An Introduction to Parallel Programming with Habanero-Java
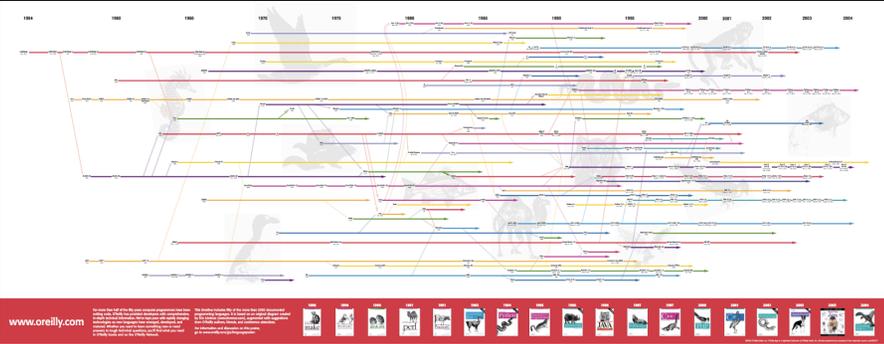
Vivek Sarkar

Dept of Computer Science
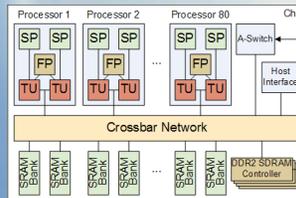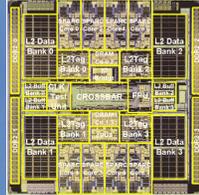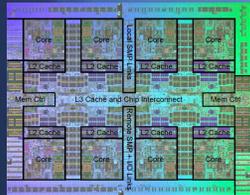
Rice University

vsarkar@rice.edu

October 26, 2011
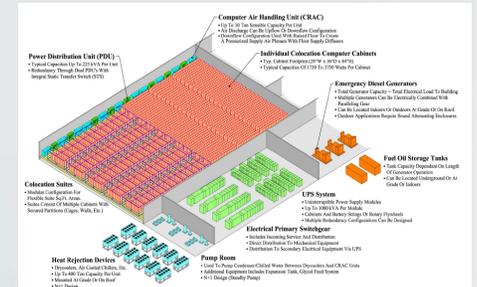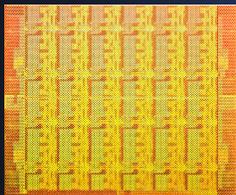
High-Level Declarative Programming

Your Favorite DSL

Orc

NESL

CnC

ArBB

Haskell

Question: How to bridge this gap???

Answer: With Mid-Level Parallel Programming Models & Runtime Systems!

Low-Level Parallel Programming

OpenCL

Java Concurrency

OpenMP

CUDA

pthreads, mutexes, …

MPI

Parallel Software Challenge & Expertise Gap

Domain Experts

Domain Experts need high level parallelism-oblivious Programming Models

Expertise gap between domain experts and concurrency experts

Concurrency Experts

Today's low-level Parallel Programming Models are only accessible to a small number of concurrency experts

RICE

CS Majors to the Rescue

Domain Experts

Software Engineers

Concurrency Experts

Domain Experts need high level parallelism-oblivious Programming Models

Concurrent Collections (CnC)

Focus of Rice Habanero Project

Software engineers need simple and portable Parallel Programming Models

Habanero Execution Model in Java, C, Scala (HJ, HC, HS)

Concurrency Experts need low-level Parallel Programming Models

RICE

4

Habanero-Java (http://habanero.rice.edu/hj)

- New pedagogic language and implementation developed at Rice since 2007
  - Derived from Java-based version of X10 language (v1.5) in 2007
    - X10 language has evolved significantly since then
  - **Habanero-Java (HJ)** is currently an extension of Java 1.4
    - All Java 5 & 6 libraries and classes can be called from HJ programs
   - Front-end support for Java 5 constructs (notably, generics) in progress
    - HJ compiler generates Java classfiles that execute with HJ runtime on a standard JRE
    - Download available at https://wiki.rice.edu/confluence/display/PARPROG/HJDownload
- HJ's parallel extensions are focused on *mid-level* task parallelism
  1. Dynamic task creation & termination: future, async, finish, force, forall, foreach
  2. Mutual exclusion and isolation: isolated
  3. Collective and point-to-point synchronization: phaser, next
  4. Locality control --- task and data distributions: places, here

- **Habanero-C** and **Habanero-Scala** are under development with similar constructs

- Reference: "Habanero-Java: the New Adventures of Old X10". PPPJ 2011, August 2011.

HJ Compilation and Execution Environment

Foo.hj

↓
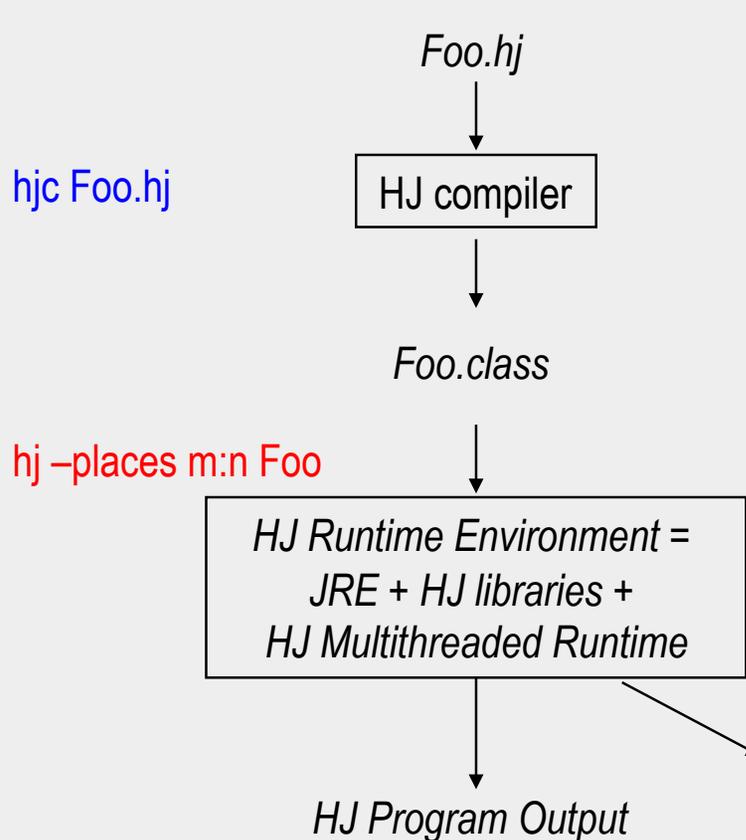
HJ source program --- must contain a class named Foo with a public static void main(String[] args) method

hjc Foo.hj

HJ compiler

↓

HJ compiler translates Foo.hj to Foo.class, and inserts calls to HJ runtime as needed

Foo.class

↓

hj –places m:n Foo

HJ Runtime Environment = JRE + HJ libraries + HJ Multithreaded Runtime

HJ runtime allocates m*n worker threads across m "places" (default values: m = 1, n = 3)

↓

HJ Program Output

HJ Abstract Performance Metrics
(optional, enabled by –perf=true option for hj command)

