COMP 322: Fundamentals of Parallel Programming

Lecture 6: Memory Models, Atomic Variables

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

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

Acknowledgments for Today's Lecture

- "Introduction to Concurrent Programming in Java", Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Brian Goetz
  - Contributing authors: Doug Lea, Tim Peierls, Brian Goetz
- "Engineering Fine-Grained Parallelism Support for Java 7", Doug Lea, July 2010
- "Java Concurrency in Practice", Brian Goetz with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea. Addison-Wesley, 2006.

Goals for Today's Lecture

- <u>Data Race Semantics and Memory Consistency Models</u>
- Java's atomic integer classes (AtomicInteger, AtomicIntegerArray, AtomicLong, AtomicLongArray)

Data Races are usually Errors, but not always

- Example of Data Race Error

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

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

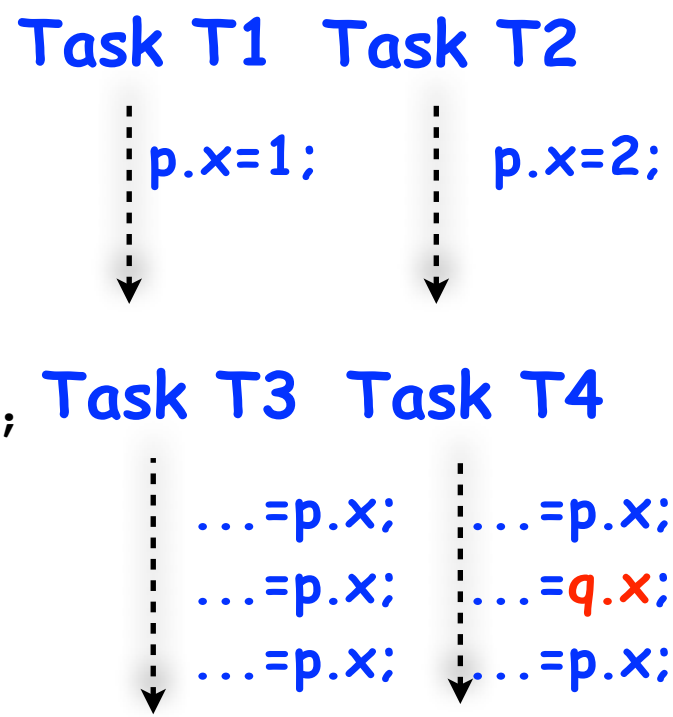
```
static int index = -1; // static field
. . .
finish for (int i = 0; i <= N - M; i++) async {
  for (j = 0; j < M; j++)
    if (text[i+j] != pattern[j]) break;
  if (j == M) index = i;          // found at offset i
}
```

