extreme Scale Computing
2. *Expressing Parallelism and Locality* in Extreme Scale Software
3. *Managing Parallelism and Locality* in Extreme Scale Software



20th century Software Stack



21st century Software Stack

Area 1: Developing Applications for Extreme Scale Computing

- **Challenge:** increased levels of concurrency in applications
- **Strong scaling:** apply more resources to the same problem size to get results faster
  - In practice, many applications are not amenable to strong scaling
  - May need to rewrite application from first principles to obtain strong scaling e.g., Qbox
- **Traditional weak scaling:** use more resources by increasing problem size
  - Reduce grid size and time-step interval
    - Memory requirement increases in ¾ power for 3-D problems and 2/3 power for 2-D problems

    ➔1000x increase in computational work requires ~ 180x and 100x in memory for 3-D and 2-D problems respectively
- **"New-era" weak scaling**
  - Application trends in which additional work is done per datum e.g., multi-scale, multi-physics, interaction analysis, data mining.
  - No increase in memory requirement

Area 2: Expressing Parallelism and Locality in Extreme Scale Software

- **Challenge:** forward-scalable and portable expression of intrinsic parallelism and locality
- **Breaking sequential habits of thought [Steele 08]**
  - Algorithms should focus on maximizing parallelism instead of minimizing operations or minimizing storage
  - Algorithms should focus on improving locality
    - Locality will have first-order impact on energy consumed
    - Explore ways to trade-off communication with extra computation
  - Use of multi-way rather than linear problem decomposition
    - e.g., use of recursive divide-and-conquer or high-level data-parallel operations instead of linear iteration for computing the sum of an array
- **Dynamic Asynchronous Parallelism** instead of Bulk Synchronous Parallelism
  - Dynamic tasks are present in shared-memory programming models e.g., OpenMP 3.0, Cilk, TBB
  - Will need to be extended to distributed global address space as in HPCS Languages --- Chapel, Fortress, X10

Area 2: Expressing Parallelism and Locality in Extreme Scale Software (contd)

- **Use of data structures and iterators** that are conducive to data parallelism
  - Vectors, arrays, streams + loop-level parallelism
  - Object-oriented collections + iterator-level parallelism
  - Irregular data structures + dynamic asynchronous parallelism
- **Portable Synchronization with Dynamic Parallelism**
  - Example: Habanero phasers derived from X10 clocks
  - Tasks register as producer (signal) and/or consumer (wait) on one or more phasers
  - Set of tasks registered with phaser can change dynamically
  - next statement guarantees that all synchronizations will be deadlock-free
- **High-level expression of massive parallelism**
  - Portable expression of horizontal and vertical locality using parameterized decomposition (partitions)
  - Algorithmic choice across scale
  - High level locality constructs e.g., places, locales

Area 3: Managing Parallelism and Locality in Extreme Scale Software

- **Operating System Challenges**
  - **OS structure has been largely unchanged since days of time-multiplexed uniprocessor systems**
- **Goal: deconstruct OS for Extreme Scale**
  - More asynchrony
  - Spatial partitioning
  - Move resource management from privileged to user mode
    - Retain OS for protection
    - Extend LightWeight Kernel (LWK) approaches
  - Efforts to increase scalability of Linux shed light on the difficulties
    - Improvements to Linux Scheduler
    - Large-page support
    - NUMA support,
    - Read Copy Update (RCU) API
    - Reducing scope of the Big Kernel Lock (BKL)

Area 3: Managing Parallelism and Locality in Extreme Scale Software (contd)

- **Runtime Challenges**
  - Lightweight runtime mechanisms for interconnection network features e.g., atomic operations, remote procedure invocation, one-sided communications and notifications
  - Runtime support for software-managed local memories
  - Runtime support for dynamic task management
  - Runtime support for memory system virtualization, including object caching and migration
  - Runtime-managed data placement in addition to the user-managed placement already available
- **Compiler challenges**
  - Target simpler low-energy processors
  - Automatic support for fine-grained parallelism (SIMD, SIMT)
  - Support for auto-tuning
  - Support for reliable software

Execution Models play a Key Role in Framing Cross-cutting Discussions

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)
??	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: Jan 2010, Full report: September 2009)

Towards an Exascale Execution Model: Desiderata and Challenges

1. **Dynamic asynchronous lightweight tasks for billion-way parallelism**
   - Motivation: maximize concurrency in face of variability
   - Challenge: adaptive scheduling with bounded resources
2. **Global name space with asynchronous one-sided communications**
   - Motivation: latency hiding, portability, interconnect variability
   - Challenge: efficient address translation and short messages
3. **Explicit horizontal and vertical locality**
   - Motivation: locality is key to efficient parallelism
   - Challenge: portable and hierarchical abstractions of locality
4. **Scalable coordination and synchronization**
   - Motivation: avoid global synchronization bottlenecks
   - Challenge: scalable implementations of mutual exclusion, producer-consumer and barrier synchronizations

➔ Von Neumann + BSP execution model will no longer suffice

Other Topics discussed in Report

- Role of Libraries
  - Properties: Domain-Specific, Numerical, Communication, Data Management, I/O
  - Enabling technologies: Standardized interfaces, Vendor support
- Role of Tools
  - Properties: Performance, Reliability, Scalability, Abstraction, Adaptation, Integration, Availability, Portability
  - Enabling technologies: Data Collection, Data Analysis, Data Mining, Companion Computations, Auto-tuning
- . . .

Summary

- Extreme Scale systems projected for 2015 – 2020 will need fundamental changes in Execution Model and System Software to address Concurrency, Energy Efficiency, and Resiliency challenges

- Applications will need to exploit a combination of strong scaling, traditional weak scaling, and new-era weak scaling techniques, in conjunction with suitable attention to energy efficiency (data movement & locality)

- Programming models will need to express all intrinsic parallelism and locality for forward scalability and portability

- 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
  - Current & future many-core systems can serve as useful intermediate platforms for evaluating intra-node software technologies for Extreme Scale

Outline

1 Exascale Software Study

2 Habanero Multicore Software Research project

Acknowledgments

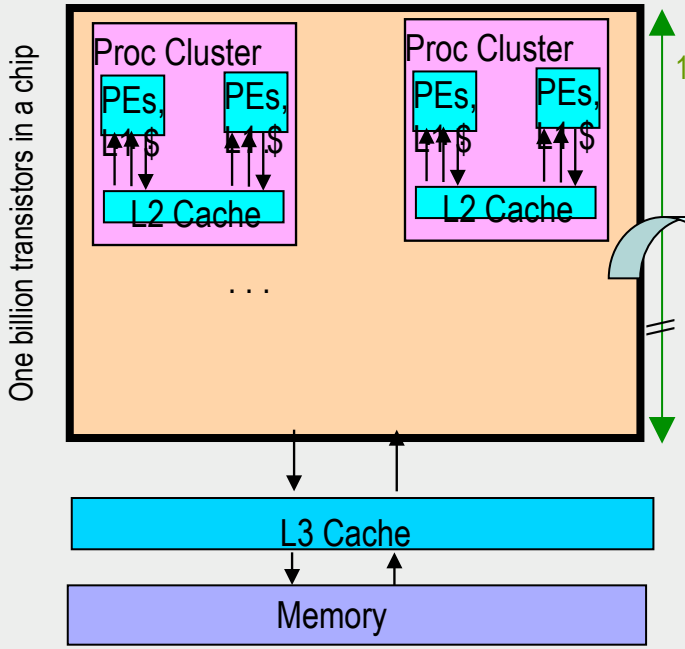
- Rice Habanero Multicore Software Research project (http://habanero.rice.edu)
  - Industry Sponsors and Donors: AMD, IBM, Intel, Microsoft, NVIDIA, Oracle/Sun
- NSF Expeditions Center for Domain-Specific Computing (CDSC)
  - UCLA, Rice University, Ohio State University, UCSB (http://www.cdsc.ucla.edu)
- NSF HECURA 2008, HECURA 2009, and CCF 2010 programs
- DARPA Exascale Software study, June 2008 - February 2009
- DARPA AACE program, March 2009 – present
- MARCO Multiscale Systems Center (http://musyc.org)
- IBM X10 project (http://x10-lang.org)
- Intel Concurrent Collections project (http://whatif.intel.com)
- COMP 322, “Fundamentals of Parallel Programming” undergraduate course, Rice University
  - https://wiki.rice.edu/confluence/display/PARPROG/COMP322
- DARPA HPCS program (for nostalgic reasons …)

Grand Challenge from DARPA HPCS program

- *Deliver 10x improvement in HPC application development productivity over today's systems by 2010, while delivering acceptable performance on large-scale systems*
- PERCS hypothesis: the two fundamental obstacles to improving HPC application development productivity are
  1. Programming complexity
  2. Expertise gap

"X10: Addressing Language, Compiler, and Runtime Challenges for Scalable Systems in 2010", V.Sarkar, LCR 2004 workshop, October 2004.

Obstacle #1 (Programming Complexity) --- High Complexity of HPC Systems Limits HPC Application Development Productivity



1995: entire chip can be accessed in 1 cycle

2010: only small fraction of chip can be accessed in 1 cycle

Major sources of complexity for application developer:

1) Severe non-uniformities in data accesses

2) Applications must exhibit large degrees of parallelism

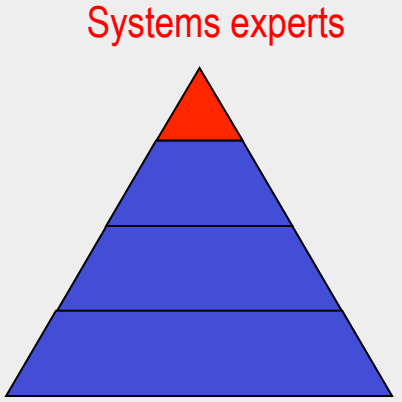
Complexity leads to increases in all phases of HPC Software Lifecycle related to parallel code

HPC Software Lifecycle

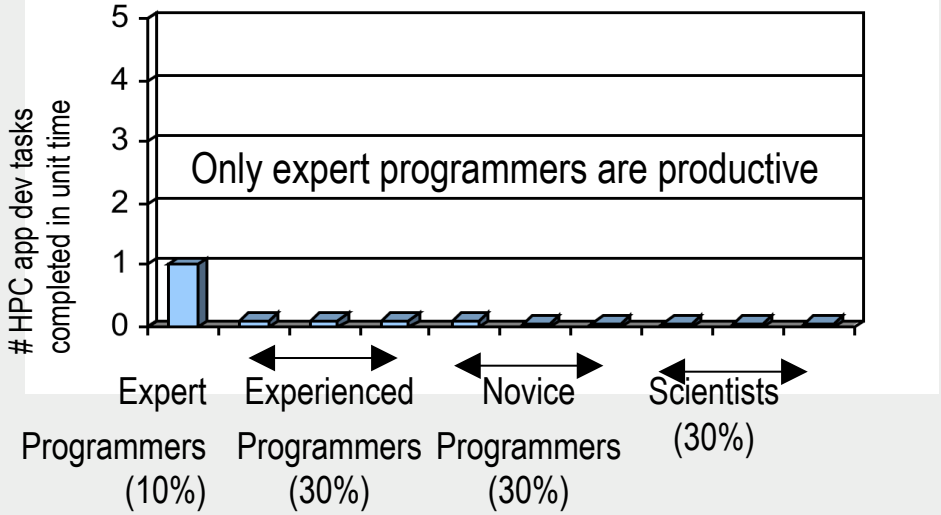
Requirements	Input Data	Algorithm Development	Written Specification	Source Code	Parallel Specification	Development of Parallel Source Code --- Design, Code, Test, Port, Scale, Optimize	Production Runs of Parallel Code	Maintenance and Porting of Parallel Code

Obstacle #2 (Expertise Gap) --- Low Availability of Expert System Programmers who can develop/scale production HPC apps

- Two classes of programming skills:
  - Expert knowledge of concurrent programming
    - top-gun knowledge of system s/w and h/w
    - find today's programming models time-consuming to use
  - Others
    - Scientists
    - Domain experts
    - . . .

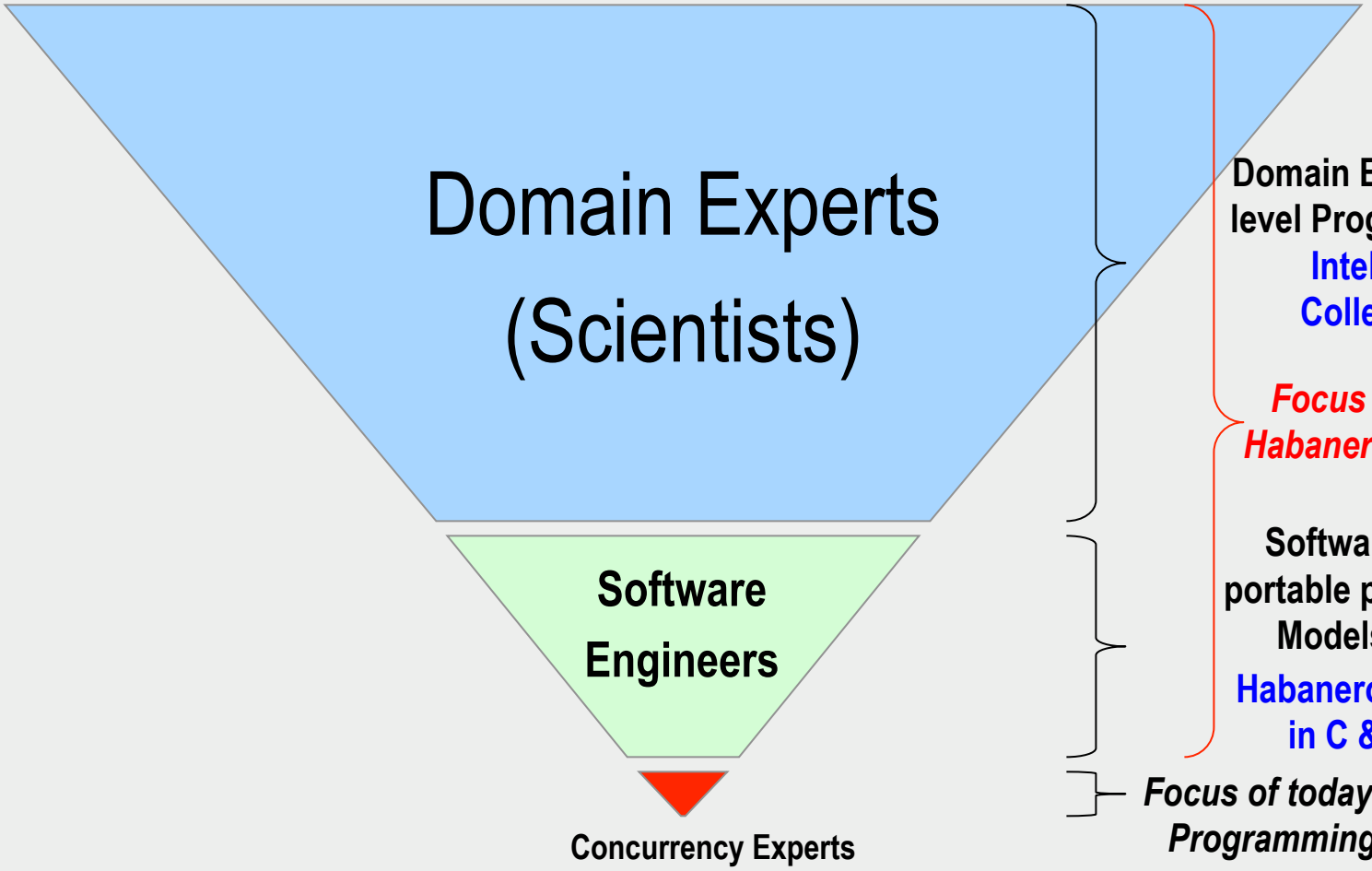


Application development productivity on current HPC systems:



Expertise profile

Motivation for Habanero project --- Inverted Pyramid of Parallel Programming Skills and Software Development

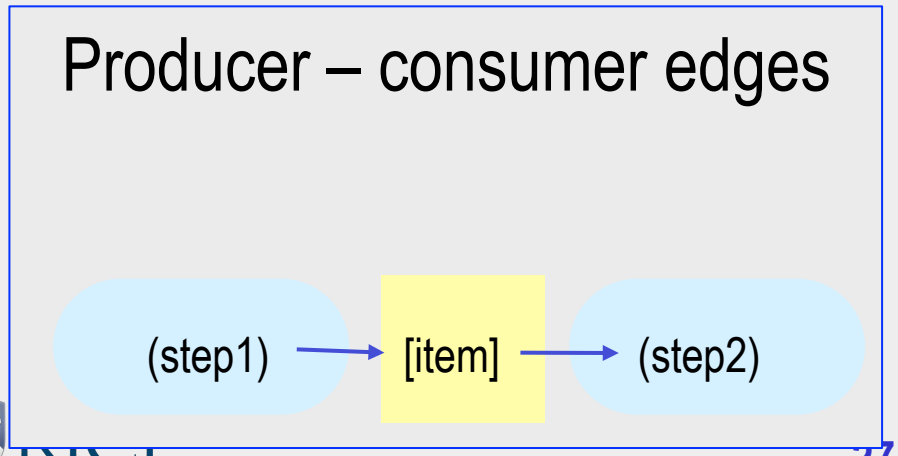


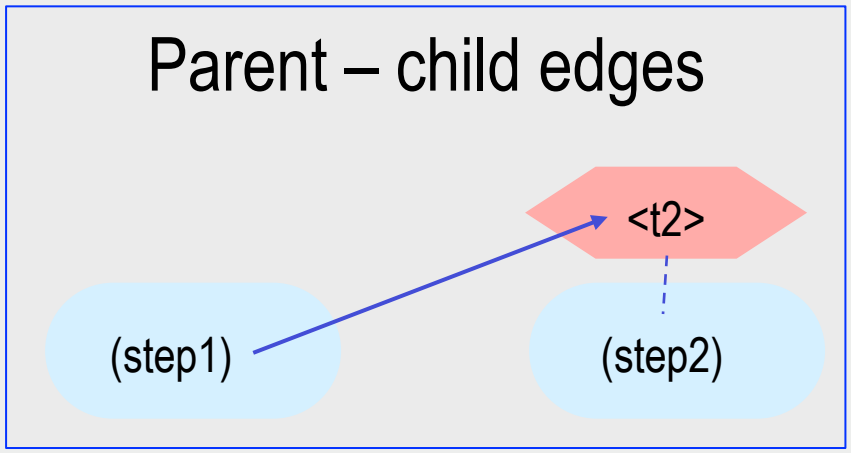
Our approach --- three levels of programming models for three different roles

Role	Current Programming Models	Habanero Programming Models (what we're working with right now)
Domain Expert	Matlab, R, SQL, Python, …	*Domain-specific programming models* based on inter-step *macro-dataflow coordination graphs* (CnC) and intra-step *domain-specific language extensions*
Software engineer	C/C++, Fortran, Java, Chapel, X10, …	*Dynamic task parallelism:* asynchronous tasks and communications with safety guarantees --- Habanero-Java, Habanero-C, X10
Concurrency Expert	Pthreads, OpenMP, MPI, UPC, CAF, …	

Programmability Gap: Future Freshman-Level Course planned for non-CS majors

- **Approach**
  - Introduce students to fundamentals of parallel programming using Intel Concurrent Collections with Python steps (CnC-Python)
  - Stealth approach: don't tell non-CS majors that they have to learn functional programming …
  - … instead, ask them to specify their program as a graph that specifies only the semantic ordering constraints in their application
  - Compositional --- step internals can be implemented in any language e.g., Matlab, Python, Java, C, C++, Scala, …





Programmability Gap: Sophomore-Level CS Course at Rice

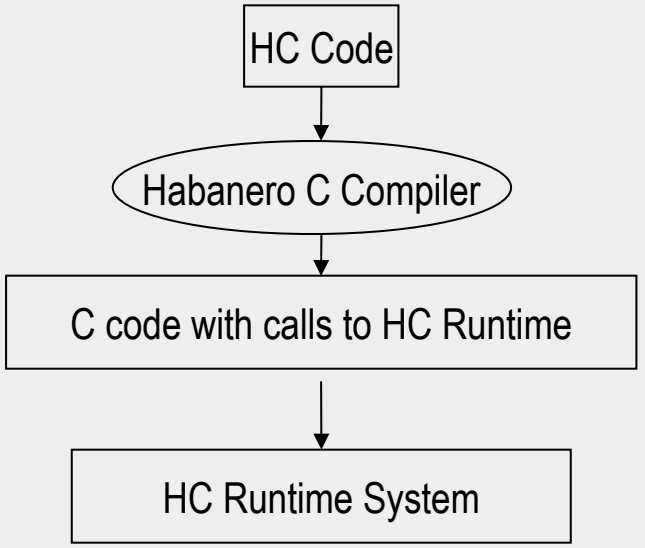
- **COMP 322: Fundamentals of Parallel Programming**
  - https://wiki.rice.edu/confluence/display/PARPROG/COMP322
- **Approach**
  - Introduce students to fundamentals of parallel programming
    - Primitive constructs for task creation & termination, collective & point-to-point synchronization, task and data distribution, and data parallelism
    - Abstract models of parallel computations and computation graphs
    - Parallel algorithms and data structures including lists, trees, graphs, matrices
    - Common parallel programming patterns including task parallelism, undirected and directed synchronization, data parallelism, dataflow, divide-and-conquer, map-reduce, concurrent event processing
  - Use Habanero-Java (HJ) as pedagogical language to understand fundamentals in majority of lectures, and then progress to standard programming models

Habanero-Java (HJ) Overview

- New language and implementation developed in Habanero Multicore Software research project at Rice (http://habanero.rice.edu/hj)
  - Derived from Java-based v1.5 of X10 language in 2007
    - X10 has evolved significantly; v2.2 is a very different language from v1.5
  - HJ is an extension of Java
    - All Java 5 & 6 libraries and classes can be called from HJ programs
- HJ extensions are focused on task parallelism
  1. Dynamic task creation & termination: forall, async, finish, force
  2. Locality control --- task and data distributions: places, here
  3. Mutual exclusion and isolation: isolated
  4. Collective and point-to-point synchronization: phaser, next
- Habanero-Scala is under development with similar constructs

Habanero-C (HC) Overview

- **Habanero-C language:** C extensions with Habanero constructs
- **Compiler:** source-to-source translation using Rose
  - Optional Rose→LLVM translator
- **Runtime:** Portable work-stealing and work-sharing schedulers
  - ~10 lines of assembly code needed for each platform
- **GPU Support**
  - Hybrid scheduling
- **FPGA support**
  - Library calls from HC tasks
- **Cluster support**
  - Shmem prototype completed (w/ Portals 4)
  - GASNet prototype recently started
- **Currently supported platforms:**
  - Intel x86 and x_64
  - Sun Niagara 2
  - Convey HC-1 + GPU (in progress)
  - Intel SCC (in progress)
  - IBM Cyclops64 (in progress)




```
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  finish {
    async IN(n) OUT(x) {x = fib(n – 1);}
    async IN(n) OUT(y) {y = fib(n - 2);}
  }
  return x + y;
}
```

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
   - *isolated*

4) Collective and point-to-point synchronization
   - *phasers*

Habanero Programming Languages

Habanero Static Compiler & Parallel Intermediate Representation

Habanero Runtime System

Two-level programming model

Declarative Coordination Language for Domain Experts,

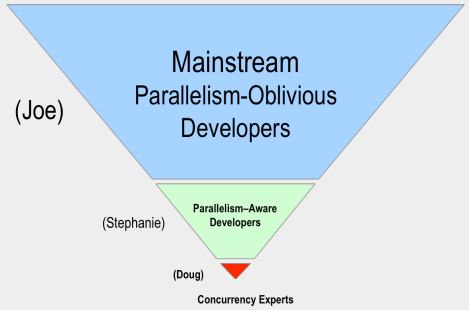
CnC (Intel Concurrent Collections)

+

Task-Parallel Languages for Parallelism-aware Developers:

Habanero-Java (from X10 v1.5),

Habanero-C, Habanero-Scala



Extreme Scale Platforms

Why focus on the Runtime System today?

- Extreme Scale computing is limited by concurrency, energy efficiency, and resiliency
- Inherent variability in UHPC system software and hardware components calls for end-to-end asynchrony in system design
- Tight integration of inter-node and intra-node parallel runtime systems has proven elusive thus far
- Important to use right runtime primitives as foundation for future languages and compilers

➔The runtime system will play a pivotal role in bridging the Programmability and Performance Gaps

Async and Finish

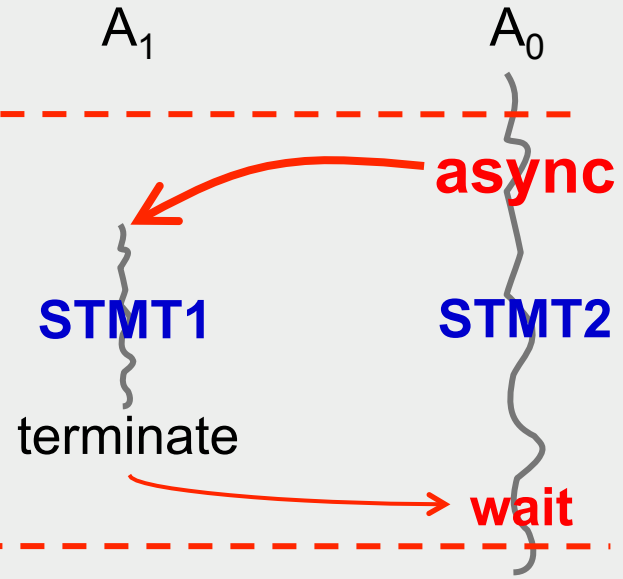
async S

- Creates a new child task that executes statement S
- Parent task moves on to statement following the async

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
- Use of finish cannot create a deadlock cycle

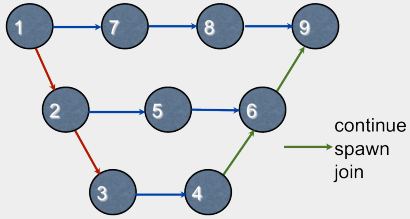
```
//A0 (Parent)
finish {    //Begin finish
  async {
    STMT1;  //A1 (Child)
  }
  STMT2;    //A0
}           //End finish
```



Comparing Async-Finish with Cilk's Spawn-Sync

- Async-Finish permits sequential and parallel calls to the same function
- Async-Finish permits an async to "escape" a parent task
- Async-Finish computations are more general than Cilk's fully strict computations

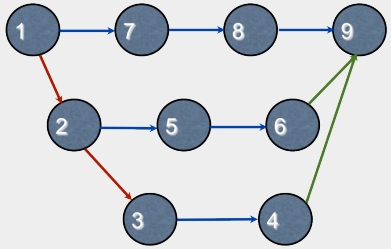
Class I: Fully-Strict Graphs




```
finish{
  async{
    finish{
      async{3;4;}
      5;6;
    }
  }
  7;8;9;}
```

- Graphs generated by Cilk style spawn and sync constructs

Class II: Terminally-Strict Graphs




```
finish{
  async{
    async{3;4;}
    5;6;
  }
  7;8;9;}
```

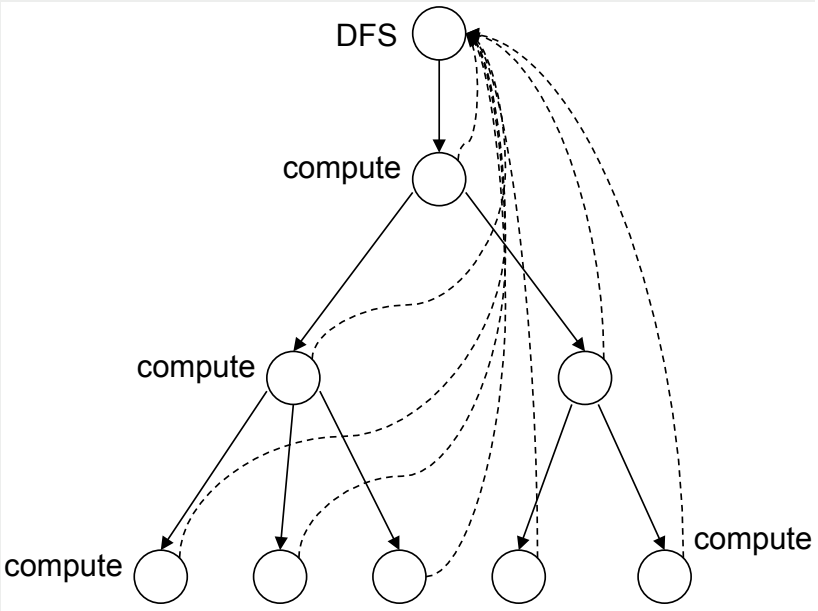
- Graphs generated by basic async and finish constructs in X10/HJ
- Superset of fully-strict graphs