COMP 322: Fundamentals of Parallel Programming

- **Sophomore-level CS Course at Rice**
  - https://wiki.rice.edu/confluence/display/PARPROG/COMP322
 - Or do a web search on "comp322 wiki"
- **Approach**
  - **Mid-level parallel programming --- "Simple things should be simple, complex things should be possible"**
  - 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 & data structures including lists, trees, graphs, matrices
    - Common parallel programming patterns
  - Use Habanero-Java (HJ) as pedagogical language for two-thirds of course, and then teach standard programming models (Java threads, MPI, CUDA) using HJ principles

Algorithmic Complexity Measures

T_P = execution time on P processors
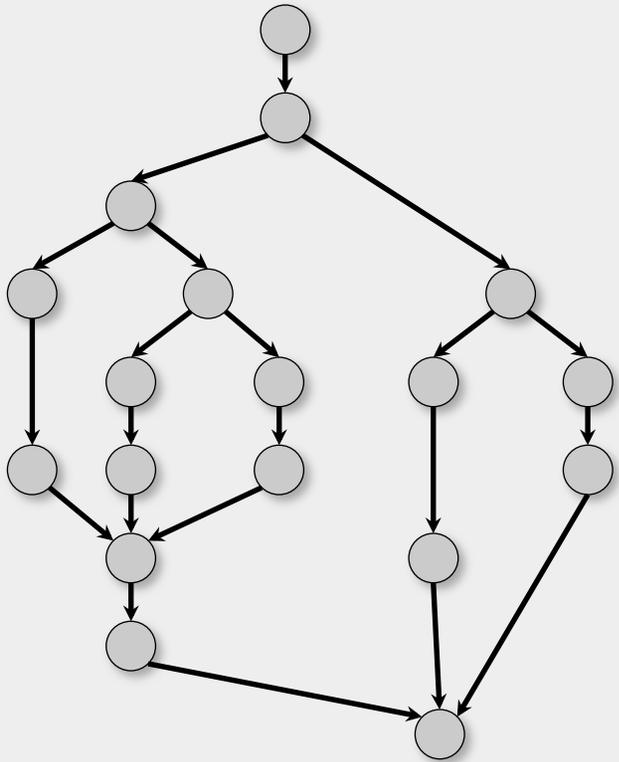
Computation graph abstraction:

Node = arbitrary sequential computation

Edge = dependence (successor node can only execute after predecessor node has completed)

Directed acyclic graph (dag)

Processor abstraction:

P identical processors

Each processor executes one node at a time

PROC$_0$ . . . PROC$_{P-1}$

Algorithmic Complexity Measures

T_P = execution time on P processors
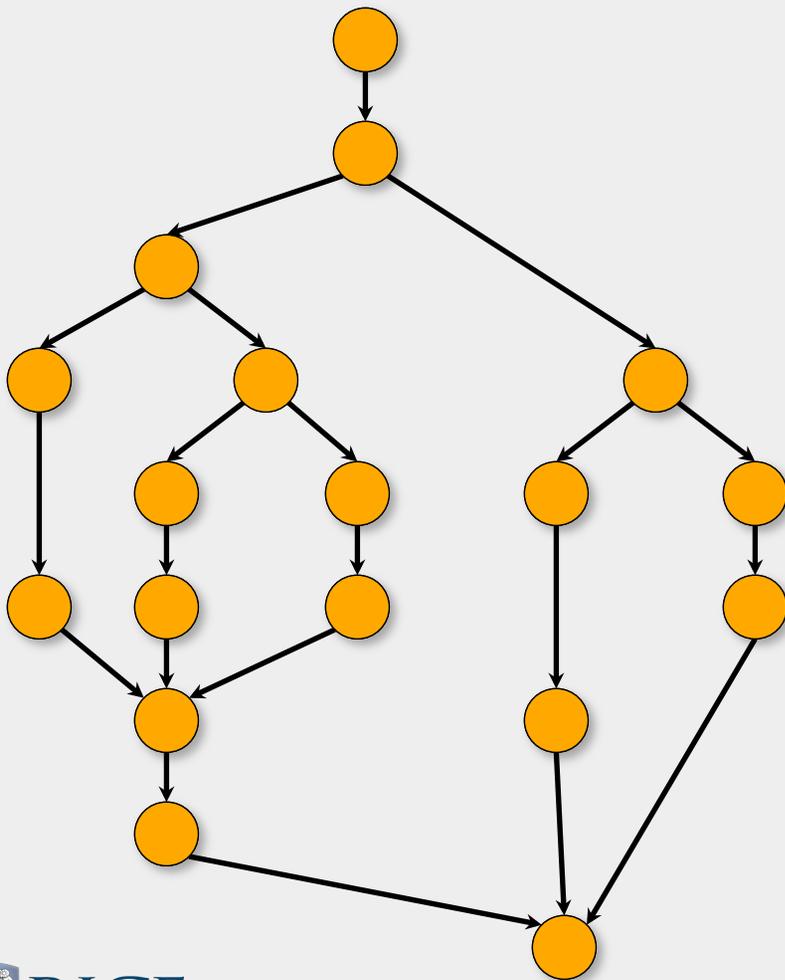
$T_1 = work$

