COMP 322: Parallel and Concurrent Programming

Lecture 31: Loop Parallelism

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Data Parallelism vs. Task Parallelism

- ***Data parallelism***: simultaneous execution of the same code across the elements of a data set
- ***Task parallelism***: simultaneous execution of multiple and different pieces of code across the same or different data sets

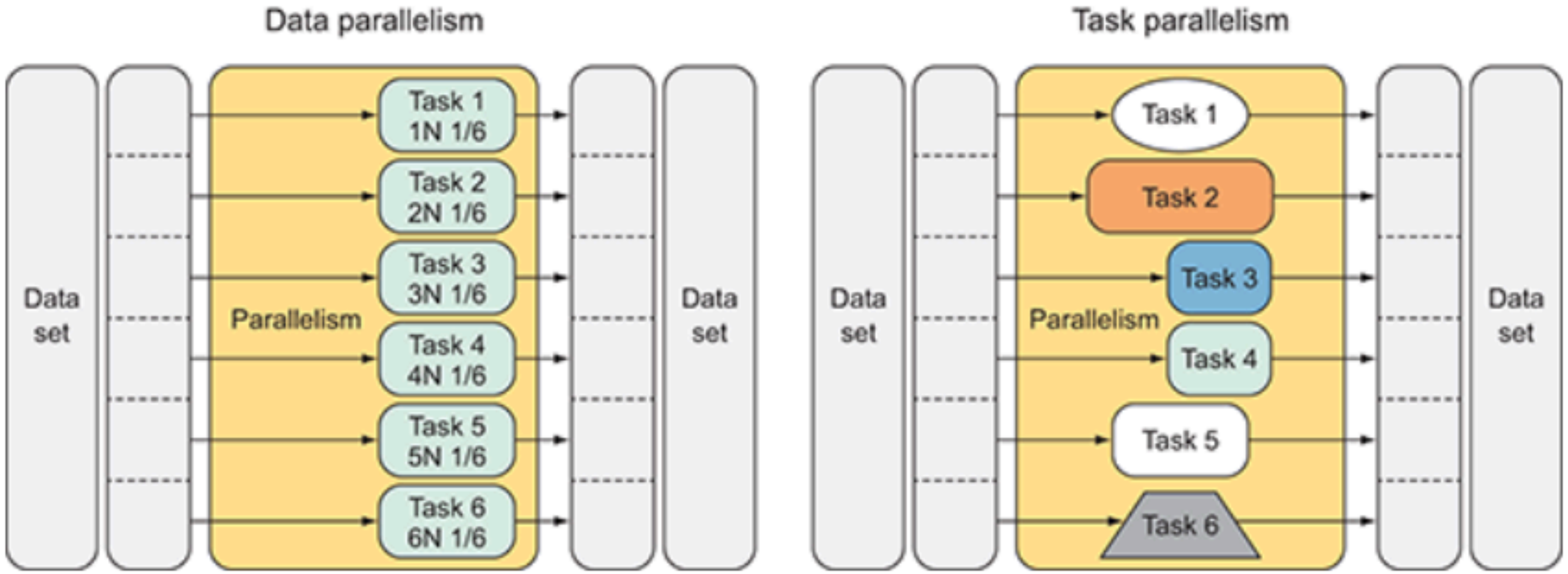
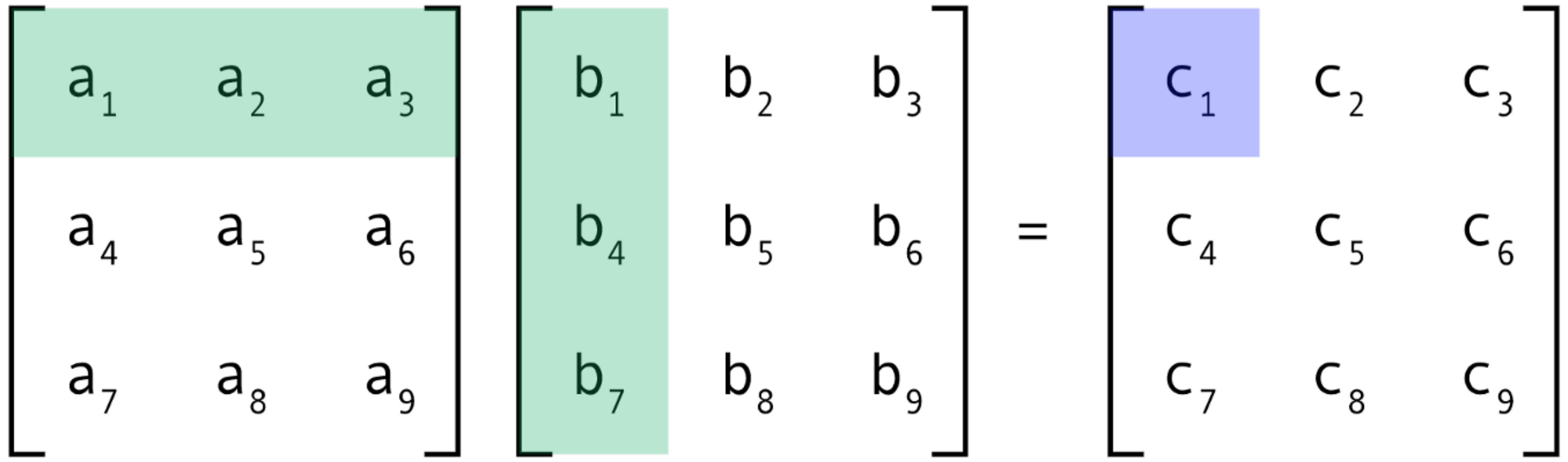


