COMP 322: Fundamentals of Parallel Programming

Lecture 7: Memory Models (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
Goals for Today’s Lecture

- Code Transformations and Memory Consistency Models
- Futures --- Tasks with Return Values
Memory Consistency Models (Recap)

- A memory consistency model, or memory model, is the part of a programming language specification that defines what write values a read may see in the presence of data races.

Example HJ program:

```java
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. }
```
A memory consistency model, or memory model, is the part of a programming language specification that defines what write values a read may see in the presence of data races.

The following reads are prohibited by Sequential Consistency (SC), but permitted by the Java Memory Model (JMM) and Habanero-Java Memory Model (HJMM).
A Code Transformation is said to be semantics-preserving if the transformed program, $P'$, exhibits the same Input-Output behavior as the original program, $P$.

For sequential programs, many local transformations are guaranteed to be semantics-preserving regardless of the context—e.g., replacing the second access of an object field or array element by a local variable containing the result of the first access, if there are no possible updates between the two accesses.

```java
P
1. static void foo(T p, T q) {
2.     System.out.println(p.x);
3.     System.out.println(q.x);
4.     System.out.println(p.x);
5. }
```

```java
P'
1. static void foo(T p, T q) {
2.     int xLocal = p.x
3.     System.out.println(xLocal);
4.     System.out.println(q.x);
5.     System.out.println(xLocal);
6. }
```
Semantics-Preserving Code Transformations in Parallel Programs

• **Question**: What should we expect if we perform a Code Transformation on a sequential region of a parallel program, if the transformation is known to be semantics-preserving for sequential programs?

• **Answer**: The transformation should be semantics-preserving for the parallel program if there are no data races. Otherwise, it depends on the memory model!

1. p.x = 0; q = p;
2. async p.x = 1;
3. async p.x = 2;
4. async foo(p, p);
5. async foo(p, q);
6. ...
7. static void foo(T p, T q) {
   8.   System.out.println(p.x);
   9.   System.out.println(q.x);
  10.   System.out.println(p.x);
  11. }

1. p.x = 0; q = p;
2. async p.x = 1;
3. async p.x = 2;
4. async foo(p, p);
5. async foo(p, q);
6. ...
7. static void foo(T p, T q) {
   8.   System.out.println(p.x);
   9.   System.out.println(q.x);
  10.   System.out.println(xLocal);
  11. }

Is this a legal transformation?

It may result in the following output:

0 0 0
1 2 1

===> Code transformation is legal for JMM & HJMM, but not for SC!
Summary of Memory Model Discussion

- Memory model specifies rules for what write values can be seen by reads in the presence of data races
  - In the absence of data races, program semantics specifies exactly one write for each read

- A local code transformation performed on a sequential code region may be semantics-preserving for sequential programs, but not necessarily for parallel programs
  - Stronger memory models (e.g., SC) are more restrictive about permissible read sets than weaker memory models (e.g., JMM, HJMM), and thus more restrictive about allowing transformations

- Different memory models are appropriate for different levels of the software stack
  - e.g., SC at the OS/HW level, JMM at the thread level, HJMM at the task level
Goals for Today’s Lecture

• Code Transformations and Memory Consistency Models
• Futures --- Tasks with Return Values
Extending Async Tasks with Return Values

• Example Scenario in PseudoCode
  1. // Parent task creates child async task
  2. container = async<int>  { return computeSum(X, low, mid); };
  3. 
  4. // Later, parent examines the return value
  5. int sum = container.get();

• Two key issues to be addressed:
  1) Distinction between container and value in container
  2) Synchronization to avoid race condition in container accesses

Parent Task

Child Task

container = async {...}
... 
container.get()
computeSum(...)
return ...