Algorithmic Complexity Measures

$$T_P = \text{execution time on } P \text{ processors}$$
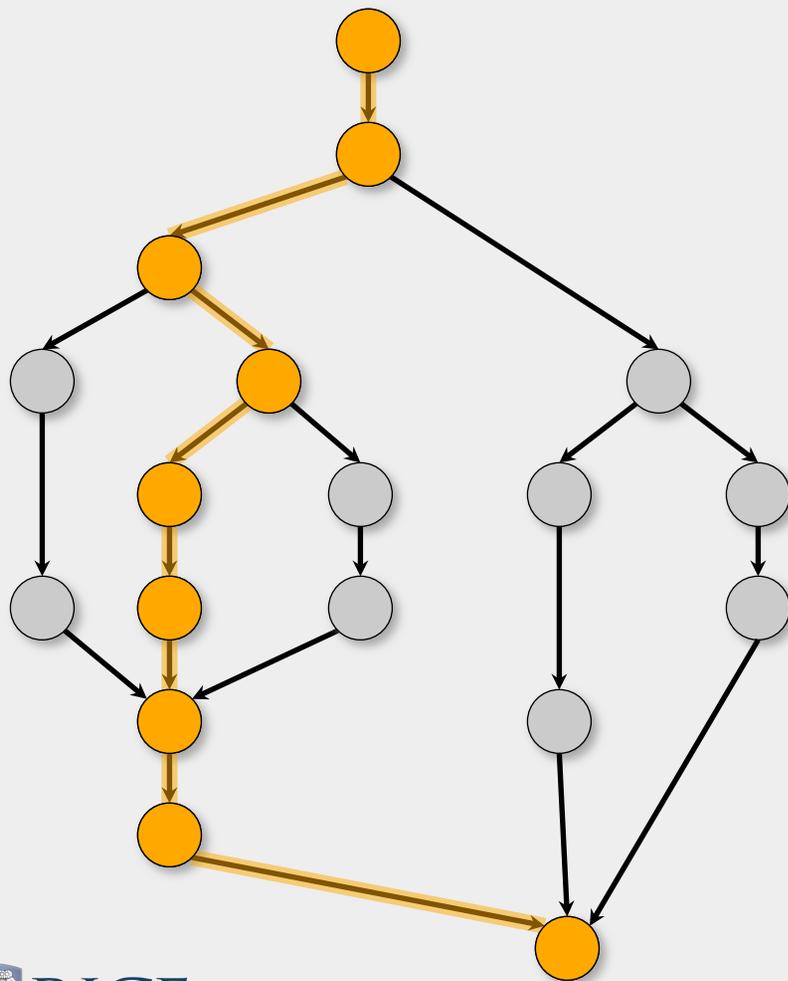
$$T_1 = work$$
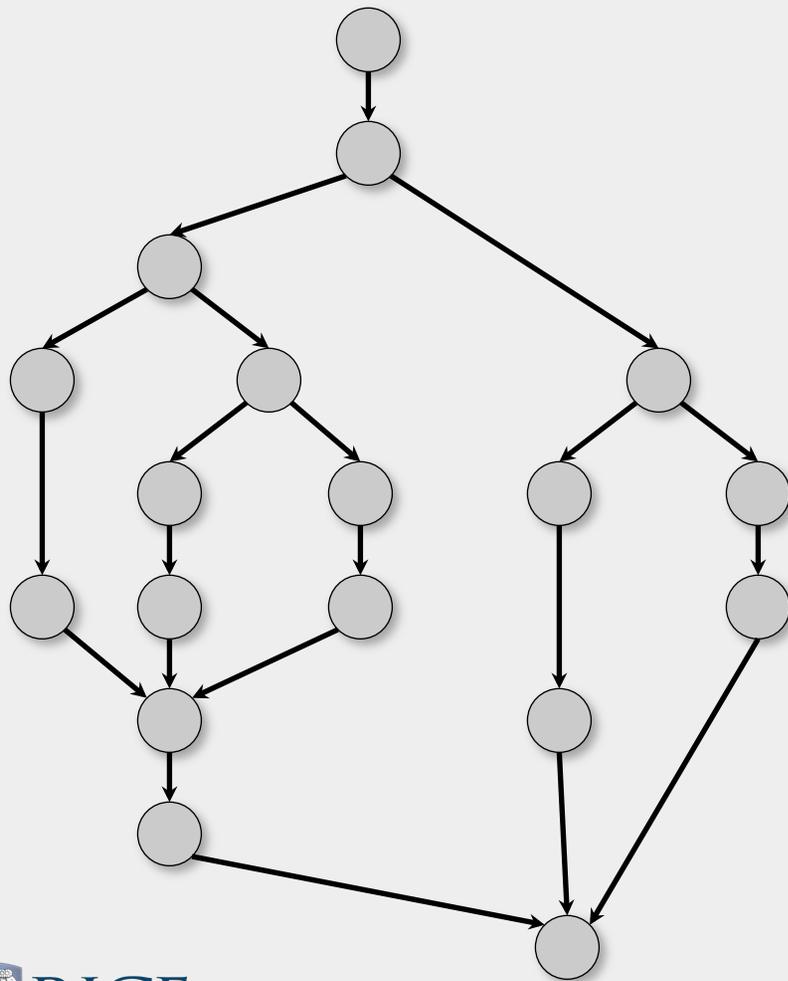
$$T_\infty = span^*$$

*Also called *critical-path length* or *computational depth*.

Algorithmic Complexity Measures

T_P = execution time on P processors

T_1 = *work*

T_∞ = *span*

LOWER BOUNDS

$T_P \geq T_1/P$

$T_P \geq T_\infty$

T_1/T_P = *speedup* on P processors

RICE

Parallelism ("Ideal Speedup")

T_P depends on the schedule of computation graph nodes on the processors

➔ Two different schedules can yield different values of T_P for the same P

For convenience, define *parallelism* (or ideal speedup) as the ratio T_1/T_∞

Parallelism is independent of P, and only depends on the computation graph

Also define *parallel slackness* as the ratio, $(T_1/T_\infty)/P$

Example 1: Array Sum (sequential version)

- Problem: compute the sum of the elements X[0] ... X[n-1] of array X

- Sequential algorithm

  - sum = 0; for (i=0 ; i< n ; i++) sum += X[i];

- Computation graph
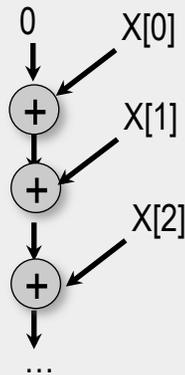
Example 1: Array Sum (parallel iterative version)

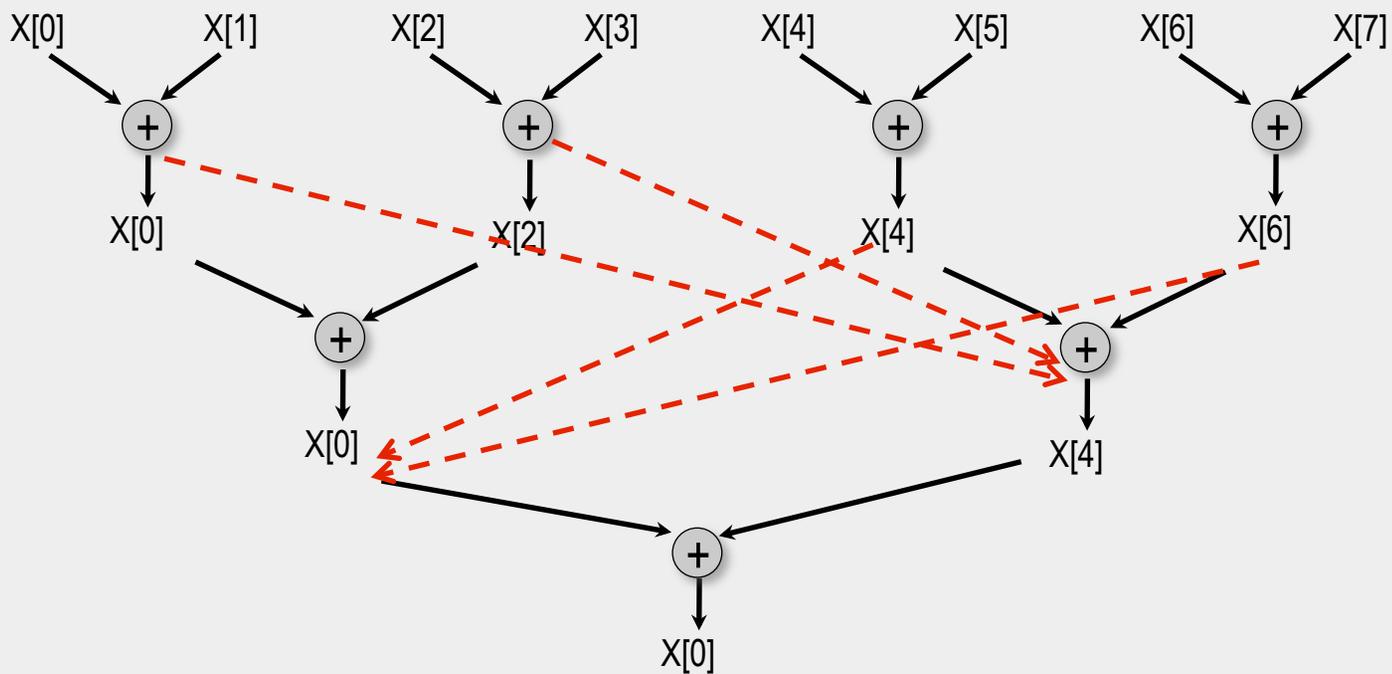- Parallel algorithm (iterative version, assumes n is a power of 2)

  ```
  for ( step = 1; step < n ; step *= 2 ) { // iterates for lg n steps
      size = n / (2*step);    // number of adds to be performed in current step
      forall ( point [i] : [0:size-1] ) X[2*i*step] += X[(2*i+1)*step];
  }
  sum = X[0];
  ```

