COMP 322: Fundamentals of Parallel Programming

Lecture 9: Data Races, Functional & Structural Determinism

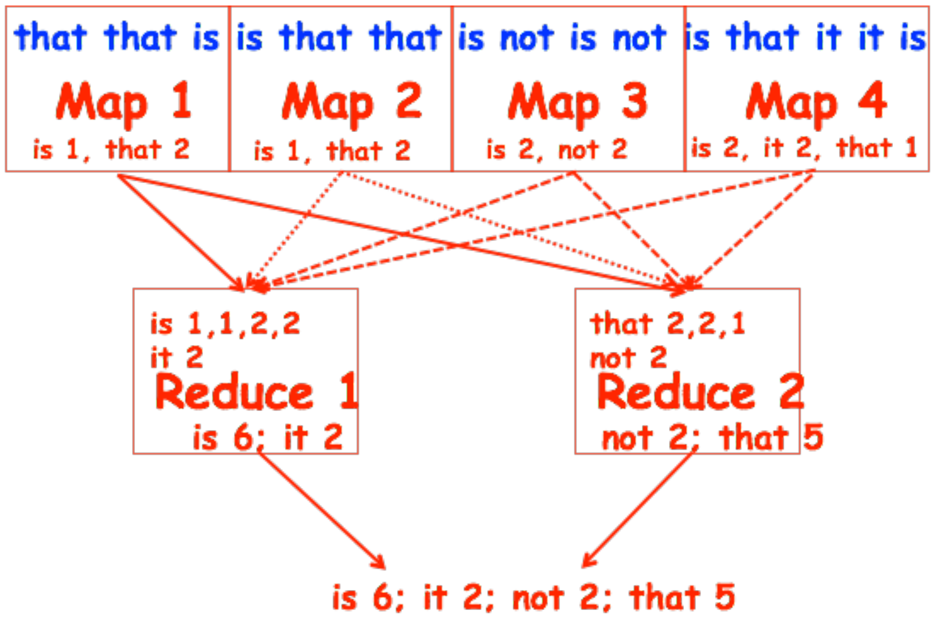
Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Worksheet #8: Analysis of Map Reduce Example

Analyze the total WORK and CPL for the Map reduce example:

- Assume that each Map step has WORK = number of input words, and CPL=1
- Assume that each Reduce step has WORK = number of input word-count pairs, and CPL = log2(# occurrences for input word with largest # pairs)



WORK/CPL for all Map steps:

- WORK = 15
- CPL = 1 (ignore impact of local sums on CPL)

WORK/CPL for Reduce 1 step:

- WORK = 5
- CPL = ceiling(log2(4)) = 2

WORK/CPL for Reduce 2 step:

- WORK = 4
- CPL = ceiling(log2(3)) = 2

Total WORK and CPL

- WORK = 15+5+4 = 24
- CPL = 1 + 2 = 3

Parallel Programming Challenges

- Correctness
  - New classes of bugs can arise in parallel programming, relative to sequential programming
    - Data races, deadlock, nondeterminism
- Performance
  - Performance of parallel program depends on underlying parallel system
    - Language compiler and runtime system
    - Processor structure and memory hierarchy
    - Degree of parallelism in program vs. hardware
- Portability
  - Functional portability: A buggy program that runs correctly on one system may not run correctly on another (or even when re-executed on the same system)
  - Performance portability: A parallel program that performs well on one system may perform poorly on another