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

Semantics of Data Races

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.  System.out.println("First read = " + p.x);
11.  System.out.println("Second read = " + q.x);
12.  System.out.println("Third read = " + p.x);
13.}
```



Can the following values be printed by tasks T3 & T4?

T3: 0, 0, 0
T4: 1, 2, 1

Program Order != Reality, for Racy Programs

- Programmer's view:
  - Everything happens in the order I indicate through the code statements that I write
- Reality (JVM/compiler & hardware processor):
  - Everything happens in whatever order yields best performance, so long as the program(mer) can't tell the difference
- For data-race-free programs
  - Program order can't be distinguished from actual order
- For "racy" programs
  - Different tasks can see different actions in memory

    At different times

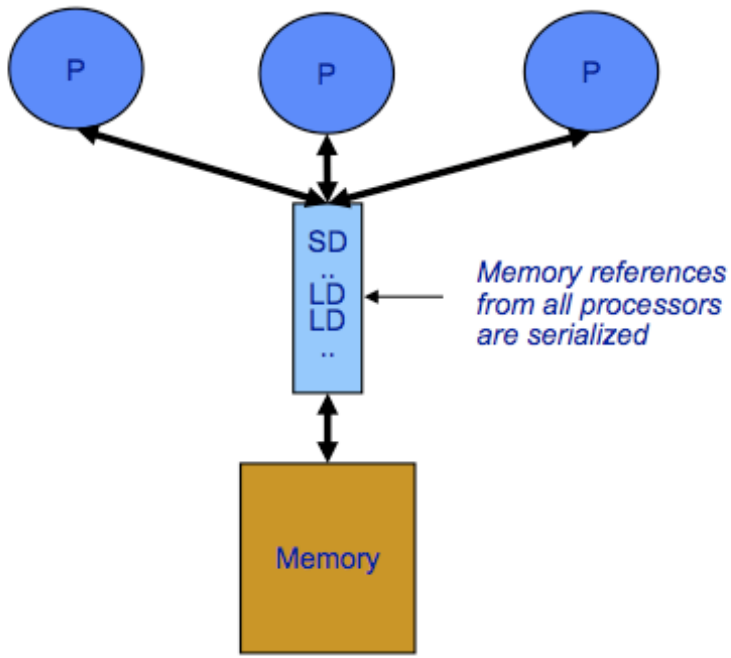
    In different orders

Memory Consistency Models

- A memory consistency model, or <u>memory model</u>, is the part of a programming language specification that defines what write values a read may see in the presence of data races.
