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

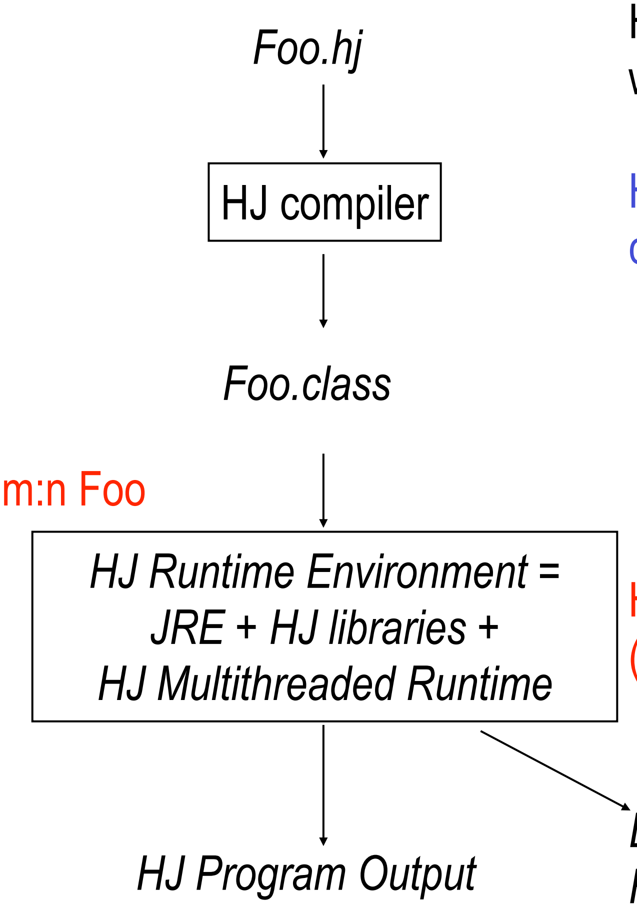
https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Goals for Today's Lecture

- <u>Abstract vs. Real performance</u>
- 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

HJ compiler translates Foo.hj to Foo.class, and inserts calls to HJ runtime as needed

Foo.class

hj –places m:n Foo

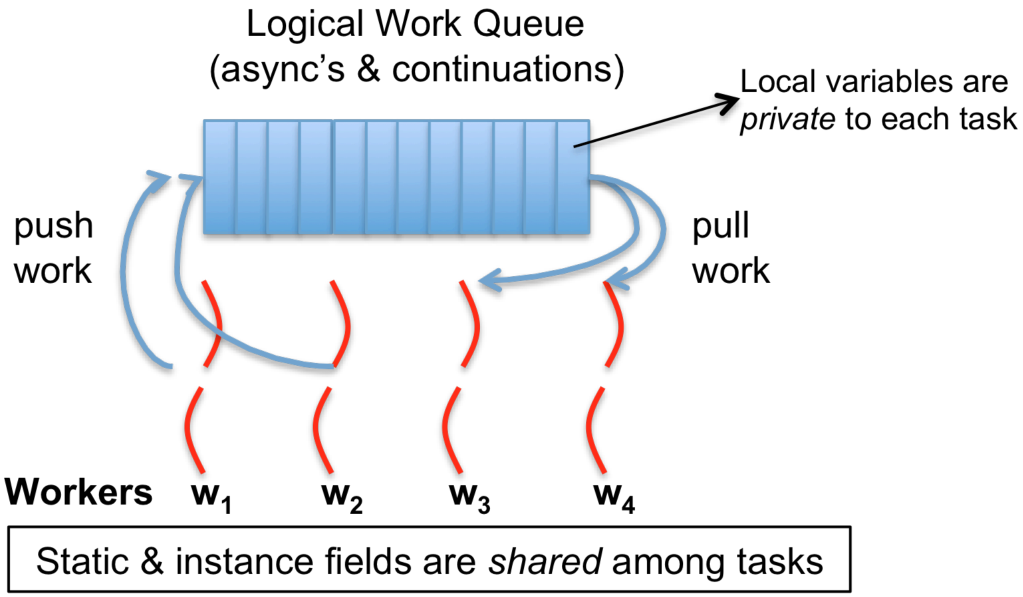
HJ Runtime Environment =
JRE + HJ libraries +
HJ Multithreaded Runtime

