CSC352
Week #6 — Spring 2017

Dominique Thiébaut
dthiebaut@smith.edu
Making the Game of Life Parallel
Serial Version

- Study it
- Run it on your laptop
- Use both dish and dish2 as the array of live cells, and see how they evolve

Other option:

2-Thread Version

• As a group, discuss the different tissues associated with parallelizing the Game of Life and running it with two threads.

• List all the issues that must be addressed on the whiteboard

• How will you verify the correctness of the parallel version?

• Play-out (human play) the execution of the 2-thread program: two people or two groups play the roles of the two threads.
Group Work!

Could be Usefull...

- **What is a BlockingQueue?**

  *BlockingQueue* is a queue which is **thread safe** to insert or retrieve elements from it. Also, it provides a mechanism which blocks requests for inserting new elements when the queue is full or requests for removing elements when the queue is empty, with the additional option to stop waiting when a specific timeout passes. This functionality makes *BlockingQueue* a nice way of implementing the Producer-Consumer pattern, as the producing thread can insert elements until the upper limit of *BlockingQueue* while the consuming thread can retrieve elements until the lower limit is reached and of course with the support of the aforementioned blocking functionality.

  [https://examples.javacodegeeks.com/core-java/util/concurrent/java-blockingqueue-example/](https://examples.javacodegeeks.com/core-java/util/concurrent/java-blockingqueue-example/)
Thread safe: Implementation is guaranteed to be free of race conditions when accessed by multiple threads simultaneously.

–Johnny Appleseed
• BlockingQueue

• Need to use an implementation of it:
  “A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element”
  • ArrayBlockingQueue
  • DelayQueue
  • LinkedBlockingQueue
  • PriorityBlockingQueue
  • SynchronousQueue

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class UsingQueues {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Integer> toWorkerQ = new ArrayBlockingQueue<Integer>(2);
        BlockingQueue<Integer> fromWorkerQ = new ArrayBlockingQueue<Integer>(2);

        // create a worker and give it the two queues
        DemoThread t = new DemoThread(fromWorkerQ, toWorkerQ);

        // start thread
        t.start();

        // wait 1/2 second
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // send work to worker
        toWorkerQ.put(100);

        // wait for answer back from worker
        int x = fromWorkerQ.take();

        // display the result
        System.out.println("x = " + x);
    }
}
/**
 * DemoThread
 */

class DemoThread extends Thread {
    BlockingQueue<Integer> sendQ;
    BlockingQueue<Integer> receiveQ;

    DemoThread( BlockingQueue<Integer> sendQ,
                 BlockingQueue<Integer> receiveQ ) {
        this.sendQ = sendQ;
        this.receiveQ = receiveQ;
    }

    public void run(){
        int x=0;

        // block until there's something in the queue
        try {
            x = receiveQ.take();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // do some computation
        x = x*2;

        // send results back
        try {
            sendQ.put( x );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
Implement the 2-Thread Game of Life in Java.

- Play out Serial Version
- Play out Parallel Version
We Stopped Here Last Time
The following slides present an approach for

1) Running experiments
2) Evaluating performance
3) Displaying a meaningful graph
The Next Slides present...

- An approach for
  - *Running* experiments *automatically*
  - *Measuring* and *recording* performance measures
  - *Filtering* and *graphing* the results
The Next Slides present...

- An **approach** for
  - *Running* experiments *automatically*
  - *Measuring* and *recording* performance measures
  - *Filtering* and *graphing* the results

bash scripts
The Next Slides present…

• An **approach** for
  
  • *Running* experiments *automatically*
  
  • *Measuring* and *recording* performance measures
  
  • *Filtering* and *graphing* the results
The Next Slides present...