Fully-strict computation

Async-finish computation

Parallel Depth-First Search Spanning Tree

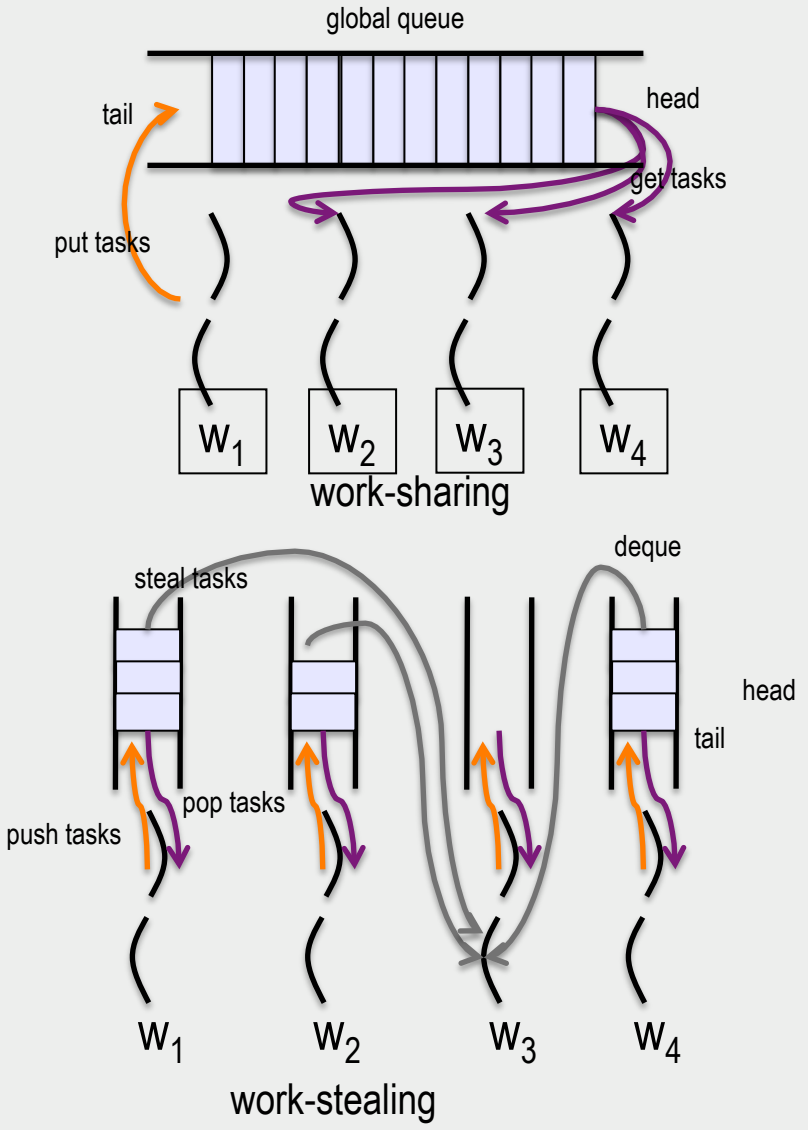
```
class V  {
  V [] neighbors; // Input adjacency list
  V parent; // Output spanning tree
  . . .
  boolean tryLabeling(V n) {
    isolated if (parent == null) parent = n;
    return parent == n;
  } // tryLabeling
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      V child = neighbors[i];
      if (child.tryLabeling(this))
          async child.compute(); //escaping async
     }
  } // compute
} // class V
root.parent = root; //Use self-cycle to identify root
finish root.compute();
```



Habanero-C runtime: Scheduling Paradigms

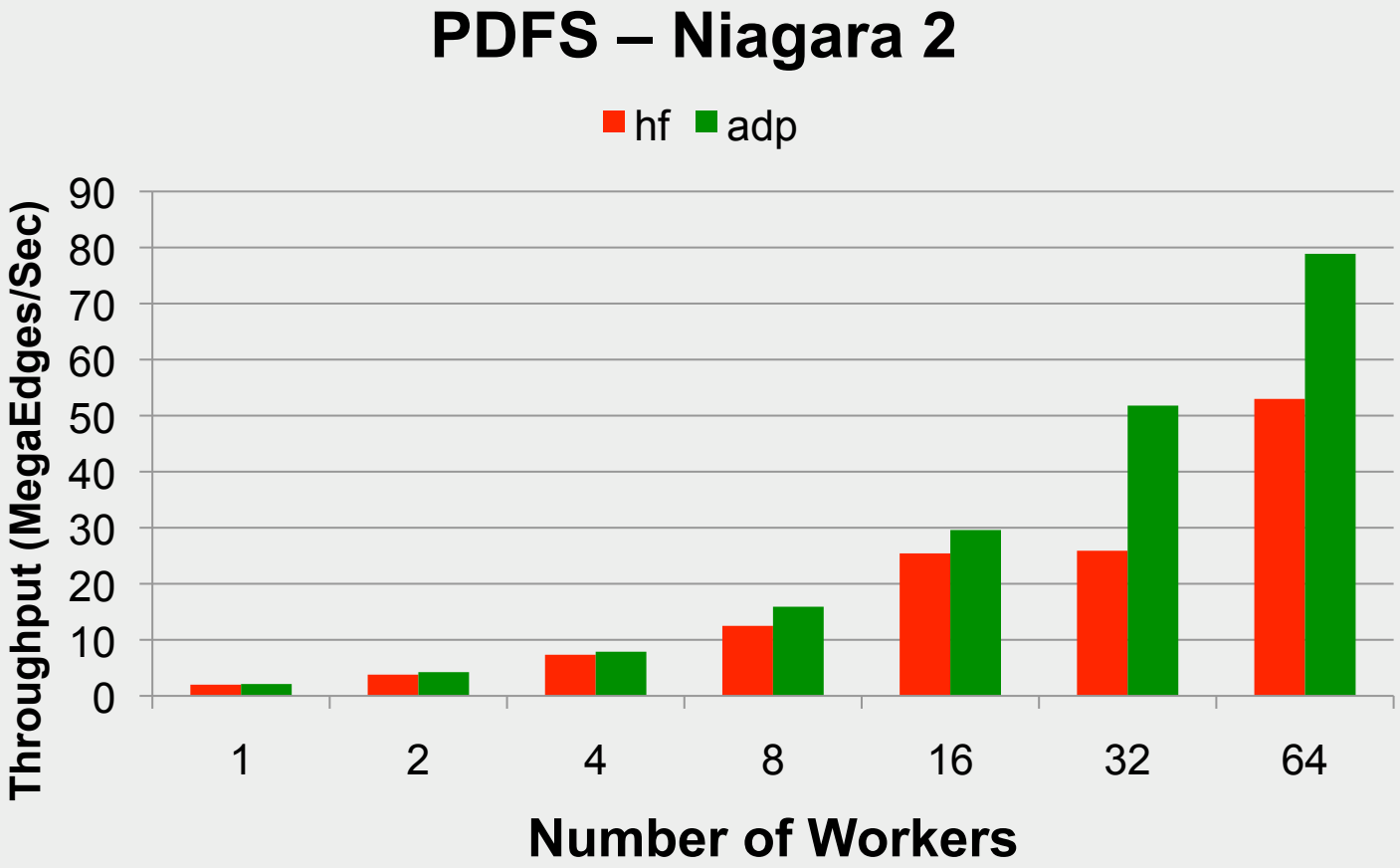
- Work-Sharing (X10 v1.5, OpenMP, …)
  - Busy worker re-distributes the task eagerly
  - Global thread/task/team queue
  - Access to the global queue needs to be synchronized: scalability bottleneck

- Work-Stealing (Cilk, TBB, …)
  - Distributed task pools: Each worker has a local double-ended queue (deque)
  - Idle worker steals the tasks from busy workers
  - Support for work-first and help-first policies



- "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.

PDFS on a Torus Graph with 4M vertices (Habanero-Java implementation)



Work-first policy is unable to complete due to stack overflow

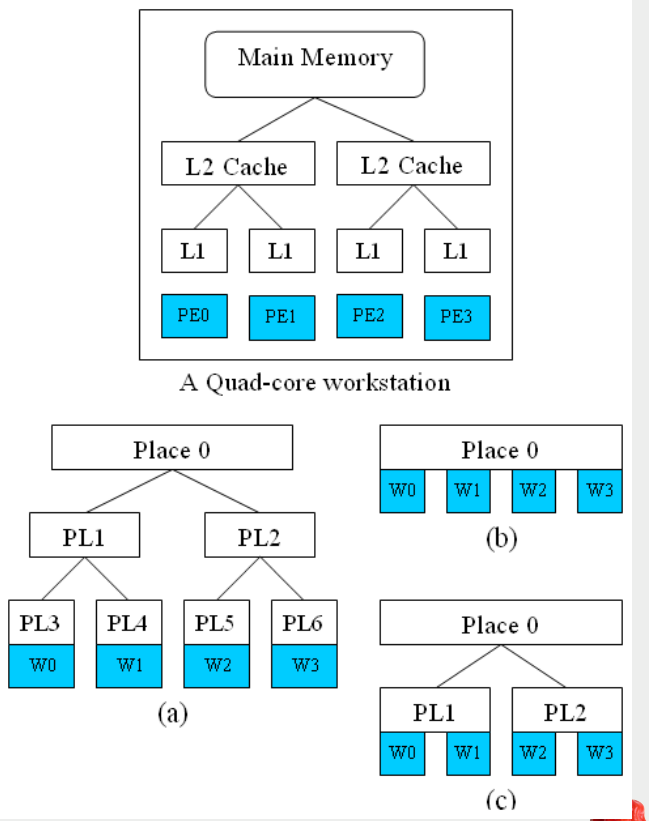
Adaptive (adp) policy performs better than help-first policy

Locality control for task and data distribution: Limitations of Past Work

- Most shared-memory execution models perpetuate the myth of uniformly accessible memory
- Partitioned Global Address Space (PGAS) models support a flat single-level partition e.g., HPF, UPC, CAF
- Past work on hierarchical memory models were limited to static parallelism e.g., Sequoia
- No support in past work for modeling hierarchical memories with dynamic parallelism

Hierarchical Place Trees (HPT)

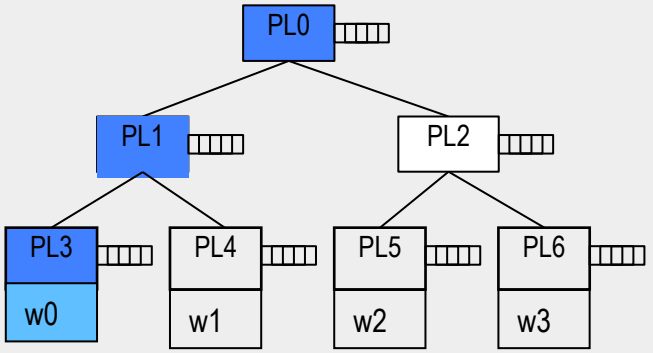
- Past approaches
  - Flat single-level partition e.g., HPF, PGAS
  - Hierarchical memory model with static parallelism e.g., Sequoia
- HPT approach
  - Hierarchical memory + Dynamic parallelism
- Place denotes memory hierarchy level
  - Cache, SDRAM, device memory, …
- Leaf places include worker threads
  - e.g., W0, W1, W2, W3
- Explore multiple HPT configurations
  - For same hardware and application
  - 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

Locality-aware Scheduling using the HPT

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

Examples of Using Places to Co-locate Computation and Data

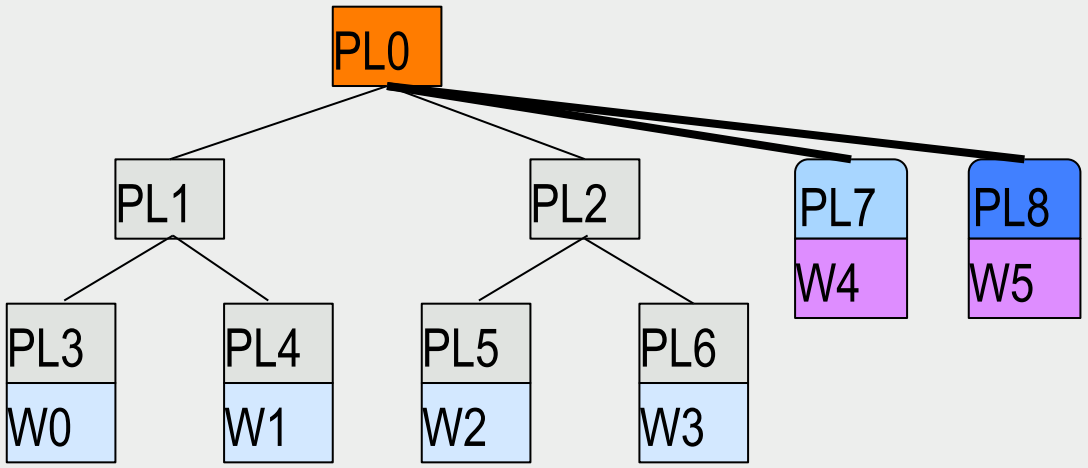
```
1) finish { // Inter-place parallelism
       final int x = … , y = … ;
       async (a) a.foo(x); // Execute at a's place
       async (b.distribution[i])
           b[i].bar(y); // Execute at b[i]'s place
   }

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);
      }
```