Image source: https://livebook.manning.com/concept/net/task-parallelism

Sequential Algorithm for Matrix Multiplication

```
// Sequential version
 for (int i = 0 ; i < n ; i++)
   for (int j = 0 ; j < n ; j++)
     c[i][j] = 0;
 for (int i = 0 ; i < n ; i++)
   for (int j = 0 ; j < n ; j++)
     for (int k = 0 ; k < n ; k++)
       c[i][j] += a[i][k] * b[k][j];
  // Print first element of output matrix
println(c[0][0]);
```

$$c[i,j] = \sum_{0 \le k < n} a[i,k] * b[k,j]$$



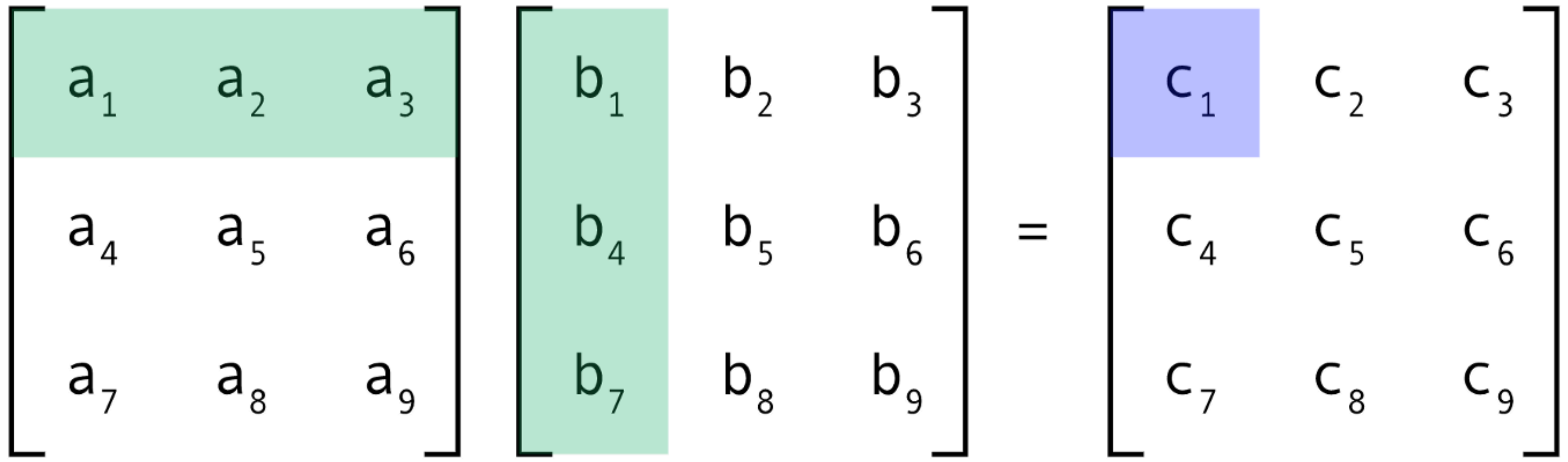
$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Image source: https://medium.com/ai%C2%B3-theory-practice-business/fastai-partii-lesson08-notes-fddcdb6526bb