- We will briefly introduce three memory models, and discuss them in more detail later in the course
  - Sequential Consistency (SC)
    - Suitable for specifying semantics at the hardware and OS levels *
  - Java Memory Model (JMM)
    - Suitable for specifying semantics at application thread level *
  - Habanero Java Memory Model (HJMM)
    - Suitable for specifying semantics at application task level *

* This is your instructor's opinion. Memory models are a very controversial topic in parallel programming!

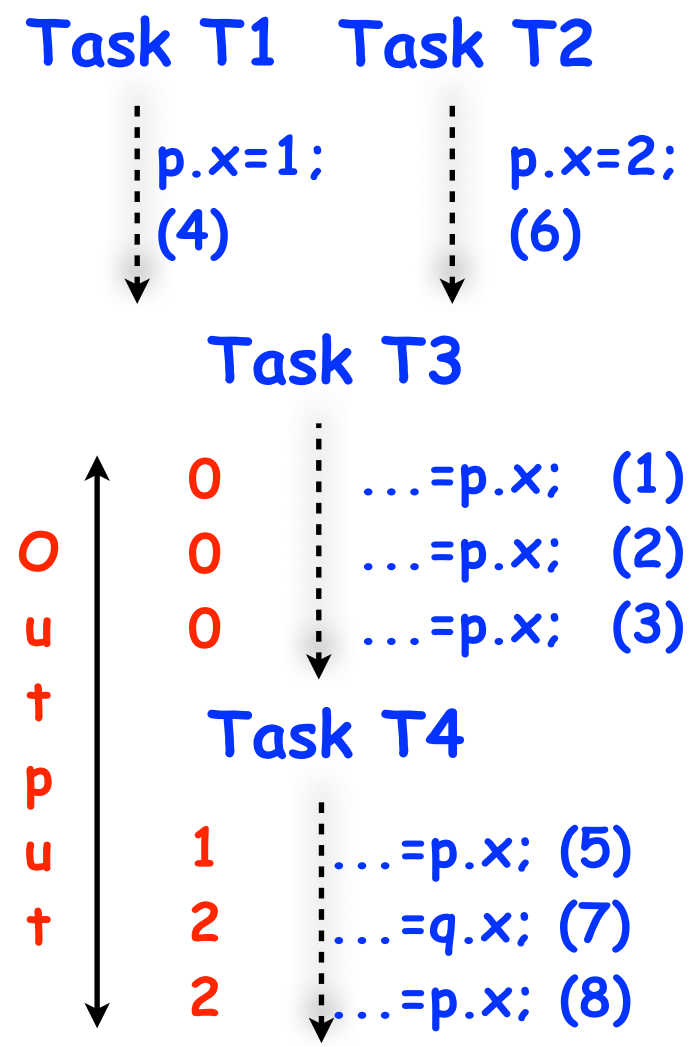
Sequential Consistency Memory Model



[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

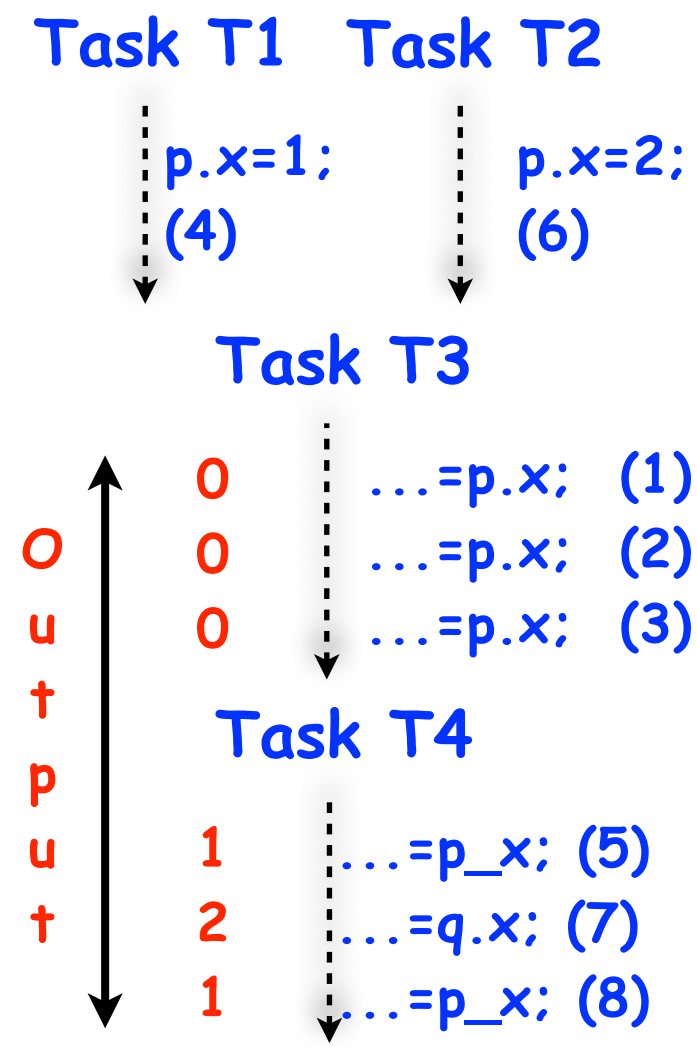
- SC constrains all memory operations across all tasks
  - Write → Read
  - Write → Write
  - Read → Read
  - Read → Write
- Simple model for reasoning about data races at the hardware level, but may lead to counter-intuitive behavior at the application level e.g.,
  - A programmer may perform modular 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. }
```