- HJ forall construct executes all iterations in parallel
  - forall body can read (but not write) outer local variables (copied on entry)

- This algorithm overwrites X (make a copy if X is needed later)

- Work = O(n), Span = O(log n), Parallelism = O(n / (log n))

- NOTE: this and the next parallel algorithm can be used for any associative operation on array elements (need not be commutative) e.g., multiplication of an array of matrices

Example 1: Array Sum
(parallel iterative version)

Computation graph for n = 8

Extra orderings due to forall construct

RICE

15

Example 1: Array Sum (parallel recursive version)

- Parallel algorithm (recursive version, assumes n is a power of 2)

```
sum = computeSum(X, 0, n-1);
int computeSum(final int[] X, final int lo, final int hi) {
    if ( lo > hi ) return 0;
    else if ( lo == hi ) return X[lo];
    else {
        int mid = (lo+hi)/2;
        final future<int> sum1 = async<int> { return computeSum(X, lo, mid); }
        final future<int> sum2 = async<int> { return computeSum(X, mid+1, hi); }
        return sum1.get() + sum2.get();
    }
} // computeSum
```
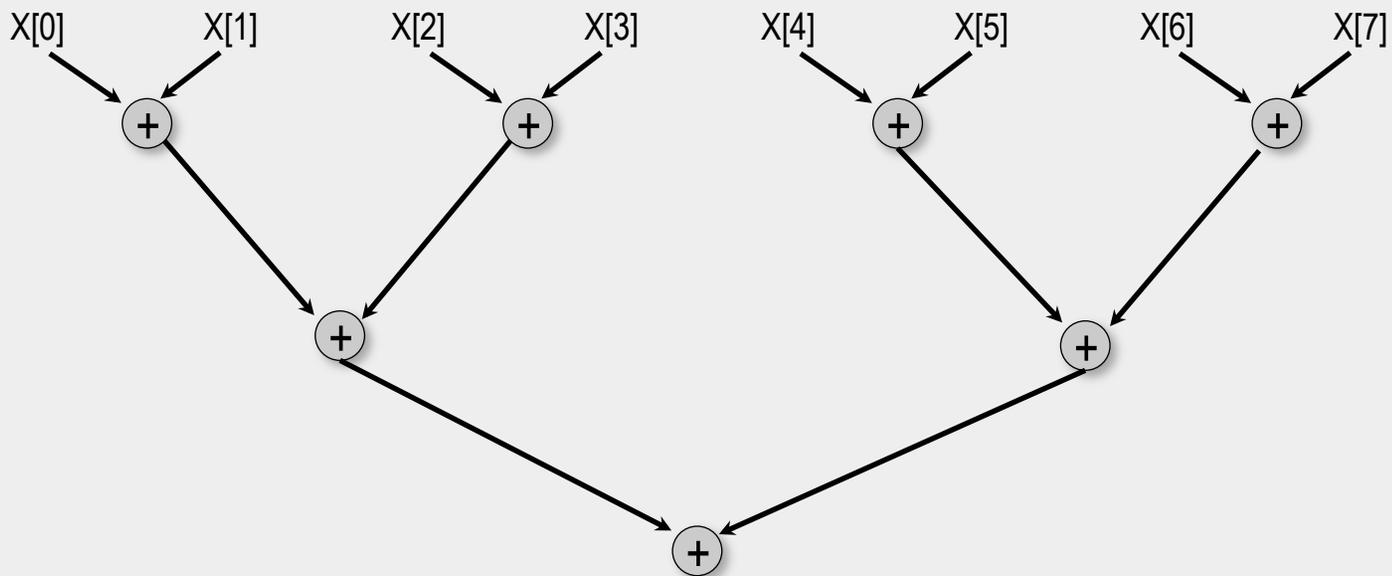
- "future" executes child expression in parallel with parent

  - get() causes the parent to wait for the child.

Example 1: Array Sum
(parallel recursive version)

Computation graph for n = 8

X[0] X[1] X[2] X[3] X[4] X[5] X[6] X[7]

Work = O(n), Span = O(log n), Parallelism = O(n / (log n))

No extra orderings as in forall case

Four classes of mid-Level Parallel Constructs

1) <u>Asynchronous tasks and data transfers</u> e.g.,

- MPI: *mpi_isend, mpi_irecv, mpi_wait*

- OpenMP: *task, taskwait*

- Cilk: *spawn, sync*

- CAF, UPC, Chapel: *function shipping*

- X10: *async, finish*

- **Habanero: *async, finish, asyncMemcpy, futures, foreach, forall, async-await***

2) <u>Collective and point-to-point synchronization & reductions</u> e.g.,

- MPI: *mpi_send, mpi_recv, mpi_barrier, mpi_reduce,*

- OpenMP: *barrier, reductions*

- Cilk: *reducers*

- CAF, UPC, Chapel: *barrier, reductions*

- X10: *clocks, finish accumulators, conditional atomic*

- **Habanero: *phasers, phaser accumulators, finish accumulators***

Four classes of mid-Level Parallel Constructs

3) <u>Mutual exclusion</u> e.g.,

- OpenMP: *atomic, critical*

- X10, Chapel, STM systems: *atomic*

- Galois: operations on unordered sets

- **Habanero:** *isolated (weak atomicity)*

4) <u>Locality control for task and data distribution</u> e.g.,

- MPI: *all-local (shared-nothing)*

- CAF, Chapel, UPC, X10: *PGAS storage model (local vs. remote)*

- Sequoia: *hierarchical storage model w/ static tasks*

- **Habanero:** *hierarchical place tree w/ dynamic parallelism, heterogeneity*

- Scalable implementations of these constructs require first-class compiler and runtime support

- Constructs can be used to raise current low-level programming models, or as target for high-level programming models

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

(Joe)

Mainstream Parallelism-Oblivious Developers

(Stephanie)

Parallelism–Aware Developers

(Doug)