• An **approach** for

  • *Running* experiments *automatically*

  • *Measuring* and *recording* performance measures

  • *Filtering* and *graphing* the results

Python and R
Defining the Number of Threads at Execution Time
public class UsingQueuesN {

    public static void main(String[] args) throws InterruptedException {
        if (args.length < 1) {
            System.out.println("Syntax: java UsingQueuesN n");
            System.out.println(" where n = # of threads");
            return;
        }

        int N = Integer.parseInt(args[0]);

        BlockingQueue<Integer> toWorkersQ = new ArrayBlockingQueue<Integer>(2*N);
        BlockingQueue<Integer> fromWorkersQ = new ArrayBlockingQueue<Integer>(2*N);

        // create a worker and give it the two queues
        DemoThreadN[] threads = new DemoThreadN[N];

        for (int i=0; i<N; i++) {
            DemoThreadN t = new DemoThreadN(i, fromWorkersQ, toWorkersQ);
            t.start();
            threads[i] = t;
        }

        // wait 1/2 second
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // send same amount of work to each worker
        for (int i=0; i<N; i++)
            toWorkersQ.put(100);

        // wait for answer back from worker
        for (int i=0; i<N; i++) {
            int x = fromWorkersQ.take();

            // display the result
            System.out.println("x = " + x);
        }
    }
}

```java
class DemoThreadN extends Thread {
    private BlockingQueue<Integer> sendQ;
    private BlockingQueue<Integer> receiveQ;
    private int Id;

    DemoThreadN( int Id,
        BlockingQueue<Integer> sendQ,
        BlockingQueue<Integer> receiveQ ) {
        this.Id = Id;
        this.sendQ = sendQ;
        this.receiveQ = receiveQ;
    }

    public void run(){
        int x=0;

        // block until there's something in the queue
        try {
            x = receiveQ.take();
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }

        // do some computation
        x = x*( Id + 1 );

        // send results back
        try {
            sendQ.put( x );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Measuring Performance
Pick the Performance Measure that is Right for Your Application

• **Speedup** = \( T(1) / T(N) \) as a function of \( N \)

• Pick the *best serial algorithm*!

• Define \( N \) (# of cores, # of threads, # of processors)

• Pick the right size problem and keep it constant (size of life grid, for example)

• Make sure data size is large enough, but fits in memory (avoid *disk thrashing*)
How can Amdahl's Law be circumvented:
How can Amdahl's Law be circumvented:
- Pick a very large data set
(side note)

- How can Amdahl's Law be circumvented:
  - Pick a very large data set
(side note)
(side note)
(side note)

No more thrashing
Measuring Performance

• Measure the *average* execution time of several runs for each case, or the *average* quantity of interest per unit of time.

• Use shell *scripts* and programming tools (See next slides)
Using Shell Scripts

http://www.science.smith.edu/dftwiki/index.php/CSC352:_Using_Bash,_an_example

# from 352b-xx account on aurora...
getcode PrintN.java
getcode processTimingData.py
getcode runPrintN.sh
The target program

```java
class PrintN {
    public static void main( String[] args ) {
        int N = Integer.parseInt( args[0] );
        System.out.println( "I got " + N );
    }
}
```

Create a program that gets its (fake) degree of parallelism from the command line.

```java
class PrintN {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        System.out.println("I got " + N);
    }
}
```

# at the Linux prompt:
```
bash
javac PrintN.java
for i in 1 2 3 4 5 6 7 8 9 10; do
    java PrintN $i
done
```

Run the program once in a loop from the command line…
class PrintN {
    public static void main( String[] args ) {
        int N = Integer.parseInt( args[0] );
        System.out.println( "I got " + N );
    }
}

# at the Linux prompt:
bash
javac PrintN.java
for i in 1 2 3 4 5 6 7 8 9 10 ; do
    java PrintN $i
done

#!/bin/bash
# runPrintN.sh
#
javac PrintN.java
for i in 1 2 3 4 5 6 7 8 9 10 ; do
    java PrintN $i
done

Embed the commands just typed at the prompt into a Bash shell script
class PrintN {
    public static void main( String[] args ) {
        int N = Integer.parseInt( args[0] );
        System.out.println( "I got " + N );
    }
}

# at the Linux prompt:
bash
javac PrintN.java
for i in 1 2 3 4 5 6 7 8 9 10 ; do
    java PrintN $i
done

#!/bin/bash
# runPrintN.sh
#
javac PrintN.java
for i in 1 2 3 4 5 6 7 8 9 10 ; do
    for j in 1 2 3 ; do
        /usr/bin/time java PrintN $i
    done
done

Run each program a few times for the same level of parallelism, and measure execution time for each run…
I got 1
real 0m0.080s
user 0m0.067s
sys 0m0.011s
I got 1
real 0m0.082s
user 0m0.067s
sys 0m0.011s
...
I got 10
real 0m0.079s
user 0m0.066s
sys 0m0.011s

Note, the time command outputs its timing information to **stderr**, while the other command and java program outputs to **stdout**...
Redirect stderr to stdout, and capture lines with "got" or "real" to a text file.
./runPrintN.sh 2>&1 | grep "got\|real" > timing.data

cat timing.data

I got 1
real 0m0.085s
I got 1
real 0m0.086s
I got 1
real 0m0.085s
I got 2
real 0m0.093s
I got 2
real 0m0.096s
...

real 0m0.079s
I got 10
real 0m0.079s
I got 10
real 0m0.079s
I got 10
real 0m0.079s

Contents of timing.data (with middle lines removed for conciseness)
Write a Python program to filter timing.data and print a simple output of x and y values.
```python
# processTimingData.py
# D. Thiebaut

