Common Correctness and Performance Issues

Unit 2.b

Acknowledgments

- Authored by
  - Sebastian Burckhardt, MSR Redmond

Concepts

Performance Concept

- Data locality, Cache Coherence
- False sharing
- Lock Overhead
- Lock Contention

Code Concept

- Interlocked
- Volatile

Correctness Concept

- Data Races
- Atomicity Violations
- Deadlocks, Lock Leveling

Parallel Performance: Not always easy

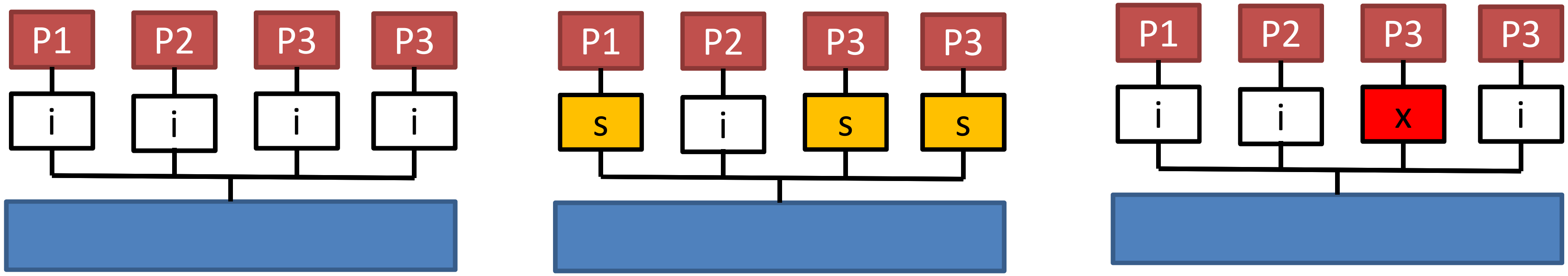
- Even if a problem is parallelizable in principle, there may be practical limitations
  - Takes time to start a task on a processor
  - Takes time to move data between processors
  - Takes time to synchronize tasks

- Anthropomorphic example: Imagine you have to write the numbers 1 through 1000 on a single sheet of paper.
  - If you are a team of 2 and well coordinated, you may indeed achieve a speed-up of about 2x
  - But can you achieve a speed-up of 100x with 100 friends?

Potential Performance Problems

- Task Overhead
  - Takes time to start a task and wait for its result
  - If amount of work done by task is very small, not worth doing in parallel
- Data Locality & Cache Behavior
  - Performance of computation depends HUGELY on how well the cache is working (i.e. how many of the memory accesses hit in the cache).
  - Naïve parallelization may cause too many cache misses, in particular if processors are “fighting” for the same cache lines

Cache Coherence



- Each cacheline, on each processor, has one of these states:
  - i - invalid : not cached here
  - s - shared : cached, but immutable
  - x - exclusive: cached, and can be read or written
- State transitions require communication between caches (cache coherence protocol)
  - If a processor writes to a line, it removes it from all other caches

Ping-Pong & False Sharing

- Ping-Pong
  - If two processors both keep writing to the same location, cache line has to go back and forth
  - Very inefficient (lots of cache misses)
- False Sharing
  - Two processors writing to two different variables may happen to write to the same cacheline
    - If both variables are allocated on the same cache line
  - Get ping-pong effect as above, and horrible performance

False Sharing Example

```
void WithFalseSharing()
{
    Random rand1 = new Random(), rand2 = new Random();
    int[] results1 = new int[20000000],
          results2 = new int[20000000];
    Parallel.Invoke(
        () => {
            for (int i = 0; i < results1.Length; i++)
                results1[i] = rand1.Next();
        },
        () => {
            for (int i = 0; i < results2.Length; i++)
                results2[i] = rand2.Next();
        });
}
```

False Sharing Example

```
void WithFalseSharing()
{
    Random rand1 = new Random(), rand2 = new Ra
    int[] results1 = new int[20000000],
          results2 = new int[20000000];
    Parallel.Invoke(
        () => {
            for (int i = 0; i < results1.Lengt
                results1[i] = rand1.Next();
        },
        () => {
            for (int i = 0; i < results2.Lengt
                results2[i] = rand2.Next();
        });
}
```

rand1, rand2
are allocated
at same time
=>
likely on same
cache line.

