COMP 322: Fundamentals of Parallel Programming

Lecture 9: Abstract vs Real Performance, seq clause, forasync loops

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Goals for Today’s Lecture

• Abstract vs. Real performance
• seq clause in async statements
• forasync loops and “chunking”
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 translates Foo.hj to Foo.class, and inserts calls to HJ runtime as needed

```
hj –places m:n Foo
```

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

```
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 Computation Graph, HJ Abstract Performance Metrics (all enabled by appropriate options)
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

Static & instance fields are *shared* among tasks
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

```java
1. finish { // F1
2.   async A1;
3.   finish { // F2
4.     async A3;
5.     async A4;
6.   }
7.   S5;
8. }
```
Work-Sharing vs. Work-Stealing
Scheduling Paradigms

• **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, better suited for loop parallelism), or
    - start working on $T_b$ first (**work-first** policy, better suited for recursive parallelism)
Context Switch

- Context Switch occurs whenever a processor
  - Deviates execution from sequential execution by not following 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 by worker e.g., save/restore of local variables
  — 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
### Scheduling Policies Currently Available in HJ

<table>
<thead>
<tr>
<th>DrHJ compiler option</th>
<th>Command-line option</th>
<th>Functional limitations</th>
<th>Performance limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>work-sharing</td>
<td>hjc -rt s</td>
<td>None - supports full HJ language</td>
<td>Creates additional worker threads when a task blocks</td>
</tr>
<tr>
<td>(default option)</td>
<td>(default option)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>work-sharing</td>
<td>hj -fj</td>
<td>None - supports full HJ language</td>
<td>May perform better than work-sharing for recursive parallelism</td>
</tr>
<tr>
<td>(Fork-Join)</td>
<td>(Same compiler option as work-sharing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>work-stealing</td>
<td>hjc -rt h</td>
<td>Only supports async, finish, forasync, atomic vars, isolated</td>
<td>Lower overheads, better for loop parallelism</td>
</tr>
<tr>
<td>(Help-First)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>work-stealing</td>
<td>hjc -rt w</td>
<td>Only supports async, finish, forasync, atomic vars, isolated</td>
<td>Lower overheads, better for recursive parallelism</td>
</tr>
<tr>
<td>(Work-First)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>work-stealing</td>
<td>hjc -rt a</td>
<td>Only supports async, finish, forasync, atomic vars, isolated</td>
<td>Lower overheads, automatically chooses between help-first and work-first policies</td>
</tr>
<tr>
<td>(Adaptive)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Iterative Fork-Join Microbenchmark

```java
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
- \(\text{Java-thread}(k)\) = 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>$t_{1wf}^w(k)$</th>
<th>$t_{1hf}^w(k)$</th>
<th>$t_{1ws}^w(k)$</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>

Help-First may perform better than Work-First on this microbenchmark if the number of workers is increased to > 1.
Goals for Today’s Lecture

- Abstract vs. Real performance
- `seq` clause in async statements
- `forasync` loops and “chunking”
Adding a Threshold Test for Efficiency

```java
1. void fib (int n) {
2.   if (n<2) {
3.     ...
4. } else {
5.     finish {
6.       async fib(n-1);
7.       async fib(n-2);
8.    } // finish
9. } // if-else
10.} // fib()
```

```java
1. void fib (int n) {
2.   if (n<2) {
3.     ...
4. } else if (n > THRESHOLD) {
5.     // PARALLEL VERSION
6.     finish {
7.       async fib(n-1);
8.       async fib(n-2);
9.    } // finish
10. } else {
11.     fib(n-1); fib(n-2);
12. } // if-else-else
13.} // fib()
```
seq clause in HJ async statement

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

- seq clause specifies condition under which async should be executed sequentially

1. void fib (int n) {
2.     if (n<2) {
3.         . . .
4.     } else {
5.         finish {
6.             async seq(n <= THRESHOLD) fib(n-1);
7.             async seq(n <= THRESHOLD) fib(n-2);
8.         }
9.     } // if-else
10.} // fib()
Example of seq clause: nqueens.hj

