
COMP 322: Principles of Parallel Programming

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



Summary of Last Lecture

- Latency
- Throughput / Bandwidth
- Little's Law
- Sources of Performance Loss (ONIC):
 1. Overhead
 2. Non-parallelizable code
 3. Idleness
 4. Contention
- Memory hierarchy and caches
- Homework #1 due in class by Sep 10th

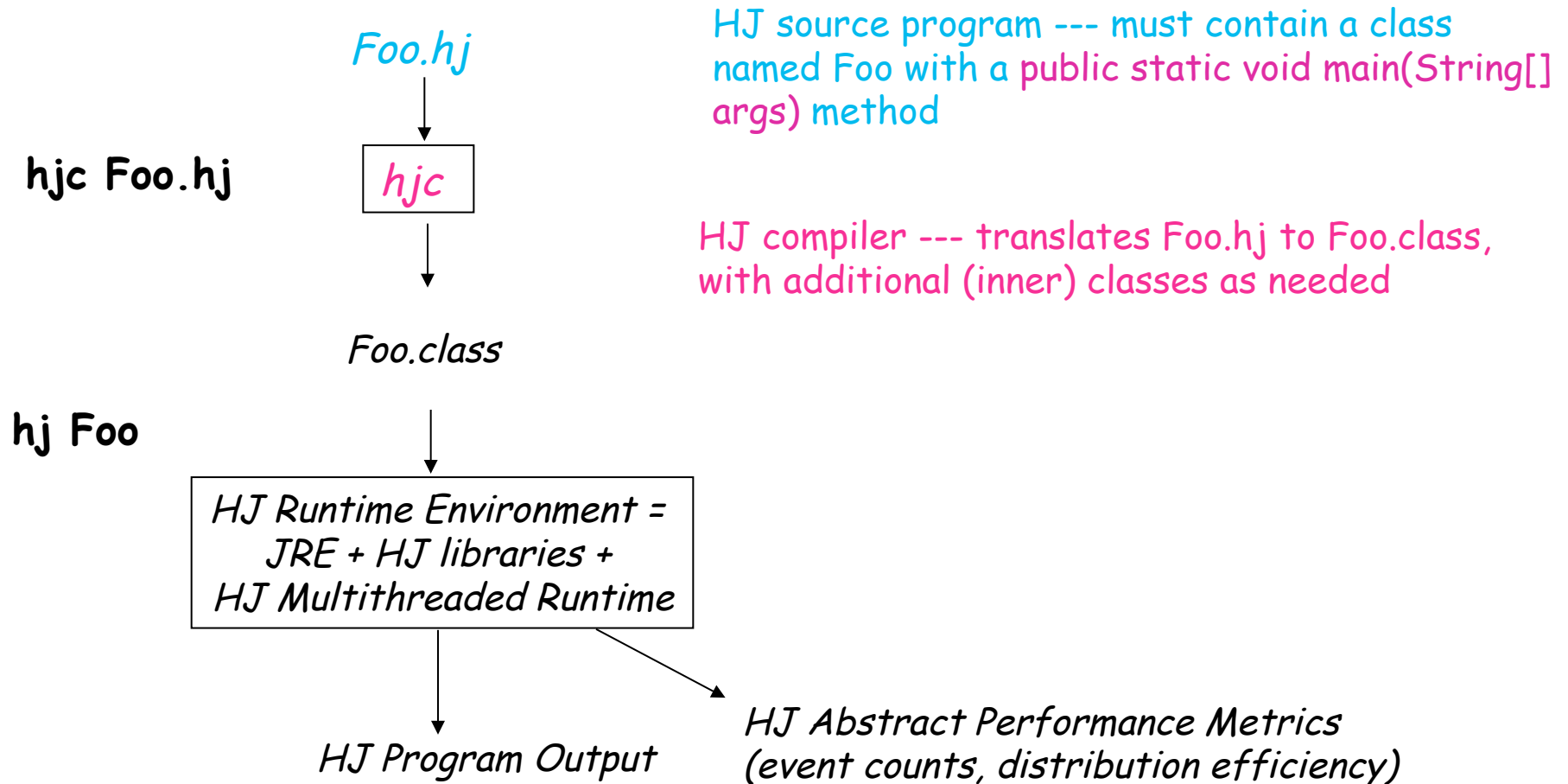
Acknowledgments for Today's Lecture

- Course text: "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
 - Includes resources available at <http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html>
- X10 tutorial given at PLDI 2007 conference, Vijay Saraswat, Vivek Sarkar, Nathaniel Nystrom