Call to Next()
writes to the
random
object
=>
Ping-Pong
Effect

False Sharing, Eliminated

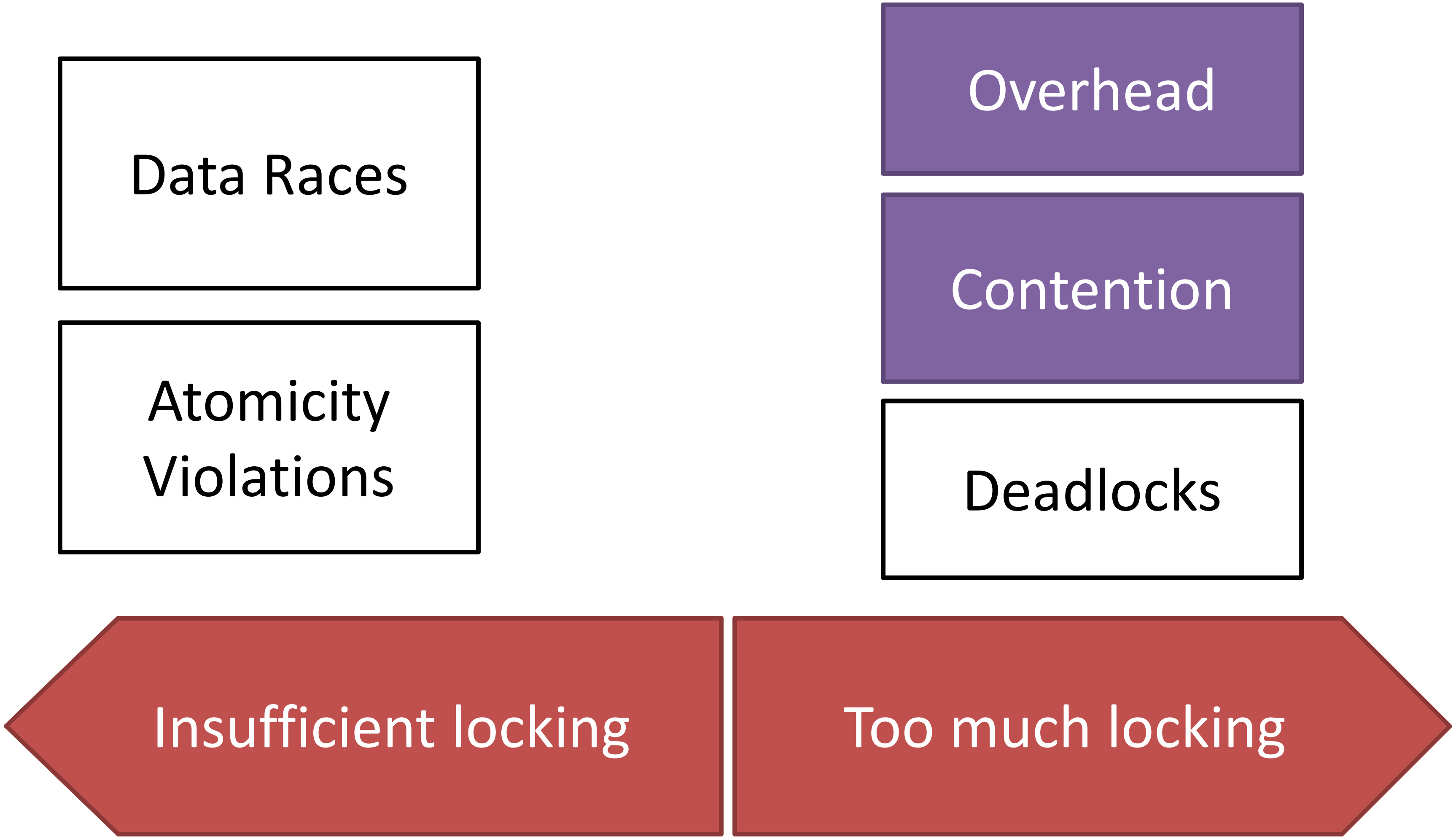
```
void WithoutFalseSharing()
{
    int[] results1, results2;
    Parallel.Invoke(
        () => {
            Random rand1 = new Random();
            results1 = new int[20000000];
            for (int i = 0; i < results1.Length; i++)
                results1[i] = rand1.Next();
        },
        () => {
            Random rand2 = new Random();
            results2 = new int[20000000];
            for (int i = 0; i < results2.Length; i++)
                results2[i] = rand2.Next();
        });
}
```

