COMP 322: Fundamentals of Parallel Programming

Lecture 6: Data Races (contd), Futures --- Tasks with Return Values

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

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

COMP 322 Grading Policy (from course web site)

- **Weekly Quizzes 10%**
  - One lecture quiz + one lab quiz per week
  - No extensions past deadline
  - Open book
- **Exams (2) 40%**
  - Take-home written exams
  - No extensions past deadline
  - Closed book
- **Quizzes and exams test your individual understanding and knowledge of the material. Collaboration on quizzes and exams is strictly forbidden.**
- **Homeworks (6) 40%**
  - Written and programming components
  - 10% per day late penalty up to 6 days (HW1 due by 5pm on Jan 23rd)
  - Send zip file to comp322-staff if you cannot submit homework via turnin
  - You can discuss homework problem approaches with others, but submission must be your own individual effort
- **Class Participation 10%**
  - Lecture worksheets, classroom, lab & on-line discussions, brilliant observations, ...
  - Lecture worksheets are graded for participation (not for quality of answer)

Outline of Today's Lecture

- **Recap of Data Races, Determinism, Memory Models**
- **Futures --- Tasks with Return Values**

- **Acknowledgments**
  - COMP 322 Module 1 handout, Chapter 4, Sections 5.1, 5.2
    - https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

Solution to Worksheet #5: Data Races and Determinism

Consider a modified String Search program that returns true if any occurrence is found, rather than the count of all occurrences:

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

Questions:

1. **Does this program have a data race?**
   Yes. Multiple async tasks can write to the same static field, found.
2. **Is it deterministic?**
   Yes. The answer will be the same regardless of the order of the writes.
3. **Is it structurally deterministic?**
   Yes. The computation graph will always be the same for the same input.

Definition of Data Races

Formally, 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
- Above definition focuses on interfering pairs of read/write accesses, but ignores the values e.g.,
  - Two parallel writes of X = 1 are considered to be a data race
  - If Y=1 initially, then a parallel read of Y and a (re)write of Y=1 are also considered to be a data race

Definitions of Determinism and Structural Determinism

- A parallel program is said to be *deterministic* with respect to its inputs if it always computes the same answer when given the same inputs.
- A parallel program is said to be *structurally deterministic* with respect to its inputs if its final computation graph is guaranteed to be the same for all executions of the program with the same inputs
  - Structural determinism is also referred to as "determinacy"
- Structural Determinism Property for HJ programs
  - If an HJ parallel program is written using only the constructs in Module 1 and is guaranteed to be data-race-free, then it must be structurally deterministic with respect to its inputs.

A Classification of Parallel Programs

Data Race Free?	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

Structural 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)

String Search variation: “Eureka” extension for existence of occurrence

```
1.  static boolean found = false; //static field
2.  . . .
3.  finish for (int i = 0; i <= N - M; i++) {
4.    if (found) break; // Eureka!
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
```

Memory Consistency Models

- A memory consistency model, or memory model, is the part of a programming specification that defines what write values a read may observe
  - — For data-race-free programs, all memory models are identical since each read can observe exactly one write value
  - ⇒ *if you only write data-race-free programs, you don't have to worry about memory models!*
- Question: why do different memory models have different rules for data races?
- Answer: because different memory models are useful at different levels of software
  - — Sequential Consistency (SC)
    - – Useful for implementing low-level synchronization primitives e.g., operating system services
  - — Java Memory Model (JMM)
    - – Useful for implementing task schedulers e.g., HJ runtime
  - — Habanero Java Memory Model (HJMM)
    - – Useful for specifying semantics at application task level e.g., HJ programs
    - – Derived from past work on "Location Consistency" memory model

HJMM
JMM
SC

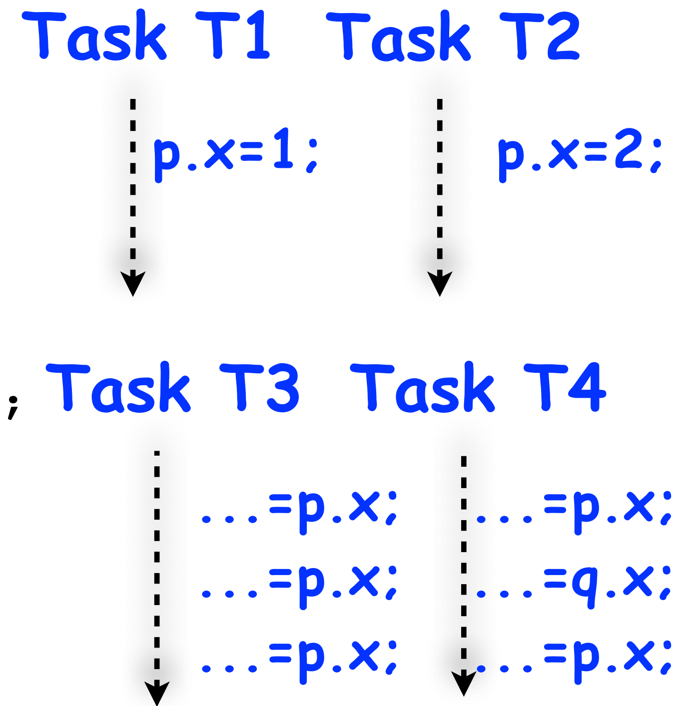
Example 1: can task T4 print 1, 2, 1?

