COMP 322: Fundamentals of Parallel Programming

Lecture 9: Abstract vs. Real Performance

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Announcement

- Homework 1 feedback and grades will be sent to your Rice email
  - We will send one email with feedback and grades on the written assignments and the programming report this weekend (will cover 75% of the homework grade)
  - We will send a second email next week with feedback and grades on the entire homework (including the remaining 25%)

- Homework 2 is due by 5pm on Wednesday, February 6

Generalized Reduce

- **Basic idea: given a binary function, *f(x,y)*, and an identity element, *i*, compute the reduction of an array X[0], X[1], ... as follows**
  - **Reduction = f(f(f(i,X[0]),X[1]) ...), which can be computed sequentially as follows**
    - **temp := i; // identity element**
    - **temp := f(temp, X[0]); // f(i,X[0])**
    - **temp := f(temp, X[1]); // f(f(i,X[0]),X[1])**
    - . . .
- **In Homework 2, you have to write an HJ program to compute the reduction in parallel i.e., to obtain the same answer as the sequential version, assuming that *f(x,y)* is associative and commutative.**
  - ***f(x,y)* is specified by the `combine()` method and the identity element is specified by the `init()` method**
- **In Worksheet 8, we studied the impact of commutativity and associativity on the applicability of finish accumulators and the parallel prefix algorithm**

Worksheet #8 solution: Associativity and Commutativity

A Finish Accumulator (FA) can be used for any *associative and commutative* binary function.
Parallel Prefix (PP) algorithm can be used for any *associative* binary function (the same applies for parallel reductions in ArraySum1 and ArraySum2).

A binary function f is *associative* if f(f(x,y),z) = f(x,f(y,z)).
A binary function f is *commutative* if f(x,y) = f(y,x).

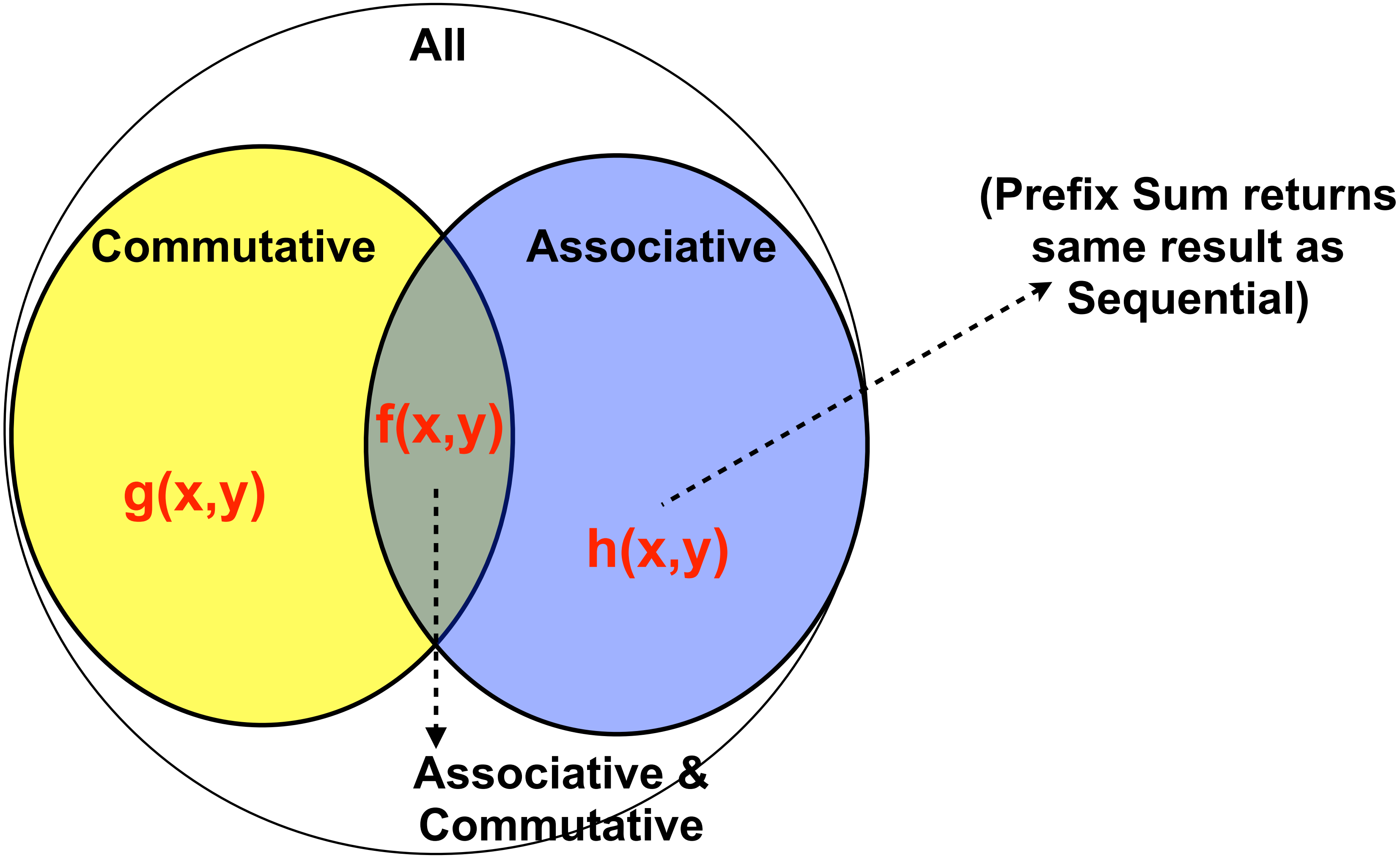
For each of the following functions, indicate if it can be used in a finish accumulator or a parallel prefix sum algorithm or both or neither.

