# COMP 322: Fundamentals of Parallel Programming

# Lecture 36: Volatile Variables, Memory Consistency Models

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

**Contact email: vsarkar@rice.edu**

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

## Worksheet #35 solution: UPC data distributions

In the following example from slide 22, assume that each UPC array is distributed by default across threads with a cyclic distribution. In the space below, identify an iteration of the upc_forall construct for which all array accesses are local, and an iteration for which all array accesses are non-local (remote). Explain your answer in each case.

```
shared int a[100],b[100], c[100];
int i;
upc_forall (i=0; i<100; i++; (i*THREADS)/100)
   a[i] = b[i] * c[i];
```

**Solution:**

- Iteration 0 has affinity with thread 0, and accesses a[0], b[0], c[0], all of which are located locally at thread 0
- Iteration 1 has affinity with thread 0, and accesses a[1], b[1], c[1], all of which are located remotely at thread 1

# Memory Visibility

- Basic question: if a memory location L is written by statement S1 in thread T1, when is that write guaranteed to be visible to a read of L in statement S2 of thread T2?
- General answer: whenever there is a directed path of edges from S1 in S2 in the computation graph
  - Computation graph edges are defined by semantics of parallel HJlib constructs — e.g., async, finish, async-await, futures, phasers, isolated, object-based isolation — and can be defined for parallel Java constructs in a similar manner
  - This directed path of edges is also referred to as a "happens-before" relation from S1 to S2

# Troublesome example

```
1. public class NoVisibility {
2.  private static boolean ready;
3.  private static int number;
4.
5.  private static class ReaderThread extends Thread {
6.    public void run() {
7.      while (!ready) Thread.yield()
8.      System.out.println(number)
9.    }
10.  }
11.
12.  public static void main(String[] args) {
13.    new ReaderThr
14.    number = 42;
15.    ready = true;
16.  }
17. }
```

No happens-before ordering between main thread and ReaderThread
==> ReaderThread may loop forever OR may print 42 OR may print 0 !!

# Volatile Variables in Java

- Java provides a "light" form of synchronization/fence operations in the form of `volatile` variables (fields)
- Volatile variables guarantee visibility
  - Reads and writes of volatile variables should be assumed to occur in "location-based isolated blocks", which are finer-grained than object-based isolated blocks
  - Adds serialization edges to computation graph due to isolated read/write operations on same volatile variable
- Incrementing a volatile variable (++v) is <u>not thread-safe</u>
  - Increment operation looks atomic, but isn't (read and write are two separate operations in "v = v + 1"). Better to use AtomicInteger instead.
- Volatile variables are often used for flag variables that implement synchronization patterns e.g.,

```
volatile boolean asleep;
foo() { ... while (! asleep) ++sheep; ... }
```

— WARNING: In the absence of volatile declaration, the above code can legally be transformed to the following (much better to use explicit synchronization construct instead e.g., Eureka or Data-Driven Task)

```
boolean asleep;
foo(){ boolean temp=asleep; ... while (! temp) ++sheep; ... }
```

# Troublesome example fixed with volatile declaration

```
1. public class NoVisibility {
2.   private static volatile boolean ready;
3.   private static volatile int number;
4.
5.   private static class ReaderThread extends Thread {
6.     public void run() {
7.       while (!ready) Thread.yield()
8.       System.out.println(number)
9.     }
10.  }
11.
12.  public static void main(String[] args) {
13.    new ReaderThread().
14.    number = 42;
15.    ready = true;
16.  }
17. }
```

Declaring number and ready as volatile ensures happens-before-edges: 14-->15-->7-->8, thereby ensuring that only 42 will be printed

# Data Races on non-volatile variables are usually errors, but not always

- Example of Data Race Error

```
1.  for ( p = first; p != null; p = p.next)
2.      async p.x = p.y + p.z;
3.  for ( p = first; p != null; p = p.next)
4.      sum += p.x;
```

- Example of intentional (benign) data race
  - Search algorithm that returns any match (need not be the first match)

```
5.  static int index = -1; // static field
6.  . . .
7.  finish for (int i = 0; i <= N - M; i++) async {
8.    for (j = 0; j < M; j++)
9.      if (text[i+j] != pattern[j]) break;
10.   if (j == M) index = i;          // found at offset i
11. }
```

