COMP 322: Fundamentals of Parallel Programming

Lecture 10: Abstract vs. Real Performance (contd), seq clause

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

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

Outline of Today's Lecture

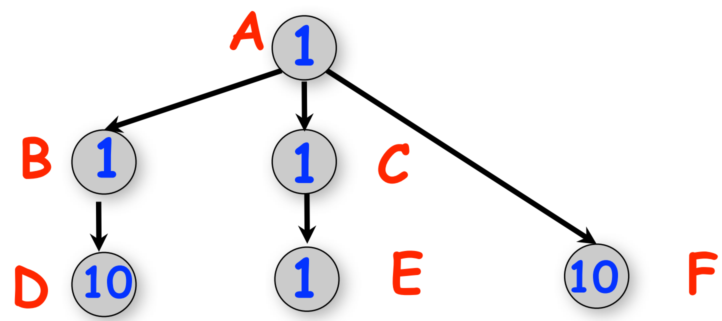
- **Abstract vs. Real performance (contd)**

- **seq clause in async statements**

Acknowledgments

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

Recap of 2-processor schedule of a Computation Graph studied in Lecture 3 (slide 11)



Schedule with execution time, $T_2 = 13$

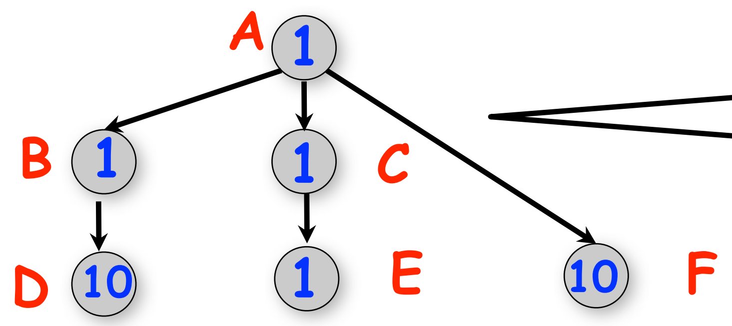
Start time	Proc 1	Proc 2
0	A	
1	B	F
2	D	F
3	D	F
4	D	F
5	D	F
6	D	F
7	D	F
8	D	F
9	D	F
10	D	F
11	D	C
12		E
13		

This schedule was obtained by mapping computation graph nodes to processor assuming:

1. ***Non-preemption (no context switch in the middle of a node)***
2. ***Greedy schedule (a processor is never idle if work is available)***

There may be multiple possible schedules with these assumptions

Two possible HJ programs for this Computation Graph (there can be others ...)



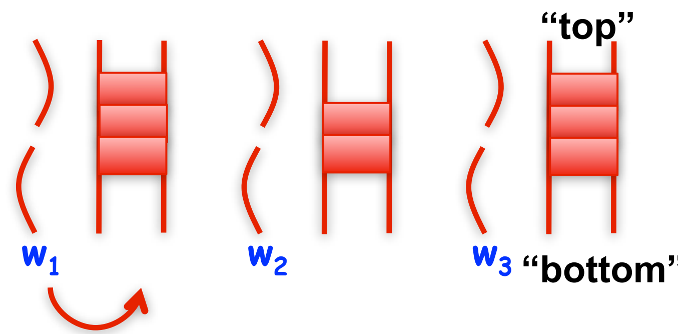
There is no significance to the left-to-right ordering of edges in a computation graph, which is why there can be multiple parallel programs for the same computation graph

```
// Program Q1
A;
finish {
  async { B; D; }
  async F;
  async { C; E; }
}
```

```
// Program Q2
A;
finish {
  async { C; E; }
  async F;
  async { B; D; }
}
```