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



Logical Work Queue
(async's & continuations)

Local variables are *private* to each task

push work

pull work

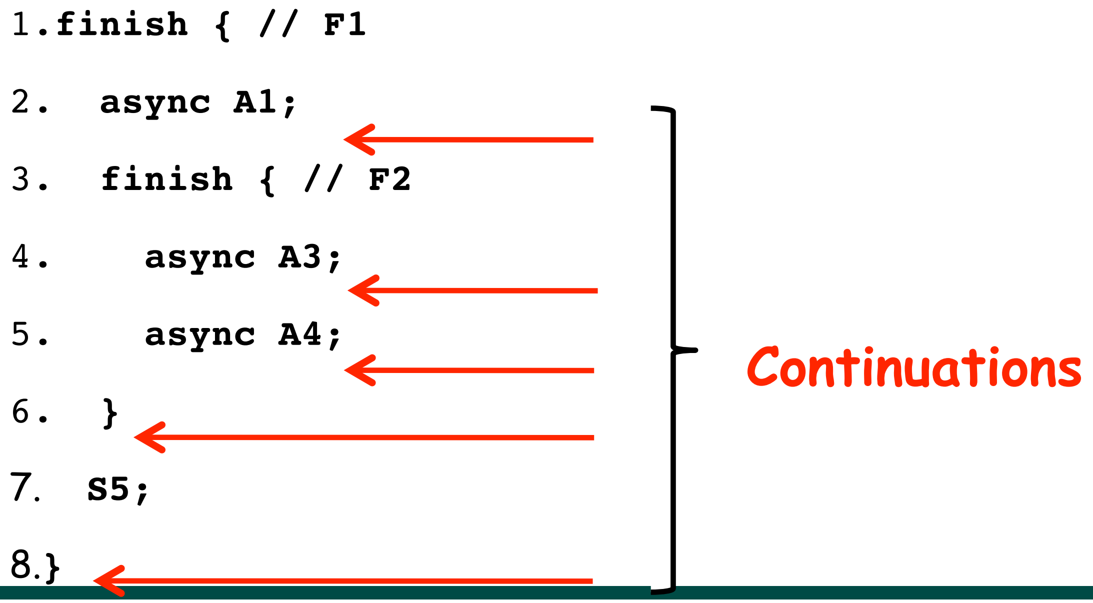
Workers w_1 w_2 w_3 w_4

Static & instance fields are *shared* among tasks

- 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

Continuations

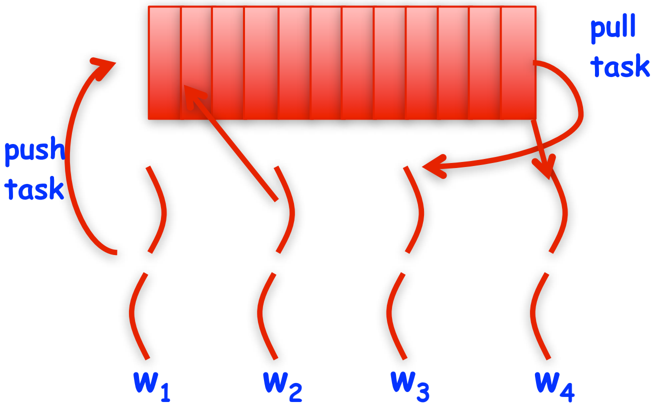
- A continuation is one of two kinds of program points
  - The point in the parent task immediately following an async
  - The point immediately following an end-finish or a future get()
- Continuations are also referred to as task-switching points
  - Program points at which a worker may switch execution between different tasks