Parallelizing loops in Matrix Multiplication using finish & async

```
// Parallel version using finish & async
finish(() → {
 for (int ii = 0 ; ii < n ; ii++)
  for (int jj = 0 ; jj < n ; jj++) {
   final int i = ii; final int j = jj;
   async(() → {c[i][j] = 0; });
  }
});
finish(() → {
 for (int ii = 0 ; ii < n ; ii++)
  for (int jj = 0 ; jj < n ; jj++){
   final int i = ii; final int j = jj;
   async(() → {
    for (int k = 0 ; k < n ; k++)
     c[i][j] += a[i][k] * b[k][j];
   });
  }
});
// Print first element of output matrix
println(c[0][0])
```

$$c[i,j] = \sum_{0 \le k < n} a[i,k] * b[k,j]$$



$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Observations on finish-for-async version

- `finish` and `async` are general constructs, and are not specific to loops
  - Not easy to discern from a quick glance which loops are sequential vs. parallel

- Loops in sequential version of matrix multiplication are "perfectly nested"
  - e.g., no intervening statement between "`for`(i = ...)" and "`for`(j = ...)"

- The ordering of loops nested between `finish` and `async` is arbitrary
  - They are parallel loops and their iterations can be executed in any order

Parallelizing loops in Matrix Multiplication example using forall

```
// Parallel version using forall
 forall(0, n-1, 0, n-1, (i, j) → {
     c[i][j] = 0;
   });
 forall(0, n-1, 0, n-1, (i, j) → {
    forseq(0, n-1, (k) → {
       c[i][j] += a[i][k] * b[k][j];
     });
  });
// Print first element of output matrix
println(c[0][0]);
```

$$c[i,j] = \sum_{0 \leq k < n} a[i,k] * b[k,j]$$

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

forall API's in HJlib (http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/Module1.html)

- `static void forall(edu.rice.hj.api.HjRegion.HjRegion1D hjRegion, edu.rice.hj.api.HjProcedureInt1D body)`

- `static void forall(edu.rice.hj.api.HjRegion.HjRegion2D hjRegion, edu.rice.hj.api.HjProcedureInt2D body)`

- `static void forall(edu.rice.hj.api.HjRegion.HjRegion3D hjRegion, edu.rice.hj.api.HjProcedureInt3D body)`

forall API’s in HJlib (http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/Module1.html)

-
  ```
  static void forall(int s0, int e0,
  edu.rice.hj.api.HjProcedure<java.lang.Integer> body)
  ```

-
  ```
  static void forall(int s0, int e0, int s1, int e1,
  edu.rice.hj.api.HjProcedureInt2D body)
  ```

-
  ```
  static <T> void forall(java.lang.Iterable<T> iterable,
  edu.rice.hj.api.HjProcedure<T> body)
  ```

- NOTE: all `forall` API’s include an implicit `finish`. `forasync` is like `forall`, but without the `finish`. Also e0 is the “end” value, not 1 + end value.

Observations on forall version

- The combination of perfectly nested finish-for–for–async constructs is replaced by a single API, `forall`
  - `forall` includes an implicit `finish`
- Multiple loops can be collapsed into a single `forall` with a multi-dimensional iteration space (can be 1D, 2D, 3D, ...)
- The iteration variable for a `forall` is a `HjPoint` (integer tuple), e.g., (i,j) is a 2-dimensional point
- The loop bounds can be specified as a rectangular `HjRegion` (product of dimension ranges), e.g., (0:n−1) x (0:n−1)
- HJlib also provides a sequential `forseq` API that can also be used to iterate sequentially over a rectangular region
  - Simplifies conversion between `forseq` and `forall`