Recap of Work-first vs. Help-first work-stealing policies

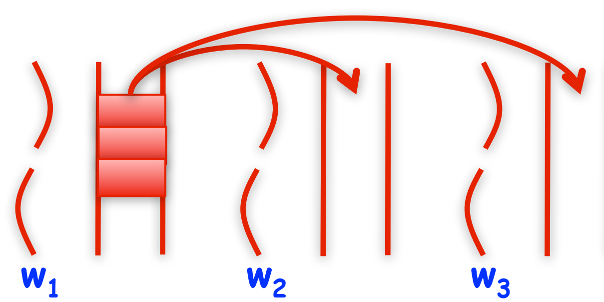
- When encountering an async
  - Help-first policy
    - Push async on “bottom” of local queue, and execute next statement
  - Work-first policy
    - Push continuation (remainder of task starting with next statement) on “bottom” of local queue, and execute async



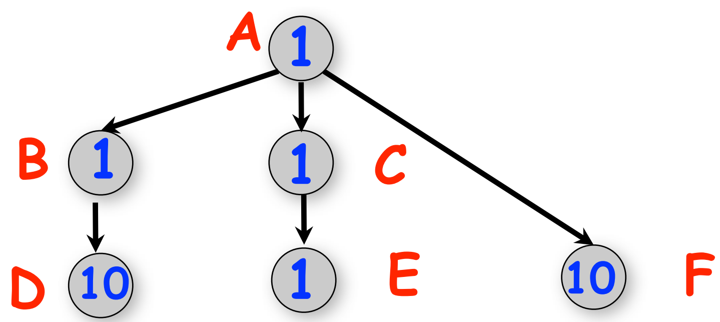
Local push/pop by w_1

- When encountering the end of a finish scope
  - Help-first policy & Work-first policy
    - Store continuation for end-finish
      - Will be resumed by last async to complete in finish scope
    - Pop most recent item from “bottom” of local queue
    - If local queue is empty, steal from “top” of another worker’s queue

Stealing by w_2 and w_3



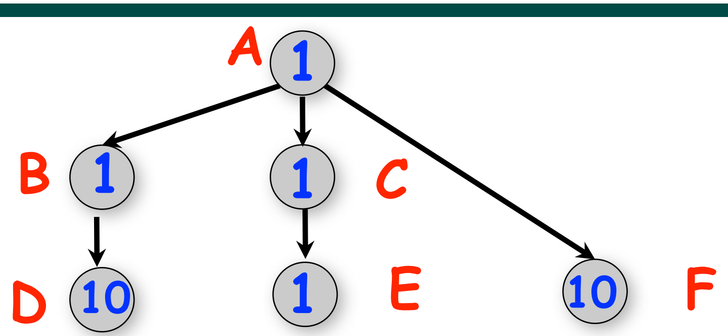
Scheduling Program Q1 using a Work-First Work-Stealing Scheduler




```
1. // Program Q1
2. A; // Executes on P1
3. finish {
4.   // P1 pushes continuation for 9,
5.   // and executes 6
6.   async { B; D; }
7.   // P2 pushes continuation for 11,
8.   // and executes 9
9.   async F;
10.   // P2 executes 11
11.  async { C; E; }
12. }
```

Start time	Proc 1	Proc 2
0	A	
1	B	F
2	D	F
3	D	F
4	D	F
5	D	F
6	D	F
7	D	F
8	D	F
9	D	F
10	D	F
11	D	C
12		E
13		

Scheduling Program Q1 using a Help-First Work-Stealing Scheduler




```
1. // Program Q1
2. A; // Executes on P1
3. finish {
4.   // P1 pushes 6, which is then
5.   // stolen by P2
6.   async { B; D; }
7.   // P1 pushes 8
8.   async F;
9.   // P1 pushes 10
10.  async { C; E; }
11. }
12. // P1 stores continuation and pops 10
13. // P1 pops 8
```

Let's try more of this in Worksheet #10 !

Start time	Proc 1	Proc 2
0	A	
1	C	B
2	E	D
3	F	D
4	F	D
5	F	D
6	F	D
7	F	D
8	F	D
9	F	D
10	F	D
11	F	D
12	F	
13		

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

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.