from __future__ import print_function

file = open( "timing.data", "r" )
lines = file.readlines()
file.close()

# create array of time averages
times = [0]*11   # 0-10, hence 11

# parse lines of text
for line in text.split( "\n" ):
    if len(line) < 2:
        continue
    if line.find( "got" ) != -1:
        n = int( line.split()[-1] )
    else:
        time = line.replace( 'm', ' ' ).replace( 's', '' ).split()[-1]
        time = float( time )
        times[n] += time

# compute averages and print them
for i in range( len( times ) )�
    if times[i] != 0:
        print( i, times[i]/3.0 )
```

Output.
Ready for plotting!

```
python processTimingData.py
1 0.08533333333333333
2 0.09233333333333333
3 0.08433333333333333
4 0.08166666666666667
5 0.079
6 0.08666666666666667
7 0.08433333333333333
8 0.07966666666666667
9 0.08066666666666667
10 0.079
```
Plotting the Resulting Timing Information With R
This R-Markdown illustrates how to quickly display a graph of the average execution times of an application running on 1 to 20 threads.

```{r}
noThreads <- c( 1, 2, 4, 8, 16, 20 )
execTimes <- c( 10, 8.5, 7.0, 6.0, 5.5, 7.3 )

jpeg( '/Users/thiebaut/Desktop/executionTimes.jpg' )
plot( noThreads, execTimes, type="b", col="blue",
     xlab="Number of Threads", ylab="Avg. Execution Time (s)"
)
dev.off()