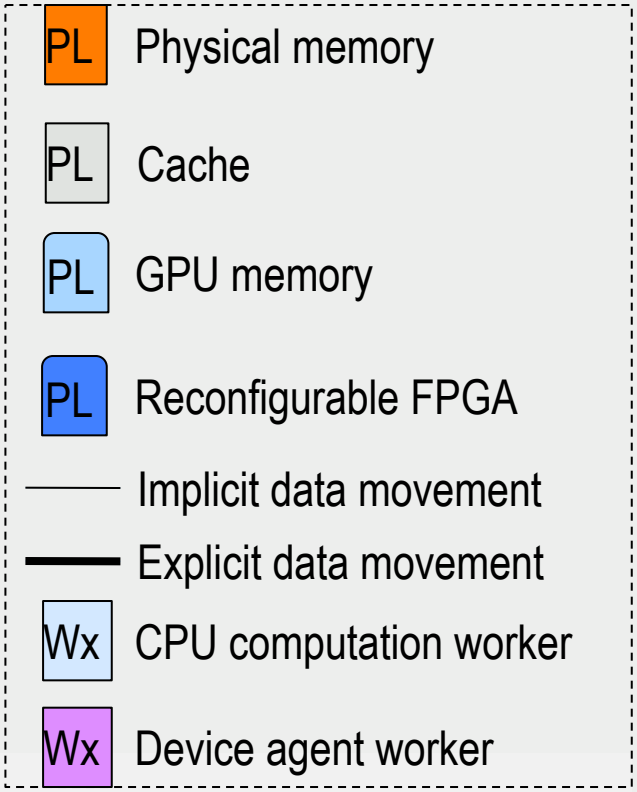

Data transfers in HPT

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 at**(<*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

HC Hierarchical Place Trees for heterogeneous architectures



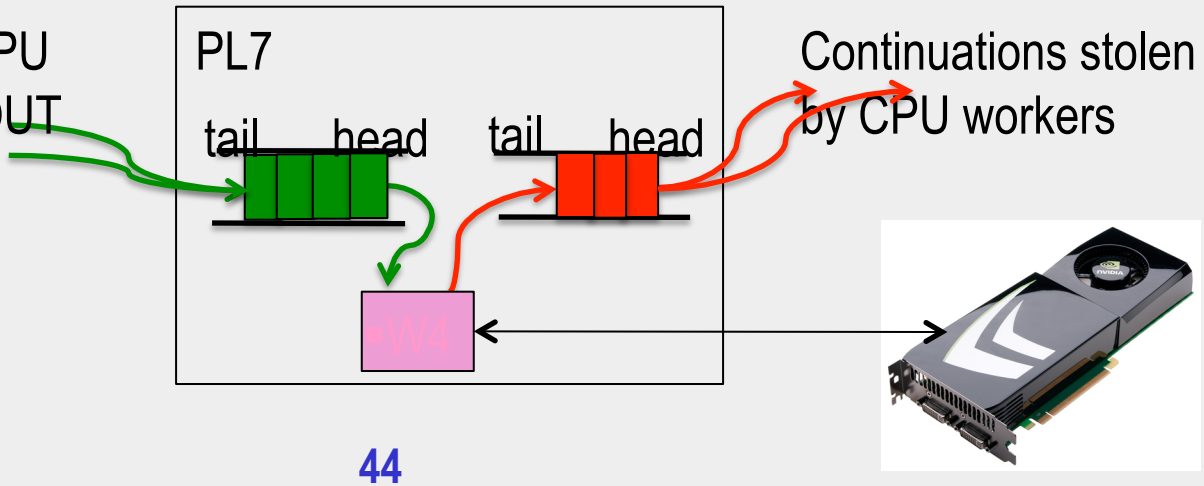


- **Devices (GPU or FPGA) are represented as memory module places and agent workers**
  - **GPU memory configuration are fixed, while FPGA memory are reconfigurable at runtime**
- **async at(P) S**
  - **Creates new activity to execute statement S at place P**
- **Physically explicit data transfer between main memory and device memory**
  - **Use of IN and OUT clauses to improve programmability of data transfers**
- **Device agent workers**
  - **Perform asynchronous data copy and task launching for device**

Hybrid Scheduling in the Habanero-C runtime

- Device place has two SC (semi-concurrent) queues: red and green
  - No locks – highly efficient
- Green queue maintains asynchronous device tasks (with IN/OUT)
  - Concurrent enqueuing device tasks by CPU workers from tail
  - Sequential dequeuing tasks by device agent workers from head
- Red queue maintains continuation of the finish scope of tasks
  - Sequential enqueuing continuation by agent workers
  - Concurrent dequeuing (steal) by CPU workers

Device tasks created from CPU worker via async at(gpl) IN OUT { … }



NSF Expeditions Center for Domain-Specific Computing (CDSC) (http://cdsc.ucla.edu)

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

CDSC Motivation: Potential for Customization

AES 128bit key 128bit data	Throughput	Power	Figure of Merit (Gb/s/W)
0.18mm CMOS	3.84 Gbits/sec	350 mW	11 (1/1)
FPGA [1]	1.32 Gbit/sec	490 mW	2.7 (1/4)
ASM StrongARM [2]	31 Mbit/sec	240 mW	0.13 (1/85)
Asm Pentium III [3]	648 Mbits/sec	41.4 W	0.015 (1/800)
C Emb. Sparc [4]	133 Kbits/sec	120 mW	0.0011 (1/10,000)
Java [5] Emb. Sparc	450 bits/sec	120 mW	0.0000037 (1/3,000,000)

[1] Amphion CS5230 on Virtex2 + Xilinx Virtex2 Power Estimator

[2] Dag Arne Osvik: 544 cycles AES – ECB on StrongArm SA-1110

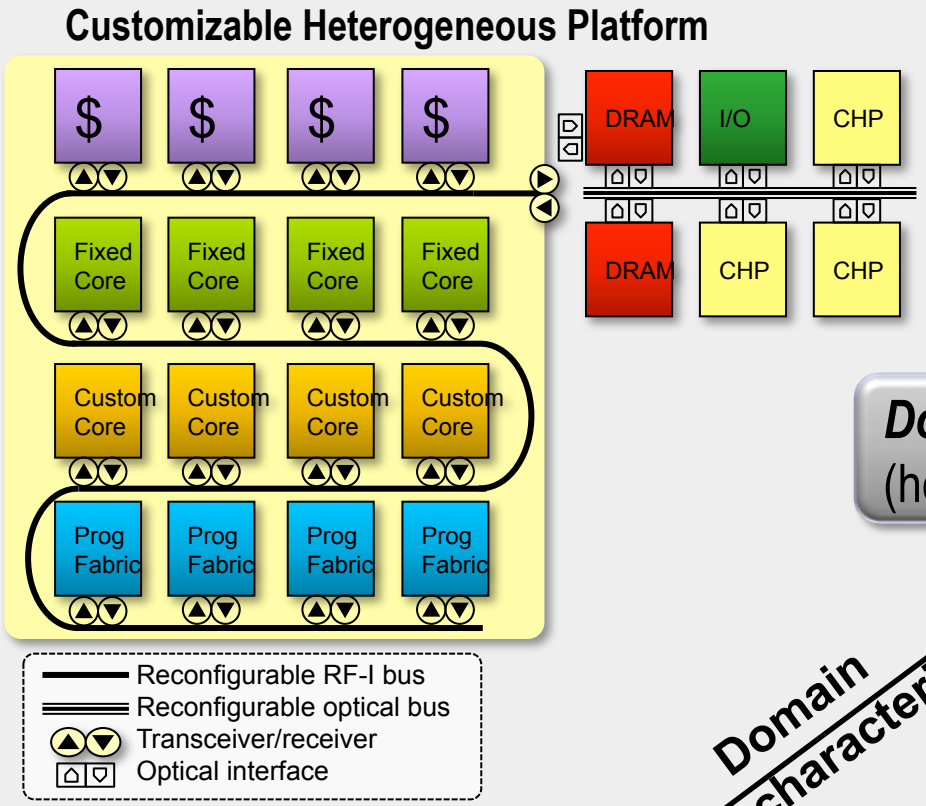
[3] Helger Lipmaa PIII assembly handcoded + Intel Pentium III (1.13 GHz) Datasheet

[4] gcc, 1 mW/MHz @ 120 Mhz Sparc – assumes 0.25 u CMOS

[5] Java on KVM (Sun J2ME, non-JIT) on 1 mW/MHz @ 120 MHz Sparc – assumes 0.25 u CMOS

Source: P. Schaumont, and I. Verbauwhede, "Domain specific codesign for embedded security," IEEE Computer 36(4), 2003.

Center for Domain-Specific Computing: Overview of Proposed Research



Key software challenges:

1) How to *model* applications in a domain?
2) How to *map* applications on to the CHP?
3) How to *bridge* from application models to CHP-deployed code?

Domain-specific-modeling
(healthcare applications)

Domain characterization

Application modeling

Architecture modeling

CHP creation and configuration
Customizable computing engines
Customizable interconnects

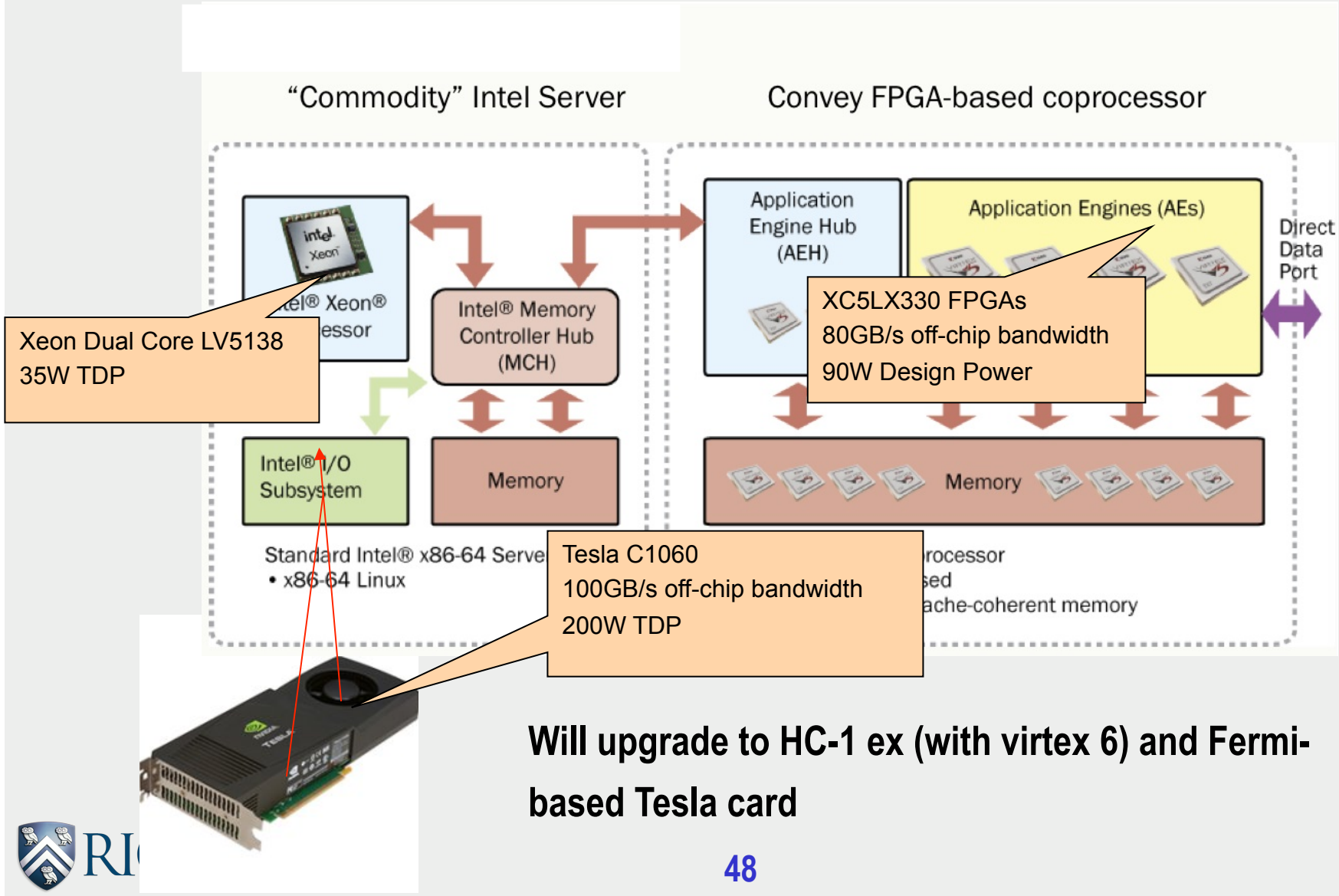
CHP mapping
Source-to-source CHP mapper
Reconfiguring & optimizing backend
Adaptive runtime

Design once (configure)