Habanero-Java (HJ) Language

- New language and implementation developed in Habanero Multicore Software research project at Rice (habanero.rice.edu)
 - Based on X10 v1.5 research language from IBM
 - Extension of Java 1.4
 - Java 5 & 6 language features (generics, metadata, etc.) are currently not supported in HJ
 - However, Java 5 & 6 libraries can be used
 - Just don't call a method that performs a blocking operation
- HJ extensions are focused on parallel programming
 1. Dynamic task creation & termination: `forall`, `async`, `finish`, `force`
 2. Mutual exclusion and isolation: `isolated`
 3. Collective and point-to-point synchronization: `phaser`, `next`
 4. Locality control --- task and data distributions: `places`, `here`

HJ Compilation and Execution Environment



Caveat: this is a research prototype implementation with many limitations. Please be patient, and report all bugs and issues!

HJ's forall statement

`forall (point [i1] : [lo1:hi1]) <body>`

`forall (point [i1,i2] : [lo1:hi1,lo2:hi2]) <body>`

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

. . .

- Main program starts as a sequential task
- forall statement creates multiple child tasks, one per iteration of the forall
 - all child tasks can execute <body> in parallel
 - child tasks are distinguished by index “points” ([i1], [i1,i2], ...)
- forall statement completes (and parent task proceeds to the following statement) when all child tasks have completed
- <body> can only access final local variables from parent
- NOTE: HJ's forall statement has slightly different syntax from Peril-L's forall statement in textbook

Array Sum Example Revisited (parallel iterative version)

- Parallel algorithm (iterative version, assumes n is a power of 2)

```
for ( step = 1; step < n ; step *= 2 ) {  
    final int size = n / (2*step);  
    final int step_f = step;  
    forall ( point [i] : [0:size-1] ) X[2*i*step_f] += X[(2*i+1)*step_f];  
}  
sum = X[0];
```
- HJ forall construct executes all iterations in parallel
 - forall body can only access outer local variables that are final
- This algorithm overwrites X (make a copy if X is needed later)
- Work = $O(n)$, Span = $O(\log n)$, Parallelism = $O(n / (\log n))$
- NOTE: this and the next parallel algorithm can be used for any associative operation on array elements (need not be commutative) e.g., multiplication of an array of matrices

Points

- A *point* is an element of an n -dimensional Cartesian space ($n \geq 1$) with integer-valued coordinates e.g., [5], [1, 2], ...
 - Dimensions 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 a point-valued expression $p1$
 - $p1.rank$ --- returns rank of point $p1$
 - $p1.get(i)$ --- returns element i of point $p1$
 - Returns element $(i \bmod p1.rank)$ 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 $<$, \leq , $>$, or \geq $p2$
 - Only defined when $p1.rank$ and $p2.rank$ are equal

Example

```
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));
    } // main()
} // TutPoint
```

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
```

forall examples: updates to a two-dimensional Java array

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

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

- What does the computation graph look like for each case?
- What is the impact of overhead on each case?

Pointwise 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 can be used instead of explicitly declaring a point variable
 - `for (point [i,j] : R) { ... }`
- The exploded syntax declares the constituent variables (i, j, \dots) as local int variables in the scope of the for loop body

Example (see TutFor.x10)

```
public class TutFor {
    public static void main(String[] args) {
        region R = [0:1,0:2];
        System.out.print("Points in region " + R + " =");
        for ( point p : R ) System.out.print(" " + p);
        System.out.println();
        // Use exploded syntax instead
        System.out.print("(i,j) pairs in region " + R + " =");
        for ( point[i,j] : R )
            System.out.print("(" + i + "," + j + ")");
        System.out.println();
    } // main()
} // TutFor
```

Console output:

```
Points in region {0:1,0:2} = [0,0] [0,1] [0,2] [1,0] [1,1] [1,2]
(i,j) pairs in region {0:1,0:2} =(0,0) (0,1) (0,2) (1,0) (1,1) (1,2)
```

Figure 4.4 Odd/Even Interchange to alphabetize a list L of records on field x ($Peril-L$)