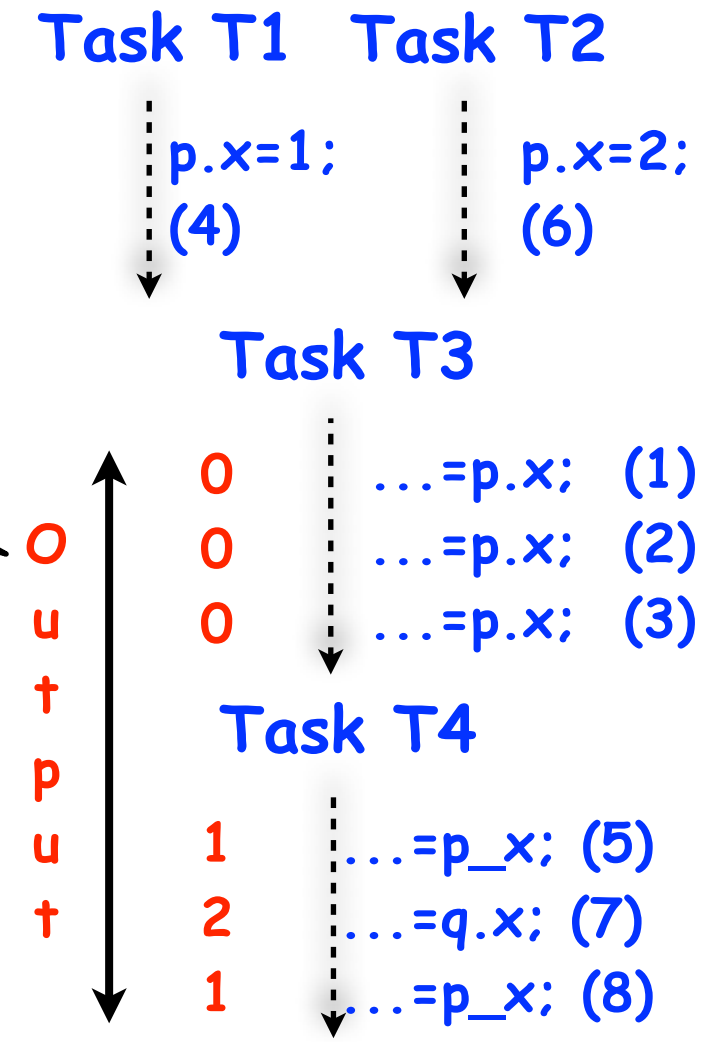


Consider a "reasonable" code transformation performed by a programmer

Example HJ program:

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

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



The Java Memory Model (JMM) and the Habanero-Java Memory Model (HJMM)

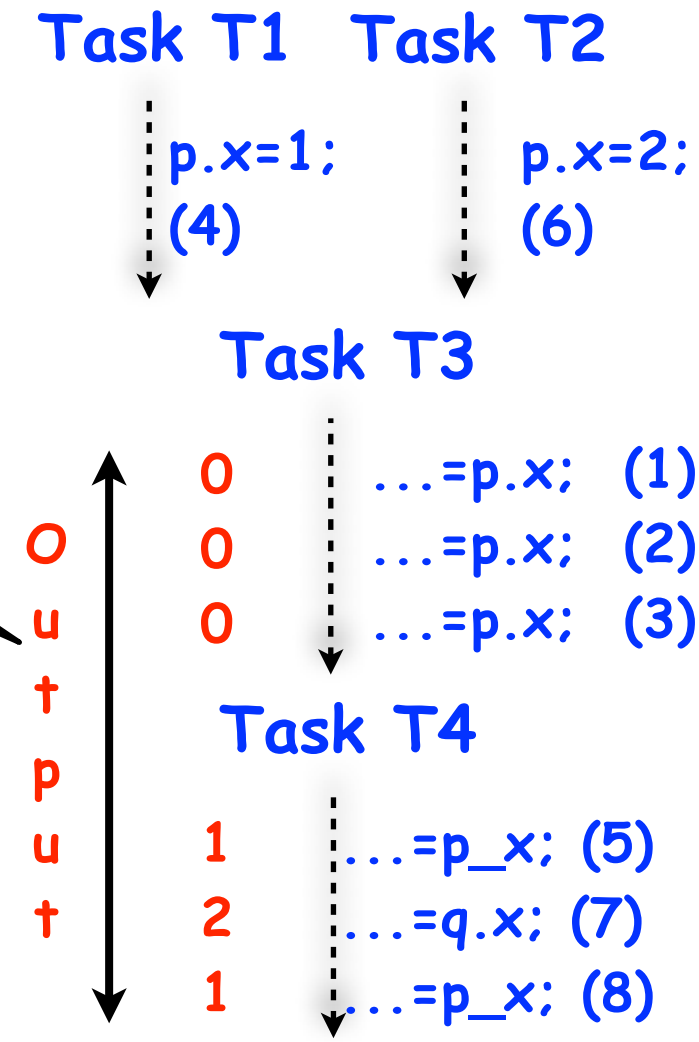
- Conceptually simple:
  - Every time a variable is written, the value is added to the set of "most recent writes" to the variable
  - A read of a variable is allowed to return ANY value from this set
- The JMM defines the rules by which values in the set are removed
  - By using ordering relationships ("happens-before") similar to the Computation Graph to determine when a value must be overwritten
- HJMM has weaker ordering rules for HJ's "isolated" statements, compared to Java's "synchronized" blocks
  - To be discussed later in the course
- Programmer's goal: through proper use of synchronization
  - Ensure the absence of data races, in which case this set will never contain more than one value and SC, JMM, HJMM will all have the same semantics