rand1, rand2 are allocated by different tasks
=>
Not likely on same cache line.

Part I

LOCKS AND PERFORMANCE

Common Problems With Locking



Data Races

Atomicity Violations

Overhead

Contention

Deadlocks

Insufficient locking

Too much locking

Example: Lock Contention

- Consider this example

```
Parallel.Invoke(
    () => { lock(gameboard) { MoveRobot(r1); } },
    () => { lock(gameboard) { MoveRobot(r2); } },
)
```

- There is no parallelism!
  - Only one task can work at a time
  - May as well write sequential code

CS13

Locking Tradeoffs

- **Coarse-Grained Locking**
  - Use few locks (e.g. single global lock)
    (i.e. many locations protected by the same lock)
  - Advantage: simple to implement, little overhead
  - Danger: lock contention may destroy parallelism

- **Fine-Grained Locking**
  - Use many locks (e.g. one lock for each object)
  - Advantage: more parallelism
  - Disadvantage: overhead, difficult to implement
  - Danger: may lead to atomicity violations
  - Danger: may lead to deadlocks

Slide 14

CS13 Second mention of atomicity violations before term is defined.
Caitlin Sadowski, 7/20/2010

Example: Locking Overhead

- Consider this sequential computation
  - Counts how many times each filename-length occurs

```
string[] filenames = /* large list of filenames */;

public void CountLengths()
{
    int[] count = new int[maxlength];
    foreach (string s in filenames)
        count[s.Length]++;
}
```

Example: Locking Overhead

- Consider this parallelization:

```
Parallel.For(0, filenames.Length, (int i) =>
          {
              int len = filenames[i].Length;
              lock (lockarray[len])
                  count[len]++;
          });
```

- Instead of a speedup we get 13x slowdown
- Problem: takes too much time to acquire and release locks

Three Main Suggestions

- Trick 1: Reduce need for locks by better partitioning the computation
- Trick 2: Reduce size of critical sections: leads to less contention; and may enable Trick 3
- Trick 3: Replace small critical sections with interlockeds and volatiles

Trick 1: Partition Computation

- Recall bad parallelization of histogram computation (13x slowdown):

```
Parallel.For(0, filenames.Length, (int i) =>
           {
               int len = filenames[i].Length;
               lock (lockarray[len])
                   count[len]++;
           });
```

- Can we reduce locking in this example?
  - Yes. Partition the computation into isolated pieces.

Partitioned Histogram Computation

```
Parallel.For(0, numpartitions, (int p) =>
{
     // create local count array
    int[] localcount = new int[maxlength];

    // count partition of filenames, store results in localcount
    for (int i = p * filenames.Length / numpartitions;
         i < (p + 1) * filenames.Length / numpartitions;
         i++)
                localcount[filenames[i].Length]++;

    // write localcounts to count – lock held only for short time
    lock (count)
    {
        for (int c = 0; c < maxlength; c++)
            count[c] += localcount[c];
    }
});
```

Trick 2: Reduce Size of Contended Critical Section

- **EXAMPLE:** Suppose
  - variable x is protected by lock a
  - lock a suffers from contention
  - compute() is a time-consuming computation that does not access x.

- Instead of

```
lock (a)
{
   x = computation() ;
}
```

- Write

```
int result = computation();
lock (a)
{
  x = result;
}
```

Trick 3: Interlocked/Volatile

- If your critical section contains a single operation only, such as
  - Reads a shared variable
  - Writes to a shared variable
  - Adds a number to a shared variable

- You can use interlocked or volatile operations instead of locks.