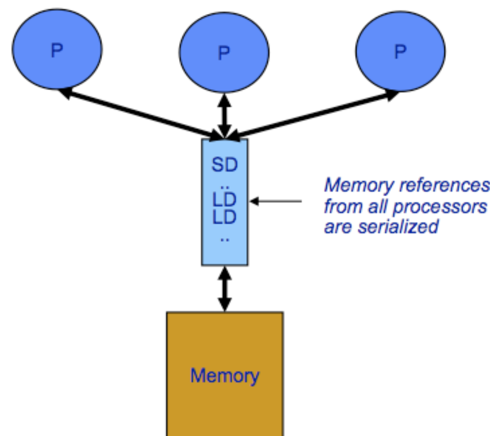
- In both cases, the semantics of data races still needs to be fully specified

# Memory Consistency Models — Rules for specifying Semantics of Data Races

- 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 model details*
- 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) — strongest (smallest set of writes for a read)
    - – 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) — weakest (largest set of writes for a read)
    - – Useful for specifying semantics at application task level e.g., HJ programs

| HJMM |
| --- |
| JMM |
| SC |

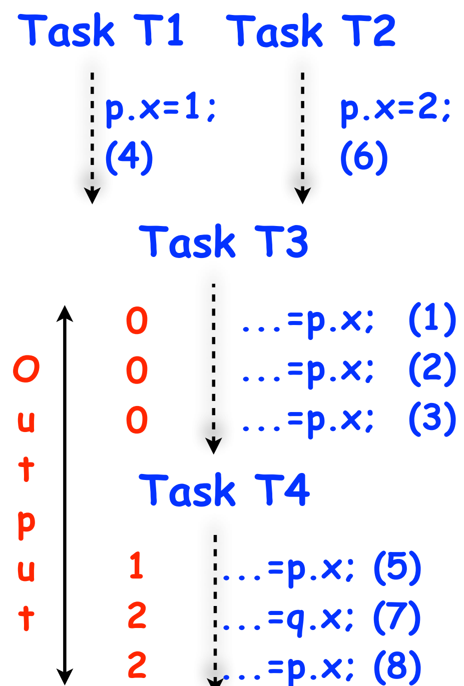
# Sequential Consistency Memory Model




[Lamport] "*A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program*"

# Sequential Consistency (SC) Memory Model

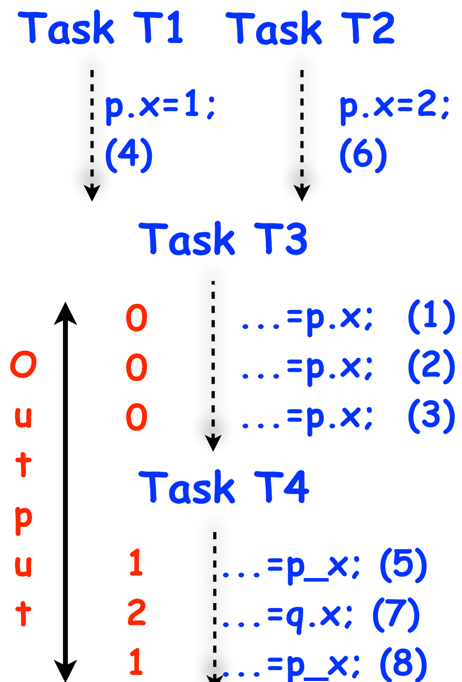
- SC constrains all memory operations across all tasks
  - Write → Read
  - Write → Write
  - Read → Read
  - Read → Write
- Simple model for reasoning about data races at the hardware level, but may lead to counter-intuitive behavior at the application level e.g.,
  - A programmer may perform modular code transformations for software engineering reasons without realizing that they are changing the program's semantics