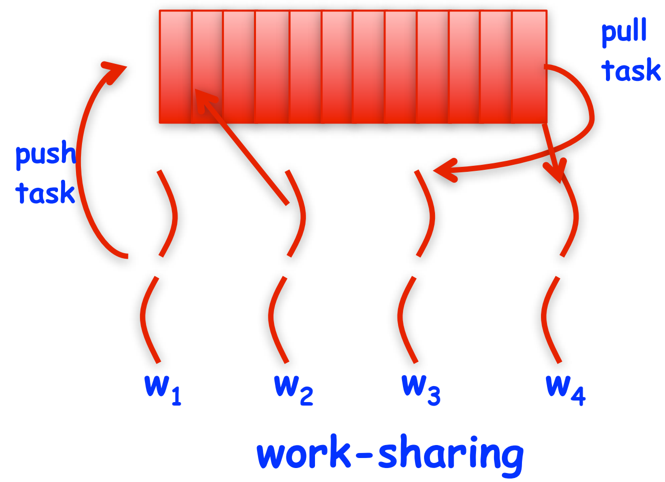
```
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
  S5;
}
```

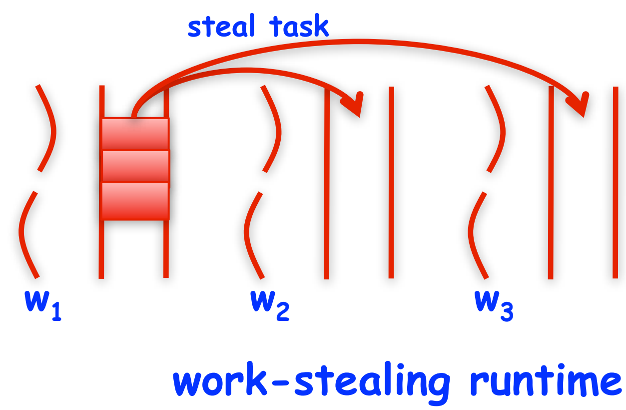
Continuations:

- after `async A1;` — **WF**
- after `async A3;` — **WF**
- after `async A4;` — **WF**
- after `}` (end of finish F2) — **WF, HF**
- after `}` (end of finish F1) — **WF, HF**

Work-Sharing vs. Work-Stealing Scheduling Paradigms (Recap)

- **Work-Sharing**
  - Busy worker eagerly distributes new work
  - Easy implementation with global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck
- **Work-Stealing**
  - 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 (help-first policy), or
    - start working on T_b first (work-first policy)





Iterative Fork-Join Microbenchmark

```
finish {
    for (int i=1; i<k; i++)
        async Ti;  // task i
    T0; //task 0
}
```

Single-Worker execution times to model overhead (Section 9.2.1)

- **k = number of tasks**
- **$t_s(k)$ = sequential time**
- **$t_1^{wf}(k)$ = 1-worker time for work-stealing with work-first policy**
- **$t_1^{hf}(k)$ = 1-worker time for work-stealing with help-first policy**
- **$t_1^{ws}(k)$ = 1-worker time for work-sharing**
- **Java-thread(k) = create a Java thread for each async**

Fork-Join Microbenchmark Measurements (execution time in micro-seconds)

k	$t_s(k)$	$t_1^{wf}(k)$	$t_1^{hf}(k)$	$t_1^{ws}(k)$	Java-thread(k)
1	0.11	0.21	0.22		
2	0.22	0.44	2.80		
4	0.44	0.88	2.95		
8	0.90	1.96	3.92	335	3,600
16	1.80	3.79	6.28		
32	3.60	7.15	10.37		
64	7.17	14.59	19.61		
128	14.47	28.34	36.31	2,600	63,700
256	28.93	56.75	73.16		
512	57.53	114.12	148.61		
1024	114.85	270.42	347.83	22,700	768,000

NOTE: Help-First usually performs better than Work-First on this benchmark when the number of workers is > 1, but not in this single-worker case

Q: Why do we have different schedulers?