forall examples: updates to two-dimensional Java array

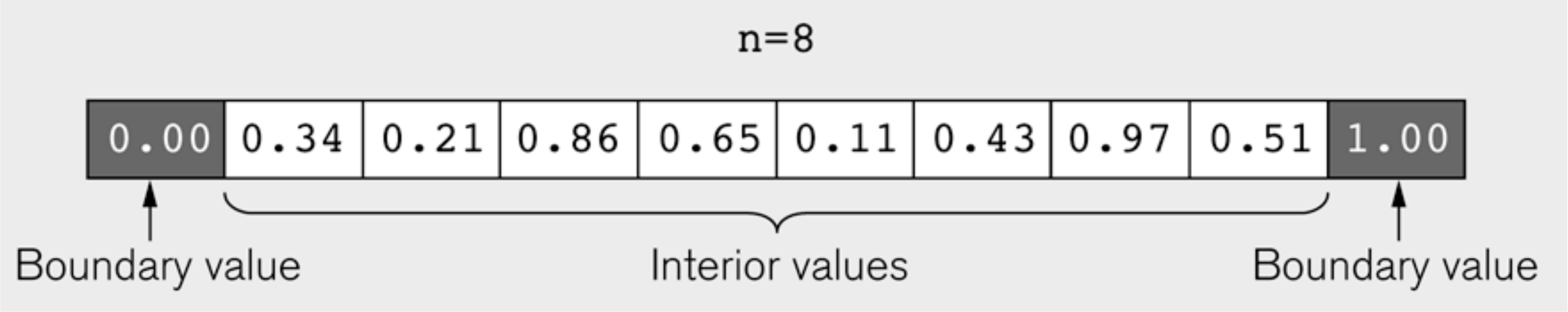
```
// Case 1: loops i,j can run in parallel
forall(0, m-1, 0, n-1, (i, j) -> { A[i][j] = F(A[i][j]);});

// Case 2: only loop i can run in parallel
forall(0, m-1, (i) -> {
  forseq(0, n-1, (j) -> { // Equivalent to “for (j=0;j<n;j++)”
     A[i][j] = F(A[i][j-1]) ;
}); });

// Case 3: only loop j can run in parallel
forseq(0, m-1, (i) -> { // Equivalent to “for (i=0;i<m;i++)”
  forall(0, n-1, (j) -> {
     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] = (myVal[i-1]+myVal[i+1])/2, for all i in 1..n



0.00	0.34	0.21	0.86	0.65	0.11	0.43	0.97	0.51	1.00

llustration of an intermediate step for n = 8 (source: Figure 6.19 in Lin-Snyder book)

Sequential code for One-Dimensional Iterative Averaging

```
1. // Intialize m, n, myVal, newVal
2. m = … ; n = … ;
3. float[] myVal = new float[n+2];
4. float[] myNew = new float[n+2];
5. forseq(0, m-1, (iter) → {
6.   // Compute MyNew as function of input array MyVal
7.     forseq(1, n, (j) → { // Create n tasks
8.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.     }); // forseq
10.   // What is the purpose of line 11 below?
11.   float[] temp=myVal; myVal=myNew; myNew=temp;
12.}); // forseq
```

14.QUESTION: can either forseq() loop execute in parallel?

HJ code for One-Dimensional Iterative Averaging

```
1. // Intialize m, n, myVal, newVal
2. m = … ; n = … ;
3. float[] myVal = new float[n+2];
4. float[] myNew = new float[n+2];
5. forseq(0, m-1, (iter) → {
6.   // Compute MyNew as function of input array MyVal
7.     forall(1, n, (j) → { // Create n tasks
8.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
9.     }); // forall
10.  // What is the purpose of line 11 below?
11.  float[] temp=myVal; myVal=myNew; myNew=temp;
12.}); // forseq
```

What about the overhead?

- It is inefficient to create forall iterations in which each iteration (async task) does very little work
- An alternate approach is "iteration grouping" or "loop chunking"

e.g., replace

```
   forall(0, 99, (i) → BODY(i)); // 100 tasks
```

with

```
      forall(0, 3, (ii) → { // 4 tasks
      // Each task executes a "chunk" of 25 iterations
          forseq(25*ii, 25*(ii+1)-1, (i) → BODY(i));

      }); // forall
```

- This is better, but it's still inconvenient for the programmer to do the "iteration grouping" or "loop chunking" explicitly

forallChunked APIs

- `forallChunked(int s0, int e0, int chunkSize, edu.rice.hj.api.HjProcedure<Integer> body)`
- Like `forall(int s0, int e0, edu.rice.hj.api.HjProcedure<Integer> body)`
- but forallChunked includes chunkSize as the third parameter
  - e.g., replace
    - `forall(0, 99, (i) → BODY(i)); // 100 tasks`
  - by
    - `forallChunked(0, 99, 100/4, (i)→BODY(i));`

Chunked Iterative Averaging

```
int nc = numWorkerThreads();
 … // Initializations
forseq(0, m-1, (iter) → {
  // Compute MyNew as function of input array MyVal
    forallChunked(1, n, n/nc, (j) → { // Create n/nc tasks
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    }); // forallChunked
  // Swap myVal & myNew;
  float[] temp=myVal; myVal=myNew; myNew=temp;
  // myNew becomes input array for next iteration
}); // forseq
```