1. void nqueens_kernel(int [] a, int depth) {
2.     if (size == depth) {
3.         total_count.addAndGet(1); // Add to solution count
4.         return;
5.     }
6.     /* try each possible position for queen at depth */
7.     for (int i = 0; i < size; i++) {
8.         async seq(depth >= cutoff_value) {
9.             /* allocate a temporary array and copy a[] into it */
10.            int [] b = new int [depth+1];
11.            System.arraycopy(a, 0, b, 0, depth);
12.            b[depth] = i;
13.            if (ok( (depth + 1), b))
14.                nqueens_kernel(b, depth+1);
15.         }
16.     }
17. }
Goals for Today’s Lecture

• Abstract vs. Real performance
• seq clause in async statements
• forasync loops and “chunking”
HJ’s pointwise for & forasync statements

Goal: capture common for-asasync pattern in a single construct for multidimensional loops e.g., replace

```java
finish {
    for (int I = 0 ; I < N ; I++)
        for (int J = 0 ; J < N ; J++)
            async
                for (int K = 0 ; K < N ; K++)
}
```

by

```java
finish forasync (point [I,J] : [0:N-1,0:N-1])
    for (point[K] : [0:N-1])
```
Observations

• Combination of for–async is replaced by a single keyword, forasync

• Multiple loops can be collapsed into a single forasync, with a multi-dimensional iteration space.

• Iteration variable for a forasync is a point (integer tuple), such as [I,J]

• Loop bounds can be specified as a rectangular region (dimension ranges) such as [0:N-1,0:N-1]

• HJ also extends the sequential for statement so as to iterate sequentially over a rectangular region
  —Simplifies conversion between for and forasync
hj.lang.point, an index type for multi-dimensional loops

- A point is an element of an n-dimensional Cartesian space (n \geq 1) with integer-valued coordinates e.g., [5], [1, 2], ...
  - Dimensions of a point are numbered from 0 to n-1
  - n is also referred to as the rank of the point

- A point variable can hold values of different ranks e.g.,
  - point p; p = [1]; ... p = [2,3]; ...

- The following operations are defined on point-valued expression p1
  - p1.rank --- returns rank of point p1
  - p1.get(i) --- returns element i of point p1
    - Returns element (i mod p1.rank) if i < 0 or i \geq p1.rank
  - p1.lt(p2), p1.le(p2), p1.gt(p2), p1.ge(p2)
    - Returns true iff p1 is lexicographically <, <=, >, or >= p2
    - Only defined when p1.rank and p1.rank are equal
```java
public class TutPoint {
    public static void main(String[] args) {
        point p1 = [1,2,3,4,5];
        point p2 = [1,2];
        point p3 = [2,1];
        System.out.println("p1 = " + p1 + " ; p1.rank = " + p1.rank + " ; p1.get(2) = " + p1.get(2));
        System.out.println("p2 = " + p2 + " ; p3 = " + p3 + " ; p2.lt(p3) = " + p2.lt(p3));
    }
}
```

**Console output:**

```
p1 = [1,2,3,4,5] ; p1.rank = 5 ; p1.get(2) = 3
p2 = [1,2] ; p3 = [2,1] ; p2.lt(p3) = true
```
A region is the set of points contained in a rectangular subspace.

A region variable can hold values of different ranks e.g.,
- \( \text{region } R; \ R = [0:10]; \ldots \ R = [-100:100, -100:100]; \ldots \ R = [0:-1]; \ldots \)

Operations
- \( R.\text{rank} ::= \# \text{ dimensions in region}; \)
- \( R.\text{size}() ::= \# \text{ points in region} \)
- \( R.\text{contains}(P) ::= \text{ predicate if region } R \text{ contains point } P \)
- \( R.\text{contains}(S) ::= \text{ predicate if region } R \text{ contains region } S \)
- \( R.\text{equal}(S) ::= \text{ true if region } R \text{ equals region } S \)
- \( R.\text{rank}(i) ::= \text{ projection of region } R \text{ on dimension } i \) (a one-dimensional region)
- \( R.\text{rank}(i).\text{low}() ::= \text{ lower bound of } i^{th} \text{ dimension of region } R \)
- \( R.\text{rank}(i).\text{high}() ::= \text{ upper bound of } i^{th} \text{ dimension of region } R \)
- \( R.\text{ordinal}(P) ::= \text{ ordinal value of point } P \text{ in region } R \)
- \( R.\text{coord}(N) ::= \text{ point in region } R \text{ with ordinal value } = N \)
Summary of forasync statement

forasync (point [i1] : [lo1:hi1]) <body>
forasync (point [i1,i2] : [lo1:hi1,lo2:hi2]) <body>
forasync (point [i1,i2,i3] : [lo1:hi1,lo2:hi2,lo3:hi3]) <body>

• forasync statement creates multiple async child tasks, one per iteration of the forasync
  — all child tasks can execute <body> in parallel
  — child tasks are distinguished by index “points” ([i1], [i1,i2], …)

• <body> can read local variables from parent (copy-in semantics like async)

• forasync needs a finish for termination, just like regular async tasks
  — Later, we will learn about replacing “finish forasync” by “forall”
Pointwise sequential for loop

- HJ extends Java’s for loop to support sequential iteration over points in region R in canonical lexicographic order
  - for ( point p : R ) . . .

