COMP 322: Fundamentals of Parallel Programming

Lecture 11: Abstract vs Real Performance, Work-sharing andWork-stealing schedulers

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

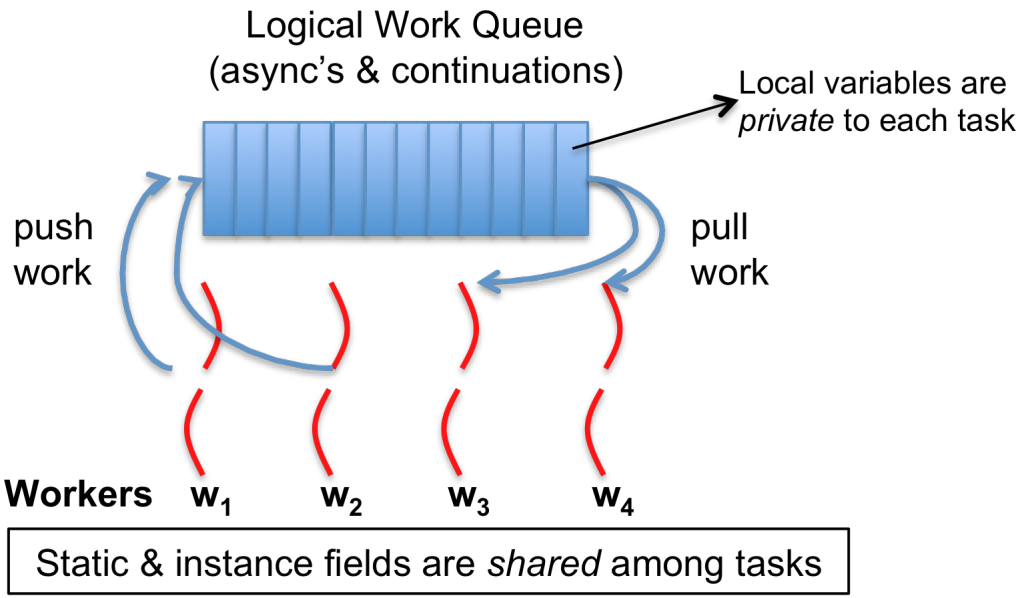
Acknowledgments for Today's Lecture

- Yi Guo. A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism. PhD thesis, Department of Computer Science, Rice University, August 2010.
- Jun Shirako for microbenchmark results.
- Lecture 11 handout

Abstract vs. Real Performance Metrics

- Abstract performance metrics are idealized
  - No penalty for fine-grained tasks and synchronization
- Many sources of overhead in practice
  - Spawn overhead
  - Join overhead
  - IEF-Join overhead
  - Isolation overhead
  - Cache overheads (not discussed in handout)
  - . . .

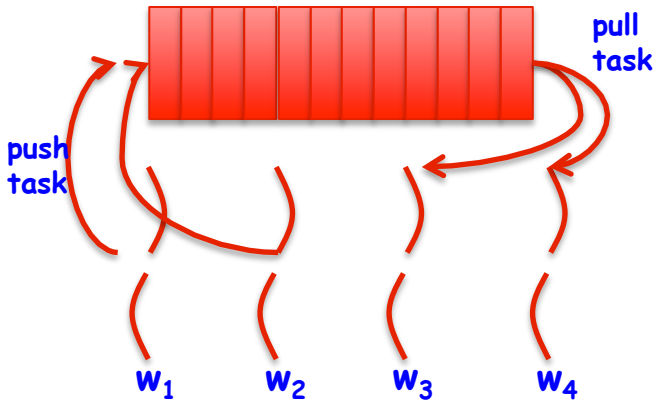
Scheduling HJ tasks on processors in a parallel machine (Lecture 2)



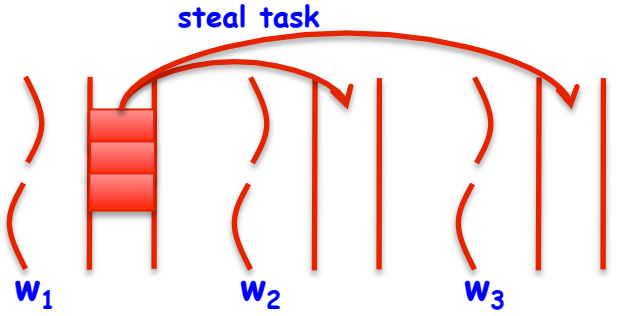
- HJ runtime creates a small number of *worker* threads, typically one per core
- Workers push async's/continuations into a logical *work queue*
  - when an async operation is performed
  - when an end-finish operation is reached
- Workers pull task/continuation work item when they are idle

