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)

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

Schedule with execution time, $T_2 = 13$

<table>
<thead>
<tr>
<th>Start time</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

- 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
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. }

<table>
<thead>
<tr>
<th>Start time</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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!
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.

1. `finish { // F1`
2. `async A1;` \(\text{WF}\)
3. `finish { // F2`
4. `async A3;` \(\text{WF}\)
5. `async A4;` \(\text{WF}\)
6. `}` \(\text{WF, HF}\)
7. `S5;`
8. `}` \(\text{WF, HF}\)
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

1. \textbf{finish} 
2. \textbf{for} (int \textit{i}=1; \textit{i}<\textit{k}; \textit{i}++) 
3. \textbf{async} \textit{T}_i; // task \textit{i} 
4. \textbf{T}_0; //task 0 
5. \textbf{}}

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

- \textit{k} = number of tasks
- \textbf{t}_s(\textit{k}) = sequential time
- \textbf{t}_1^\text{wf}(\textit{k}) = 1-worker time for work-stealing with work-first policy
- \textbf{t}_1^\text{hf}(\textit{k}) = 1-worker time for work-stealing with help-first policy
- \textbf{t}_1^\text{ws}(\textit{k}) = 1-worker time for work-sharing
- Java-thread(\textit{k}) = create a Java thread for each async
Fork-Join Microbenchmark Measurements
(execution time in micro-seconds)

<table>
<thead>
<tr>
<th>k</th>
<th>$t_s(k)$</th>
<th>$t_{wf}^w(k)$</th>
<th>$t_{hf}^w(k)$</th>
<th>$t_{ws}^w(k)$</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>

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

<table>
<thead>
<tr>
<th>DrHJ compiler option</th>
<th>SUMMARY</th>
<th>Functional limitations</th>
<th>Performance characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>work-sharing (Default option)</td>
<td>Supports full HJ language, but can lead to “max threads” error.</td>
<td>1) Supports full lang</td>
<td>3) Creates additional</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) Supports perf metrics</td>
<td>worker threads when a task blocks</td>
</tr>
<tr>
<td>work-sharing (Fork-Join variant)</td>
<td></td>
<td>1) + 2)</td>
<td>3) + 4) may perform better than work-sharing for recursive parallelism</td>
</tr>
<tr>
<td>work-stealing (Help-First policy)</td>
<td>Supports restricted HJ language, but avoids “max threads” error.</td>
<td>5) Only supports async, finish, forasync, isolated, atomic vars</td>
<td>6) Fixed number of worker threads</td>
</tr>
<tr>
<td>work-stealing (Work-First policy)</td>
<td></td>
<td>5) + 8) Supports data race detection</td>
<td>7) better for loop parallelism</td>
</tr>
<tr>
<td>work-stealing (Adaptive policy)</td>
<td></td>
<td>Same as 5)</td>
<td>10) automatically chooses between help-first and work-first policies</td>
</tr>
</tbody>
</table>
| 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

<table>
<thead>
<tr>
<th>DrHJ compiler option</th>
<th>Command-line option</th>
<th>Functional characteristics</th>
<th>Performance characteristics</th>
</tr>
</thead>
</table>
| **work-sharing**       | 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 |
| (Default option)       |                      |                            |                             |
| **work-sharing**       | Compile: hjc -rt s (default)  
Runtime: hj -fj | 1) + 2) | 3) + 4) may perform better than work-sharing for recursive parallelism |
| (Fork-Join variant)    |                      |                            |                             |
| **work-stealing**      | 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 |
| (Help-First policy)    |                      |                            |                             |
| **work-stealing**      | Compile: hjc -rt w  
Runtime: hj (no option needed) | 5) + 8) Supports data race detection | 6) + 9) better for recursive parallelism |
| (Work-First policy)    |                      |                            |                             |
| **work-stealing**      | 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 |
| (Adaptive policy)      |                      |                            |                             |
| **cooperative**        | Compile: hjc -rt c  
Runtime: hj (no option needed) | Currently supports 5) + Futures --- goal is to support everything! | Same as 6) |
| (under development)    |                      |                            |                             |
Outline of Today’s Lecture

• Abstract vs. Real performance

• seq clause in async statements

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)

1. static int computeSum(int[] X, int lo, int hi) {
2.     if ( lo > hi ) return 0;
3.     else if ( lo == hi ) return X[lo];
4.     else {
5.         int mid = (lo+hi)/2;
6.         final future<int> sum1 =
7.             async<int> { return computeSum(X, lo, mid); };
8.         final future<int> sum2 =
9.             async<int> { return computeSum(X, mid+1, hi); };
10.     // Parent now waits for the container values
11.     return sum1.get() + sum2.get();
12. }
13. } // computeSum
14. int sum = computeSum(X, 0, X.length-1); // main program

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

1. // Non-Future example
2. async seq(size < thresholdSize) computeSum(X, lo, mid);
3.
4. // Future example
5. final future<int> sum1 = async<int> seq(size < thresholdSize)
6.     { 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

1. static int thresholdSize = 1000000;
2. ...
3. static int computeSum(int[] X, int lo, int hi) {
4.   if ( lo > hi ) return 0;
5.   else if ( lo == hi ) return X[lo];
6.   else {
7.     int mid = (lo+hi)/2; size = hi-lo+1;
8.     final future<int> sum1 = async<int> seq(size < thresholdSize)
9.       { return computeSum(X, lo, mid); };
10.    final future<int> sum1 = async<int> seq(size < thresholdSize)
11.       { return computeSum(X, mid+1, hi); };
12.   // Parent now waits for the container values
13.   return sum1.get() + sum2.get();
14. }
15. } // computeSum
16. ...
17. 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

a = [ ]

async

a = [0]

async

a = [1]

cutoff

depth = 1

a = [0 2]

a = [0 3]

a = [1 3]

a = [0 3 1]

a = [1 3 0]

a = [1 3 0 2]
Parallel Solution to NQueens with Finish Accumulator and seq clause

1. static accumulator count;
2. ....
3. count = accumulator.factory.accumulator(SUM, int.class);
4. finish(a) nqueens_kernel(new int[0], 0);
5. System.out.println("No. of solutions = " + count.get().intValue());
6. ....
7. void nqueens_kernel(int [] a, int depth) {
8.    if (size == depth) count.put(1);
9.    else
10.       /* try each possible position for queen at depth */
11.      for (int i = 0; i < size; i++) async seq(depth >= cutoff) {
12.         /* allocate a temporary array and copy array a into it */
13.         int [] b = new int [depth+1];
14.         System.arraycopy(a, 0, b, 0, depth);
15.         b[depth] = i;
16.         if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
17.      } // for-async
18. } // 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)

```
1. // Program Q2
2. A;
3. finish {
4.   async { C; E; }
5.   async F;
6.   async { B; D; }
7. }
```

Work-First Schedule

<table>
<thead>
<tr>
<th>Start time</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name 1: _____________________
Name 2: _____________________
Worksheet #10: Scheduling Program Q2 using a Work-First & Help-First Schedulers (contd)

1. // Program Q2
2. A;
3. finish {
4.     async { C; E; }
5.     async F;
6.     async { B; D; }
7. }

Help-First Schedule

<table>
<thead>
<tr>
<th>Start time</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>