1) f(x,y) = x+y, for integers x, y, is associative and commutative
⇒ both FA and PP can be used

2) g(x,y) = (x+y)/2, for integers x, y, is commutative but not associative
⇒ neither FA nor PP can be used

3) h(s1,s2) = concat(s1, s2) for strings s1, s2 e.g., h("ab","cd") = "abcd" is associative but not commutative
⇒ PP can be used, but not FA

Venn diagram of binary functions



(Prefix Sum & Finish Accumulator return same result as Sequential)

Why does the String Concatenation function h(s1,s2) not work with Finish Accumulators?

- **Because it is not commutative**
- **Consider the following example (pseudo-code):**

```
accumulator acc = new custom accumulator for function h;
finish(acc) {
    async { ... a.put(“ab”);}
    async { ... a.put(“cd”);}
    async { ... a.put(“ef”);}
    async { ... a.put(“gh”);}
}
print acc.get();
```

- **Since the order of the four put() operations is nondeterministic, the final result can be any permutation of the four strings, when using a finish accumulator**
- **However, parallel prefix (and tree reduction) will compute h(h(“ab”,”cd”), h(“ef”,”gh”)), which is correct due to associativity**

Why does the pairwise-average function g(x,y) not work with Finish Accumulators or Parallel Prefix?

- **Because g(x,y) = (x+y)/2 is not associative**
- **Consider the following finish accumulator example (pseudo-code):**

```
accumulator = new custom accumulator for function g;
// assume that accumulator is initialized to zero
finish {
   async { ... a.put(2);} // result := g(result, 2)
   async { ... a.put(4);} // result := g(result,”4”)
}
```

- **Since the order of the two asyncs is nondeterministic, the final result can be g(g(0,2),4) = 2.5 or g(g(0,4),2) = 2**
- **A similar demonstration can be made for Parallel Prefix since its result can be g(g(0,2),4) = 2.5 or g(0,g(2,4)) = 1.5**