Example: Use Interlocked Operation

BEFORE:

```
Parallel.For(0, filenames.Length, (int i) =>
      {
          int len = filenames[i].Length;
          lock (lockarray[len])
              count[len]++;
      });
```

AFTER:

```
Parallel.For(0, filenames.Length, (int i) =>
    {
      Interlocked.Increment(ref count[filenames[i].Length]);
    });
```

Volatile Variables and Fields

- Add “volatile” type qualifier to field or variable
  - Means every access to that field or variable is considered a ‘volatile’ access
- If a critical section protects a single read or a single write, we can use a volatile read or write instead.

Example: Volatile/Interlockeds Can Replace Locks

```
class MyCounter()
{
   Object mylock =  new Object();
   int balance;
   public void Deposit(int what)
    {
      lock(mylock)
          balance = balance + what;
    }
    public int GetBalance()
    {
       lock(mylock)
           return balance;
    }
    public void SetBalance(int val)
    {
        lock(mylock)
           balance = val;
    }
```

```
class MyCounter()
{
   volatile int balance;

   public void Deposit(int what)
   {
      Interlocked.Add(ref balance, what)
   }
    public int GetBalance()
    {
       return balance; /* volatile read */
    }
    public int GetBalance(int val)
    {
       balance = val; /* volatile write */
    }
```

Performance of Interlocked/Volatile

- Depends on architecture
  - Measure what you want to know… don't rely on people telling you
- That said, typically, on x86 multiprocessors:
  - Interlocked is somewhat faster than locking
    - Particularly fast if access goes to a cache line that is already in X state.
  - Volatile read/write is MUCH faster than locking
    - Speed of volatile read/write is almost exactly same as speed of normal read/write (gets compiled to same instruction)

Interlocked, Volatile, And Race Detection

- Race detector will not report races between
  - Interlocked access & volatile access
  - volatile access & volatile access
  - Interlocked access & Interlocked access
- Race detector does report data races between
  - Interlocked access & normal access
  - Volatile access & normal access

Part I

CASE STUDY: ANTISOCIAL ROBOTS

Parallel Loop in AntiSocialRobots

```
Parallel.ForEach(_robots, SimulateOneStep);

void SimulateOneStep(Robot r) {
      ...
      foreach (Robot s in _robots)
      {
            ...
      }
      ...
      if (...)
      {
            ....
      }
}
```

read position of all other robots to figure out into which cell this robot wants to move

If the cell it wants to move to is free, move it there.

apply in parallel to each robot

Bug 1: Data Race on Robot.Location

```
class Robot
{
    ...
    public RoomPoint Location;
}
```

```
Parallel.ForEach(_robots, SimulateOneStep);

void SimulateOneStep(Robot r) {
    ...
    foreach (Robot s in _robots)
    {
        ...
        RoomPoint ptS = s.Location;
        ...
    }
    ...
    if (...)
    {
        ...
        r.Location = new RoomPoint(ptR.X, ptR.Y);
    }
}
```

read position of all other robots to figure out into which cell this robot wants to move

If the cell it wants to move to is free, move it there.

Fix: Protect Robot.Location with Lock

- We can use the lock of the *Robot* object to protect the field *Location*

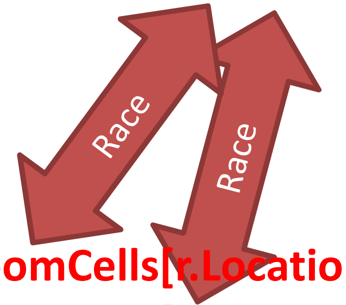
```
class Robot
{
    …
    public RoomPoint Location;
 }
```

```
lock s { …
    RoomPoint ptS = s.Location;
… }

lock r { …
     r.Location = new RoomPoint(ptR.X, ptR.Y);
}
```

No more races on Robot.Location

Bug 2: Data Race on roomCells

```
Parallel.ForEach(_robots, SimulateOneStep);

void SimulateOneStep(Robot r) {
    …
    foreach (Robot s in _robots)
    {
    }
    …
    if (… && null == roomCells[ptR.X, ptR.Y])
    {
        _roomCells[r.Location.X, r.Location.Y] = null;
        _roomCells[ptR.X, ptR.Y] = r;
        …
    }
}
```



```
public class MainWindow
{
    Robot[,] _roomCells;
}
```

If the cell it wants to move to is free, move it there.