Code Transformation Example

Example HJ program:

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

This output is legal under the JMM and HJMM!



Goals for Today's Lecture

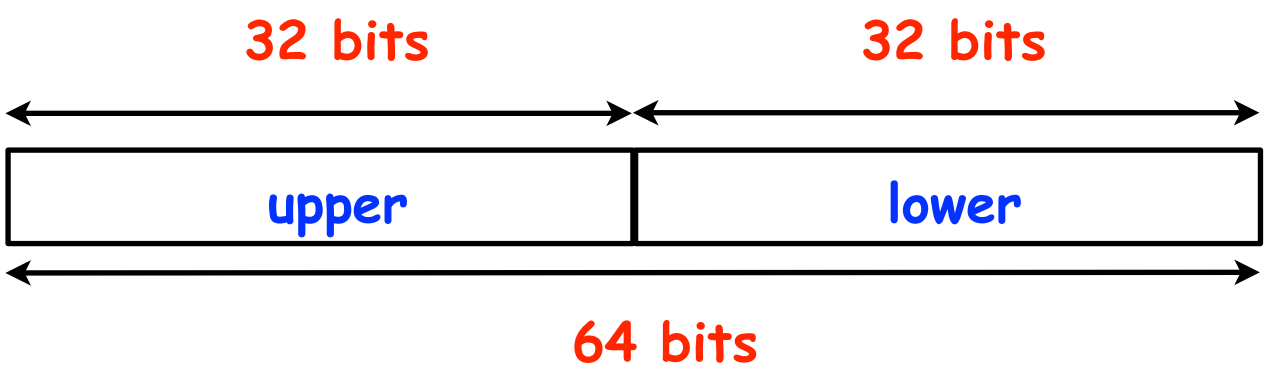
- Data Race Semantics and Memory Consistency Models
- <u>Java's atomic integer classes (AtomicInteger, AtomicIntegerArray, AtomicLong, AtomicLongArray)</u>

Atomic Accesses

- An atomic action happens all at once
  - Subcomponents of one atomic action cannot be interleaved with subcomponents of another atomic action
- Reads and write for reference variables and primitives (except `long` and `double`) are atomic
- Basic safety guarantee: No "out-of-thin-air" values for references and primitives (except for `long` and `double`)
  - A read always returns a value written by some task, some time in the past

Why reads and writes on long/double values may be non-atomic

```
long x; // upper = lower = 0
async { x = 1L << 32 + 1L; } // lower=1; upper=1;
async { x = 2L << 32 + 2L; } // lower=2; upper=2;
async { System.out.println(x); }
// Possible output value includes
// 1L << 32 + 2L (lower=2, upper=1)
```