# Consider a “reasonable” code transformation performed by a programmer

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.}
```

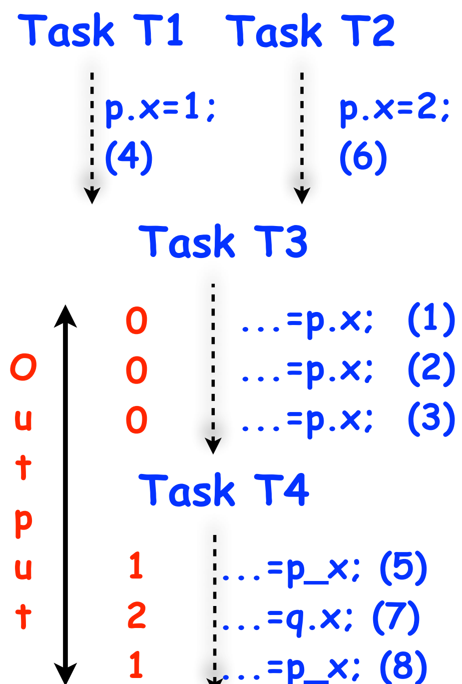


# Consider a “reasonable” code transformation performed by a programmer

Example HJ program:

```
1. p.x = 0; q = p;
2. async p.x = 1; // Task T1
3. async p.x = 2
4. async {
5.   Syst
6.   Syst
7.   Syste
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.}
```

This reasonable code transformation resulted in an illegal output, under the SC model!

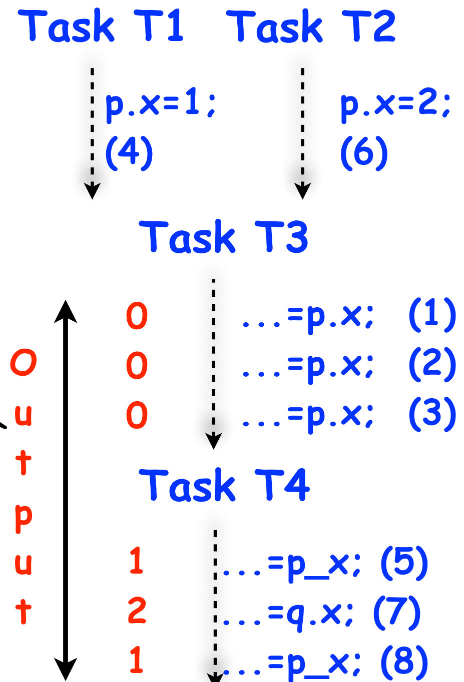


# Code Transformation Example

Example HJ program:

```
1. p.x = 0; q = p;
2. async p.x = 1; // Task T1
3. async p.x = 2
4. async {
5.    Syst
6.    Syst
7.    Syste
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.}
```

This output is legal under the JMM and HJMM!

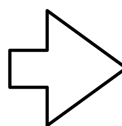


# Semantics-Preserving Code Transformations in Sequential Programs

- 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

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. }
```