Protecting roomCells w/ single lock

Single lock protects the whole array.
Destroys parallelism!

```
public class MainWindow
{
    Robot[,] _roomCells;
}
```

```
lock (this)
{
        if (… && roomCells[ptR.X, ptR.Y] == null)
        {

            _roomCells[r.Location.X, r.Location.Y] = null;
           _roomCells[ptR.X, ptR.Y] = r;
            …
         }
 }
```

Protecting roomCells w/ fine-grained locks

```
Object[,] _cellLocks = new Object[ROOM_SIZE, ROOM_SIZE];
```

Use one lock per cell.

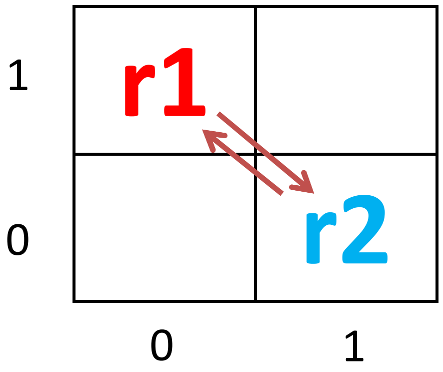
```
lock _celllocks[newLoc.X, newLoc.Y]
 {
          lock _celllocks[oldLoc.X, oldLoc.Y]
          {
              ....
             _roomCells[oldLoc.X, oldLoc.Y] = null;
             _roomCells[newLoc.X, newLoc.Y] = r;
             ...
          }
```

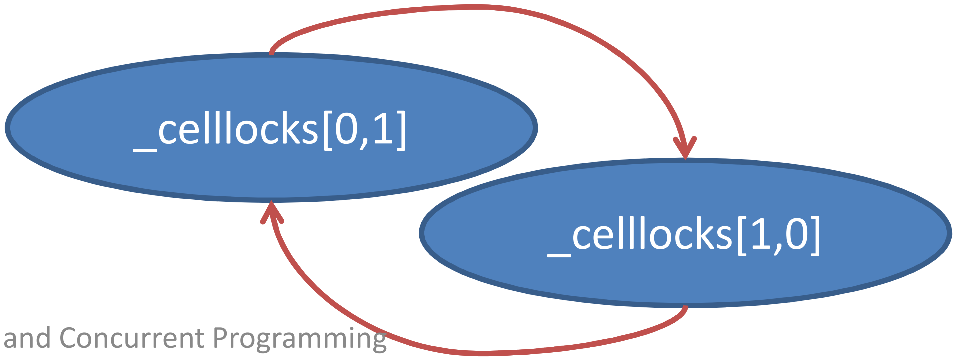
Bug 3: Deadlock.

Cycle in lock acquisition graph

(lock order not consistent)

```
lock _celllocks[newLoc.X, newLoc.Y]
   {
            lock _celllocks[oldLoc.X, oldLoc.Y]
             {                    …
             }
   }
```