Concurrency Experts
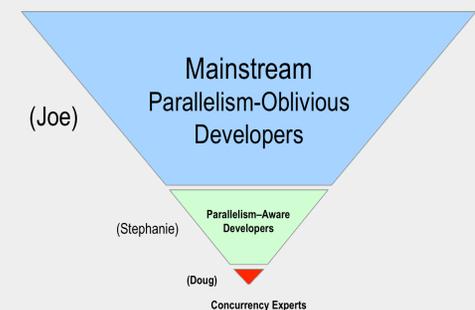
Extreme Scale Platforms

RICE

20

HJ Futures: Functional Tasks with Return Values

async<T> { <Stmt-Block> }

- Creates a new child task that executes Stmt-Block, which must terminate with a return statement returning a value of type T

- Async expression returns a reference to a *container* of type future<T>, and parent task can proceed immediately to operation following the async

- Values of type future<T> can only be assigned to *final variables*

Expr.get()

- Evaluates Expr, and blocks if Expr's value is unavailable

- Expr must be of type future<T>

- Return value from Expr.get() will then be T

- *Assignment of future references to final variables guarantees deadlock freedom with get() operations*
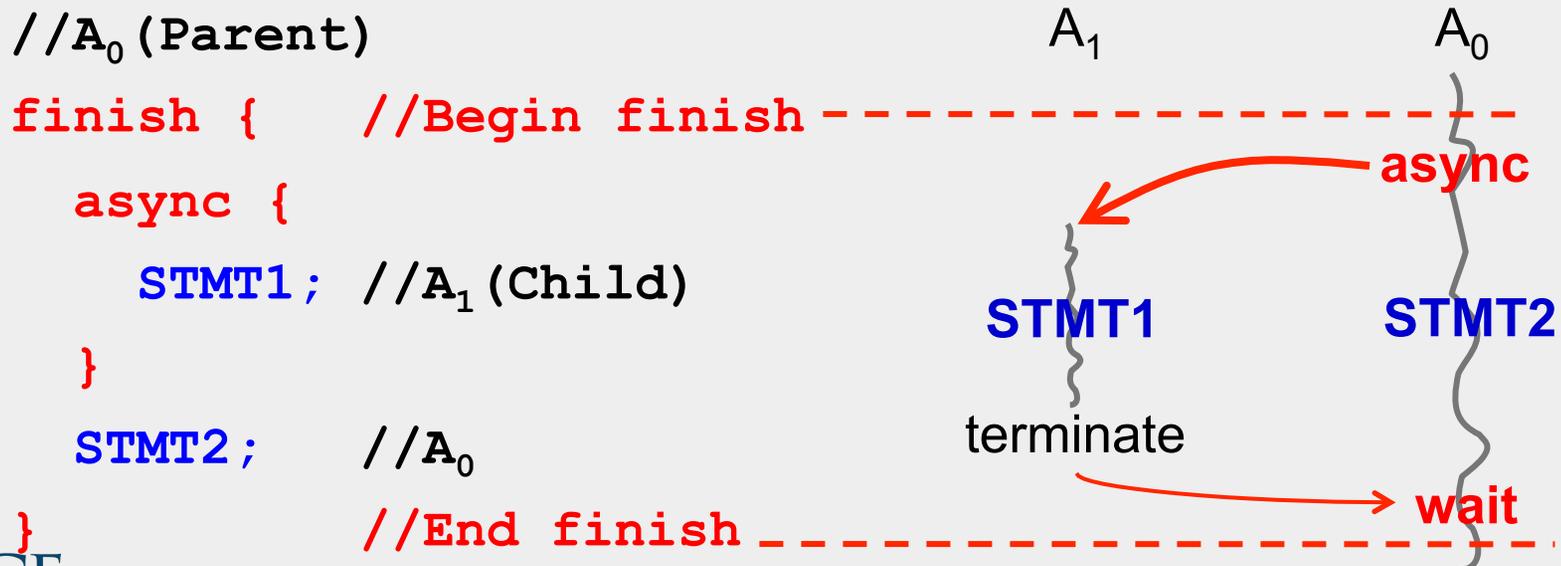
HJ Async and Finish: Imperative Tasks

async [seq(cond)] S

- Creates a new child task that executes statement S ; parent task can proceed immediately to operation following the async

- Optional "seq" clause

    - async seq(cond) <stmt> ≡

    if (cond) <stmt> else async <stmt>

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 synchronization cannot create a deadlock cycle*

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

A_1 A_0

async

STMT1   STMT2

terminate

wait

22

Task Creation: Library Approach

```
List<Callable<Void>> list = …
for(int i = …) {
  list.add(new Callable<Void>() {
    public Void call() {
     // some computation
    }
   });
}
executor.invokeAll(list);
```

- **What do we need ?**
  - A task executor
  - A task interface
  - A task implementation
- **Drawback**
  - Readability
  - Programming chores
    - Manage tasks
    - Schedule tasks
  - Error-prone!

RICE

23

Task Creation: Language Approach

- What a mainstream programmer wants ?

```
finish {
    for(int i = …) {
        async {
            // some computation
        }
    }
}
```

- Run "this statement in parallel
  - Simple task creation
  - No task management
  - No explicit task scheduling

RICE

24

HJ isolated statement

isolated <body>

- Two tasks executing isolated statements with interfering accesses must perform the isolated statement in mutual exclusion
    - Two instances of isolated statements, $\langle stmt1 \rangle$ and $\langle stmt2 \rangle$, are said to interfere with each other if both access a shared location, such that at least one of the accesses is a write.
    - ➔ Weak atomicity guarantee: no mutual exclusion applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances

- Isolated statements may be nested (redundant)

- Isolated statements must not contain any other parallel statement:
  async, finish, get, forall

- In case of exception, all updates performed by <body> before throwing the exception will be observable after exiting <body>

Parallel Spanning Tree Algorithm using Finish, Async, and Isolated constructs

```
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();
```


DFS

compute

compute

compute
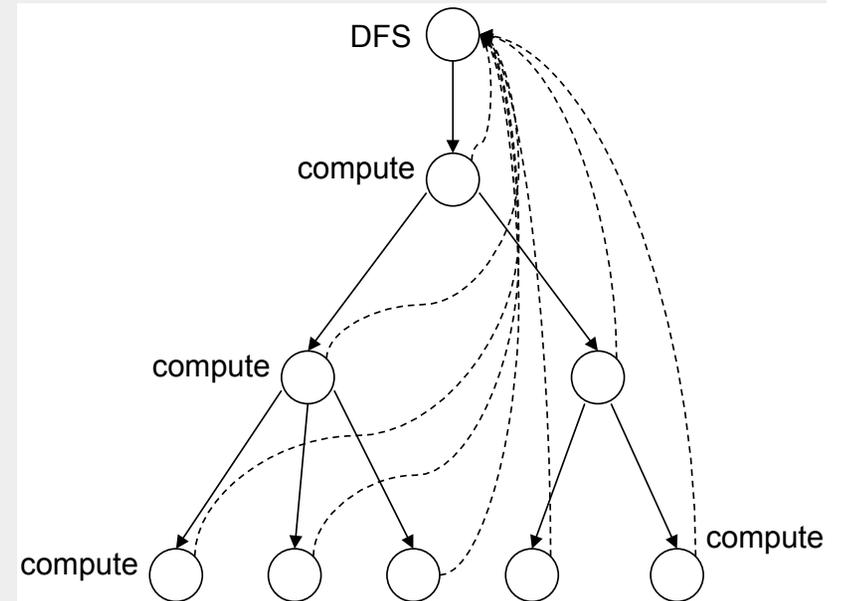
compute

compute

→ Async edge

┈┈┈> Finish edge

26

Computation Graphs for HJ Programs

- A Computation Graph (CG) is an abstract data structure that captures the dynamic execution of an HJ program
- The nodes in the CG are *steps* in the program's execution
  - A step is a sequential subcomputation of a task that contains no continuation points
  - When a worker starts executing a step, it can execute the entire step without interruption
  - Steps need not be maximal i.e., it is acceptable to split a step into smaller steps if so desired