A: Because they have different performance & functional characteristics

DrHJ compiler option	SUMMARY	Functional limitations	Performance characteristics
work-sharing (Default option)	Supports full HJ language, but can lead to "max threads" error.	1) Supports full lang 2) Supports perf metrics	3) Creates additional worker threads when a task blocks
work-sharing (Fork-Join variant)		1) + 2)	3) + 4) may perform better than work-sharing for recursive parallelism
work-stealing (Help-First policy)	Supports restricted HJ language, but avoids "max threads" error.	5) Only supports async, finish, forasync, isolated, atomic vars	6) Fixed number of worker threads 7) better for loop parallelism
work-stealing (Work-First policy)		5) + 8) Supports data race detection	6) + 9) better for recursive parallelism
work-stealing (Adaptive policy)		Same as 5)	10) automatically chooses between help-first and work-first policies
cooperative *(under development)*	Holy Grail --- full HJ without "max threads" error!	*Currently supports 5) + Futures --- goal is to support everything!*	*Same as 6)*

Scheduling Policies Currently Available in HJ

DrHJ compiler option	Command-line option	Functional characteristics	Performance characteristics
work-sharing (Default option)	Compile: hjc -rt s (default) Runtime: hj (no option needed)	1) Supports full lang 2) Supports perf metrics	3) Creates additional worker threads when a task blocks
work-sharing (Fork-Join variant)	Compile: hjc -rt s (default) Runtime: hj -fj	1) + 2)	3) + 4) may perform better than work-sharing for recursive parallelism
work-stealing (Help-First policy)	Compile: hjc -rt h Runtime: hj (no option needed)	5) Only supports async, finish, forasync, isolated, atomic vars	6) Fixed number of worker threads 7) better for loop parallelism
work-stealing (Work-First policy)	Compile: hjc -rt w Runtime: hj (no option needed)	5) + 8) Supports data race detection	6) + 9) better for recursive parallelism
work-stealing (Adaptive policy)	Compile: hjc -rt h Runtime: hj (no option needed)	Same as 5)	10) automatically chooses between help-first and work-first policies on each async
cooperative *(under development)*	*Compile: hjc -rt c* *Runtime: hj* *(no option needed)*	*Currently supports 5) + Futures --- goal is to support everything!*	*Same as 6)*

Outline of Today's Lecture

- **Abstract vs. Real performance**

- **<u>seq clause in async statements</u>**

Acknowledgments

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

Problem: creating too many small async tasks can be a source of overhead (ArraySum2)

```
static int computeSum(int[] X, int lo, 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); };
    // Parent now waits for the container values
    return sum1.get() + sum2.get();
   }
 } // computeSum
int sum = computeSum(X, 0, X.length-1); // main program
```

Creating one async per tree node leads to too many tasks!

Common fix in parallel divide-and-conquer algorithms --- add a “threshold” test

```
// Minimum size for which an async task is justified
int thresholdSize = 1000000;
. . .
int mid = (lo+hi)/2;
int size = hi - lo + 1;
if (size < thresholdSize) { // Sequential case
  sum1 = computeSum(X, lo, mid); sum2 = computeSum(X, mid+1, hi);
}
else { // Parallel case --- pseudocode
  sum1 = async computeSum(X, lo, mid); sum2 = async computeSum(X, mid+1, 
hi);
}
. . .
```

- **The “size < thresholdSize” condition ensures that async tasks are only created for upper nodes in the reduction tree; lower nodes (closer to the leaves) are executed sequentially.**
- **A large thresholdSize value leads to larger async tasks with less (shallower) parallelism**
- **A small thresholdSize value leads to smaller async tasks with more (deeper) parallelism**

seq clause in HJ async statement

```
async seq(cond) <stmt> ≡ if (cond) <stmt> else async <stmt>
```

```
// Non-Future example
async seq(size < thresholdSize) computeSum(X, lo, mid);

// Future example
final future<int> sum1 = async<int> seq(size < thresholdSize)
                                { return computeSum(X, lo, mid); };
```