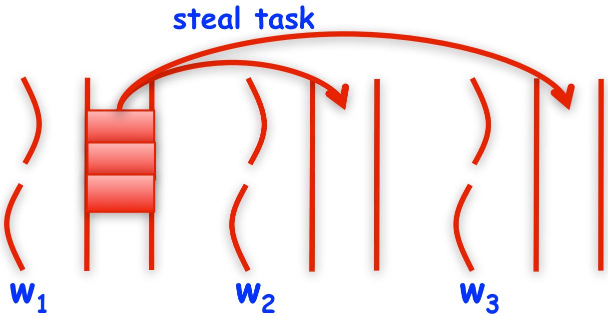
```
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
 S5;
}
```

Work-Sharing vs. Work-Stealing Scheduling Paradigms

- <u>**Work-Sharing**</u>
  - Busy worker eagerly distributes new work
  - Easy implementation with global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck
- <u>**Work-Stealing**</u>
  - 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 (<u>help-first</u> policy, better suited for loop parallelism), or
    - start working on T_b first (<u>work-first</u> policy, better suited for recursive parallelism)



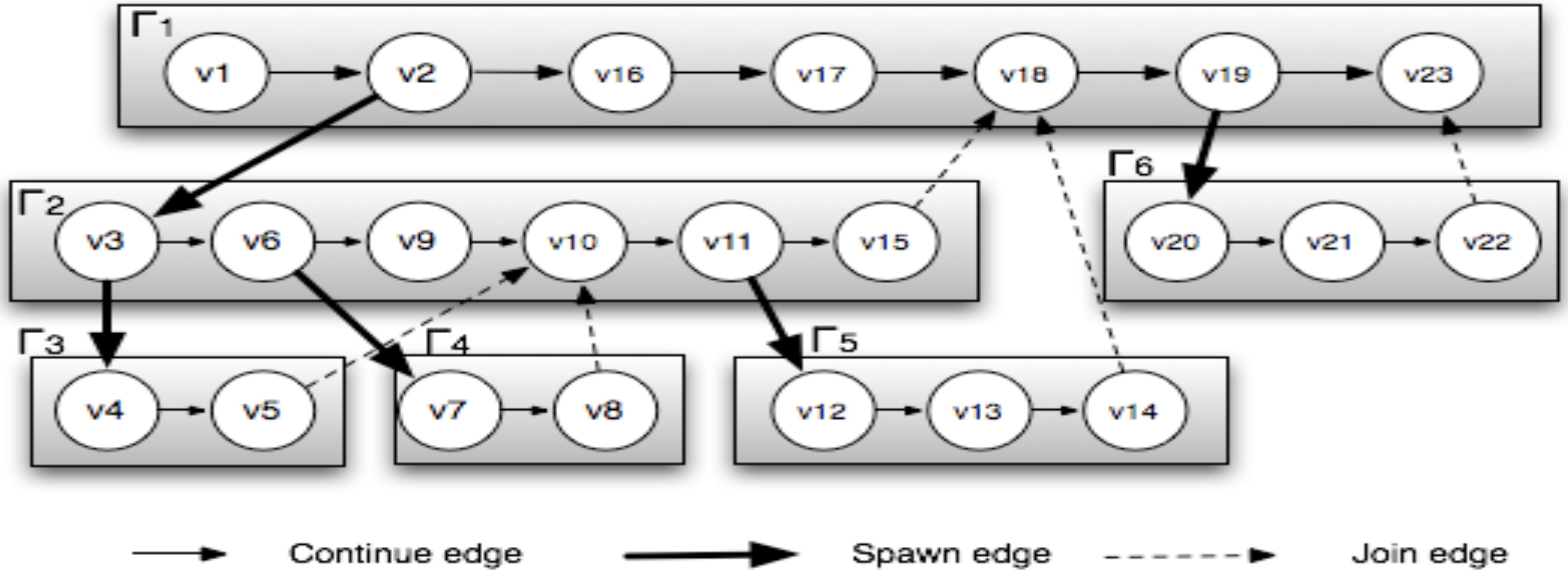
work-sharing



work-stealing

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

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

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

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

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

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

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

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

Example of seq clause: nqueens.hj

```
void nqueens_kernel(int [] a, int depth) {
   if (size == depth) {
     total_count.addAndGet(1); // Add to solution count
     return;
   }
   /* try each possible position for queen at depth */
   for (int i =  0; i < size; i++) {
     async seq(depth >= cutoff_value) {
       /* allocate a temporary array and copy a[] into it */
       int [] b = new int [depth+1];
       System.arraycopy(a, 0, b, 0, depth);
       b[depth] = i;
       if (ok( (depth +  1), b))
         nqueens_kernel(b, depth+1);
     }
   }
 }
```