<u>Example HJ program:</u>

```
1. p.x = 0; q = p;
2. async p.x = 1; // Task T1
3. async p.x = 2; // Task T2
4. async { // Task T3
5.    System.out.println("First read = " + p.x);
6.    System.out.println("Second read = " + p.x);
7.    System.out.println("Third read = " + p.x)
8. }
9. async { // Task T4
10.   System.out.println("First read = " + p.x);
11.   System.out.println("Second read = " + q.x);
12.   System.out.println("Third read = " + p.x);
13.}
```

Answer:

- *SC ⇒ No*
- *JMM ⇒ Maybe*
- *HJMM ⇒ Yes*



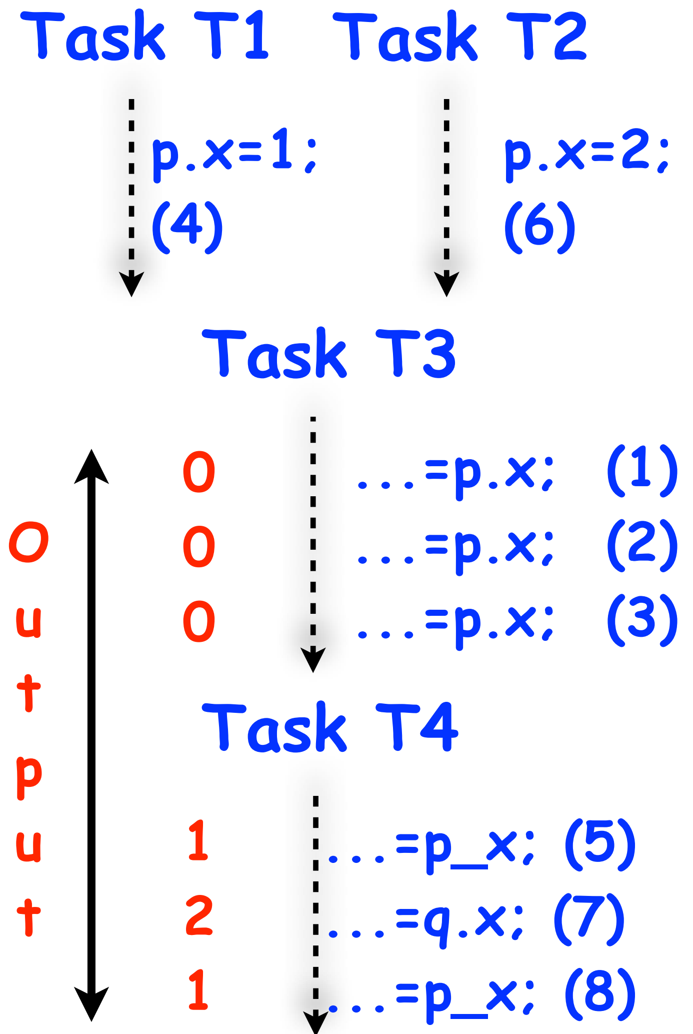
Example 2: can p.x be replaced by a local variable in task T4?

Example HJ program:

```
1.  p.x = 0; q = p;
2.  async p.x = 1; // Task T1
3.  async p.x = 2; // Task T2
4.  async { // Task T3
5.    System.out.println("First read = " + p.x);
6.    System.out.println("Second read = " + p.x);
7.    System.out.println("Third read = " + p.x)
8.  }
9.  async { // Task T4
10.   // Assume programmer doesn't know that p=q
11.   int p_x = p.x;
12.   System.out.println("First read = " + p_x);
13.   System.out.println("Second read = " + q.x);
14.   System.out.println("Third read = " + p_x);
15. }
```

Answer:

- *SC ⇒ No*
- *JMM ⇒ Maybe*
- *HJMM ⇒ Yes*



Outline of Today's Lecture

- **Recap of Data Races, Determinism, Memory Models**
- **Futures --- Tasks with Return Values**

- **Acknowledgments**
  - —COMP 322 Module 1 handout, Chapter 4, Sections 5.1, 5.2
    - https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

