COMP 322: Fundamentals of Parallel Programming

Lecture 6: Data Races and How to Avoid Them

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

Announcements

- Homework 3 is due by 5pm on Monday, Feb 7th
  - This is a programming assignment with abstract performance metrics
  - To prepare for HW3, please make sure that you can compile and run the programs from Lab 2 on your own, using the –perf option. In case of problems, please send email to comp322-staff @ mailman.rice.edu

Acknowledgments for Today's Lecture

- COMP 322 Lecture 6 handout

Example of Incorrect Parallelization from Homework 1

```
1.  // Sequential version
2.  for ( p = first; p != null; p = p.next) p.x = p.y + p.z;
3.  for ( p = first; p != null; p = p.next) sum += p.x;
4.
5.  // Incorrect parallel version
6.  for ( p = first; p != null; p = p.next)
7.      async p.x = p.y + p.z;
8.  for ( p = first; p != null; p = p.next)
9.  sum += p.x;
```

Why was this version incorrect?

What does its computation graph say about writes to p.x in line 7 and reads of p.x in line 9?

Formal Definition of Data Races

Formally, a data race occurs on location L in a program execution with computation graph *CG* if there exist steps S1 and S2 in computation graph *CG* such that:

1. S1 does not depend on S2 and S2 does not depend on S1 i.e., there is no path of dependence edges from S1 to S2 or from S2 to S1 in *CG*, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

Data races are challenging because it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program testing.

Thus, no amount of testing may be able to detect errors that might only become manifest in production use.

Observations

1. *Immutability property: there cannot be a data race on shared immutable data.*
   - A location, L, is immutable if it is only written during initialization, and can only be read after initialization. In this case, no read can potentially execute in parallel with the write.

2. *Single-task ownership property: there cannot be a data race on a location that is only read or written by a single task.*
   - Define: step S in computation graph CG "owns" location L if S performs a read or write access on L. If step S belongs to Task T, we can also say that Task T owns L when executing S.
   - Consider a location L that is only owned by steps that belong to the same task, T. Since all steps in Task T must be connected by *continue* edges in CG, all reads and writes to L must be ordered by the dependences in CG. Therefore, no data race is possible on location L.

Observations (contd)

3. *Ownership-transfer property: there cannot be a data race on a location if all steps that read or write it are totally ordered in CG.* (Generalization of single-task-ownership property.)
   - Think of the ownership of L being ``transferred'' from one step to another, even across task boundaries, as execution follows the path of dependence edges in the total order.

4. *Local-variable ownership property: there cannot be a data race on a local variable.*
   - If L is a local variable, it can only be written by the task in which it is declared (L's owner). The *copy-in* semantics for local variables ensures that the value of the local variable is copied on async creation thus guaranteeing that there is no race condition between the read access in the descendant task and the write access in L's owner.

Observations (contd)

5. *Determinism property: if a parallel program with async, finish, future and get operations can never have a data race, then it must be deterministic with respect to its inputs.*
   - A computation is said to be "deterministic with respect to its inputs" if it always computes the same answer, when given the same inputs.
   - For the class of parallel programs that we have studied thus far, the absence of data races is sufficient to guarantee that the parallel program must be deterministic with respect to its inputs.
   - Such programs are said to to be "data-race-free". Programs that may exhibit data races are said to be "racy".

Avoiding Data Races: Immutability Tip

- Use immutable objects and arrays as far as possible
  - May require making copies of objects and arrays instead of just modifying a single field or array element
  - Copying overhead may be prohibitive in some cases, but acceptable in others

- Example with java.lang.String

```
finish {
  String s1 = "XYZ";
  async { String s2 = s1.toLowerCase(); ... }
  System.out.println(s1);
}
```


Avoiding Data Races: Single-task ownership tip

- If an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task.
  - Entails making copies when sharing the object with other tasks.
  - As in the Immutability tip, copying overhead may be prohibitive in some cases, but acceptable in others.
- Example

```
finish { // Task T1 owns A
  int[] A = new int[n]; // ... initialize array A ...
  // create a copy of array A in B
  int[] B = new int[A.length]; System.arraycopy(A,0,B,0,A.length);
  async { // Task T2 owns B
    int sum = computeSum(B,0,B.length-1); // Modifies B (Lecture 5)
    System.out.println("sum = " + sum);
  }
  // ... update Array A ...
  System.out.println(Arrays.toString(A)); //printed by task T1
}
```