Customization settings

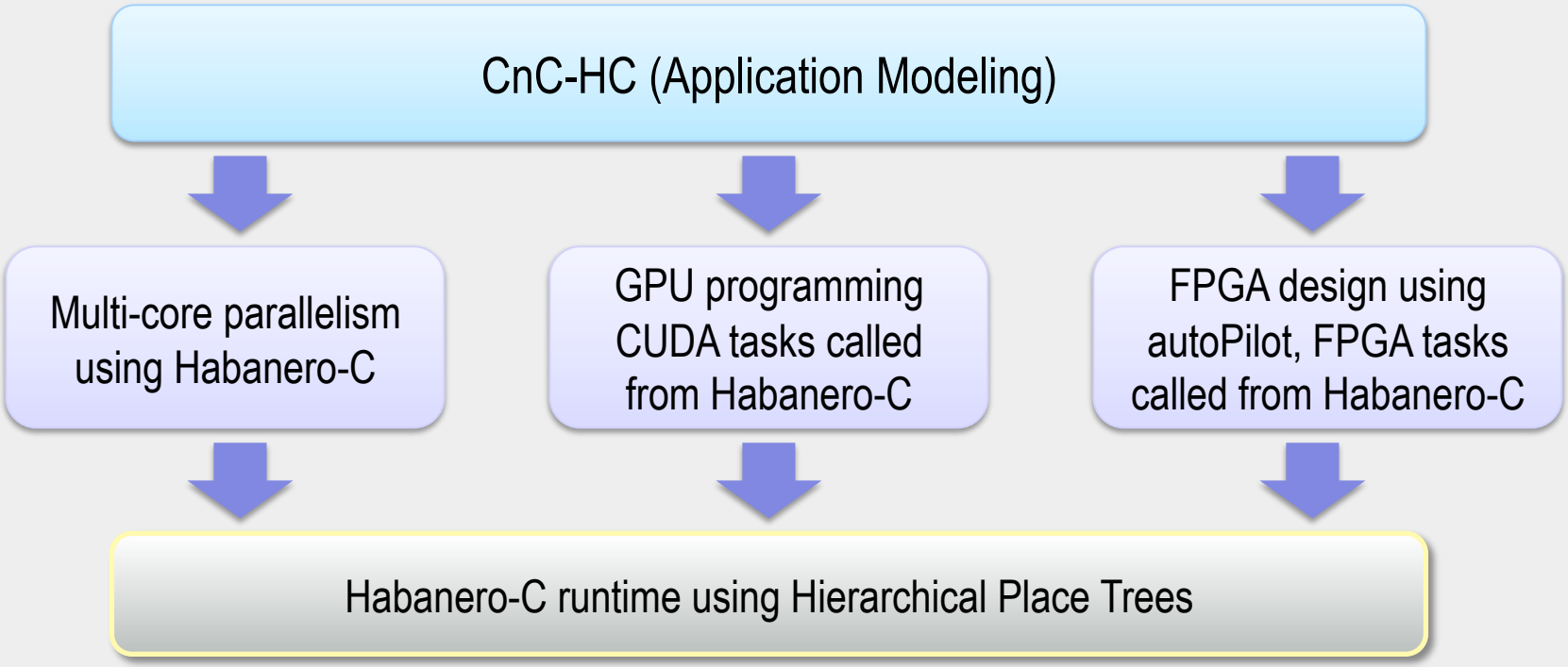
Invoke many times (customize)

Server-class CHP testbed: Convey HC-1 + GPU



Will upgrade to HC-1 ex (with virtex 6) and Fermi-based Tesla card

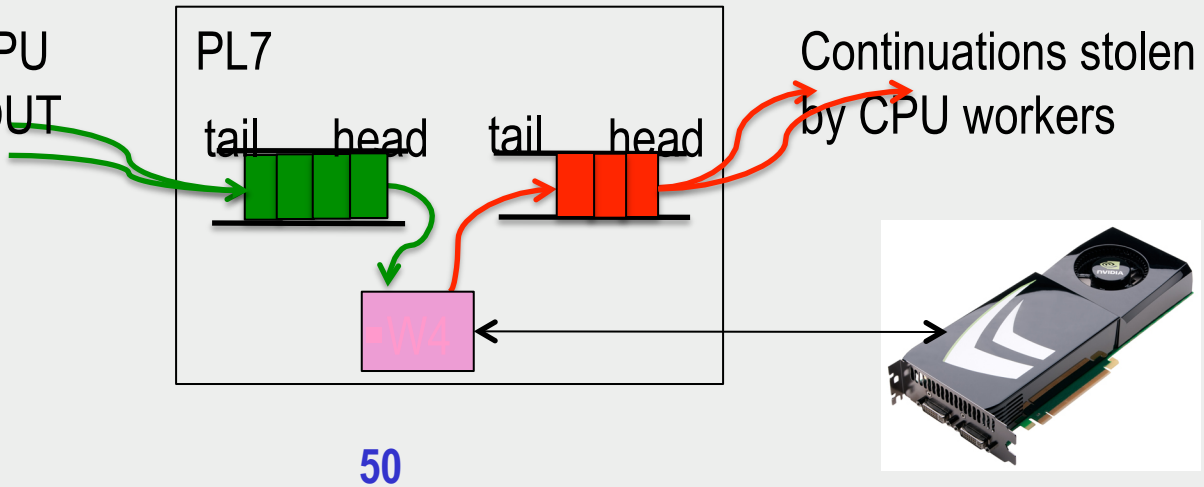
Toolchain for Server-class CHP testbed



Hybrid Scheduling in the Habanero-C runtime

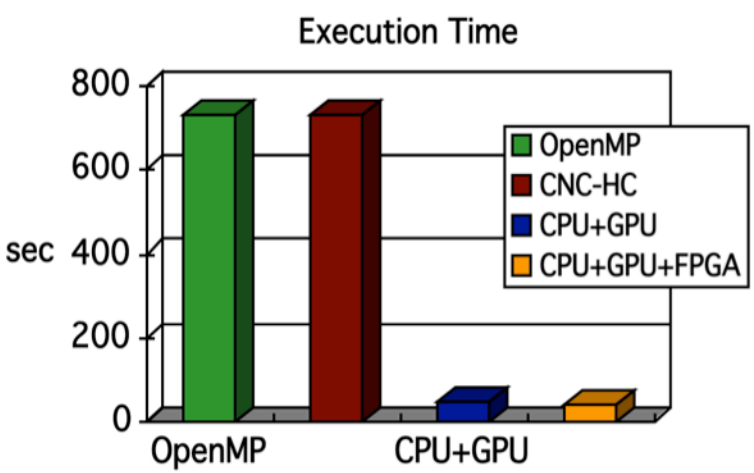
- Device place has two SC (semi-concurrent) queues: red and green
  - No locks – highly efficient
- Green queue maintains asynchronous device tasks (with IN/OUT)
  - Concurrent enqueuing device tasks by CPU workers from tail
  - Sequential dequeuing tasks by device agent workers from head
- Red queue maintains continuation of the finish scope of tasks
  - Sequential enqueuing continuation by agent workers
  - Concurrent dequeuing (steal) by CPU workers

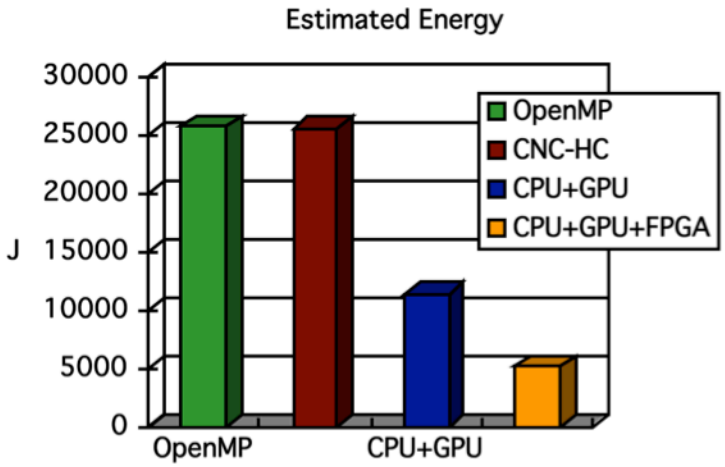
Device tasks created from CPU worker via async at(gpl) IN OUT { … }



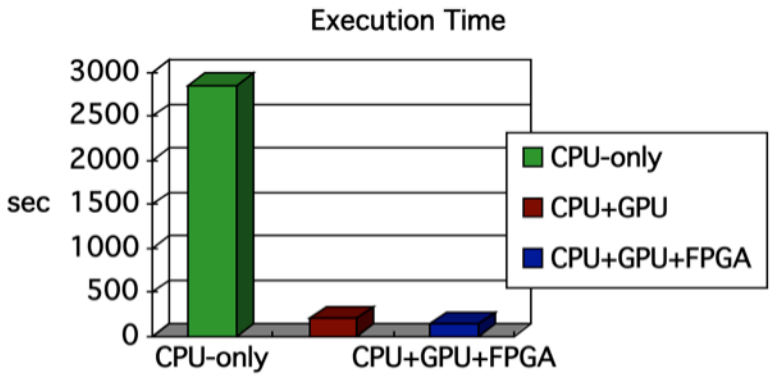
Experimental Results on Convey HC-1 testbed

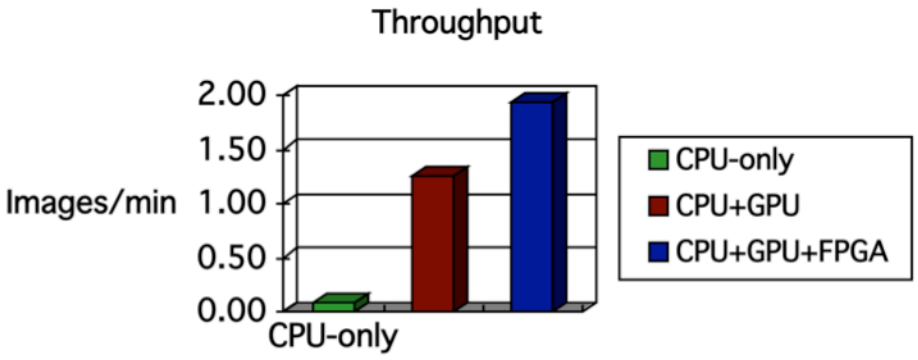
- **Pipeline: Denoise-->Registration (200 iterations)-->Segmentation (100 iterations)**





- **Multi-images (4 images)**





CPUs and GPUs have fundamentally different design philosophies

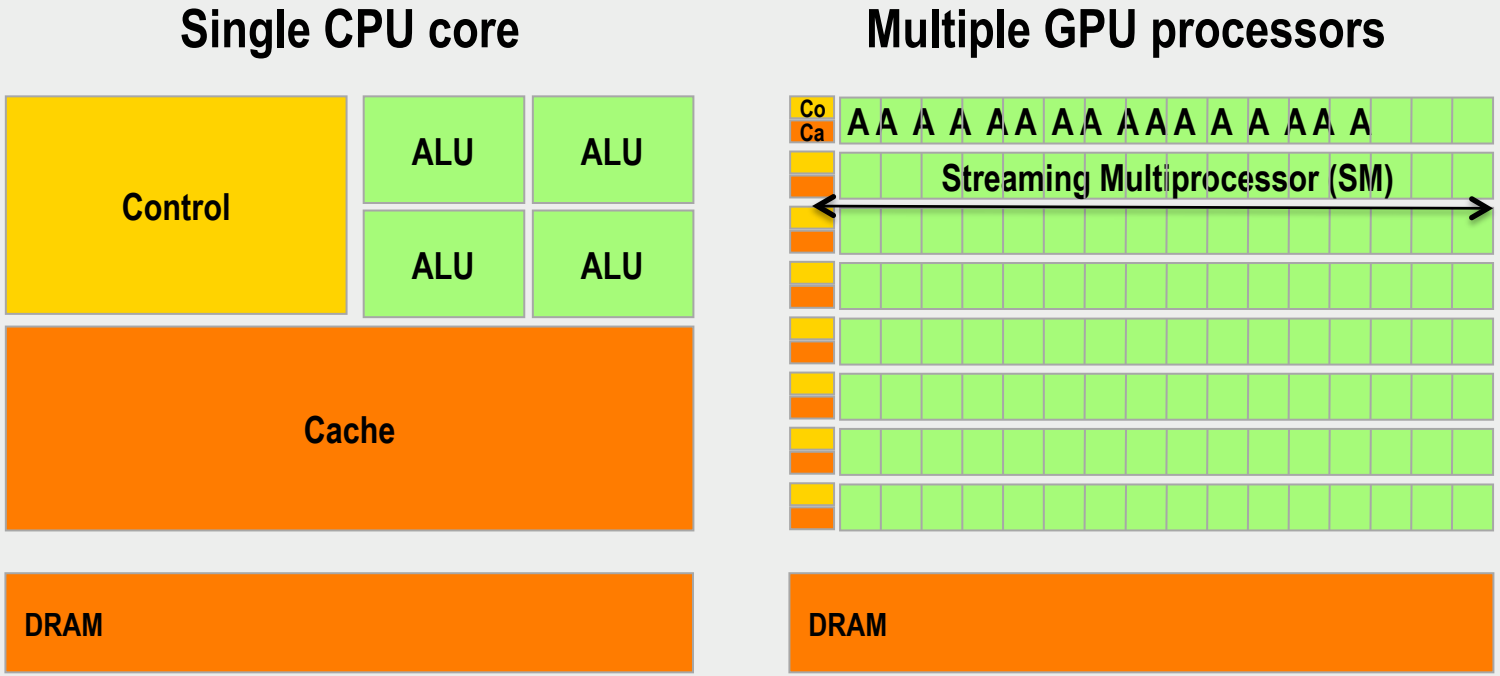
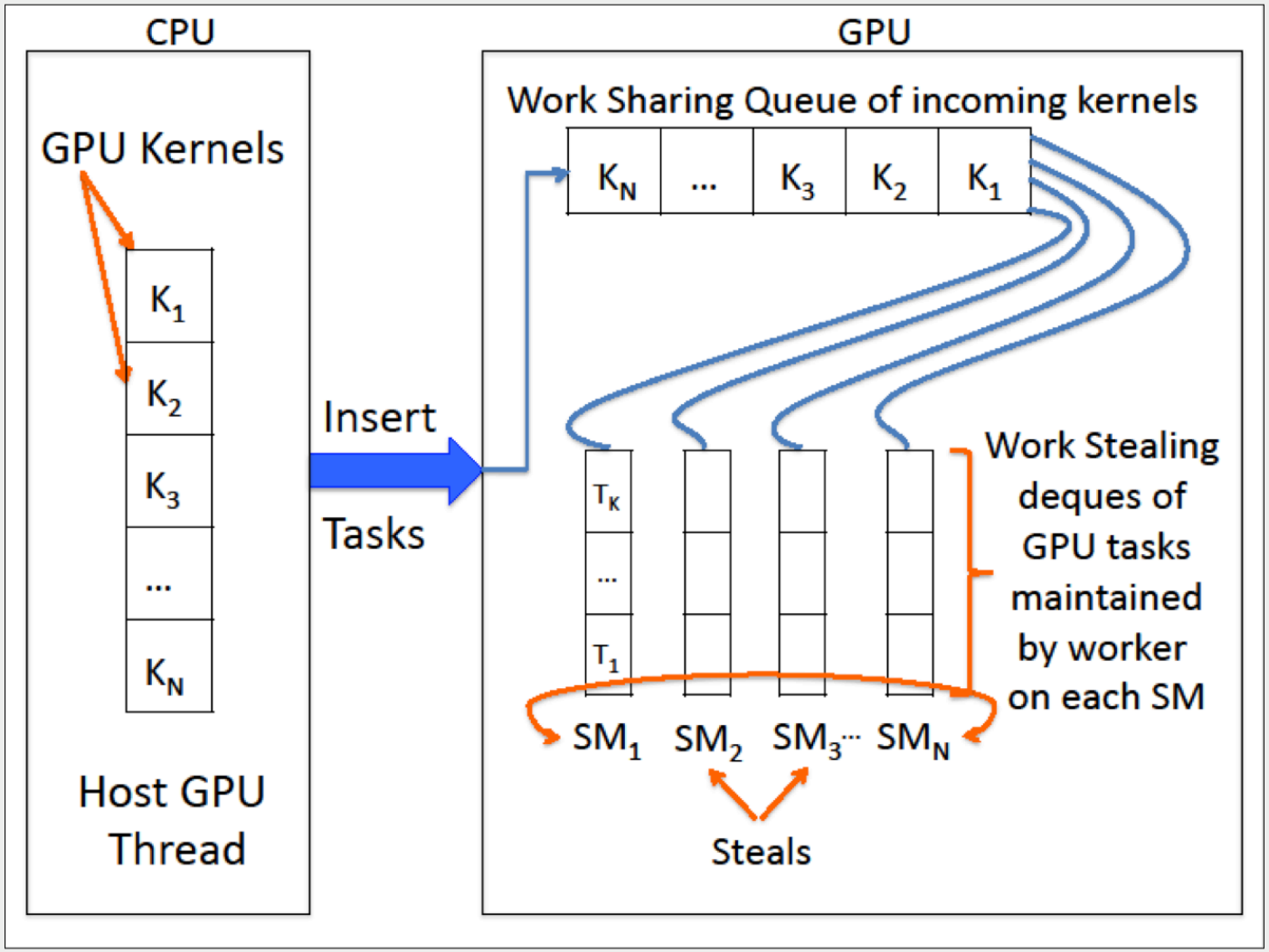