Example HJ Program Decomposed into Non-Maximal Steps (v1 … v23)

```
// Task T1
v1; v2;
finish {
  async {
    // Task T2
    v3;
    finish {
      async { v4; v5; } // Task T3
      v6;
      async { v7; v8; } // Task T4
      v9;
    } // finish
    v10; v11;
```

```
// Task T2 (contd)
    async { v12; v13;
            v14; } // Task T5
    v15;
  } // end of task T2
  v16; v17; // back in Task T1
} // finish
v18; v19;
finish {
  async {
    // Task T6
    v20; v21; v22; }
}
v23;
```
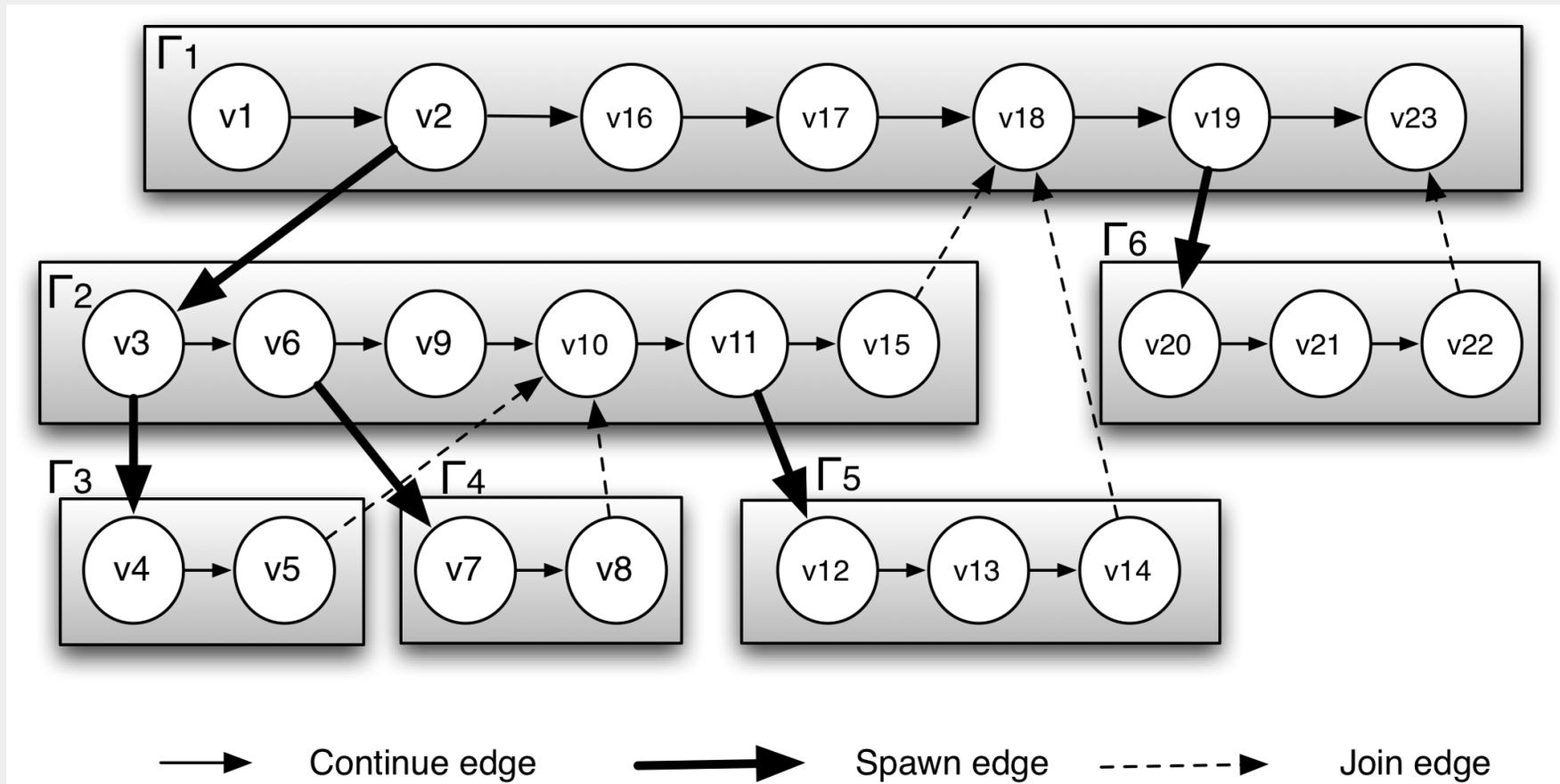
Computation Graph Edges

- CG edges represent ordering constraints
- There are three kinds of CG edges of interest in an HJ program with finish &async operations
    1. *Continue* edges define sequencing of steps within a task
    2. *Spawn* edges connect parent tasks to child async tasks
    3. *Join* edges connect async tasks to their Immediately Enclosing Finish (IEF) operations

Computation Graph for previous HJ Example

Continue edge ⟶ Spawn edge ⟹ Join edge ⇢

Observation: Step v16 can potentially execute in parallel with steps v3 … v15

Dependences in a Computation Graph

- Given edge (A,B) in a CG, node B can only start execution after node A has completed

- We say that *node Y depends on node X* if there is a path of directed edges from X to Y in the CG
  - Also referred to as a "dependence from node X to node Y" or a "dependence from node Y on node X"

- Nodes X and Y can *potentially execute in parallel* if there is no dependence from X to Y or from Y to X

- Dependence is a *transitive* relation
  - if B depends on A and C depends on B, then C must depend on A

- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself

- Computation graphs are examples of *directed acyclic graphs* (dags)

Complexity Measures for Computation Graphs

Define

- time(N) = execution time of node N

- WORK(G) = sum of time(N), for all nodes N in CG G
  - WORK(G) is the total amount of work to be performed in G

- CPL(G) = length of a longest path in CG G, when adding up the execution times of all nodes in the path
  - Such paths are called *critical paths*
  - CPL(G) is the length of these paths (*critical path length*)

Example

Assume time(N) = 1 for all nodes in this graph

WORK(G) = 18

Example (contd)

Assume time(N) = 1 for all nodes in this graph

CPL(G) = 9

Lower Bounds on Execution Time

- t_P = execution time of computation graph on P processors
- Observations
  - t_1 = WORK(G)
  - t_∞ = CPL(G)
- Lower bounds
  - Capacity bound: $t_P \geq WORK(G)/P$
  - Critical path bound: $t_P \geq CPL(G)$
- Putting it together
  - $t_P \geq \max(WORK(G)/P, CPL(G))$