Fix: Choose Consistent Lock Order

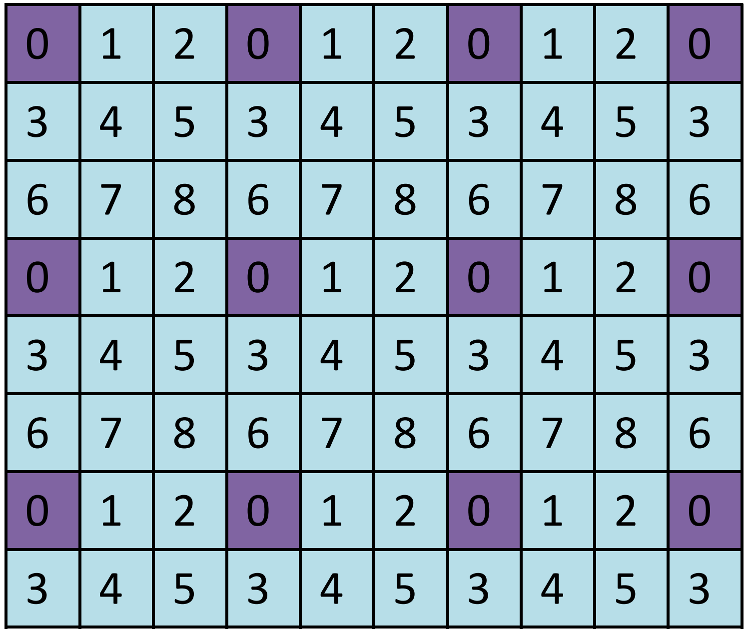
```
// lock level of _cellLocks[X, Y] is
// Y * ROOM_SIZE + X

object firstlock = _cellLocks[newLoc.X, newLoc.Y];
object secondlock = _cellLocks[origLoc.X, origLoc.Y];

// if necessary swap locks to ensure consistent order
if ((newLoc.Y * ROOM_SIZE + newLoc.X) >
                 (origLoc.Y * ROOM_SIZE + origLoc.X))
{
    object tmp = firstlock;
    firstlock = secondlock;
    secondlock = tmp;
}

lock (firstlock)
{
    lock (secondlock)
    {
```

Problem solved… or is it?

- We’ve successfully fixed the data races in antisocial robots using locks
- Was not as easy as it looked at first
  - Final design: use 3 critical sections and sophisticated lock acquisition order scheme
- What have we learned?
  - Designing good locking is a lot of work.
  - Can we solve this problem without locks?

Antisocial Robots Without Locks

- Label all cells with a number between 0 and 8 as follows:

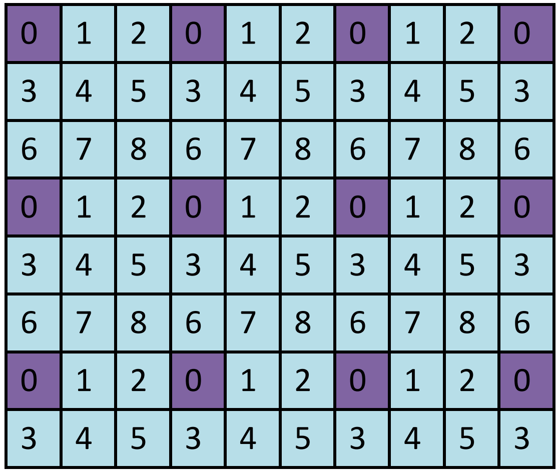


0	1	2	0	1	2	0	1	2	0
3	4	5	3	4	5	3	4	5	3
6	7	8	6	7	8	6	7	8	6
0	1	2	0	1	2	0	1	2	0
3	4	5	3	4	5	3	4	5	3
6	7	8	6	7	8	6	7	8	6
0	1	2	0	1	2	0	1	2	0
3	4	5	3	4	5	3	4	5	3

- Stripe the computation!
- In each turn, perform these 9 steps in sequence*:*
  - Move all robots in cells labeled 0 in parallel.
  - Move all robots in cells labeled 1 in parallel.
  - Move all robots in cells labeled 2 in parallel.
  - …
  - Move all robots in cells labeled 8 in parallel.

- No interference!
  - Within each step, robots are too far apart to interfere
  - Across steps, there is no parallelism

Antisocial Robots Without Locks



0	1	2	0	1	2	0	1	2	0
3	4	5	3	4	5	3	4	5	3
6	7	8	6	7	8	6	7	8	6
0	1	2	0	1	2	0	1	2	0
3	4	5	3	4	5	3	4	5	3
6	7	8	6	7	8	6	7	8	6
0	1	2	0	1	2	0	1	2	0
3	4	5	3	4	5	3	4	5	3

```
for (int stripe = 0; stripe < 9; stripe++)
    Parallel.ForEach(_robots, (Robot r) =>
      {
         if (r.lastmoved < _frameIndex
            && (r.Location.X % 3) == (stripe % 3)
            && (r.Location.Y % 3) == (stripe / 3))
         {
            SimulateOneStep(r);
            r.lastmoved = _frameIndex;
         }
      });
```