Work-Sharing vs. Work-Stealing Scheduling Paradigms

- Work-Sharing
  - Busy worker re-distributes the task eagerly
  - Easy implementation through global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck
- Work-Stealing
  - Busy worker pays little overhead to enable stealing
  - Idle worker steals the tasks from busy workers
  - Distributed task pools
  - Better scalability
- Two Work-Stealing policies
  - When T_a spawns T_b, the processor will
    - start working on T_b first (work-first policy)
    - stay on T_a , making T_b available for execution by another processor (help-first policy)



work-sharing



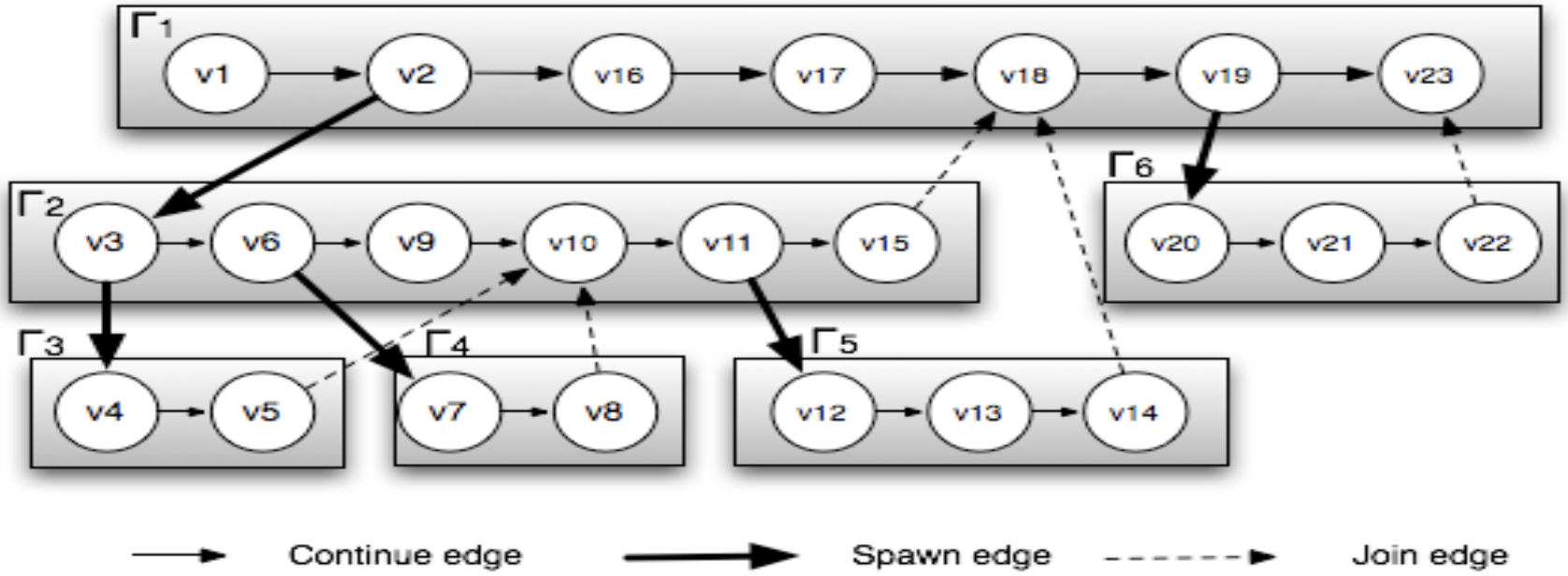
work-stealing

Specifying Scheduling Policies in HJ

- Work-sharing is the default. Normal compilation and execution with hjc and hj commands uses the work-sharing policies
  - Work-sharing supports all parallel constructs in HJ
- Work-stealing can be enabled by an option
  - "hjc –rt w" compiles a program for work-stealing scheduling with the work-first policy
  - "hjc –rt h" compiles a program for work-stealing scheduling with the help-first policy
  - Work-stealing only supports finish, async, and isolated statements
    - Work-stealing support for future get() and phasers is in progress
- In all cases, "hj –places 1:n" creates n workers in 1 place
  - You will learn about places later in the course
  - Caveat: the work-sharing scheduler creates additiona threads if some worker threads get blocked

Context Switch

- Context Switch occurs when the processor
  - Deviates execution from the serial depth-first schedule, AND
  - does not follow continue edges



- Two examples of context switches:
  - Case 1: …..v12 v13 v14 → context switch → v18 …..
  - Case 2: v1 v2 v3 v6 v9 → context switch → v4 v5 ….

Context Switch (cond.)

- Why are context switches expensive?
  - —Execution context needs special handling
  - —Cache may be cold
- When does a context switch occur?
  - —In work-first policy, every steal will trigger a context switch of the victim
  - —In help-first policy, every task is executed after a context switch

Iterative Fork-Join Microbenchmark

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

- 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

Table 1: 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

Adding a Threshold Test for Efficiency

```
void fib (int n) {
    if (n<2) {
        . . .
    } else {
        finish {
            async fib(n-1);
            async fib(n-2);
        }
    }
}
```

```
void fib (int n) {
    if (n<2) {
        . . .
    } else if ( n > THRESHOLD) { //
    PARALLEL VERSION
        finish {
            async fib(n-1);
            async fib(n-2);
        }
    }
    else { // SEQUENTIAL VERSION
            fib(n-1); fib(n-2);
    }
}
```


seq clause in HJ async statement

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

```
void fib (int n) {
    if (n<2) {
        . . .
    } else {
        finish {
            async seq(n <= THRESHOLD) fib(n-1);
            async seq(n <= THRESHOLD) fib(n-2);
        }
    }
}
```