COMP 322: Fundamentals of Parallel Programming

Lecture 8: Data Races, Functional & Structural Determinism

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

http://comp322.rice.edu

Worksheet #7 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 e.g., an accumulator can be implemented using one "partial accumulator" per processor core.
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
⇒ *Incorrect answers found in some worksheets: Associative / Both / Neither*

c) h(s1,s2) = concat(s1, s2) for strings s1, s2, e.g., h("ab","cd") = "abcd", is associative but not commutative
⇒ *Incorrect answers found in some worksheets: Commutative / Neither*

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

Example of a Data Race

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1,sum2 are static/object 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 and 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

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.

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
 }
} // 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
```


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
```