- **“seq” clause specifies condition under which async should be executed sequentially**
    - **False ⇒ an async is created**
    - **True ⇒ the parent executes async body sequentially**
- **Avoids the need to duplicate code for both cases**
- **Also simplifies use of final variables (needed for futures)**

Use of seq clause in ArraySum2

```
static int thresholdSize = 1000000;
. . .
static int computeSum(int[] X, int lo, int hi) {
  if ( lo > hi ) return 0;
  else if ( lo == hi ) return X[lo];
  else {
    int mid = (lo+hi)/2; size = hi-lo+1;
    final future<int> sum1 = async<int> seq(size < thresholdSize)
                                  { return computeSum(X, lo, mid); };
    final future<int> sum1 = async<int> seq(size < thresholdSize)
                                  { return computeSum(X, mid+1, hi); };
    // Parent now waits for the container values
    return sum1.get() + sum2.get();
   }
 } // computeSum
 . . .
int sum = computeSum(X, 0, X.length-1); // main program
```

Threshold Condition depends on application

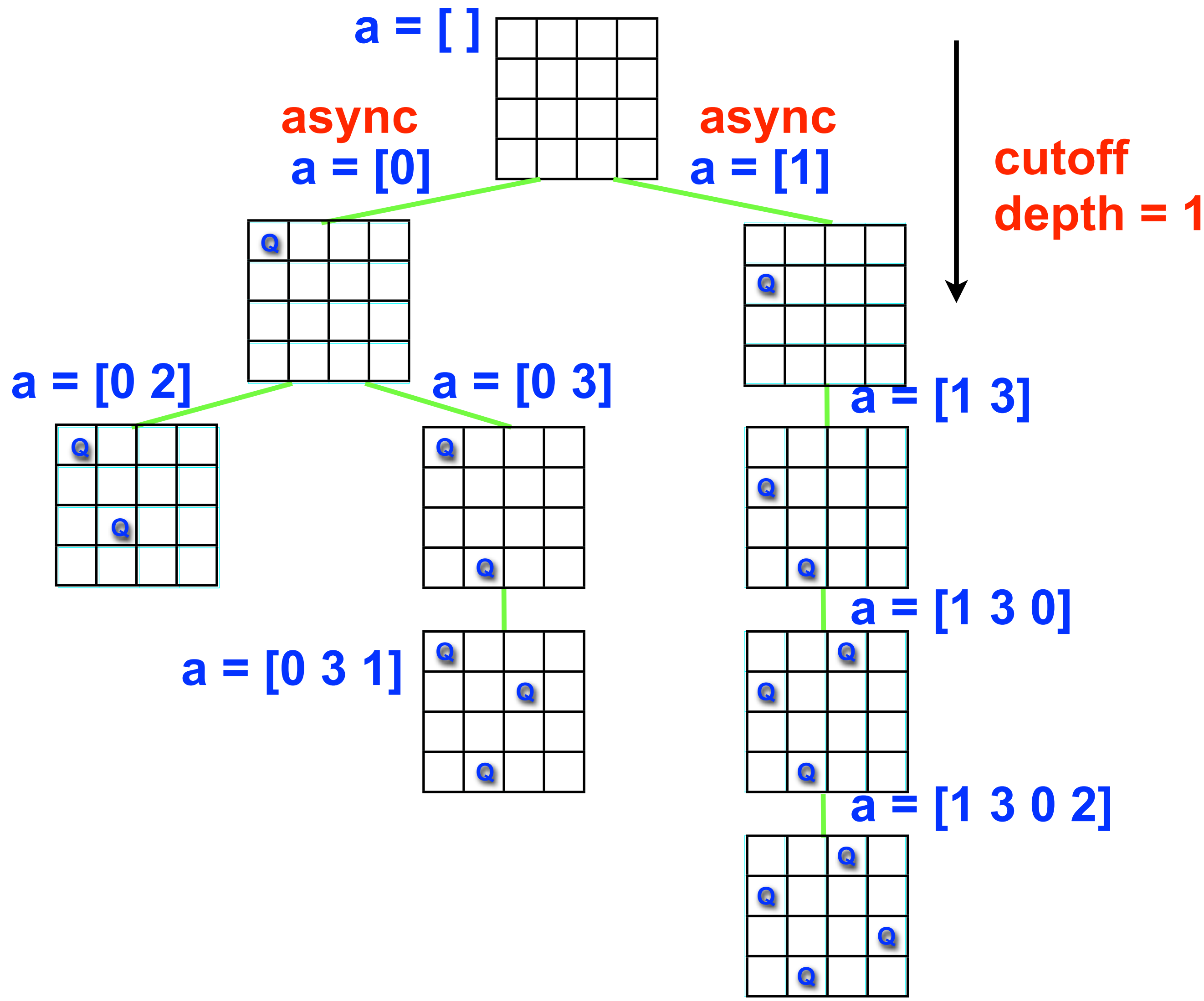
empty board

place 1st queen

place 2nd queen

place 3rd queen

place 4th queen



Parallel Solution to NQueens with Finish Accumulator and seq clause