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-async pattern in a single construct for multidimensional loops e.g., replace

```
finish {
  for (int I = 0 ; I < N ; I++)
    for (int J = 0 ; J < N ; J++)
      async
        for (int K = 0 ; K < N ; K++)
          C[I][J] += A[I][K] * B[K][J];
}
```

by

```
finish forasync (point [I,J] : [0:N-1,0:N-1])
  for (point[K] : [0:N-1])
    C[I][J] += A[I][K] * B[K][J];
```

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>=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 >= 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

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
```

hj.lang.region, a rectangular iteration space for multi-dimensional loops

A region is the set of *points* contained in a rectangular subspace

A region variable can hold values of different ranks e.g.,

- region R; R = [0:10]; … R = [-100:100, -100:100]; … R = [0:-1]; …

Operations

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

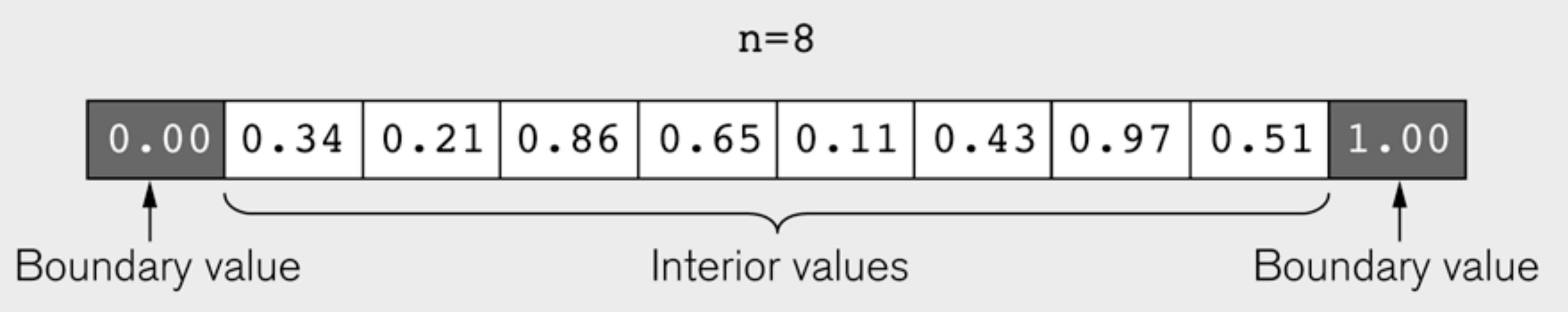
```
// Case 1: loops i,j can run in parallel
forasync (point[i,j] : [0:m-1,0:n-1]) A[i][j] = F(A[i][j]) ;


// 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, myVal[0] = 0 and myVal[n+1] = 1.
- In each iteration, each interior element 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 myVal[i] = i/(n+1)
  - In this case, myVal[i] = (myVal[i-1]+myVal[i+1])/2, for all i in 1..n



HJ code for One-Dimensional Iterative Averaging using nested for-finish-forasync structure

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

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

- How many tasks does this chunked version create?