Figure source: David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

Motivation for Inter-SM Work-Stealing

- CUDA programming model launches a batch of data-parallel threads across a grid of SM's
- Can we do better with dynamic task parallelism?
- Our approach
  - Manage task execution across multiple streaming multiprocessors (SMs) in a GPU device by introducing a hybrid work-stealing/ work-sharing runtime system
  - Implement one deque per SM
    - No problem supporting divergence across SMs
  - Manage multiple CUDA devices for the user
  - Hide device memory allocation and communication from user

Inter-SM Work-Stealing



In this slide, kernel = thread block (not a standard CUDA grid)

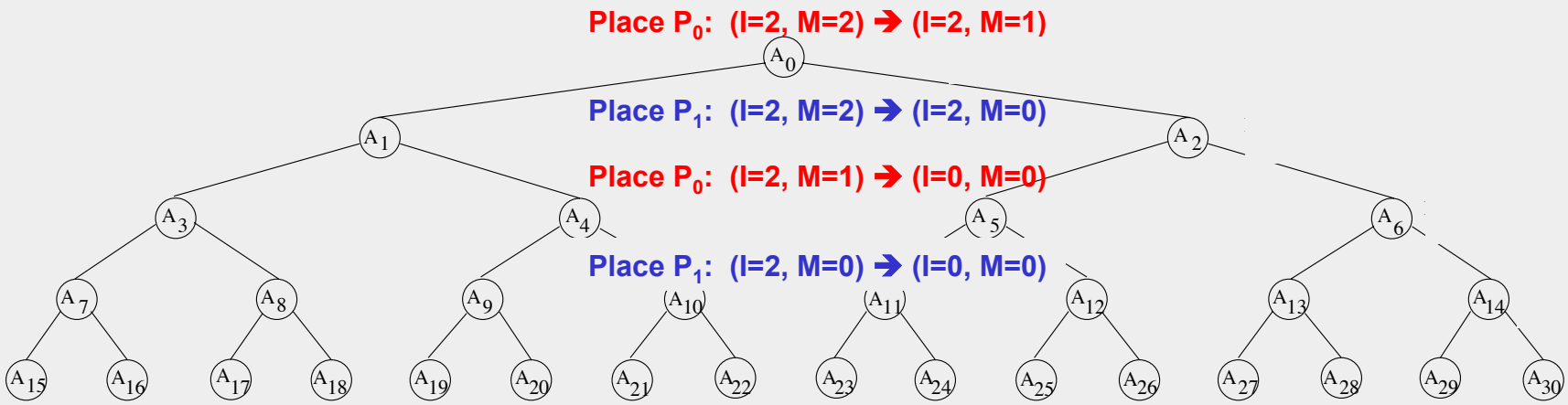
Deadlock-Free Scheduling of X10 Computations with Bounded Resources [SPAA 2007]

(S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. Shyamasundar, K. Yelick)

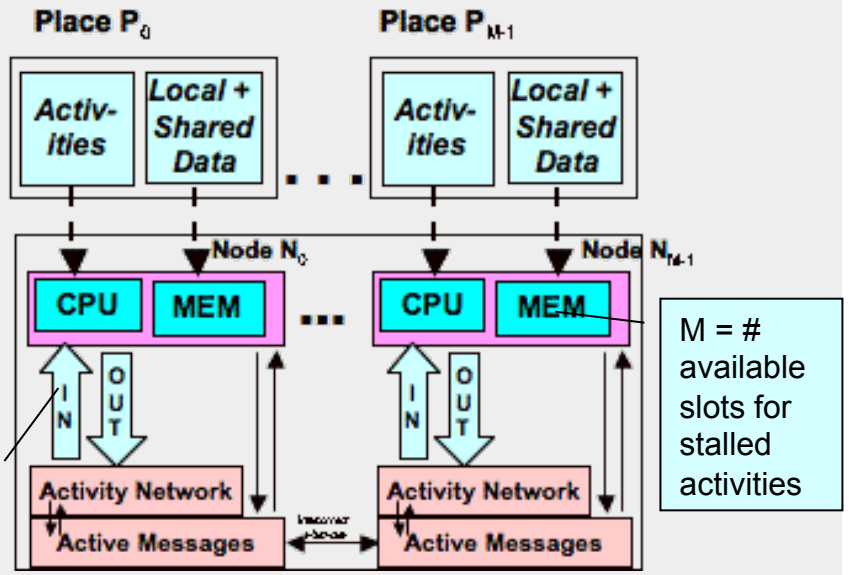
- Motivation: combine global address space, dynamic parallelism, and locality control in an Active Message system like GASNet while avoiding resource deadlocks

	Locality Control	Dynamic Parallelism	Global Address Space
MPI	**X**		
OpenMP, Cilk		**X**	
+ Shared Virtual Memory		**X**	**X**
HPF, UPC, Co-Array Fortran, Titanium	**X**		**X**
X10, Fortress, Chapel	**X**	**X**	**X**

Uniprocessor Cluster Deployment: Physical Deadlock Scenario




```
void foo (place p, ...) {
   if (...) {
      ... // recursion termination condition
   } else {
l1:   finish {
l2:      async (p.next()) { foo (here, ...); }
l3:      async (p.next()) { foo (here, ...); }
l4:   }
   }
   ... // Other computation
}
void main () {
l5:   finish async { foo (here, ...); };
```



I = # available slots in input activity buffer

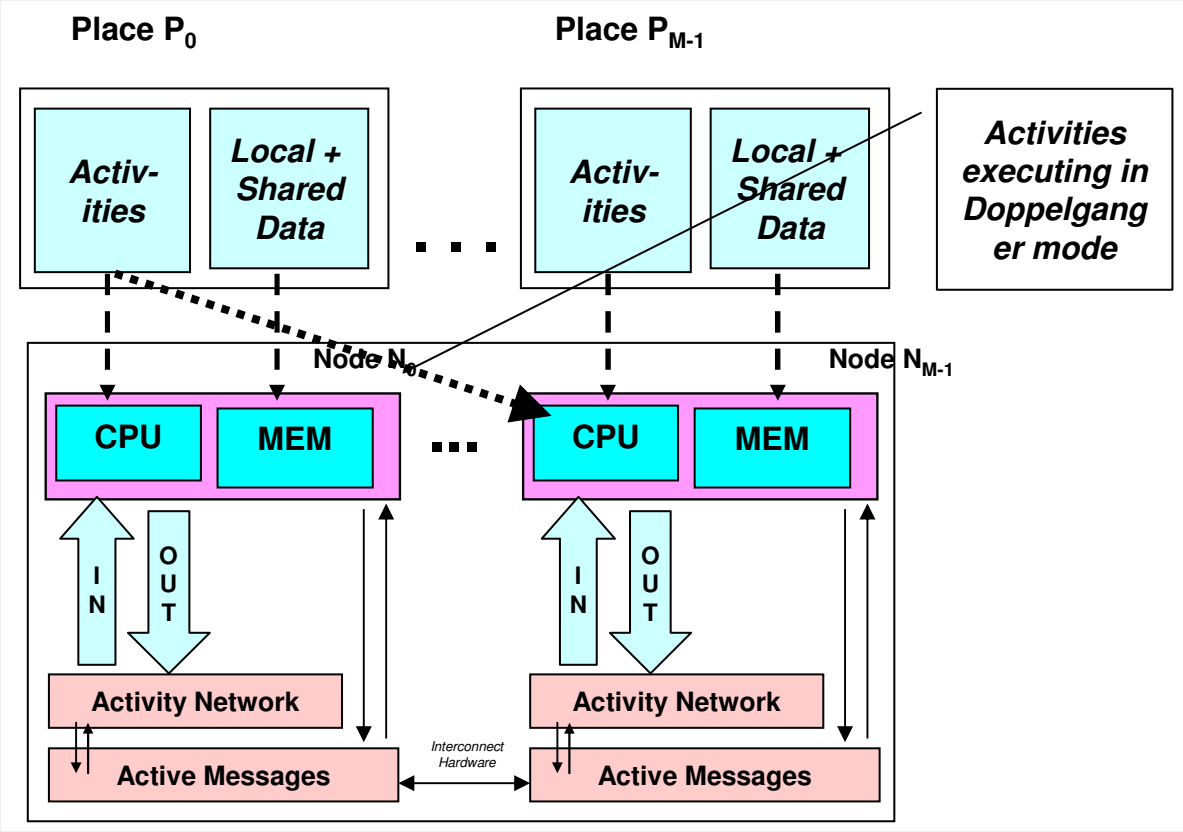
M = # available slots for stalled activities

RICE

Deadlock Avoidance in Active Messages

- Active Messages (e.g., AM-2, GASNet)
  - Supports unordered reliable delivery of request/reply messages
  - Reply is the only communication permitted in an AM request
  - Message injection is the only blocking operation permitted
- Virtual networks
  - Partition physical network into independent communication channels, so that resource starvation in one virtual network does not block progress in another
- AM(N) protocol for Active Messages uses N virtual networks V_0, … V_{N-1} in increasing priority
  - External messages must be injected in V_0
  - Response to message received on V_i must be injected in V_{i+1}
  - Handler for V_{N-1} cannot inject messages
  - AM(N) protocol can be implemented without deadlock on bounded resources

Doppelganger Mode (Figure 6)



- Allow an activity to run on a node different from its designated node, but keep data at designated node node
- Use Active Message network to support remote data accesses and remote execution of atomic blocks

Deadlock-Free Scheduling on Uniprocessor Cluster Deployments

- Each activity can be in one of three states:
  - *Active:* executing on a processor
  - *Enqueued:* maintain dequeue of activities on each processor
    - Parent of local async pushed at bottom of dequeue
    - Remote async is pushed on top of remote dequeue
      - If there are unsufficient resources then remote async is executed "locally" in Doppelganger mode
  - *Stalled:* Stalled Activity Buffer (SAB) holds activities waiting on a descendant activity pushed to a remote node
    - When a stalled activity is enabled it is inserted in "middle-insertion point" of dequeue
- Space usage per node is <= $2*R*S_{max} + R*S_1$ bytes, where
  - R = bound on in/out queue
  - S_{max} = size of largest activiation record (in bytes)
  - S_1 = space required by 1-processor execution schedule
- See Section 4 for details

The best laid plans …

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
3701 NORTH FAIRFAX DRIVE
ARLINGTON, VA 22203-1714

August 18, 2010

Dr. Vivek Sarkar
William Marsh Rice University
6100 Main Street, MS132
P. O. Box 1892
Houston, TX 77005-1892

Dear Dr. Sarkar:

Your proposal, "A Scalable, Hierarchical and Adaptive Runtime Environment for UHPC," submitted in response to the Defense Advanced Research Projects Agency (DARPA) Omnipresent High Performance Computing (OHPC) Broad Agency Announcement 10-78, was received by DARPA and assigned control number P-1070-113336.

Reviewed in accordance with the criteria set forth in that announcement, your proposal has been selected for a potential contract award. A Government agent will contact you in the near future to initiate the negotiation process. Should the negotiating parties not come to terms, DARPA is not required to make a contract award. This letter is not a notice of award nor an authorization to incur costs.

Thank you for your submission to this solicitation. Should you have any questions, please don't hesitate to contact me at DARPA-BAA-10-78@darpa.mil.