container

return value
HJ Futures: Tasks with Return Values

```plaintext
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()

• Guaranteed absence of race conditions in Future Task example
  — 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
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: \[ \text{int sum} = \text{sum1.get()} + \text{sum2.get();} \]
      can be split into three sub-statements
    - 14a \[ \text{int temp1 = sum1.get();} \]
    - 14b \[ \text{int temp2 = sum2.get();} \]
    - 14c \[ \text{int sum = temp1 + temp2;}} \]
- 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: Generation of computation graphs and data race detection in current HJ implementation do not support futures as yet
Reduction Tree Schema in ArraySum1 (Recap)

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;
6.         final future<int> sum1 =
7.             async<int> { return computeSum(X, lo, mid); };
8.         final future<int> sum2 =
9.             async<int> { return computeSum(X, mid+1, hi); };
10.     } // Parent now waits for the container values
11.     return sum1.get() + sum2.get();
12. } // computeSum
13. int sum = computeSum(X, 0, X.length-1); // main program
Extension of ArraySum2 to reduce an arbitrary associative function, \( f \)

1. static int reduce(int[] X, int lo, int hi) {
2.   if ( lo > hi ) return identity();
3.   else if ( lo == hi ) return X[lo];
4.   else {
5.     int mid = (lo+hi)/2;
6.     final future<int> sum1 =
7.       async<int> {return computeSum(X, lo, mid)};
8.     final future<int> sum2 =
9.       async<int> {return computeSum(X, mid+1, hi)};
10.    return f(sum1.get(), sum2.get());
11.  }
12. } // computeSum
13. int retVal = reduce(X, 0, X.length-1); // main program
Extra dependences in ArraySum1 program (for-finish-for-async)

- Extra dependence edges due to finish-async stages (not present in ArraySum2 version with futures)

- Which of ArraySum1 or ArraySum2 will perform better if the time taken by the reduction operator depends on its inputs e.g., as in WordCount?
Why must Future References be declared as final?

static future<int> f1=null;
static future<int> f2=null;

void main(String[] args) {
    f1 = async<int> {return a1();};
    f2 = async<int> {return a2();};

    int a1() { // Task T1
        while (f2 == null); // spin loop
        return f2.get(); //T1 waits for T2
    }

    int a2() { // Task T2
        while (f1 == null); // spin loop
        return f1.get(); //T2 waits for T1
    }

    cyclic wait condition

    • Above situation cannot arise in HJ because f1 and f2 must be final
    • Final declaration ensures that variable (handle) cannot be modified after initialization
    • WARNING: such spin loops are an example of bad parallel programming practice in application code (they should only be used by expert systems programmers, and even then sparingly)
      • Their semantics depends on the memory model!
Future Tasks with void Return Type

- **Key difference between regular async's and future tasks** is that future tasks have a `future<T>` return value.
- **We can get an intermediate capability by setting** `T=void` **as shown.**
- **Can be useful if a task needs to synchronize on a specific task** (instead of finish), but doesn't need a future object to communicate a return value.

```java
1. sum1 = 0; sum2 = 0; // Task T1
2. // Assume that sum1 & sum2 are fields
3. final future<void> a1 = async<void> {
4.   for (int i=0; i < X.length/2; i++)
5.     sum1 += X[i]; // Task T2
6. }
7. final future<void> a2 = async<void> {
8.   for (int i=X.length/2; i < X.length; i++)
9.     sum2 += X[i]; // Task T3
10. }
11. //Task T1 waits for Tasks T2 and T3
12. a1.get(); a2.get();
13. // Now fields sum1 and sum2 can be read
```
Future Tasks can generate more general Computation Graphs than regular Async Tasks

Can you write a finish-async HJ program that generates the following Computation Graph?
Using Future Tasks to generate previous Computation Graph

1. // NOTE: return statement is optional
2. // when return type is void
3. final future<void> A = async<void>
4.   { . . . };
5. final future<void> B = async<void>
6.   { A.get(); . . . };
7. final future<void> C = async<void>
8.   { A.get(); . . . };
9. final future<void> D = async<void>
10.  { B.get(); C.get(); . . . };
11. final future<void> E = async<void>
12.  { C.get(); . . . };
13. final future<void> F = async<void>
14.  { D.get(); E.get(); . . . }

Computation Graph
Homework 2 Reminder

• Programming assignment, due Monday, Jan 30th

• Post questions on Piazza (preferred), or send email to comp322-staff at mailman.rice.edu

• You should plan to use turn-in script for HW2 submission
  — Contact teaching staff if you cannot access turn-in by following the instructions for Lab 1

• See course web site for penalties for late submissions