Outline of Today's Lecture

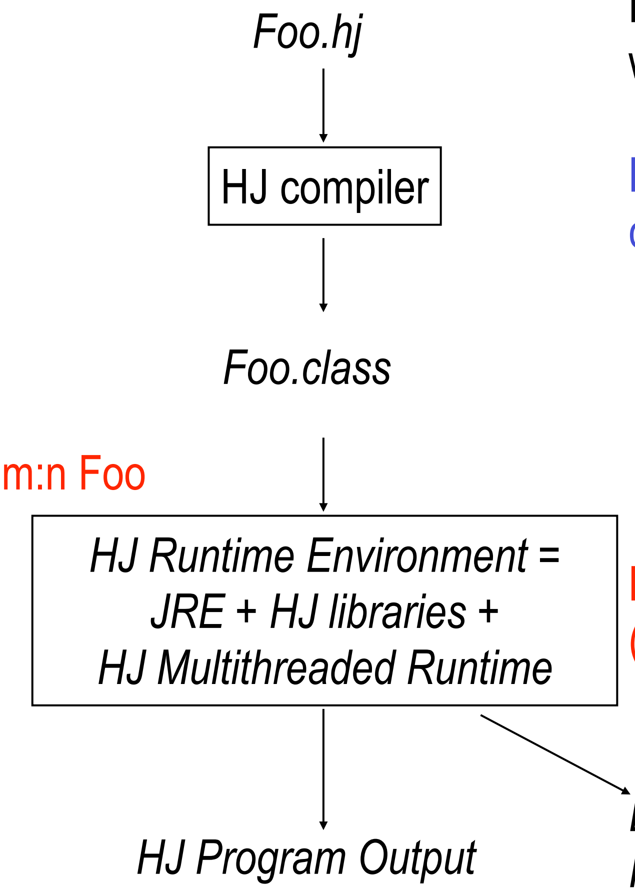
- **Abstract vs. Real performance**

Acknowledgments

- COMP 322 Module 1 handout, Sections 9.1, 9.2, 9.3

HJ Compilation and Execution Environment

DrHJ IDE (optional)



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

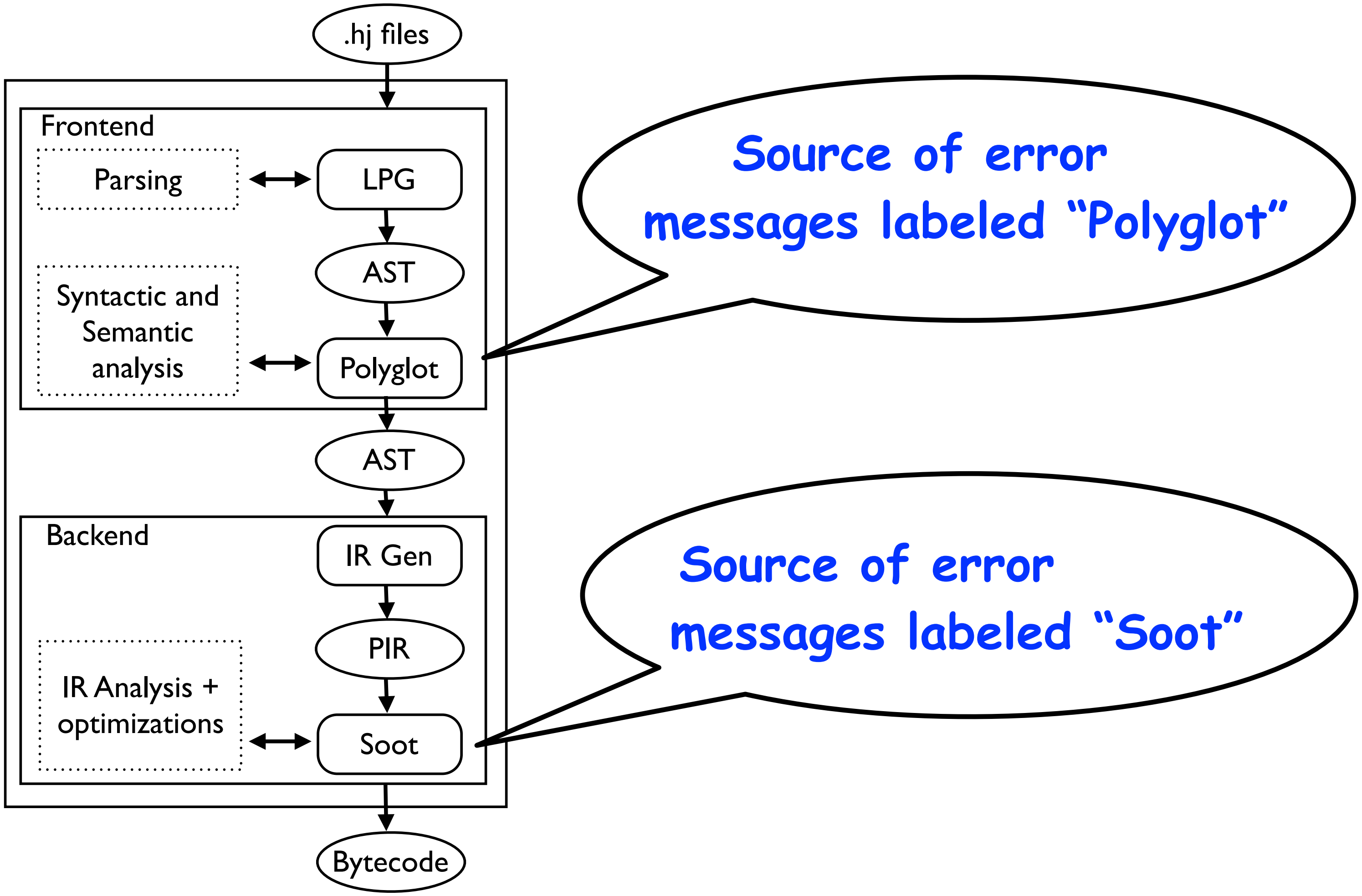
HJ Runtime Environment =
JRE + HJ libraries +
HJ Multithreaded Runtime