Sincerely,

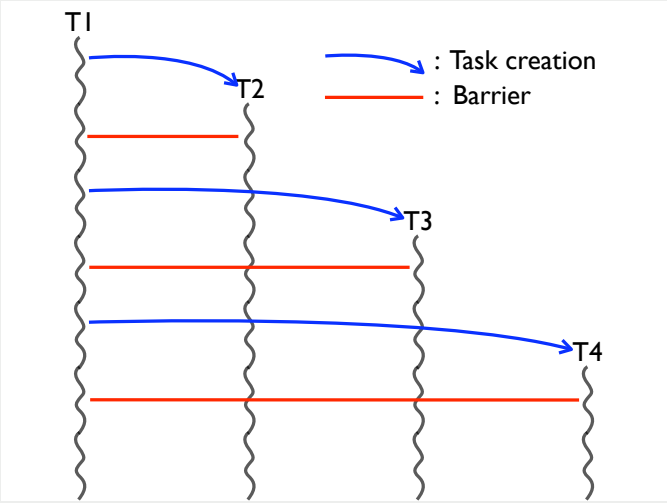
Dr. William Harrod
Program Manager
Information Innovation Office

cc:
Ms. Wendy Smith, DARPA IIO
Ms. Susan Shean, DARPA CMO

RICE

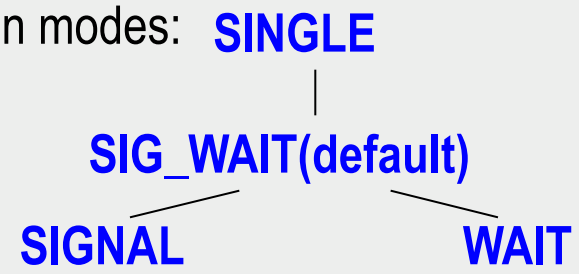
Phasers

- New synchronization construct **designed to integrate**
  - Asynchronous barriers
  - Asynchronous point-to-point synchronizations
  - Asynchronous collectives
  - Streaming computations
  - Dynamic parallelism
- **Support for "fuzzy barriers" and "single" statements**
- **Phase ordering property**
- **Deadlock freedom with "next" operations**
  - **Generalization of deadlock freedom property in X10 clocks**
- **Amenable to efficient hierarchical and implementations**
- **Subset of phaser functionality incorporated in Java 7**
- **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



Phaser Operations in Habanero Java

- **Phaser allocation**
  - **phaser ph = new phaser(mode);**
  - Phaser **ph** is allocated with **registration mode**
  - *Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)*
  - Registration modes:



Registration mode defines capability

There is a lattice ordering of capabilities

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

next / signal / wait operations

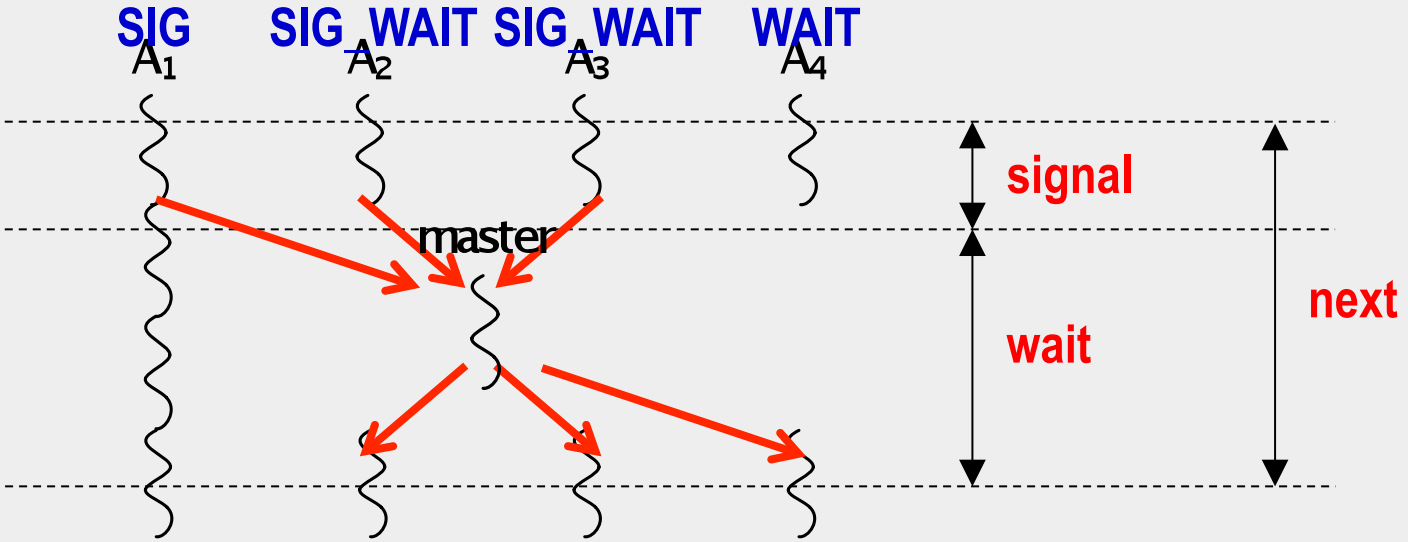
next = { **Notify "I reached next" = signal (or ph.signal())**
Wait for others to notify = wait

Semantics of next depends on registration mode

SIG_WAIT: **next = signal + wait**

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

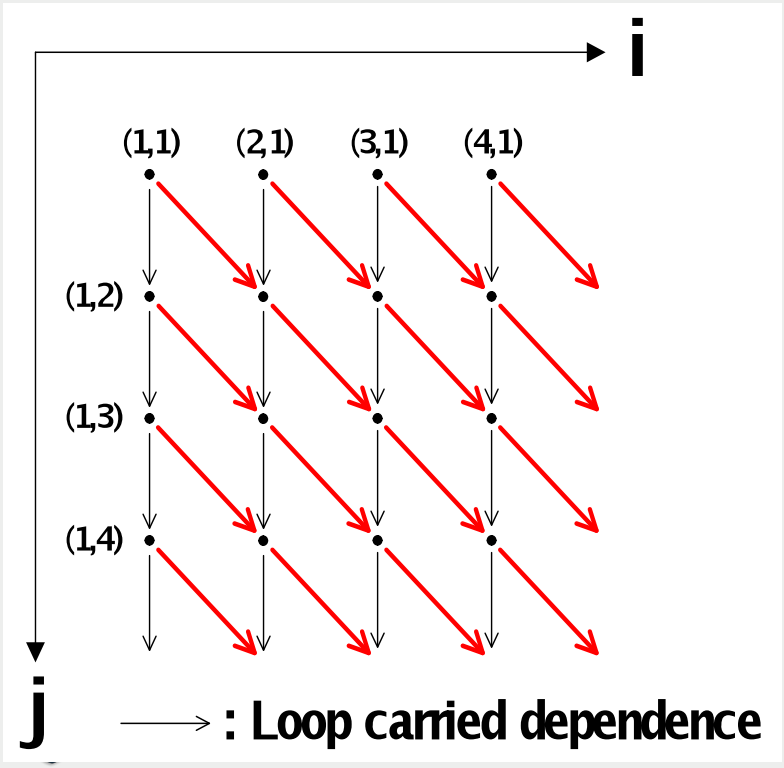
WAIT: **next = wait** (Don't disturb any task)

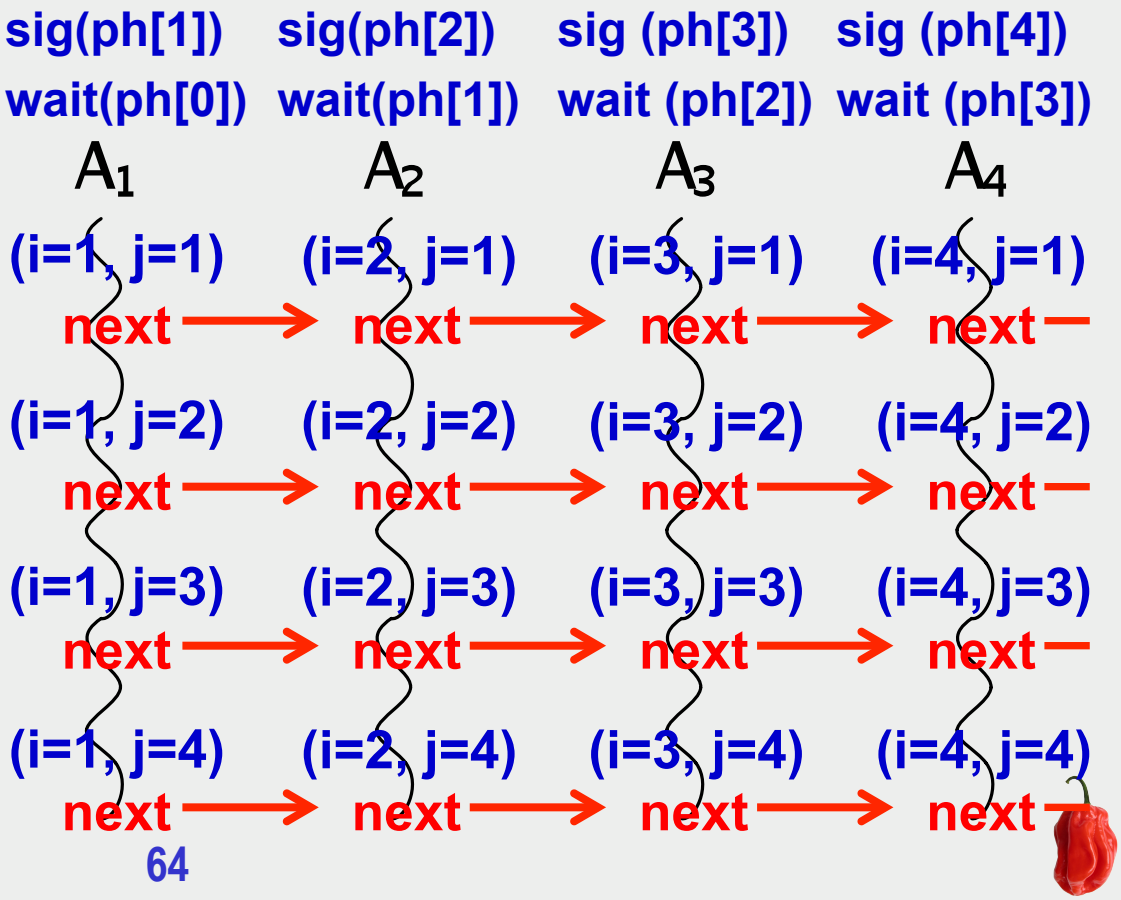


A master task receives all signals and broadcasts a barrier completion

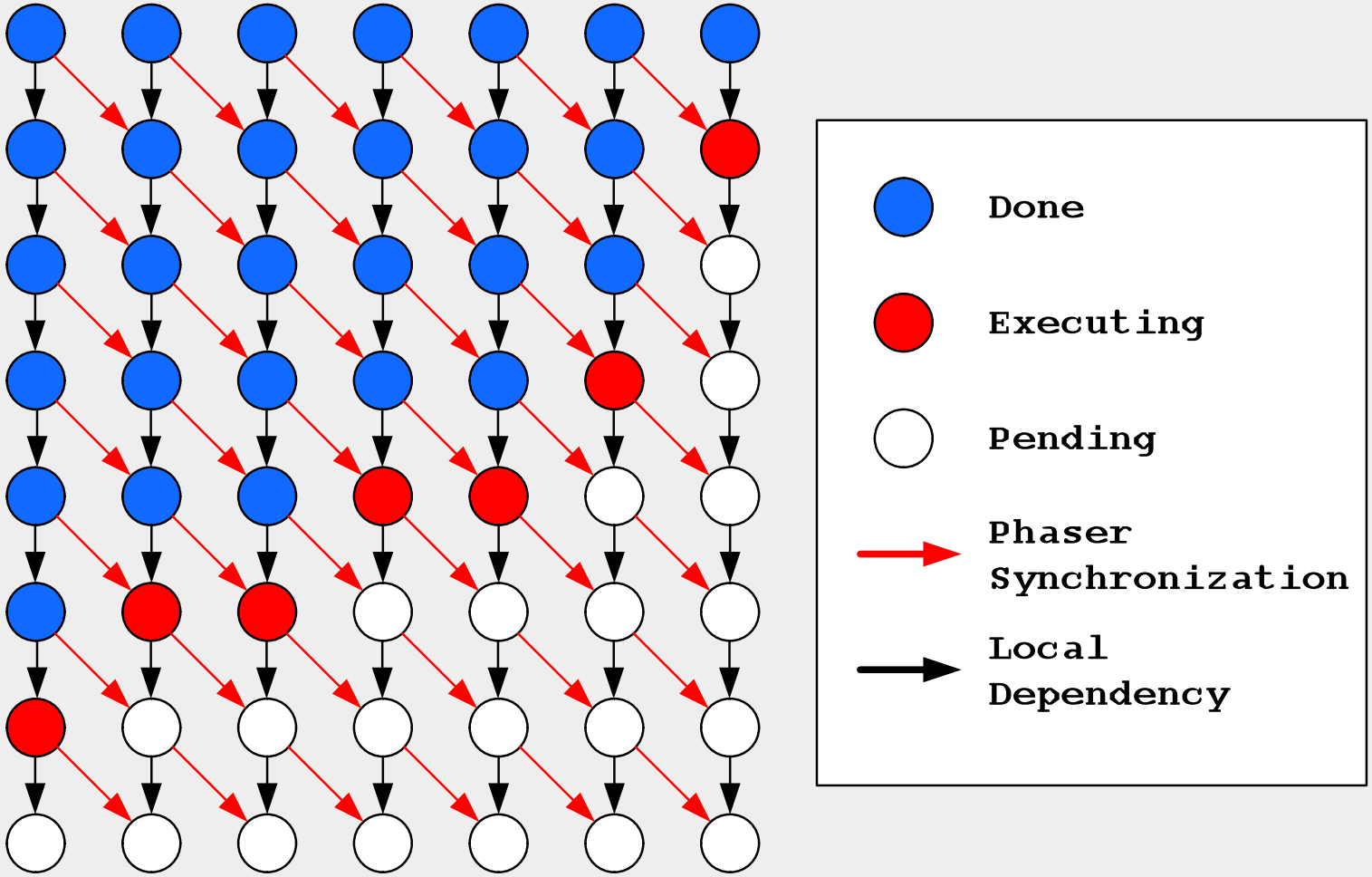
Example of Point-to-point Synchronization with Phaser Array

```
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
```