Example of a Data Race

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1,sum2 are shared fields
async { // Task T1 computes sum of upper half of array
  for(int i=X.length/2; i < X.length; i++)
    sum2 += X[i];
}
// Continue in T0, compute sum of lower half of array
for(int i=0; i < X.length/2; i++) sum1 += X[i];
return sum1 + sum2;
```

Data race between accesses of sum2 in async and in main program (due to missing finish)

Data Races (Recap from Lecture 2)

A data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1, i.e., S1 and S2 can potentially execute in parallel, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

- A data-race is usually considered an error. The result of a read operation in a data race is undefined. The result of a write operation is undefined if there are two or more writes to the same location.
  - Note that our definition of data race includes the case that both S1 and S2 write the same value in location L, even if that may not be considered an error.
- Above definition includes all "potential" data races i.e., we consider it to be a data race even if S1 and S2 end up executing on the same processor.

Recap of Java's Storage Model

Java's storage model contains three memory regions:

1. Static Data: region of memory reserved for variables that are not allocated or destroyed during a class' lifetime, such as static fields.

   Static fields can be shared among threads/tasks

2. Heap Data: region of memory for all objects and arrays (implicitly/explicitly created by "new").

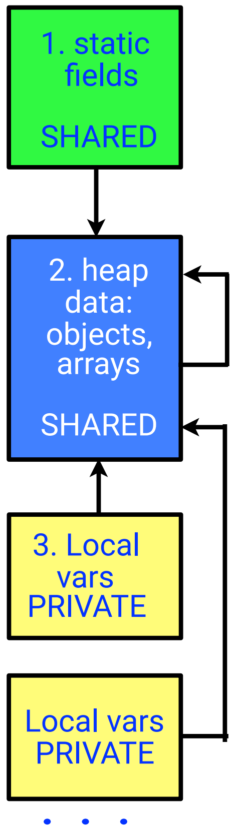
   Heap data can be shared among threads/tasks

3. Stack Data: Each time you call a method, Java allocates a new block of memory called a stack frame to hold its local variables

   Local variables are private to a given thread/task

   No data races possible on local variables

NOTE: all references (pointers) must point to heap data --- no references can point to static or stack data



Four Observations related to Data Races

1. Immutability property: there cannot be a data race on shared immutable variables
   - 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.

- Parallel programming tip: use immutable objects and arrays to avoid data races
  - Will require making copies of objects and arrays for updates
  - Copying overhead may be prohibitive in some cases, but acceptable in others
  - NOTE: future values are also immutable

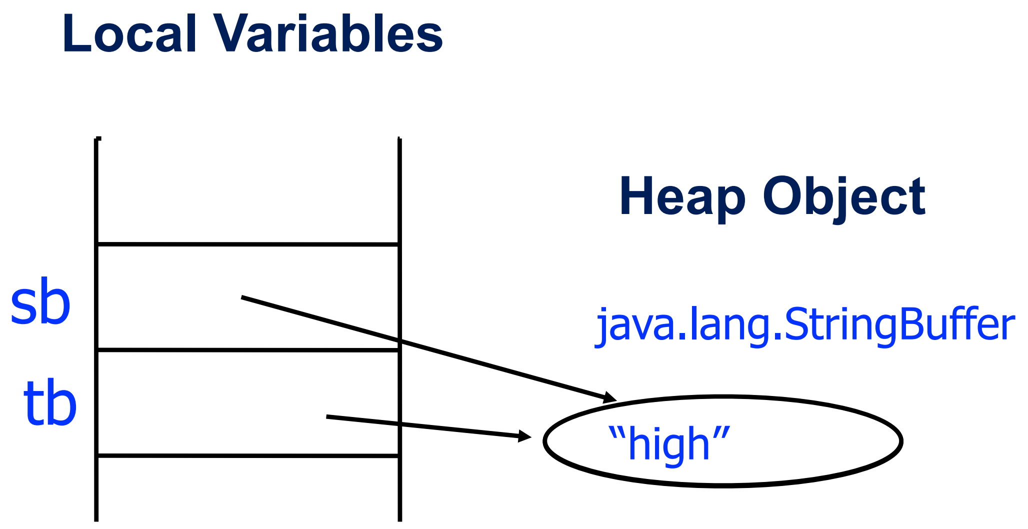
Example with java.lang.String

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

Example of a Mutable Object

- If an object is modified, all references to the object see the new value

```
StringBuffer sb = new StringBuffer(“hi”);
StringBuffer tb = sb;
tb.append (“gh”);
```



Observations

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.

- Parallel programming tip: if an object or array needs to be written multiple times after initialization, then try and restrict its ownership to a single task.
  - Will require making copies when sharing the object or array with other tasks.

Example of Single-task ownership with Copying

- 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 with Immutability, 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
    computePartialSums(B);// Updates B
    System.out.println(Arrays.toString(B));
  }
  . . .
  System.out.println(Arrays.toString(A)); //printed by task T1
}
```

Observations (contd)

3. <u>Ownership-transfer property:</u> there cannot be a data race on a location if all steps that read or write it are totally ordered in CG (i.e., if the steps belong to a single directed path)
   - 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.