plot( noThreads, execTimes, type="b", col="blue",
     xlab="Number of Threads", ylab="Avg. Execution Time (s)"
)
```

![Graph of execution times](./executionTimes.jpg)
Make sure that the graph clearly shows **POINTS** and that the lines are understood to show the trend.
We Stopped Here Last Time
Some Comments On Papers
What is Scalability?

- Ideal is to get $N$ times more work done on $N$ processors

- Strong scaling: compute a fixed-size problem $N$ times faster
  - Speedup $S = T_1 / T_N$; linear speedup occurs when $S = N$
  - Can’t achieve it due to Amdahl’s Law (no speedup for serial parts)

- Weak scaling: compute a problem $N$ times bigger in the same amount of time
  - Speedup depends on the amount of serial work remaining constant or increasing slowly as the size of the problem grows
  - Assumes amount of communication among processors also remains constant or grows slowly
Strong vs Weak Scaling

**Strong Scaling**

- Single Proc
- Single RAM

**Weak Scaling**

- Multiple Pros
- Multiple RAMs
Strong vs Weak Scaling
https://www.top500.org/lists/2016/11/

**Top500 List**

#1 National Supercomputing Center in Wuxi China 10,649,600 cores

<table>
<thead>
<tr>
<th>Name</th>
<th>The Number</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>quadrillion</td>
<td>$10^{15}$</td>
<td>peta</td>
</tr>
<tr>
<td>trillion</td>
<td>$1,000,000,000,000,000$</td>
<td>tera</td>
</tr>
<tr>
<td>billion</td>
<td>$1,000,000,000$</td>
<td>giga</td>
</tr>
<tr>
<td>million</td>
<td>$1,000,000$</td>
<td>mega</td>
</tr>
<tr>
<td>thousand</td>
<td>$1,000$</td>
<td>kilo</td>
</tr>
</tbody>
</table>
https://www.top500.org/lists/2016/11/

### Top500 List

<table>
<thead>
<tr>
<th>CPU</th>
<th>MHz</th>
<th>MFlops</th>
<th>MFlops (no opt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i5 2467M</td>
<td>2000</td>
<td>1092.56</td>
<td>121.25</td>
</tr>
<tr>
<td>Celeron C2 M</td>
<td>2400</td>
<td>1315.42</td>
<td>195.13</td>
</tr>
<tr>
<td>Core 2 Duo 1 CP</td>
<td>3000</td>
<td>1412.83</td>
<td>244.43</td>
</tr>
<tr>
<td>Phenom II</td>
<td>4000</td>
<td>2004.31</td>
<td>381.97</td>
</tr>
<tr>
<td>Core i7 930</td>
<td>4400</td>
<td>2529.73</td>
<td>746.01</td>
</tr>
<tr>
<td>Core i7 860</td>
<td>4800</td>
<td>2671.15</td>
<td>892.04</td>
</tr>
<tr>
<td>Core i7 3930K</td>
<td>5200</td>
<td>2864.05</td>
<td>895.54</td>
</tr>
<tr>
<td>Core i7 3930K</td>
<td>5600</td>
<td>3112.94</td>
<td>926.92</td>
</tr>
<tr>
<td>Core i7 3930K</td>
<td>6000</td>
<td>3361.82</td>
<td>958.43</td>
</tr>
</tbody>
</table>

- Rated as 2800 MHz but running at up to 3460 MHz using Turbo Boost
- Rated as 3200 MHz but running at up to 3800 MHz OC OverClocked ~4720 MHz

### Performance Metrics

- 10^7 cores, 93 10^15 Flops
- 4 cores, 3 10^9 Flops
- 2.5 10^6 more cores, 30 10^6 more computing power
FIGURE 1. Supercomputer performance over time as tracked by the TOP500 list. The red and orange lines show performance of the first (number 1) and last (number 500) systems, respectively, and the blue line shows average performance of all systems. Dashed lines are fitted exponential growth curves before and after 2008 for the orange line and before and after 2013 for the blue line.

From: https://www.nextplatform.com/2015/11/25/2241/
Advanced Concepts on Threads
The Basics

- **Threads Operation**
  - `run()`/`start()`
  - `yield()`
  - `sleep()`
  - `join()`
  - `wait()`, `notify()`, and also `notifyAll()`
States of a Thread

New thread $\rightarrow$ running $\rightarrow$ Terminated

- waiting on an object
- sleeping
- blocking on I/O
- blocked on a lock
How does one get the state?

- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIME_WAITING
- TERMINATED

getState()
Threads good not only for speedup
Threads good not only for speedup
Important Concepts

- CPU Bound Processes/Threads
- I/O Bound Processes/Threads
Time Scale

- Why I/O recognizing I/O-bound process is important
- CPU cycle: 1 ns
- RAM cycle: 100-500 ns
- Disk access = seek + latency
  - seek = 1 ms
  - latency = 1/2 rotation, at 10,000 RPM
- Question: How long does the processor wait for data from disk?
Problems Associated with Sharing Data

- The Dining-Philosophers Problem
  http://vip.cs.utsa.edu/nsf/pubs/starving/starving.html
Problems Associated with Sharing Data

- The Dining-Philosophers Problem
  [Link](http://vip.cs.utsa.edu/nsf/pubs/starving/starving.html)

Diagram:

- Thinking
- Eating
- Hungry
Starvation
Starvation

Deadlock
Thread Scheduling

• What is the policy?

• Java doc says: *Implemented in the JVM, preemptive, based on priority.* (No mention of time-slices.)

• 1 = low priority, 5 = main, 10 = high priority

• `getPriority()` & `setPriority()`

• However, most OS implement *time-slices* (quanta), roughly 1ms, *preemptive,* and *round-robin* $\implies$ JVMs do the same
Rule #1 for Preventing Deadlocks

• **Grab all** the shared data-structures that you need first

• If you can’t, **release** them all

• **Wait a random amount** of time and try again
Rule #2 for Preventing Starvation

- In Dining Philosophers situation, **do not allow** a philosopher to **eat twice** before one has had a chance to eat once ("polite" algorithm of [http://vip.cs.utsa.edu/nsf/pubs/starving/starving.html](http://vip.cs.utsa.edu/nsf/pubs/starving/starving.html))
Crash Course on C

(Switch to Separate Set of Slides)