Extending Async Tasks with Return Values

- **Example Scenario in PseudoCode**

```
1. // Parent task creates child async task
2. final future<int> container =
3.                     async<int>  { return computeSum(X, low, mid); };
4. . . .
5. // Later, parent examines the return value
6. int sum = container.get();
```

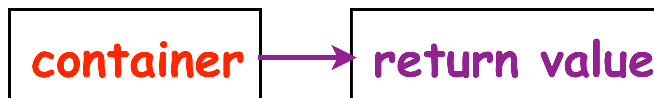
- **Two issues to be addressed:**
  1) Distinction between container and value in container (box)
  2) Synchronization to avoid race condition in container accesses

Parent Task

```
container = async {...}
. . .
container.get()
```

Child Task

```
computeSum(...)
return ...
```



HJ Futures: Tasks with Return Values

```
async<T> { Stmt-Block }
```

- **Creates a new child task that executes Stmt-Block, which must terminate with a return statement returning a value of type T**
- **Async expression returns a reference to a *container* of type future<T>**
- **Values of type future<T> can only be assigned to *final variables***

```
Expr.get()
```

- **Evaluates Expr, and blocks if Expr's value is unavailable**
- **Expr must be of type future<T>**
- **Return value from Expr.get() will then be T**
- **Unlike finish which waits for *all* tasks in the finish scope, a get() operation only waits for the specified async expression**

Example: Two-way Parallel Array Sum using Future Tasks

```
1.  // Parent Task T1 (main program)
2.  // Compute sum1 (lower half) and sum2 (upper half) in parallel
3.  final future<int> sum1 = async<int> { // Future Task T2
4.    int sum = 0;
5.    for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
6.    return sum;
7.  }; //NOTE: semicolon needed to terminate assignment to sum1
8.  final future<int> sum2 = async<int> { // Future Task T3
9.    int sum = 0;
10.   for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
11.   return sum;
12. }; //NOTE: semicolon needed to terminate assignment to sum2
13. //Task T1 waits for Tasks T2 and T3 to complete
14. int total = sum1.get() + sum2.get();
```

Why are these semicolons needed?

Future Task Declarations and Uses

- Variable of type future<T> is a reference to a future object
  - Container for return value of T from future task
  - The reference to the container is also known as a “handle”
- Two operations that can be performed on variable V1 of type future<T1> (assume that type T2 is a subtype of type T1):
  - Assignment: V1 can be assigned value of type future<T2>
  - Blocking read: V1.get() waits until the future task referred to by V1 has completed, and then propagates the return value
- Future task body must start with a type declaration, async<T1>, where T1 is the type of the task's return value
- Future task body must consist of a statement block enclosed in { } braces, terminating with a return statement

Comparison of Future Task and Regular Async Versions of Two-Way Array Sum

- Future task version initializes two references to future objects, sum1 and sum2, and both are declared as final
- No finish construct needed in this example
  - Instead parent task waits for child tasks by performing sum1.get() and sum2.get()
- Easier to guaranteed absence of race conditions in Future Task version
  - No race on sum because it is a local variable in tasks T2 and T3
  - No race on future variables, sum1 and sum2, because of blocking-read semantics

Reduction Tree Schema in ArraySum1 (Recap)

X[0] X[1] X[2] X[3] X[4] X[5] X[6] X[7]

stride = 1, size = 4

X[0] X[2] X[4] X[6]

stride = 2, size = 2

X[0] X[4]

stride = 4, size = 1

X[0]

Questions:

- ***How can we implement this schema using future tasks instead?***
- ***Can we avoid overwriting elements of array X?***

Array Sum using Future Tasks (ArraySum2)