HJ runtime allocates m*n worker threads across m “places” (default values: m = 1 place, n = # hardware cores/threads)

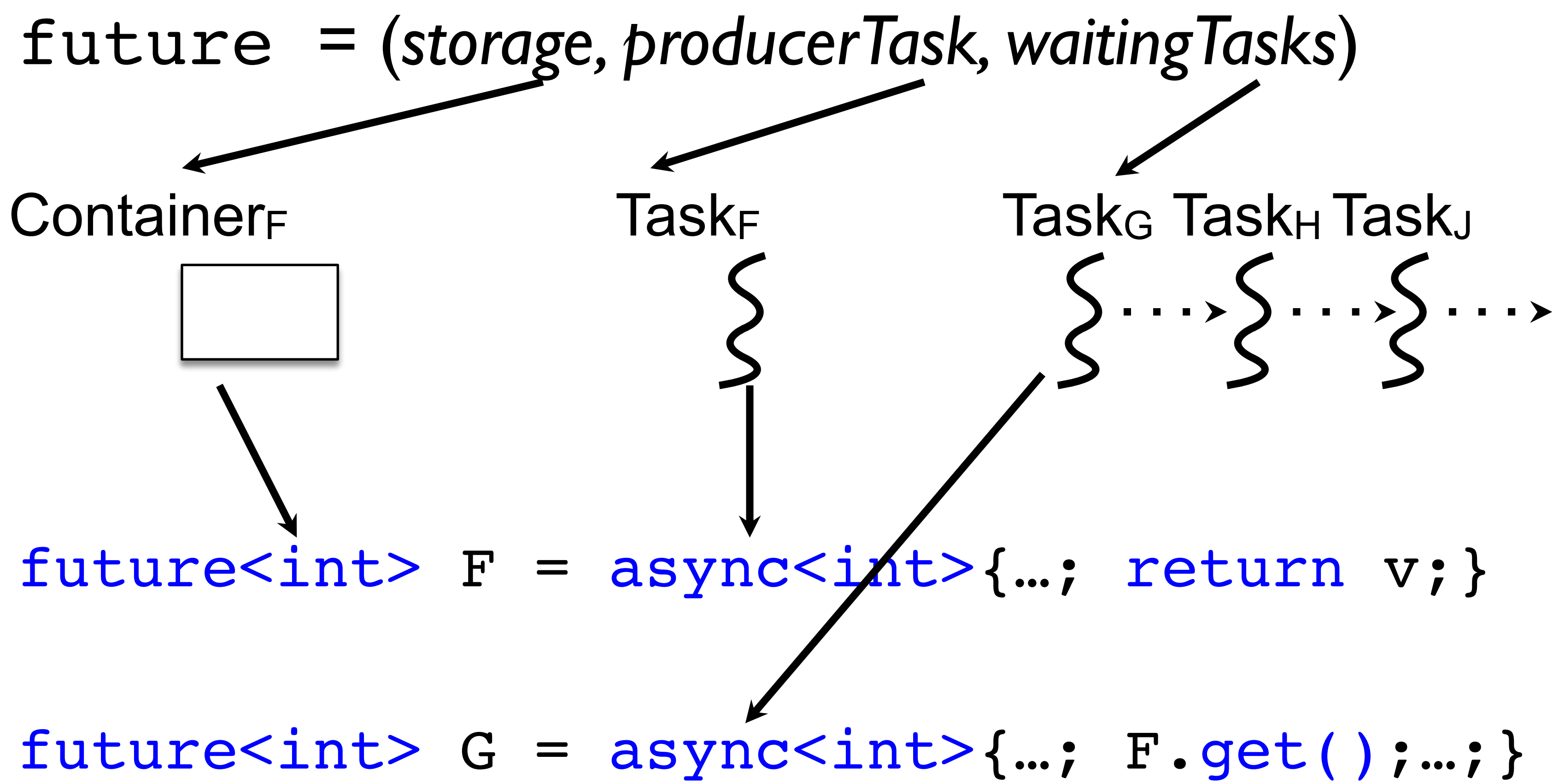
HJ Program Output

Data Race Detection Output,
HJ Abstract Performance Metrics
(all enabled by appropriate options)