Greedy-Scheduling Theorem (Upper Bound)
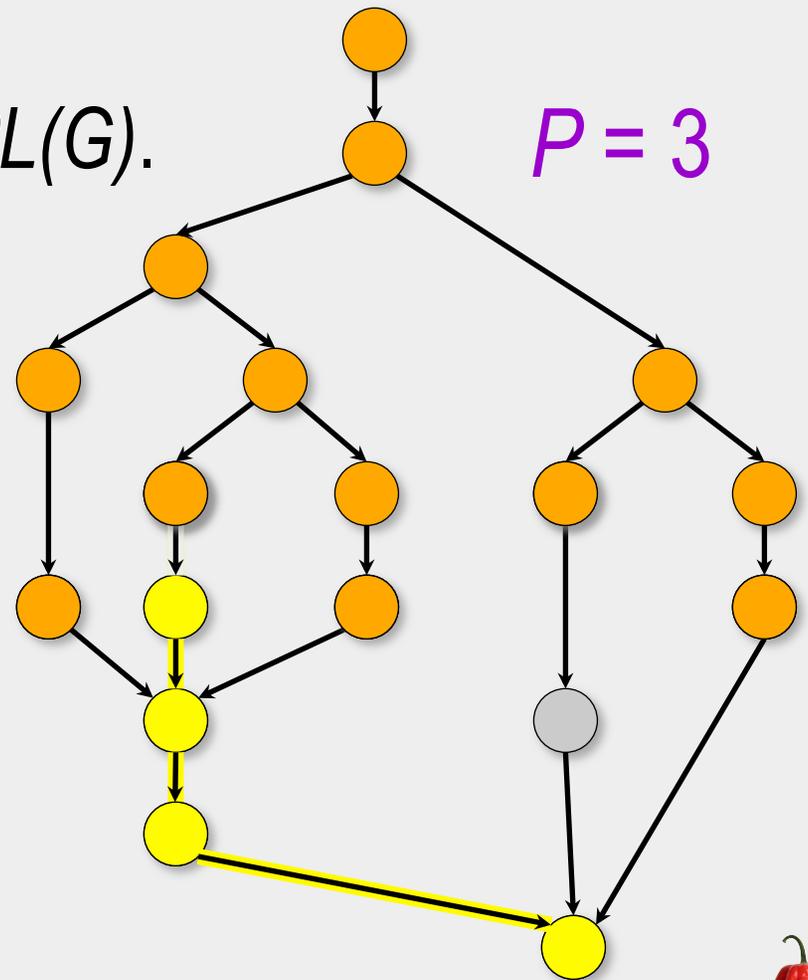
Theorem [Graham '66]. Any greedy scheduler achieves
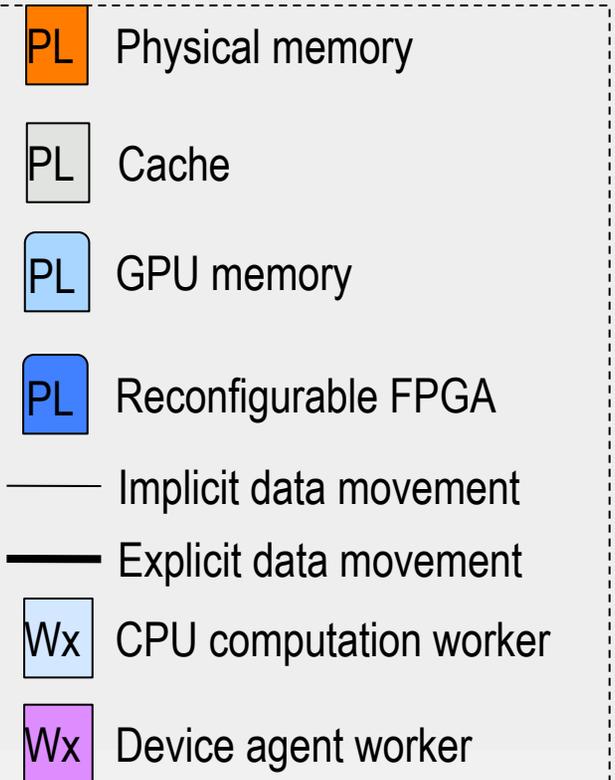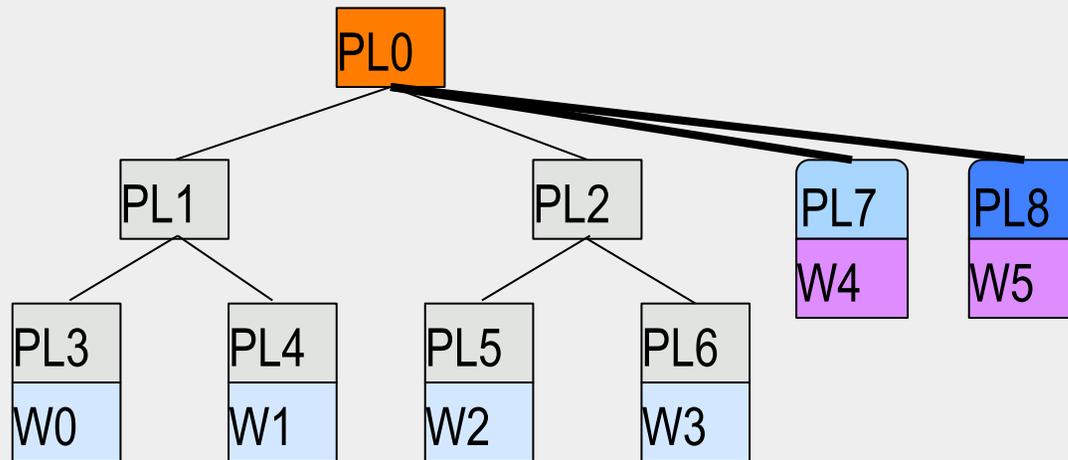
$$t_P \leq WORK(G)/P + CPL(G).$$

$P = 3$

Proof sketch.

complete steps ≤ *WORK(G)/P*, since each complete step performs *P* work.

incomplete steps ≤ *CPL(G)*, since each incomplete step reduces the span of the unexecuted dag by 1. ∎

# Hierarchical Place Trees for Locality and Heterogeneity

- **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**

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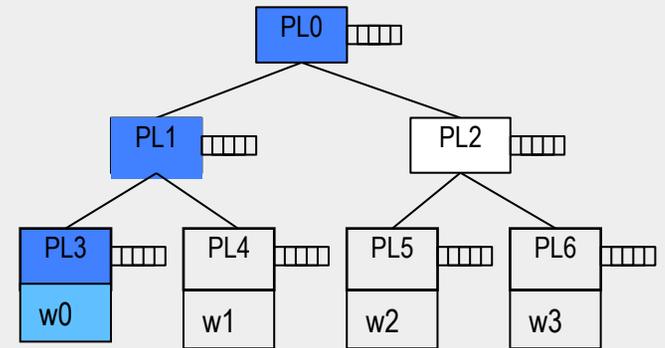
- A worker executes tasks from ancestor places from bottom-up
  - W0 executes tasks from PL3, PL1, PL0
- Tasks in a place queue can be executed by all workers in the place's subtree
  - Task in PL2 can be executed by workers W2 or W3