```
1  bool continue=true;
2  rec L[n];
3  while(continue) do
4  {
5      forall(i in(1:n-2:2))
6      {
7          rec temp;
8          if(strcmp(L[i].x,L[i+1].x)>0)
9          {
10             temp=L[i];
11             L[i]=L[i+1];
12             L[i+1]=temp;
13         }
14     }
15     forall(i in(0:n-2:2))
16     {
17         rec temp;
18         bool done = true;
19         if(strcmp(L[i].x,L[i+1].x)>0)
20         {
21             temp=L[i];
22             L[i]=L[i+1];
23             L[i+1]=temp;
24             done=false;
25         }
26         continue=(&&/done);
27     }
28 }
```

The data is global

Stride by 2

Is odd/even pair misordered?

Yes, fix

Stride by 2

Set up for termination test

Is even/odd pair misordered?

Yes, interchange

Not done yet

Were any changes made?

Modified Hello World Example

```
rank.count = 0; // rank object contains an int field, id
forall (point[i] : [0:m-1]) {
    int r = rank.count++;
    StringArray[i] = "Hello, World from task with rank = " + r; }
```

- **Problem:** what happens when two forall iterations perform `rank.count++` in parallel?
 - Race condition! Result is undefined in general.

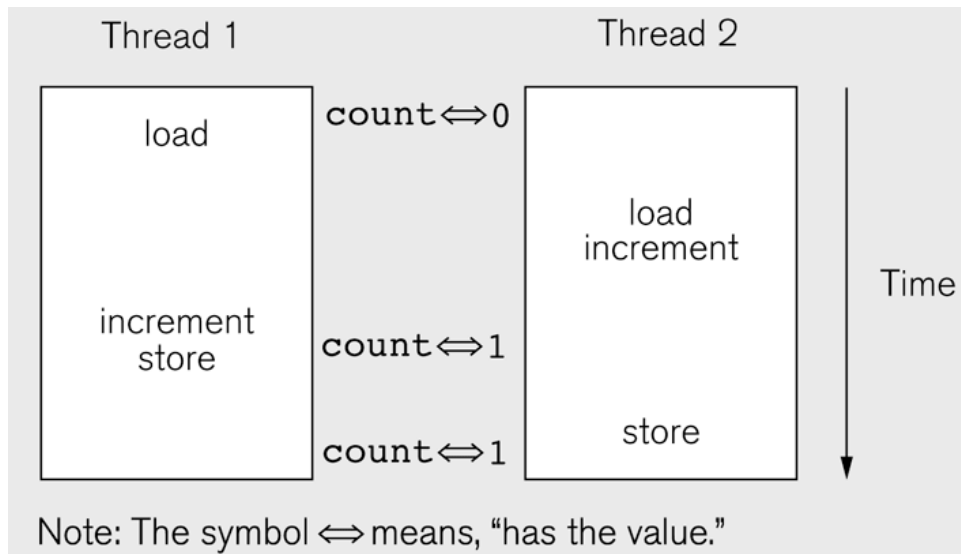


Figure 1.8 One of several possible interleavings of references to the unprotected variable `count`, illustrating a race condition.

isolated statement

`isolated <body>`

- Only one task can execute an isolated statement at a time
 - `<body>` is executed in isolation of other instances of `<body>`
 - Guarantees *mutual exclusion* between any pair of isolated statement instances
 - No guarantee applies to non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested; use of isolated for inner statements is redundant
- Isolated statements must not contain any other parallel statement: `forall`, `async`, `finish`, `force`, `next`
- **NOTE:** HJ's isolated statement is same as Peril-L's exclusive statement

Modified Hello World Example Revisited

```
rank.count = 0; // rank object contains an int field, id
forall (point[i] : [0:m-1]) {
    int r;
    isolated {r = rank.count++;}
    StringArray[i] = "Hello, World from task with rank = " + r;
}
```

- Isolated statements can be used to fix race conditions as shown above, but their use can also decrease parallelism
- Let T_I = total time spent in all isolated statements
- Can you prove the following lower and upper bounds?
– $\max(T_1/P, T_\infty, T_I) \leq T_P \leq T_1/P + T_\infty + T_I$