Under the hood look at the HJ Compiler



Under the Hood View of Futures in HJ Runtime System



`future` = (*storage*, *producerTask*, *waitingTasks*)

Container$_F$

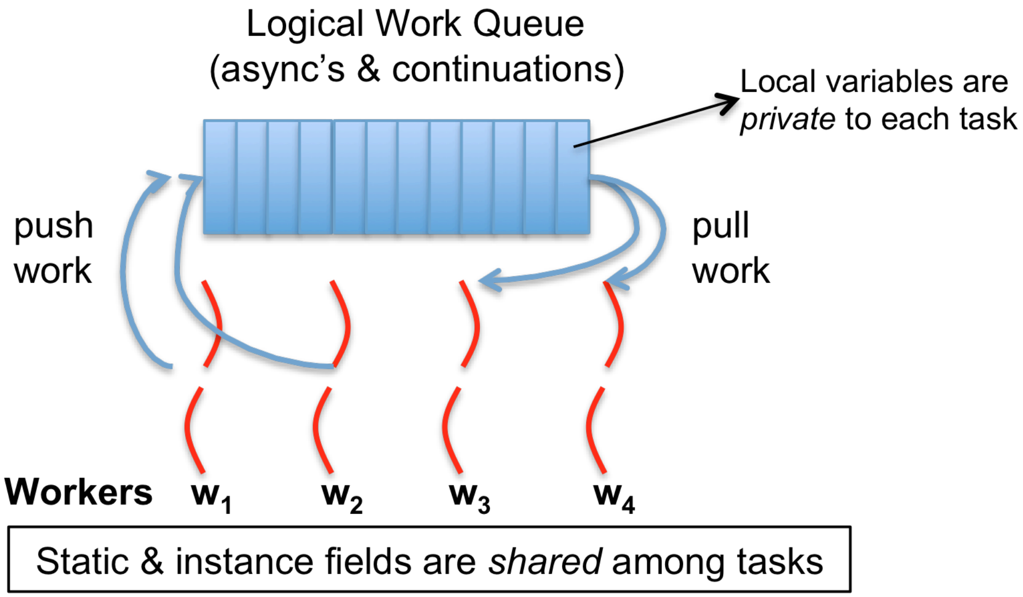
Task$_F$

Task$_G$ Task$_H$ Task$_J$

```
future<int> F = async<int>{…; return v;}

future<int> G = async<int>{…; F.get();…;}
```

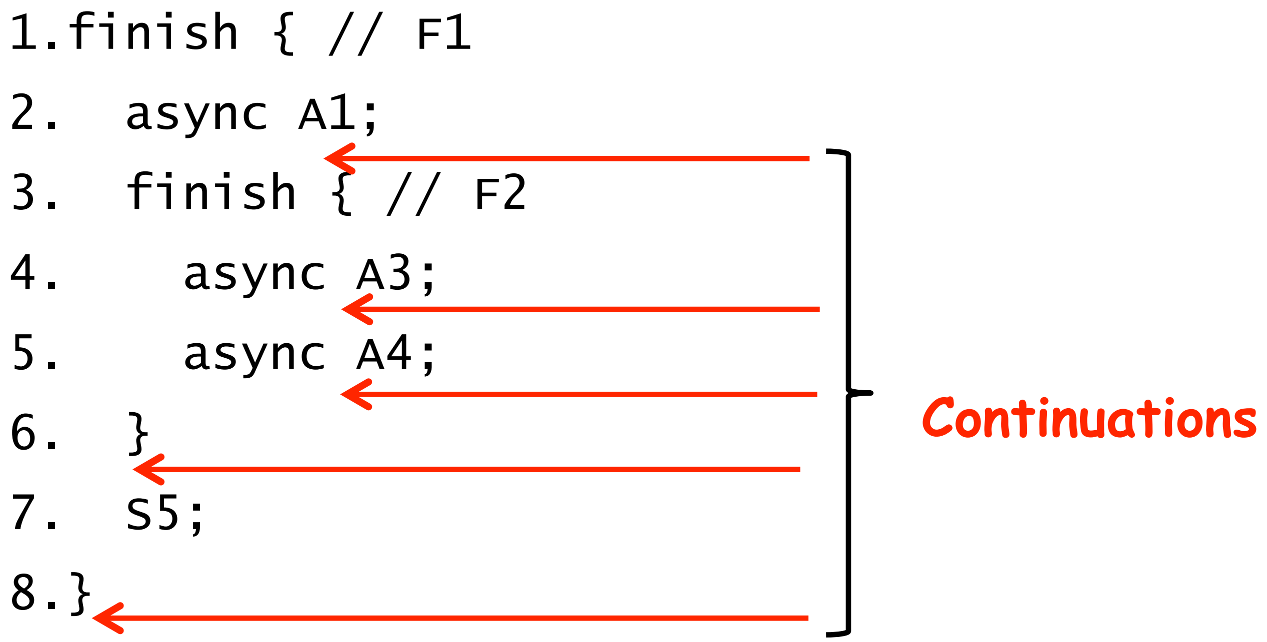
Scheduling HJ tasks on processors in a parallel machine