Implementing Shared Counters

- There are many algorithms in which parallel tasks need to atomically increment a shared counter
  - Challenge: an increment (x = x+1) consists of a read and a write
  - Even if the read and write are individually atomic, the increment operation is not
- Java provides a library of "atomic variables" for which each individual method can be assumed to be an atomic operation
- Atomic operations can be safely invoked on parallel tasks, but they may increase the critical path length of your parallel program
  - Not a problem if the remaining parallel (non-atomic) work is large

java.util.concurrent library

- <u>Atomic variables</u>
  - Efficient implementations of special-case patterns of isolated statements
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, …
  - Data structures designed for concurrent environments
- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Synchronizers: Semaphore, Latch, Barrier, Exchanger, Phaser
  - Tools for thread coordination
- <u>WARNING: only a small subset of the full java.util.concurrent library can safely be used in HJ programs</u>
  - Atomic variables are part of the safe subset
  - We will study the full library later this semester as part of Java Concurrency

java.util.concurrent.atomic.AtomicInteger

- Constructors
  - `new AtomicInteger()`
    - Creates a new AtomicInteger with initial value 0
  - `new AtomicInteger(int initialValue)`
    - Creates a new AtomicInteger with the given initial value
- Selected methods
  - `int addAndGet(int delta)`
    - Atomically adds delta to the current value of the atomic variable, and returns the new value
  - `int getAndAdd(int delta)`
    - Atomically returns the current value of the atomic variable, and adds delta to the current value
- Similar interfaces available for `LongInteger`
  - No worry about lower/upper half issues when using a LongInteger atomic variable

Summing Values from Multiple Async's in same Finish Scope

- With ArraySum, you learned how to sum an array to a single value
- How can we perform a sum on values generated by dynamic async statements?
- Example 1: compute sum of elem values from async tasks in a loop

```
finish while (...)
  async { ...; elem = ...; ...; }
```

- Example 2: compute sum of elem values from async tasks in a recursive method

```
void visit(...)
{ ...; elem = ...; async visit(...); ...; }
... finish visit(...); ...
```


Solution for Examples 1 and 2 using AtomicInteger

```
1.  import java.util.concurrent.atomic.AtomicInteger;
2.  // Example 1: compute sum from async tasks in a loop
3.  AtomicInteger a1 = new AtomicInteger();
4.    finish while(...)
5.     async { ...; elem = ...; a1.addAndGet(elem); ...; }
6.  // Example 2: compute sum in a recursive method
7.  AtomicInteger a2 = new AtomicInteger();
8.  void visit(...)
9.  { ...; elem = ...; a2.addAndGet(elem);
10.   async visit(...); ...;
11. }
12. ... finish visit(...); ...
```

Work-Sharing Pattern using AtomicInteger

```
1.  import java.util.concurrent.atomic.AtomicInteger;
2.  . . .
3.  String[] X = ... ; int numTasks = ...;
4.  AtomicInteger a = new AtomicInteger();
5.  . . .
6.  finish for (int i=0; i<numTasks; i++ )
7.    async {
8.      do {
9.          int j = a.getAndAdd(1);
10.         // can also use a.getAndIncrement()
11.         if (j >= X.length) break;
12.         . . . // Process X[j]
13.       } while (true);
14.     } // finish-for-async
```


Solution Counting Pattern using AtomicInteger

```
import java.util.concurrent.atomic.AtomicInteger;
. . .
AtomicInteger count = new AtomicInteger();
finish nqueens_kernel(new int[0], 0);
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) count.addAndGet(1);
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) async {
      /* allocate a temporary array and copy array a into it */
      int [] b = new int [depth+1];
      System.arraycopy(a, 0, b, 0, depth);
      b[depth] = i;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```


java.util.concurrent.atomic.AtomicIntegerArray

- Constructors
  - `new AtomicIntegerArray(int length)`
    - Creates a new array of AtomicInteger's with initial values of 0
- Selected methods
  - `int addAndGet(int i, int delta)`
    - Atomically adds delta to the i'th element of the atomic array, and returns the new value
  - `int getAndAdd(int i, int delta)`
    - Atomically returns the current value of the i'th element of the atomic array, and adds delta to the current value
- Similar interfaces available for `LongIntegerArray`