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

1. // Start of Task T0 (main program)
2. sum1 = 0; sum2 = 0; // sum1,sum2 are static/object fields
3. async { // Task T1 computes sum of upper half of array
4.   for(int i=X.length/2; i < X.length; i++)
5.     sum2 += X[i];
6. }
7. // Continue in T0 and compute sum of lower half of array
8. for(int i=0; i < X.length/2; i++) sum1 += X[i];
9. 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) \( S_1 \) and \( S_2 \) in \( CG \) such that:

1. \( S_1 \) does not depend on \( S_2 \) and \( S_2 \) does not depend on \( S_1 \), i.e., \( S_1 \) and \( S_2 \) can potentially execute in parallel, and
2. Both \( S_1 \) and \( S_2 \) 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 \( S_1 \) and \( S_2 \) 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

1. for (int i = 0; i <= N - M; i++) {
2.     for (j = 0; j < M; j++) {
3.         if (text[i+j] != pattern[j]) break;
4.     } // for j
5.     if (j == M) {
6.         // pattern found
7.         // update flag/count/index as needed
8.         // exit for-i loop if needed
9.         . . .
10.    }
11. } // for i

Version 1 of Parallel Search: Count of all occurrences

1. // Count all occurrences
2. a = new Accumulator(SUM, int)
3. finish(a) {
4.     for (int ii = 0; ii <= N - M; ii++) {
5.         int i = ii;
6.         async {
7.             for (j = 0; j < M; j++)
8.                 if (text[i+j] != pattern[j]) break;
9.             if (j == M) a.put(1); // Increment count
10.         } // async
11.     }
12. } // finish
13. print a.get(); // Output
### Version 2 of Parallel Search: Existence of an occurrence

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

### Version 3 of Parallel Search: Index of an occurrence

1. `index = -1;` // object or static field
2. . . .
3. `finish` for (int `i = 0; i <= N - M; i++`)
4. `async` {
   5.   for (j = 0; j < M; j++)
   6.     if (text[i+j] != pattern[j]) break;
   7.     if (j == M) `index = i;` // found at i
   8. } // finish-for-async
9. `print index` // Output
Version 4 of Parallel Search: Optimized existence of an occurrence

1. `found = false; // object or static field`
2. ...  
3. `finish` for (int `i = 0; i <= N - M; i++`) {
4. if (found) break; // Optimization!
5. `async` {
6. for (j = 0; j < M; j++)
7. if (text[i+j] != pattern[j]) break;
8. if (j == M) found = true;
9. } // async
10. } // finish-for

Version 5 of Parallel Search: Optimized index of an occurrence

1. `index = -1; // object or static field`
2. ...  
3. `finish` for (int `i = 0; i <= N - M; i++`) {
4. if (index != -1) break; // Optimization!
5. `async` {
6. for (j = 0; j < M; j++)
7. if (text[i+j] != pattern[j]) break;
8. if (j == M) index = i;
9. } // async
10. } // finish-for