COMP 322: Fundamentals of Parallel Programming

Lecture 2: Async-Finish Parallel Programming and Computation Graphs

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

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

• Cilk lectures, http://supertech.csail.mit.edu/cilk/

• PrimeSieve.java example
  – http://introcs.cs.princeton.edu/java/14array/PrimeSieve.java.html
Goals for Today’s Lecture

- Discussion of Async and Finish constructs
- Understanding when two statements can run in parallel
- Understanding limits to ideal parallelism (critical path length)
Async and Finish Statements for Task Creation and Termination (Recap)

async S

- Creates a new child task that executes statement S

finish S

- Execute S, but wait until all asyncs in S’s scope have terminated.

// T₀ (Parent task)
STMT₀;
finish {  //Begin finish
    async {
        STMT₁;  //T₁ (Child task)
    }
    STMT₂;   //Continue in T₀
        //Wait for T₁
}         //End finish
STMT₃;   //Continue in T₀

T₁

T₀

STMT₀

fork

STMT₁

join

STMT₂

STMT₃
Some Properties of Async & Finish constructs

1. Scope of async/finish can be any arbitrary statement
   - async/finish constructs can be arbitrarily nested e.g.,
   - finish { async S1; finish { async S2; S3; } S4; } S5;

2. A method may return before all its async's have terminated
   - Enclose method body in a finish if you don't want this to happen
   - main() method is enclosed in an implicit finish e.g.,
   - main(){ foo();} void foo() {async S1; S2; return;}

3. Each dynamic async task will have a unique Immediately Enclosing Finish (IEF) at runtime

4. Async/finish constructs cannot “deadlock”
   - Cannot have a situation where both task A waits for task B to finish, and task B waits for task A to finish

5. Async tasks can read/write shared data via objects and arrays
   - Local variables have special restrictions (next slide)
Local Variables

Three rules for accessing local variables across tasks in HJ:

1) An async may read the value of any final outer local var

   final int i1 = 1; async { ... = i1; /* i1=1 */ }

2) An async may read the value of any non-final outer local var
   (copied on entry to async like method parameters)

   int i2 = 2; // i2=2 is copied on entry to the async
   async { ... = i2; /* i2=2*/ }

   i2 = 3; // This assignment is not seen by the above async

3) An async is not permitted to modify an outer local var

   int[] A; async { A = ...; /*ERROR*/ A[i] = ...; /*OK*/ }
Converting sequential Java programs to parallel Async-Finish HJ programs

One possible approach:

1. **Create “ideal” parallel version**
   - Insert async’s at all points where parallelism can logically be exploited
   - Insert finish’s to ensure that the parallel version produces the same results as the sequential version

2. **Transform ideal parallelism to useful parallelism**
   - Merge or remove async’s to amortize overhead
   - Replace finish by more efficient synchronization constructs (to be covered later in course)
Java Example: Sieve of Eratosthenes

1. // initially assume all integers are prime
2. boolean[] isPrime = new boolean[N + 1];
3. for (int i = 2; i <= N; i++) isPrime[i] = true;
4. // mark non-primes <= N using Sieve of Eratosthenes
5. for (int i = 2; i*i <= N; i++)
6.   // if i is prime, then mark multiples of i as nonprime
7.   if (isPrime[i])
8.     for (int j = i; i*j <= N; j++)
9.       isPrime[i*j] = false;
10. // count primes
11. int primes = 0;
12. for (int i = 2; i <= N; i++) if (isPrime[i]) primes++;

How should we parallelize the sieve computation in lines 5-9?
Ideal Parallelization of Sieve Computation

1. // initially assume all integers are prime
2. boolean[] isPrime = new boolean[N + 1];
3. for (int i = 2; i <= N; i++) isPrime[i] = true;
4. // mark non-primes <= N using Sieve of Eratosthenes
5. for (int i = 2; i*i <= N; i++)
6.   // if i is prime, then mark multiples of i as nonprime
7.     if (isPrime[i])
8.       finish for (int j = i; i*j <= N; j++)
9.         async isPrime[i*j] = false;
10. // count primes
11. int primes = 0;
12. for (int i = 2; i <= N; i++) if (isPrime[i]) primes++;

Is this approach correct? Is it efficient?
Which statements can potentially be executed in parallel with each other?

1. `finish { // F1`
2. `async A1;`
3. `finish { // F2`
4. `async A3;`
5. `async A4;`
6. `} // F2`
7. `S5;`
8. `} // F1`
Computation Graphs for HJ Programs

- A Computation Graph (CG) captures the dynamic execution of an HJ program, for a specific input.
- CG nodes are “steps” in the program’s execution:
  - A step is a sequential subcomputation without any async, begin-finish and end-finish operations.
- CG edges represent ordering constraints:
  - “Continue” edges define sequencing of steps within a task.
  - “Spawn” edges connect parent tasks to child async tasks.
  - “Join” edges connect the end of each async task to its IEF’s end-finish operations.
Example HJ Program with statements v1 ... v23

// Task T1
v1; v2;
finish {
    async {
        // Task T2
        v3;
        finish {
            async {
                // Task T3
                v4; v5;
            } // Task T3
            v6;
            async {
                // Task T4
                v7; v8;
            } // Task T4
            v9;
        } // finish
        v10; v11;
    } // Task T2
    async {
        v12; v13;
        v14; } // Task T5
        v15;
} // end of task T2
v16; v17; // back in Task T1
} // finish
v18; v19;
finish {
    async {
        // Task T6
        v20; v21; v22;
    }
}
v23;
Example: Step \( v_{16} \) can potentially execute in parallel with steps \( v_3 \) ... \( v_{15} \)
Complexity Measures for Computation Graphs

Define

- $\text{TIME}(N) = \text{execution time of node } N$
- $\text{WORK}(G) = \text{sum of } \text{TIME}(N)\text{, for all nodes } N \text{ in CG } G$
  - $\text{WORK}(G)$ is the total work to be performed in $G$
- $\text{CPL}(G) = \text{length of a longest path in CG } G\text{, when adding up execution times of all nodes in the path}$
  - Such paths are called critical paths
  - $\text{CPL}(G)$ is the length of these paths (critical path length)
Ideal Speedup

Define **ideal speedup** of Computation G Graph as the ratio, \( \text{WORK}(G)/\text{CPL}(G) \)

Ideal Speedup is independent of the number of processors that the program executes on, and only depends on the computation graph.
Example

- Assume time(N) = 1 for all nodes in this graph

\[
\text{WORK}(G) = 18
\]
Example (contd)

- Assume \( \text{time}(N) = 1 \) for all nodes in this graph

\[
\text{CPL}(G) = 9
\]

Ideal speedup
\[
= \frac{\text{WORK}(G)}{\text{CPL}(G)}
= 2
\]
Homework 1 Reminder

• **Written assignment, due by Friday, Jan 13th**

• **Submit a softcopy of your solution in Word, PDF, or plain text format**
  — Try and use turn-in script for submission, if possible
  — Otherwise, email your homework to comp322-staff at mailman.rice.edu

• **See course web site for penalties for late submissions**
  — Send me email if you have an extenuating circumstance for delay