- **HJ runtime creates a small number of worker threads, typically one per core**
- **Workers push async's and/or "continuations" into a logical work queue**
    - **when an async operation is performed**
    - **when an end-finish operation is reached**
- **Workers pull task/continuation work item when they are idle**

Continuations

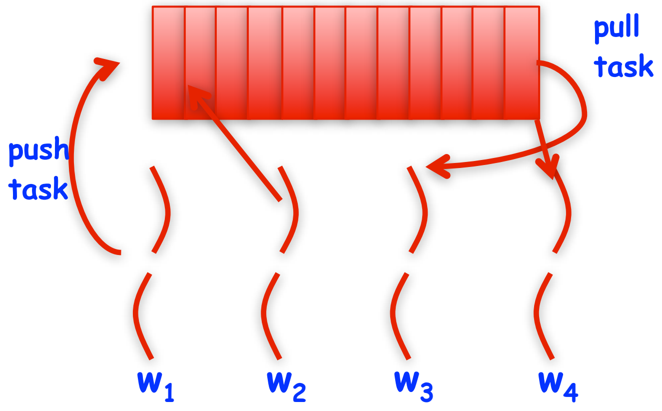
- **A continuation is one of two kinds of program points**
  - The point in the parent task immediately following an async
  - The point immediately following an end-finish or a future get()
- **Continuations are also referred to as task-switching points**
  - Program points at which a worker may switch execution between different tasks (depends on scheduling policy)



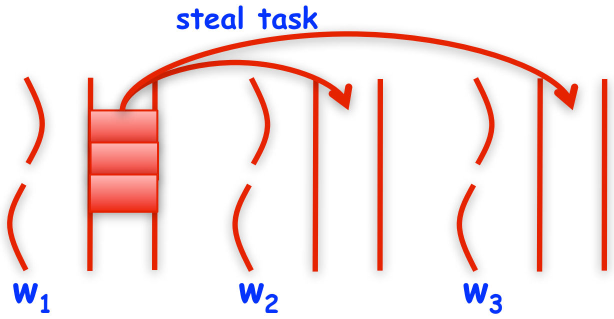
```
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
  S5;
}
```

Work-Sharing vs. Work-Stealing Scheduling Paradigms

- **<u>Work-Sharing</u>**
  - —Busy worker eagerly distributes new work
  - —Easy implementation with global task pool
  - —Access to the global pool needs to be synchronized: scalability bottleneck
- **<u>Work-Stealing</u>**
  - —Busy worker incurs little overhead to create work
  - —Idle worker "steals" the tasks from busy workers
  - —Distributed task pools lead to improved scalability
  - —When task T_a spawns T_b, the worker can
    - –stay on T_a, making T_b available for execution by another processor (<u>help-first</u> policy), or
    - –start working on T_b first (<u>work-first</u> policy)



work-sharing



work-stealing runtime

Work-first vs. Help-first work-stealing policies on 2 processors

```
1. finish {
2.  // Start of Task T0 (main program)
3.  sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
4.  async { // Task T1 computes sum of upper half of array
5.    for(int i=X.length/2; i < X.length; i++)
6.      sum2 += X[i];
7.  }
8.  // T0 computes sum of lower half of array
9.  for(int i=0; i < X.length/2; i++) sum1 += X[i];
10. }
11. // Task T0 waits for Task T1 (join)
12. return sum1 + sum2;
13.} // finish
```