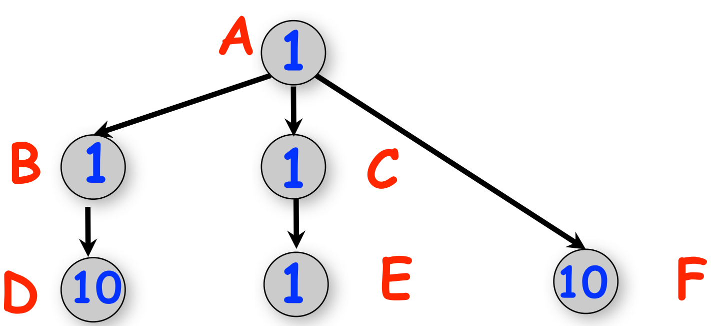
```
static accumulator count;
. . .
count = accumulator.factory.accumulator(SUM, int.class);
finish(a) nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " + count.get().intValue());
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) count.put(1);
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) async seq(depth >= cutoff) {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

Worksheet #10: Scheduling Program Q2 using a Work-First & Help-First Schedulers

Name 1: ______________________

Name 2: ______________________

Complete work-first and help-first schedules for the program shown below (using step times from the computation graph)

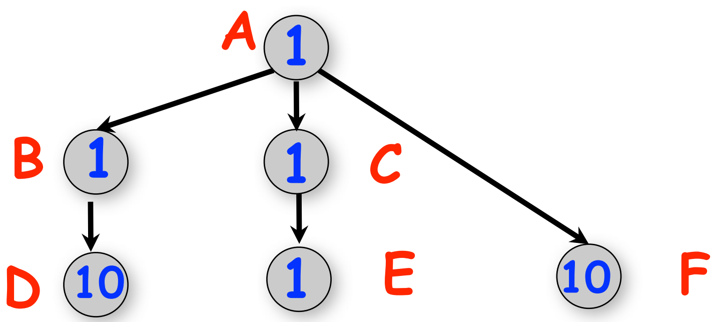



```
// Program Q2
A;
finish {
  async { C; E; }
  async F;
  async { B; D; }
}
```

Work-First Schedule

Start time	Proc 1	Proc 2
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		

Worksheet #10: Scheduling Program Q2 using a Work-First & Help-First Schedulers (contd)




```
// Program Q2
A;
finish {
  async { C; E; }
  async F;
  async { B; D; }
}
```

Help-First Schedule

Start time	Proc 1	Proc 2
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		