# Logical Structure of a CUDA kernel invocation in HJ terms

```
1  finish async at(GPU) {
2    // Parallel execution of blocks in grid
3    forall (point[blockIdx.x,blockIdx.y] : [0:gridDim.x-1,0:gridDim.y-1]) {
4      // Parallel execution of threads in block (blockIdx.x,blockIdx.y)
5      forall (point[threadIdx.x,threadIdx.y,threadIdx.z]
6                    : [0:blockDim.x-1,0:blockDim.y-1,0:blockDim.z-1]) {
7        // Perform kernel computation as function of blockIdx.x,blockIdx.y
8        // and threadIdx.x,threadIdx.y,threadIdx.z
9        . . .
10       next; // barrier synchronizes inner forall only (__syncthreads)
11       . . .
12     } // forall threadIdx.x,threadIdx.y,threadIdx.z
13   } // forall blockIdx.x, blockIdx.y
14 } // finish async (GPU)
```

Listing 1: Logical structure of a CUDA kernel invocation

- Future work: automatic generation of CUDA/ OpenCL from above HJ code structure

RICE

Classification and Properties of Parallel Programs
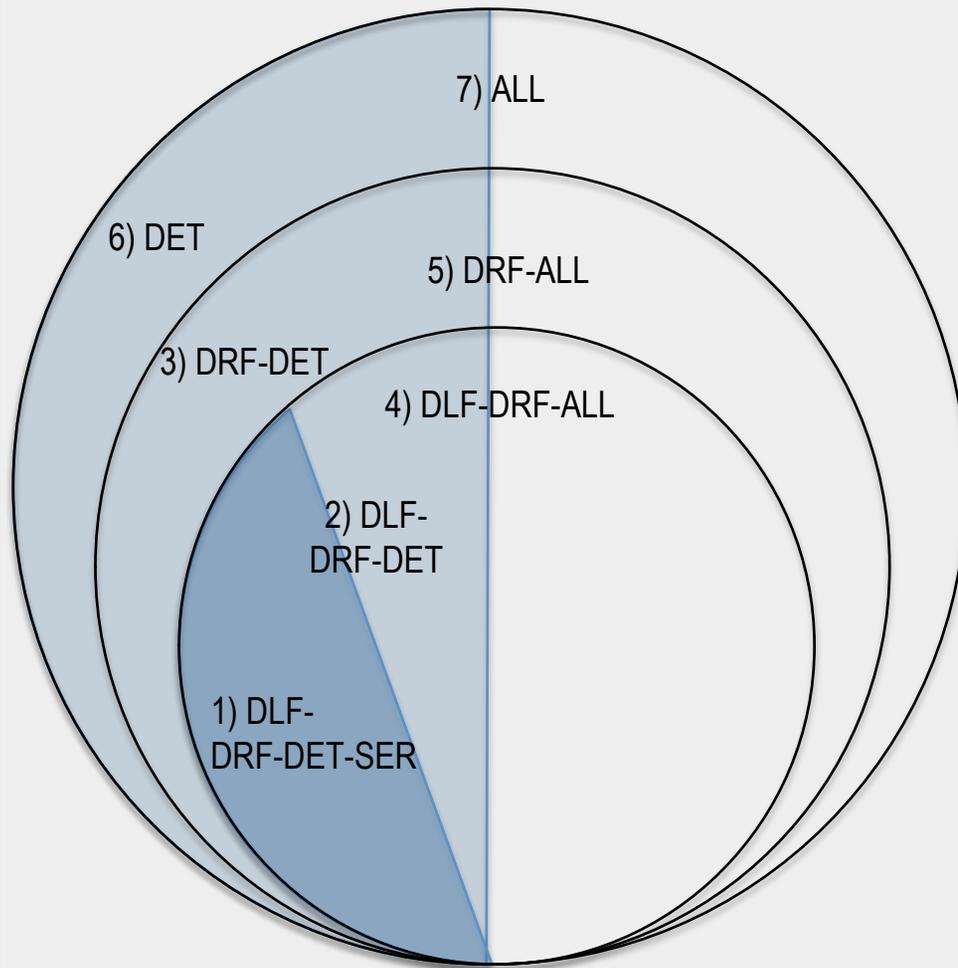
- Legend
  - DET = Deterministic
  - DRF = Data-Race-Free
  - DLF = DeadLock-Free
  - SER = Serializable

- Subsets of task-parallel constructs can be used to guarantee membership in certain classes e.g.,

- *If an HJ program is data-race-free and only uses async, finish, and phaser constructs (no mutual exclusion), then it is guaranteed to belong to the DLF-DRF-DET class*

- *Adding async await yields programs in the DRF-DET class*

- *Adding isolated yields programs in the DRF-ALL class*

Summary

- **Habanero-Java is a safe and powerful mid-level parallel language**
- **Safety**
  - Deadlock freedom for any HJ program using finish, async, futures, phasers, isolated
  - Data-race freedom for values accessed through futures and data-driven futures
- **Expressiveness**
  - Orthogonal classes of parallel constructs enables programmers with a basic knowledge of Java to get started quickly with expressing a wide range of parallel patterns
- **Performance**
  - HJ runs on standard JRE's, and has been shown to deliver good performance on a wide range of multicore SMPs (16-core Xeon, 32-core Power7, 64 & 128-thread Nlagara2)
- **HJ's mid-level constructs are a good match for**
  - Undergraduate level teaching
  - Multicore software research
  - Future JVM support
  - . . .

Conclusions

- **Habanero-Java is a safe and powerful mid-level parallel language**
- **Safety**
  - Deadlock freedom for any HJ program using finish, async, futures, phasers, isolated
  - Data-race freedom for values accessed through futures and data-driven futures
- **Expressiveness**
  - Orthogonal classes of parallel constructs enables programmers with a basic knowledge of Java to get started quickly with expressing a wide range of parallel patterns
- **Performance**
  - HJ runs on standard JRE's, and has been shown to deliver good performance on a wide range of multicore SMPs (16-core Xeon, 32-core Power7, 64 & 128-thread NIagara2)
- **HJ's mid-level constructs are a good match for**
  - Future JVM support
  - Undergraduate level teaching
  - Multicore software research
  - . . .

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
  - Kumud Bhandari, Sanjay Chatterjee, Shams Imam, Deepak Majeti, Raghavan Raman, Dragoș Sbîrlea, Alina Sbîrlea, Kamal Sharma, Rishi Surendran, Sağnak Taşırlar, Nick Vrvilo
- Undergraduate Students
  - Max Grossman, Vijay Rajaram, Yunming Zhang
- Other collaborators at Rice
  - Rich Baraniuk, Corky Cartwright, Swarat Chaudhuri, Keith Cooper, Tim Harvey, Roberto Lublinerman, John Mellor-Crummey, Karthik Murthy, David Peixotto, Bill Scherer, Linda Torczon, Lin Zhong, …

RICE