COMP 322: Fundamentals of Parallel Programming

Lecture 7: Data Races, Functional & Structural Determinism

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #6 solution: Associativity and Commutativity

Recap:
A binary function f is *associative* if f(f(x,y),z) = f(x,f(y,z)).
A binary function f is *commutative* if f(x,y) = f(y,x).

Worksheet problems:
1) Claim: a Finish Accumulator (FA) can only be used with operators that are *associative and commutative.* Why? What can go wrong with accumulators if the operator is non-associative or non-commutative?
You may get different answers in different executions if the operator is non-associative or non-commutative

2) For each of the following functions, indicate if it is associative and/or commutative.
a) f(x,y) = x+y, for integers x, y, is associative and commutative

b) g(x,y) = (x+y)/2, for integers x, y, is commutative but not associative

c) h(s1,s2) = concat(s1, s2) for strings s1, s2, e.g., h("ab","cd") = "abcd", is associative but not commutative

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
  - A buggy program that runs correctly on one system may not run correctly on another (or even when re-executed on the same system)
  - A parallel program that performs well on one system may perform poorly on another

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 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.
- A program is *data-race-free* it cannot exhibit a data race for any input
- Above definition includes all "potential" data races i.e., we consider it to be a data race even if S1 and S2 are scheduled on the same processor.

Example of a Data Race

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

Data race between accesses of sum1 in async and in main program

Formal Definition of Data Races

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., 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. (L must be a shared location i.e., a static field, instance field, or array element.)

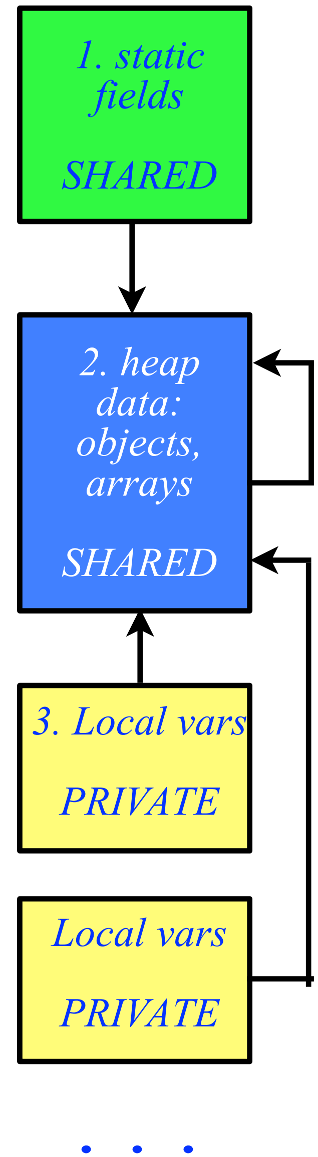
- A program is *data-race-free* it cannot exhibit a data race for any input
- Above definition includes all "potential" data races i.e., it's considered a data race even if S1 and S2 execute on the same processor

Recap of Java's Storage Model

Java's storage model contains three memory regions:

1. **<u>Static Data</u>: region of memory reserved for variables that are not allocated or destroyed during a class' lifetime, such as <u>static fields</u>.**
   - **Static fields can be <u>shared</u> among threads/tasks**
2. **<u>Heap Data</u>: region of memory for dynamically allocated <u>objects</u> and <u>arrays</u> (created by "new").**
   - **Heap data can be <u>shared</u> among threads/tasks**
3. **<u>Stack Data</u>: Each time you call a method, Java allocates a new block of memory called a stack frame to hold its <u>local variables</u>**
   - **Local variables are <u>private</u> 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



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*

Functional + Structural Determinism (V1 of Parallel Search)

```
// Count all occurrences
a = new ACCUM(SUM, int)
finish(a) 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) a.put(1);        // found
 }
print a.get();
```

Functional + Structural Determinism (V2 of Parallel Search)

```
// Existence of an occurrence
found = false
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;
 }
 print found
```

Functional Nondeterminism + Structural Determinism

```
// Index of an occurrence
static int index = -1; // 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
}
```

Functionally Determinism + Structural Nondeterminism (V4 of Parallel Search)

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

Functionally Nondeterminism + Structural Nondeterminism (V5 of

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

A Classification of Parallel Programs

Data Race Free?	Functionally Deterministic?	Structurally Deterministic?	Example: String Search variation
Yes	Yes	Yes	Count of all occurrences
No	Yes	Yes	Existence of an occurrence
No	No	Yes	Index of any occurrence
No	Yes	No	"Eureka" extension for existence of an occurrence: do not create more async tasks after occurrence is found
No	No	No	"Eureka" extension for index of an occurrence: do not create more async tasks after occurrence is found

Data-Race-Free Determinism Property implies that it is not possible to write an HJ program with Yes in column 1, and No in column 2 or column 3 (when only using Module 1 constructs)