- **Help-first policy: Worker 0 executes lines 1, 2, 3 in T0, pushes out async on line 4, and then executes lines 8, 9 in Task T0. Worker 1 steals async on line 4 and executes task T1.**

- **Work-first policy: Worker 0 executes lines 1, 2, 3 in T0, pushes out continuation on line 8, and then executes async in task T0. Worker 1 steals continuation at line 8 in T0.**

Work-first vs. Help-first work-stealing policies on 2 processors (contd)

```
finish {
  // Start of Task T0 (main program)
  sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
  async { // Task T1 computes sum of upper half of array
    for(int i=X.length/2; i < X.length; i++)
      sum2 += X[i];
  }
  // T0 computes sum of lower half of array
  for(int i=0; i < X.length/2; i++) sum1 += X[i];
}
// Task T0 waits for Task T1 (join)
return sum1 + sum2;
} // finish
```

Help-First worker does not switch tasks
Work-first worker will switch tasks

Continuations

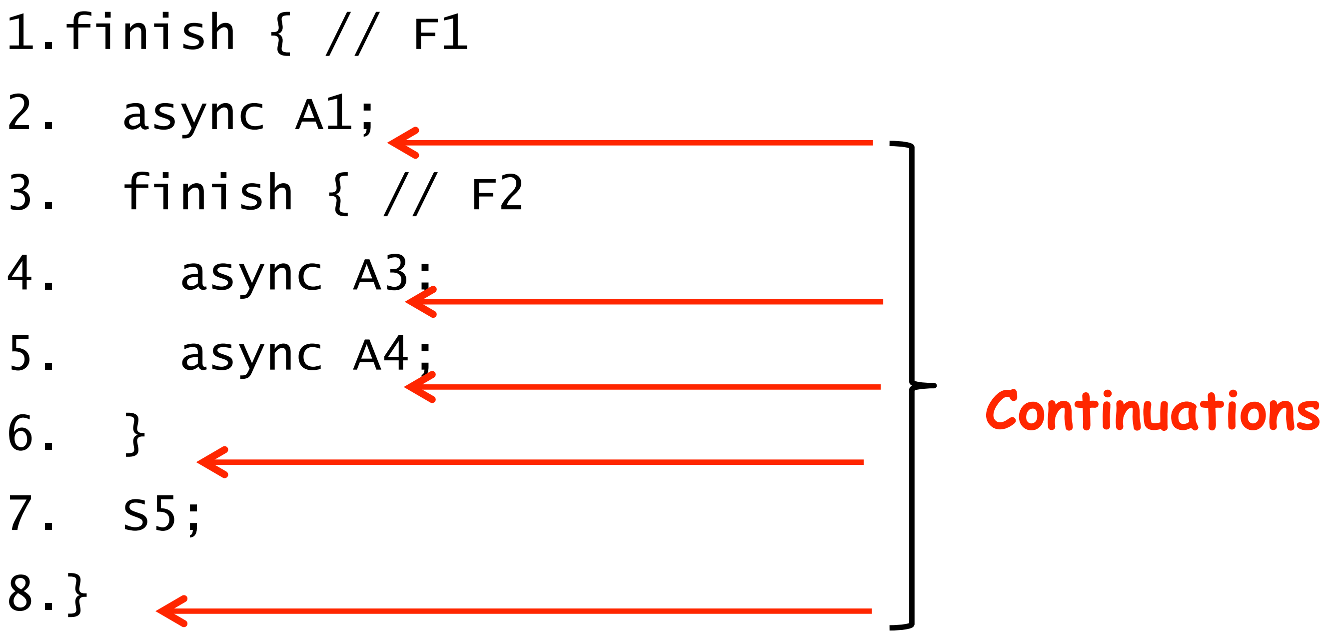
Help-First worker can switch tasks
Work-first worker can switch tasks

Let's try more of this in Worksheet #9 !

Worksheet #9: Continuations and Work-First vs. Help-First Work-Stealing Policies

Name 1: ______________________ **Name 2:** ______________________

For each of the continuations below, label it as "WF" if a work-first worker can switch tasks at that point and as "HF" if a help-first worker can switch tasks at that point. Some continuations may have both labels.



```
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
  S5;
}
```