- Standard point operations can be used to extract individual index values from point p
  - for ( point p : R ) { int i = p.get(0); int j = p.get(1); . . . }

- Or an “exploded” syntax is commonly used instead of explicitly declaring a point variable
  - for ( point [i,j] : R ) { . . . }

- The exploded syntax declares the constituent variables (i, j, ...) as local int variables in the scope of the for loop body
forasync examples: updates to a two-dimensional Java array

// Case 1: loops i,j can run in parallel
forasync (point[i,j] : [0:m-1,0:n-1]) A[i][j] = F(A[i][j]) ;

// Case 2: only loop i can run in parallel
forasync (point[i] : [1:m-1])
    for (point[j] : [1:n-1]) // Equivalent to “for (j=1;j<n;j++)”
        A[i][j] = F(A[i][j-1]) ;

// Case 3: only loop j can run in parallel
for (point[i] : [1:m-1]) // Equivalent to “for (i=1;i<m;j++)”
    finish forasync (point[j] : [1:n-1])
        A[i][j] = F(A[i-1][j]) ;
One-Dimensional Iterative Averaging Example

- Initialize a one-dimensional array of \((n+2)\) double's with boundary conditions, \(\text{myVal}[0] = 0\) and \(\text{myVal}[n+1] = 1\).

- In each iteration, each interior element \(\text{myVal}[i]\) in \(1..n\) is replaced by the average of its left and right neighbors.
  - Two separate arrays are used in each iteration, one for old values and the other for the new values

- After a sufficient number of iterations, we expect each element of the array to converge to \(\text{myVal}[i] = i/(n+1)\)
  - In this case, \(\text{myVal}[i] = (\text{myVal}[i-1] + \text{myVal}[i+1])/2\), for all \(i\) in \(1..n\)

![Illustration of an intermediate step for \(n = 8\) (source: Figure 6.19 in Lin-Snyder book)](image)
HJ code for One-Dimensional Iterative Averaging using nested for-finish-forasync structure

1. for (point [iter] : [0:iterations-1]) {
2.   // Compute MyNew as function of input array MyVal
3.   finish forasync (point [j] : [1:n]) { // Create n tasks
4.     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
5.   } // finish forasync
6.   temp=myVal; myVal=myNew; myNew=temp; // Swap myVal & myNew;
7.   // myNew becomes input array for next iteration
8. } // for

• How many tasks does this version create?
• This is an idealized version with no “chunking” of forasync iterations
Chunking of forasync loops for efficiency

// Original forasync loop iterates over region R
forasync (point \([i,j] : R\)) <body>

// Chunked forasync loop iterates over Ci*Cj chunks with
// point \([ii,jj] \) in region chunks(R,\([Ci,Cj]\)).
// Forasync body contains inner for loop iterating over
// myChunk(R,\([ii,jj]\))
forasync (point \([ii,jj] : \) chunks(R,\([Ci,Cj]\)))

for (point \([i,j] : \) myChunk(R,\([ii,jj]\)))
Example: HJ code for One-Dimensional Iterative Averaging with chunked for-finish-forasync-for

```java
1. for (point [iter] : [0:iterations-1]) {
2.   // Compute MyNew as function of input array MyVal
3.   int Cj = ...; // Set to desired number of chunks
4.   int iters = (n+Cj-1)/Cj; // Max iterations per chunk
5.   finish forasync (point [jj]:[1:Cj]) {
6.     for (point [j]:[1+(jj-1)*iters : Math.min(jj*iters,n)])
7.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
8.   } // finish forasync
9.   temp=myVal; myVal=myNew; myNew=temp; // Swap myVal & myNew;
10.  // myNew becomes input array for next iteration
11.} // for
```

- How many tasks does this chunked version create?