Avoiding Data Races: Ownership-transfer tip

- If an object or array needs to be written by multiple tasks, then try and restrict its ownership so that all read/write steps are ordered by a chain of dependences in the CG.
  - Ownership transfer occurs when we cross task boundaries in the chain of dependences
- Example

```
finish { // Task T1 owns A
  int[] A = new int[n]; // ... initialize array A ...
  // Task T1 initially owns B
  int[] B = new int[A.length]; System.arraycopy(A,0,B,0,A.length);
  async { // Task T2 now owns B
    int sum = computeSum(B,0,B.length-1); // Modifies B (Lecture 5)
    System.out.println("sum = " + sum);
  }
  // ... update and print Array A ...
}
```


Assumptions that can be made in the presence of Data Races

- Example

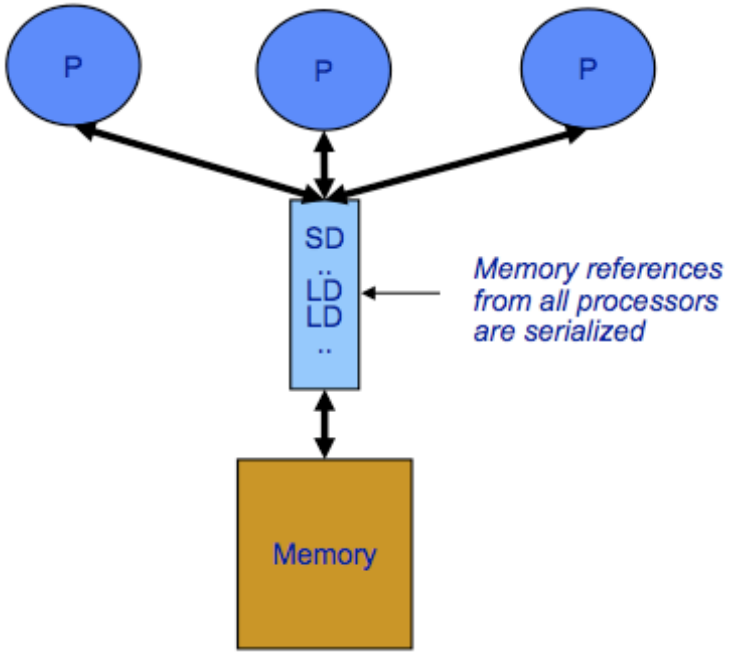
```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x = 2; // Task T2
async { // Task T3
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p.x);
}
 async { // Task T4
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + p.x);
  System.out.println("Third read = " + p.x);
}
```

What values can be printed by tasks T3 and T4?

Memory Models

- A memory consistency model, or *memory model*, is the part of a programming language specification that defines what write values a read may see in the presence of data races.
- We will briefly discuss three memory models
  - Sequential Consistency (SC)
  - Location Consistency (LC)
  - C++ Memory Model

Sequential Consistency



[Lamport] "*A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program*"

Example Revisited

- SC model will not permit Task T3 to print "0, 1, 2" and Task T4 to print "0, 2, 1"

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x = 2; // Task T2
async { // Task T3
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p.x);
}
 async { // Task T4
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + p.x);
  System.out.println("Third read = " + p.x);
}
```


Example Revisited

- What if the programmer transformed the body of Task T3?

```
p.x = 0; q = p;
async p.x = 1; // Task T1
async p.x = 2; // Task T2
async { // Task T3
   int p_x = p.x;
  System.out.println("First read = " + p_x);
  System.out.println("Second read = " + q.x);
  System.out.println("Third read = " + p_x);
}
 async { // Task T4
  System.out.println("First read = " + p.x);
  System.out.println("Second read = " + p.x);
  System.out.println("Third read = " + p.x);
}
```