- Parallel programming tip:
  - If an object or array needs to be written multiple times after initialization and also accessed by multiple tasks, then try and ensure that all the steps that read or write a location L in the object/array are totally ordered by dependences in CG.
    - Ownership transfer might even be necessary to support single-task ownership. In the previous example, since Task T1 initializes array B as a copy of array A, T1 is the original owner of A. The ownership of B is then transferred from T1 to T2 when Task T2 is created.

Observations (contd)

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 *"implicitly final"* semantics for accessing outer local variables ensures that there is no race condition between the read access in the child task and the write access in L's owner (parent task).

- Parallel programming tip:
  - You do not need to worry about data races on local variables, since they are not possible. However, local variables in Java are restricted to contain primitive data types (such as int) and references to objects and arrays. In the case of object/array references, be aware that there may be a data race on the underlying object even if there is no data race on the local variable that refers to (points to) the object.

Functional vs. Structural Determinism

- A parallel program is said to be *functionally deterministic* if it always computes the same answer when given the same input
- A parallel program is said to be *structurally deterministic* if it always produces the same computation graph when given the same input
- *Data-Race-Free Determinism Property*
  - If a parallel program is written using the constructs learned so far (finish, async, futures) *and* is known to be data-race-free, *then it must be both functionally deterministic and structurally deterministic*

Example: Sequential search for pattern in text

```
for (int i = 0; i <= N - M; i++) {
  for (j = 0; j < M; j++) {
    if (text[i+j] != pattern[j]) break;
  } // for j
  if (j == M) {
    // pattern found
    // update flag/count/index as needed
    // exit for-i loop if needed
    . . .
  }
} // for i
```

Version 1 of Parallel Search: Count of all occurrences

```
// Count all occurrences
a = new Accumulator(SUM, int)
finish(a) {
  for (int ii = 0; ii <= N - M; ii++) {
    int i = ii;
    async {
      for (j = 0; j < M; j++)
        if (text[i+j] != pattern[j]) break;
      if (j == M) a.put(1); // Increment count
    } // async
  } // for
} // finish
print a.get(); // Output
```

Version 2 of Parallel Search: Existence of an occurrence

```
found = false; // object or static field
finish for (int i = 0; i <= N - M; i++)
  async {
    for (j = 0; j < M; j++)
      if (text[i+j] != pattern[j]) break;
    if (j == M) found = true;
  } // finish-for-async
 print found // Output
```

Version 3 of Parallel Search: Index of an occurrence

```
index = -1; // object or static field
. . .
finish for (int i = 0; i <= N - M; i++)
  async {
    for (j = 0; j < M; j++)
      if (text[i+j] != pattern[j]) break;
    if (j == M) index = i; // found at i
  } // finish-for-async
print index // Output
```

Version 4 of Parallel Search: Optimized existence of an occurrence

```
found = false; // object or static field
. . .
finish for (int i = 0; i <= N - M; i++) {
 if (found) break; // Optimization!
 async {
   for (j = 0; j < M; j++)
     if (text[i+j] != pattern[j]) break;
   if (j == M) found = true;
 } // async
} // finish-for
print found // Output
```

Version 5 of Parallel Search: Optimized index of an occurrence

```
index = -1; // // object or static field
. . .
finish for (int i = 0; i <= N - M; i++) {
  if (index != -1) break; // Optimization!
  async {
    for (j = 0; j < M; j++)
      if (text[i+j] != pattern[j]) break;
    if (j == M) index = i;
   } // async
} // finish-for
print index // Output
```

Announcements & Reminders

- HW2 is available and due by 11:59pm on Wednesday, Feb 6th
- See course web site for all work assignments and due dates
- Use Piazza (public or private posts, as appropriate) for all communications re. COMP 322
- See Office Hours link on course web site for latest office hours schedule.