Recursive divide-and-conquer pattern

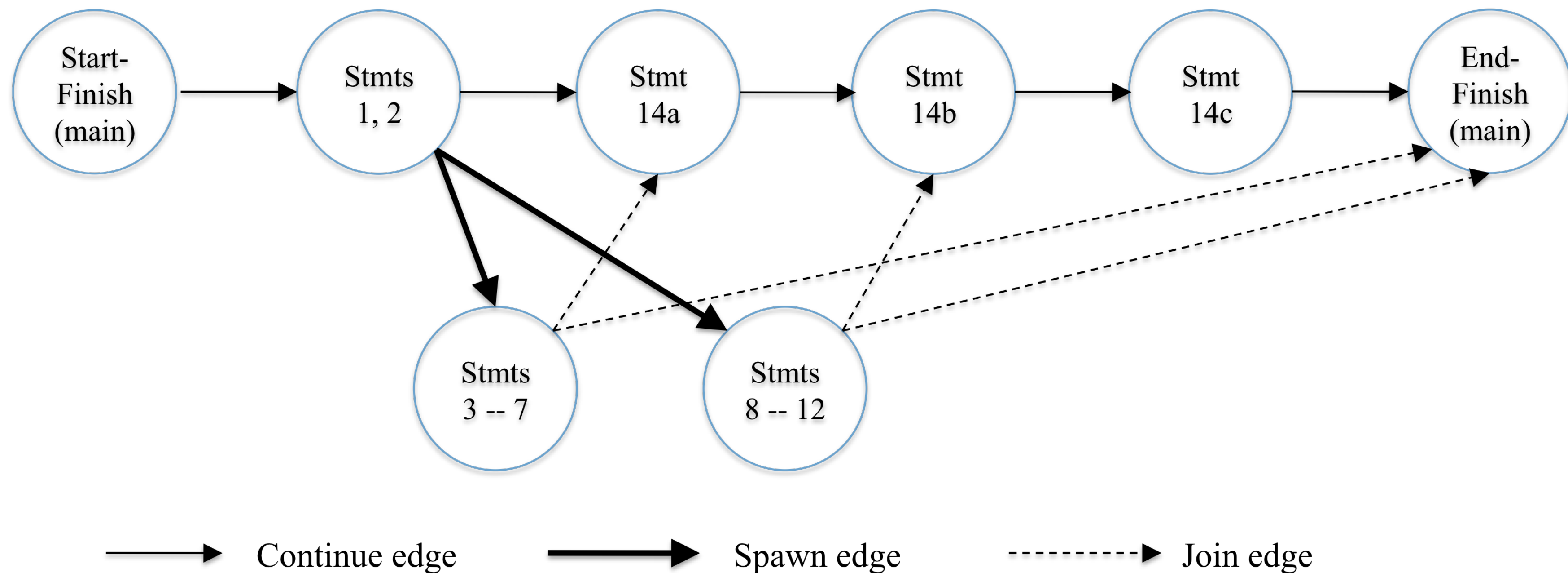
```
1.  static int computeSum(int[] X, int lo, int hi) {
2.    if ( lo > hi ) return 0;
3.    else if ( lo == hi ) return X[lo];
4.    else {
5.      int mid = (lo+hi)/2;
        final future<int> sum1 =
6.            async<int> { return computeSum(X, lo, mid); };
7.      final future<int> sum2 =
8.           async<int> { return computeSum(X, mid+1, hi); };
9.     // Parent now waits for the container values
10.     return sum1.get() + sum2.get();
11.    }
12.  } // computeSum
13. int sum = computeSum(X, 0, X.length-1); // main program
```

Computation Graph Extensions for Future Tasks

- Since a get() is a blocking operation, it must occur on boundaries of CG nodes/steps
  - May require splitting a statement into sub-statements e.g.,
    - 14: int sum = sum1.get() + sum2.get();

    can be split into three sub-statements
    - 14a int temp1 = sum1.get();
    - 14b int temp2 = sum2.get();
    - 14c int sum = temp1 + temp2;
- *Spawn* edge connects parent task to child future task, as before
- *Join edge* connects end of future task to Immediately Enclosing Finish (IEF), as before
- *Additional join edges* are inserted from end of future task to each get() operation on future object

Computation Graph for Two-way Parallel Array Sum using Future Tasks



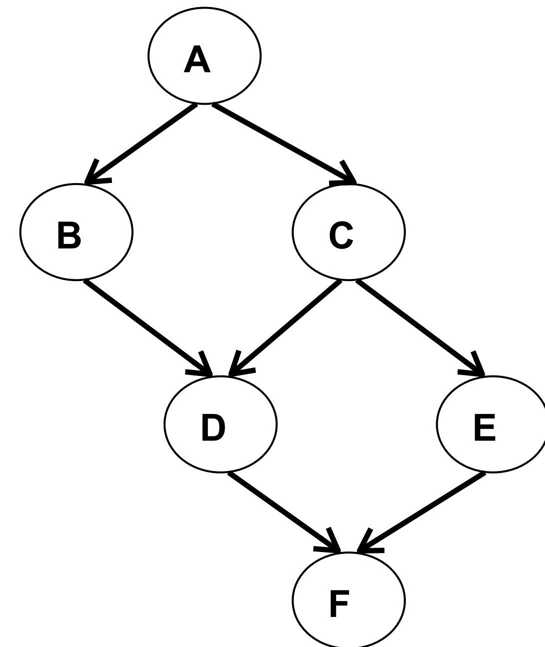
NOTE: DrHJ's data race detection tool does not support futures as yet (it only supports finish, async, and isolated constructs)

Worksheet #6: Computation Graphs for Async-Finish and Future Constructs

Name 1: ______________________ Name 2: ______________________

1) Can you write an HJ program with <u>async-finish</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

2) Can you write an HJ program with <u>future async-get</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.



Use the space below for your answers