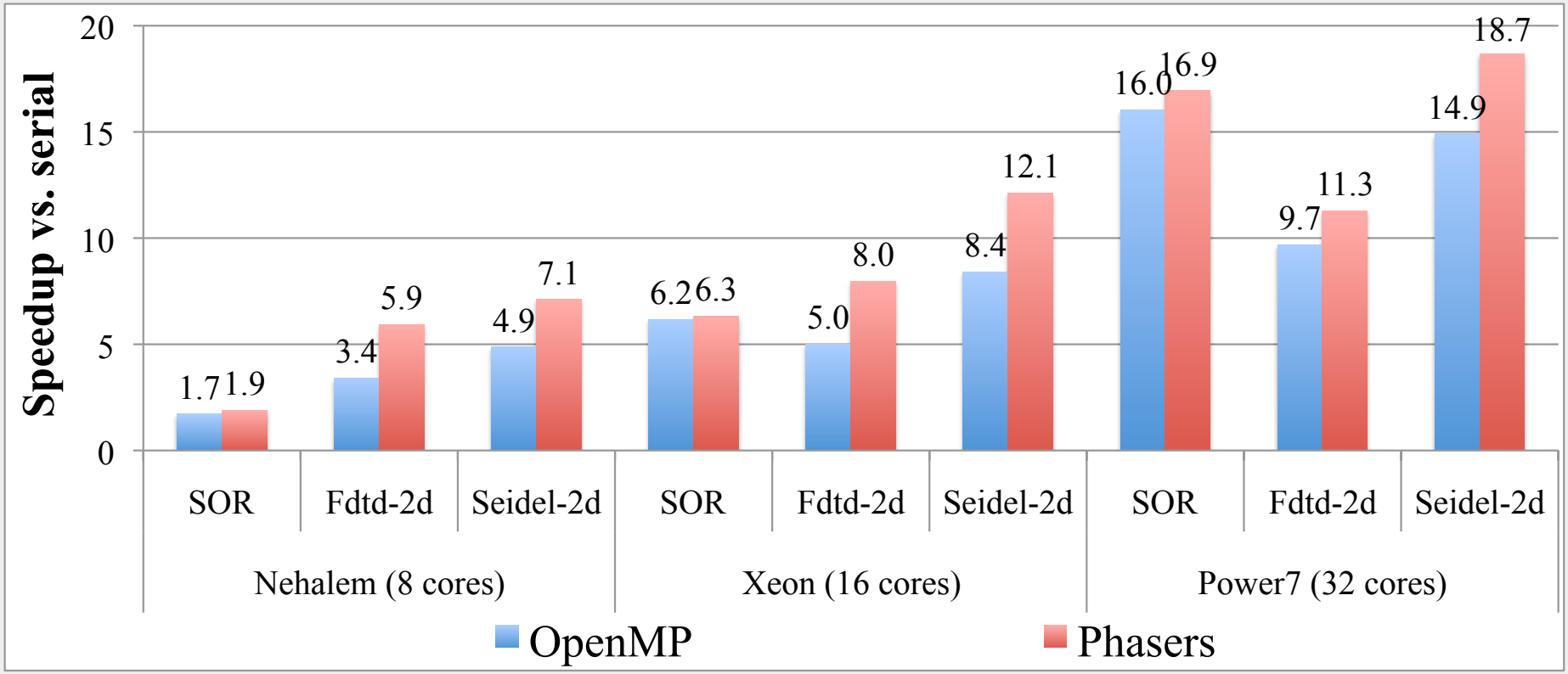




Example of Point-to-point Synchronization with Phaser Array (contd)



Preliminary Results with Phaser Point-to-Point Synchronization in OpenMP



"Unifying Barrier and Point-to-Point Synchronization in OpenMP with Phasers", J. Shirako, K. Sharma, V. Sarkar, IWOMP 2011, June 2011.

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();
    …
}
```

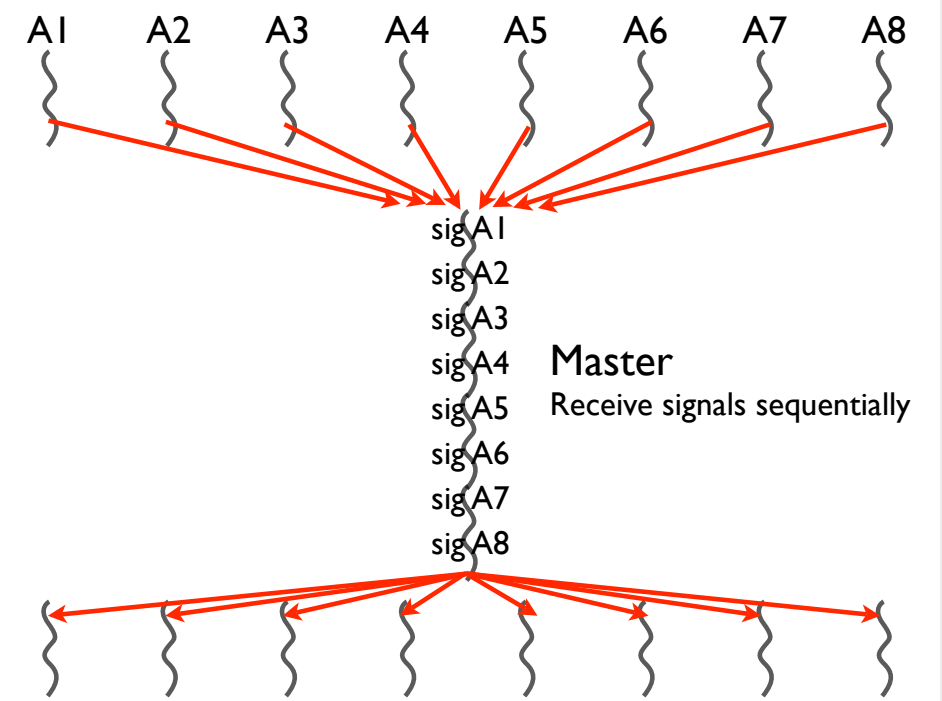
send: Send a value to accumulator

next: Barrier operation; advance the phase

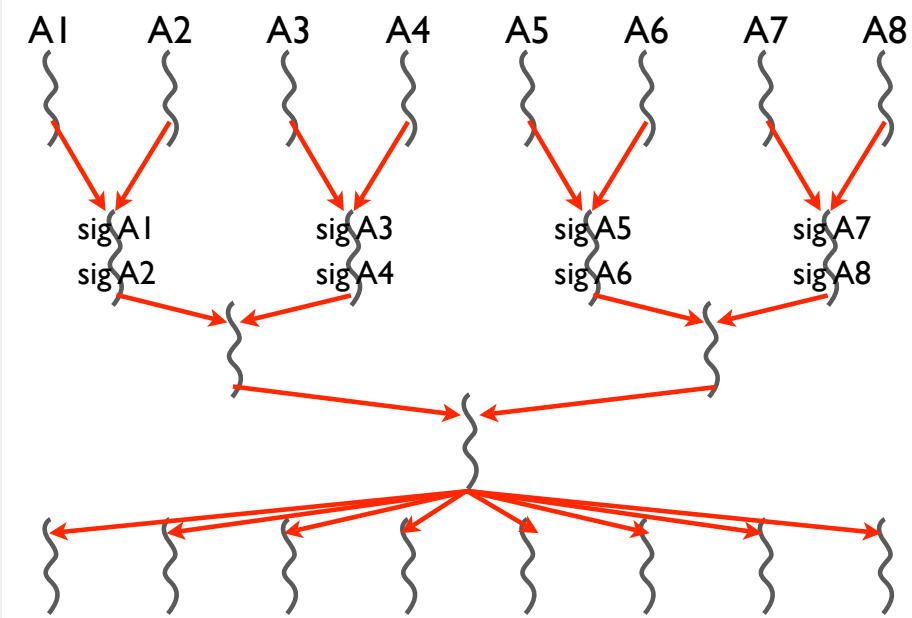
result: Get the result from previous phase (no race condition)

Flat vs. Hierarchical Implementations of Phasers and Accumulators

Single Level (Flat)



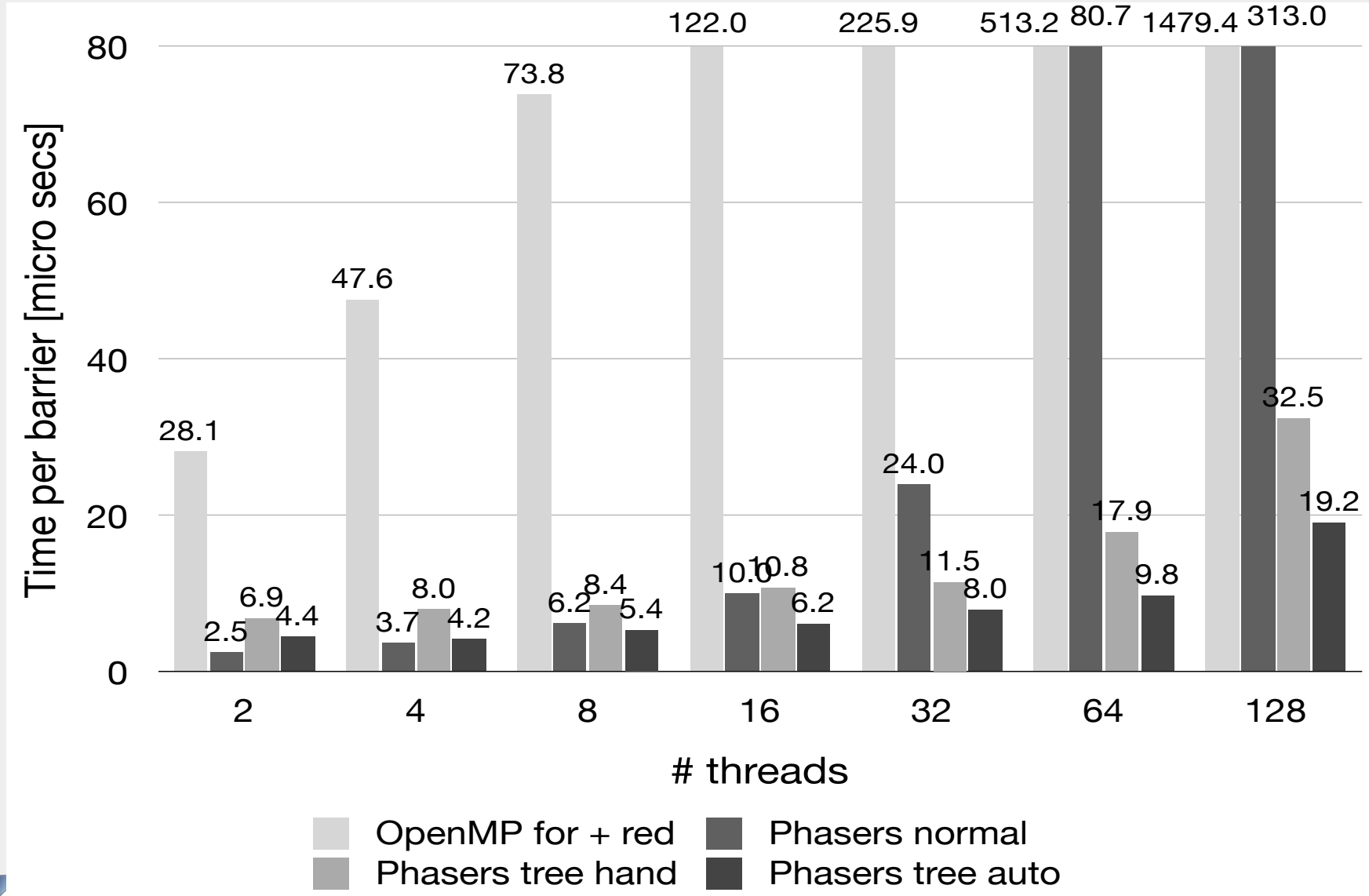
Hierarchical



Sub-masters in the same tier receive signals in parallel

"Hierarchical Phasers for Scalable Synchronization and Reduction", J.Shirako, V.Sarkar, IPDPS 2010

Barrier and Reduction Microbenchmark on 128-thread Niagara 2



Role of Compilers for Extreme Scale Systems

- Compilers have reliably contributed to programmer productivity for 50+ years
- Compiler foundations need to be revisited for Extreme Scale programs with explicit parallelism and locality
- Compiler research under way in Habanero project
  - Communication Optimizations for Distributed-Memory X10 Programs
  - Reducing Task Creation and Termination Overhead
  - Chunking Parallel Loops in the Presence of Synchronization
  - . . .
- Details can be found in PLDI 2011 tutorial and related papers

Conclusions

- Extreme Scale systems projected for 2015 – 2020 will need fundamental changes in Execution Model and System Software 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
- Urgent need for execution models that can span multiple scales of parallelism and heterogeneity – multicore, accelerators, multi-node, HPC cluster, data center cluster
- Well-designed runtime primitives can enable synergistic innovation in languages and compilers
- This talk presented early experiences in the Habanero project, and key primitives that could be useful in a unified execution model across heterogeneous levels of parallelism

System Software Under Development in Habanero Group

- "Prototype" implementations for research, teaching, and evaluation
  - Habanero-Java (HJ) compiler, runtime, and datarace detection tool
  - Habanero-C (HC) compiler and runtime
  - Concurrent Collections (CnC)
    - CnC-HJ --- Java-based runtime supports integration of multiple languages for application prototyping (Java, C, C++, Matlab, Python, …)
    - CnC-HC --- Unified C-based runtime supports execution across heterogeneous processors
  - Pace compiler --- includes integration of Rose and LLVM
  - Extensions to X10 compiler --- communication optimizations for the APGAS runtime
- "Research" implementations
  - CnC-Hadoop --- implementing the CnC model on Hadoop clusters
  - Addition of phasers to Microsoft Concurrent Runtime (ConcRT)
  - Habanero-Scala (HS) compiler and runtime
  - . . .

Acknowledgments: Habanero Team

- Faculty
  - Vivek Sarkar
- Senior Research Scientist
  - Michael Burke
- Research Scientists
  - Zoran Budimlić, Philippe Charles, Jun Shirako, Jisheng Zhao
- Research Programmer
  - Vincent Cavé
- Postdoctoral Researcher
  - Edwin Westbrook
- PhD Students
  - Sanjay Chatterjee, Shams Imam, Deepak Majeti, Raghavan Raman, Dragoș Sbîrlea, Kamal Sharma, Alina Sbîrlea, Sağnak Taşırlar
- Undergraduate Students
  - Max Grossman, Jungwoo Lee, Jarred Payne
- Other collaborators at Rice
  - Rich Baraniuk, Corky Cartwright, Keith Cooper, Tim Harvey, John Mellor-Crummey, David Peixotto, Bill Scherer, Linda Torczon, Lin Zhong, …

RICE