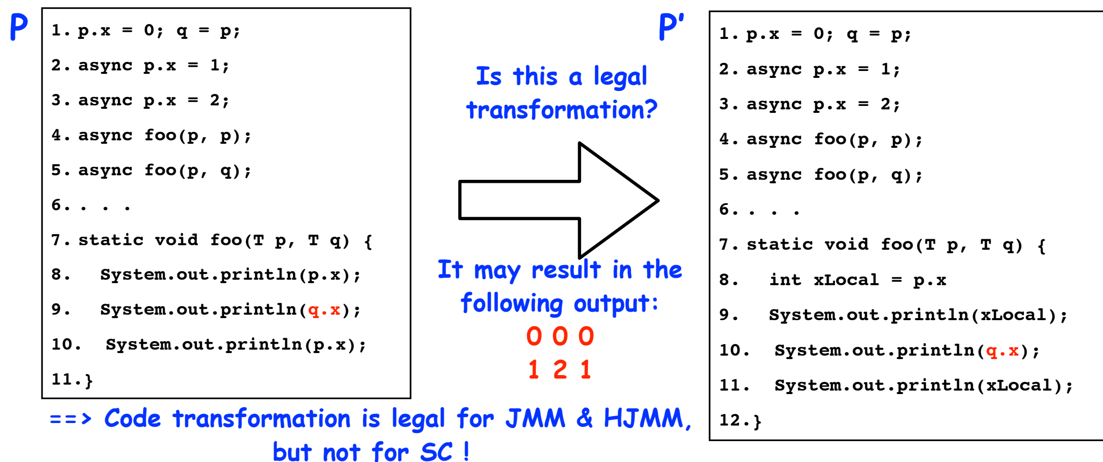


P'

```
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. }
```

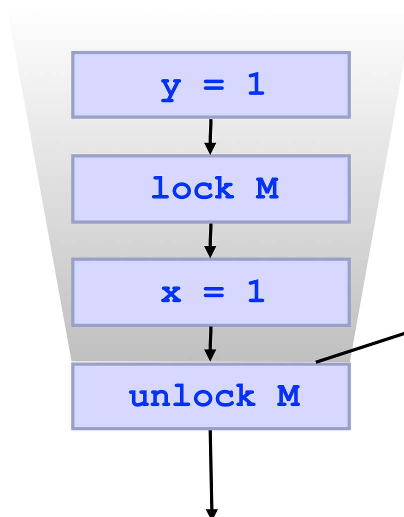
# Semantics-Preserving Code Transformations in Parallel Programs

- 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!

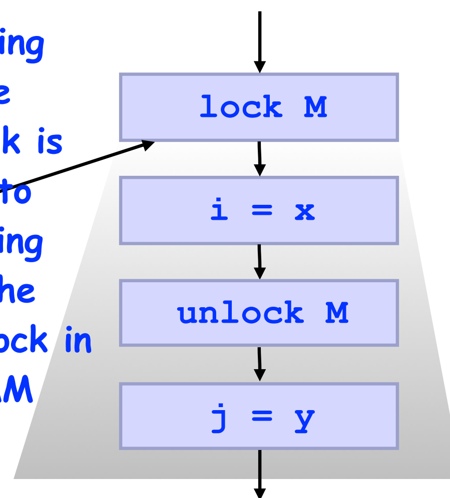



==> Code transformation is legal for JMM & HJMM, but not for SC !

# When are actions visible and ordered with other Threads in the JMM?




Everything before the unlock is visible to everything after the matching lock in the JMM




lock/unlock operations can come from synchronized statement or from explicit calls to locking libraries

# Troublesome example fixed with empty synchronized statements instead of

```
1. public class NoVisibility {
2.  private static boolean ready;
3.  private static int number;
4.  private static final Object a = new Object();
5.
6.  private static class ReaderThread extends Thread {
7.    public void run() {
8.      synchronized(a){}
9.      while (!ready) { Thread.yield(); synchronized(a){} }
10.     System.out.println(number);
11.    }
12.  }
13.
14.  public static void main(String[] args) {
15.    new ReaderThread().start();
16.    number = 42;
17.    ready = true; synchronized(a){}
18.  }
19. }
```

Empty synchronized statement is NOT a no-op in Java. It acts as a memory "fence".

# Empty isolated statements are no-ops in HJ

```
1. public class NoVisibility {
2.  private static boolean ready;
3.  private static int number;
4.
5.  private static class ReaderThread extends Thread {
6.    public void run() {
7.      isolated{}
8.      while (!ready) { Thread.yield(); isolated{} }
9.      System.out.println(number);
10.    }
11.  }
12.
13.  public static void main(String[] args) {
14.    new ReaderThread().start();
15.    number = 42;
16.    ready = true; isolated {}
17.  }
18. }
```

Empty isolated statement is a no-op in HJ. ReaderThread may loop forever OR may print 42 OR may print 0.

# Better to use explicit synchronization in HJ instead

```
1. public class NoVisibility {
2.  private static boolean ready;
3.  private static int number;
4.  private static DataDrivenFuture<Boolean>
5.      readyDDF = new DataDrivenFuture<Boolean>();
6.
7.  public static void main(String[] args) {
8.    async await(readyDDF) {System.out.println(number);}
9.    number = 42;
10.   readyDDF.put(true);
11.  }
12. }
```

# 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

| HJMM |
| --- |
| JMM |
| SC |