COMP 322: Fundamentals of Parallel Programming

Lecture 11: Loop-Level Parallelism, Parallel Matrix Multiplication, Iteration Grouping (Chunking)

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/
1) Consider the compute method on slide 9. Let us suppose we supply it with an 8 element array with values [0,1,2,3,4,5,6,7] and THRESHOLD value of 2. Draw a computation graph corresponding to a call to compute with the appropriate fork and join edges.

2) Define each direct (sequential) computation as 2 units of work and each recursive subdivision as one unit of work. What is the total work? What is the critical path length?

TOTAL WORK = 14, CPL = 6 (critical path is highlighted as dashed edges)

NOTE: each call to compute() takes 2 units because THRESHOLD = 2
Outline of Today’s Lecture

- Loop-Level Parallelism, Parallel Matrix Multiplication
  - [Topics 3.1, 3.2]
- Grouping/chunking of parallel loop iterations
  - [Topic 3.3]
Sequential Algorithm for Matrix Multiplication

1. // Sequential version
2. for (int i = 0 ; i < n ; i++)
3.     for (int j = 0 ; j < n ; j++)
4.         c[i][j] = 0;
5. for (int i = 0 ; i < n ; i++)
6.     for (int j = 0 ; j < n ; j++)
7.         for (int k = 0 ; k < n ; k++)
8.             c[i][j] += a[i][k] * b[k][j];
9. // Print first element of output matrix
10. println(c[0][0]);

\[ c[i,j] = \sum_{0 \leq k < n} a[i,k] \cdot b[k,j] \]
Parallelizing the loops in Matrix Multiplication example using finish & async

1. // Parallel version using finish & async
2. finish() -> {
3.   for (int ii = 0 ; ii < n ; ii++)
4.       for (int jj = 0 ; jj < n ; jj++) {
5.           int i = ii; int j = jj;
6.           async() -> {c[i][j] = 0; }};
7.       }
8.   }
9. finish() -> {
10.  for (int ii = 0 ; ii < n ; ii++)
11.     for (int jj = 0 ; jj < n ; jj++){
12.        int i = ii; int j = jj;
13.        async() -> {
14.            for (int k = 0 ; k < n ; k++)
15.                c[i][j] += a[i][k] * b[k][j];
16.            }};
17.     }
18.   }
19. // Print first element of output matrix
20. println(c[0][0])

\[ c[i,j] = \sum_{0 \leq k < n} a[i,k] \times b[k,j] \]
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 the loops in Matrix Multiplication example using forall

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

1. // Parallel version using forall
2. forall(0, n-1, 0, n-1, (i, j) -> {
3.     c[i][j] = 0;
4. });
5. forall(0, n-1, 0, n-1, (i, j) -> {
6.     forseq(0, n-1, (k) -> {
7.         c[i][j] += a[i][k] * b[k][j];
8.     });
9. });
10. // Print first element of output matrix
11. println(c[0][0]);
forall API’s in HJlib

- 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)

- 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 a 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;j++)”
    forall(0, n-1, (j) -> {
        A[i][j] = F(A[i-1][j]) ;
    });
});
What about overheads?

- As you will see in next week’s lab and in Homework 2, 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

    ```java
    forall(0, 99, (i) -> BODY(i)); // 100 tasks
    ```

  — by

    ```java
    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));`
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

Illustration of an intermediate step for n = 8 (source: Figure 6.19 in Lin-Snyder book)
Sequential code for One-Dimensional Iterative Averaging that uses two copies of the array

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. } // forseq
11. // What is the purpose of line 11 below?
12. float[] temp=myVal; myVal=myNew; myNew=temp;
13. } // forseq

QUESTION: can either forseq() loop execute in parallel?
Example: HJ code for One-Dimensional Iterative Averaging using nested forseq-forall structure

1. // Initialize 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.   }); // forseq
10.  // What is the purpose of line 11 below?
11.  float[] temp=myVal; myVal=myNew; myNew=temp;
12.  // myNew becomes input array for next iteration
13. }); // forseq
Example: HJ code for One-Dimensional Iterative Averaging with forseq-forall structure w/ chunking

1. int nc = numWorkerThreads();
2.   ... // Initializations
3. forseq(0, m-1, (iter) -> {
4.     // Compute MyNew as function of input array MyVal
5.         forallChunked(1, n, n/nc, (j) -> {
6.             myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
7.         }); // forall
8.     // Swap myVal & myNew;
9.     float[] temp=myVal; myVal=myNew; myNew=temp;
10.    // myNew becomes input array for next iteration
11. }); // for