COMP 322: Fundamentals of Parallel Programming

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

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Acknowledgments for Today’s Lecture


• 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)
    – . . .
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

Static & instance fields are shared among tasks
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)
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 additional 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 \(\rightarrow\) context switch \(\rightarrow\) v18 ..... 
  - **Case 2:** v1 v2 v3 v6 v9 \(\rightarrow\) context switch \(\rightarrow\) 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
- $t_{1wf}(k) =$ 1-worker time for work-stealing with work-first policy
- $t_{1hf}(k) =$ 1-worker time for work-stealing with help-first policy
- $t_{1ws}(k) =$ 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)

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

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

```c
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);
        }
    }
}
```