Parallel Programming

Recitation Session 3

Thomas Weibel <weibelt@ethz.ch>

Laboratory for Software Technology,
Swiss Federal Institute of Technology Zürich

March 18, 2010

Executive Summary

- Java Questions?
  - Please send me an email if you want me to discuss something specific in the recitation session
- Synchronization
- Loop Examples
  - Pre Increment
  - Post Increment
- MergeSort: How to parallelize?
- Performance Measurement
  - Harsh realities of parallelization
  - Amdahl's law

Outline

Solution: Questions Part 2

Question

Why is it not sufficient to add the `synchronized` keyword to the `read()` and `write()` methods to guarantee the specified behavior of the producer/consumer problem?

Answer

Synchronization ensures that the producer and the consumer can not access the buffer at the same time.
But it does not prevent the consumer to read a value more than one time or the producer to overwrite a value that was not read.

Solution: Questions Part 2

Question

Would it be safe to use a boolean variable as a "guard" within the `read()` and `write()` methods instead of using the `synchronized` keyword?

Answer

No, reading and writing a value is not atomic!
Why is `i++` is not atomic?

Solution: Questions Part 3

Question

Would it suffice to use a simple `synchronized(this)` within the `run()` method of each the producer and the consumer to guard the updating of the buffer?

Answer

No, since producer and consumer are different objects with different locks $\rightarrow$ no mutual exclusion guaranteed

Solution: Questions Part 3

Question

What is the object that should be used as the shared monitor and therefore the object upon which the threads are `synchronized()`?

Answer

The shared instance of `UnsafeBuffer`.

Solution: Questions Part 3

Question

What could you have used instead?

Answer

A dedicated shared lock object.

Solution: Questions Part 3

Question

What are the potential advantages/disadvantages of synchronizing the producer/consumer over synchronizing the buffer?

Advantages

- Can use arbitrary (also unsafe!) buffers
- Can do things in the Producer/Consumer that need to be done before the other thread can use the buffer, for example print something to the console.

Disadvantages

- More work to do :-)
- More error-prone

Outline

Loop 1

```
int j, m;

System.out.println("Loop 1");
j = 0;
while (j < 10) {
  j = j + 1;
  System.out.print(" " + j);
}
```

Loop 2

```
System.out.println("Loop 2");
j = 0;
while (j++ < 10) {
  System.out.print(" " + j);
}
```

Loop 3

```
System.out.println("Loop 3");
j = 0;
while (++j < 10) {
  System.out.print(" " + j);
}
```

Loop 4

```
System.out.println("Loop 4");
j = 0;
m = 0;
while (j++ < 10) {
  if (++m == j) {
    System.out.print("j=" + j +
                     " m=" + m + "\n");
  }
}
```

Loop 5

```
System.out.println("Loop 5";)
j = 0;
m = 0;
while (j++ < 10) {
  if (m++ == j) {
    System.out.print("j=" + j +
                     " m=" + m + "\n");
  }
}
```

Loop 6

```
System.out.println("Loop 6");
j = 0;
m = 0;
while (j < 10) {
  if (j == (++m)) {
    System.out.print("j=" + j +
                     " m=" + m + "\n");
  }
  j++;
}
```

Loop 7

```
System.out.println("Loop 7");
j = 0;
m = 0;
while (j < 10) {
  if (m++ == j) {
    System.out.print("j=" + j +
                     " m=" + m + "\n");
  }
  j++;
}
```

Outline

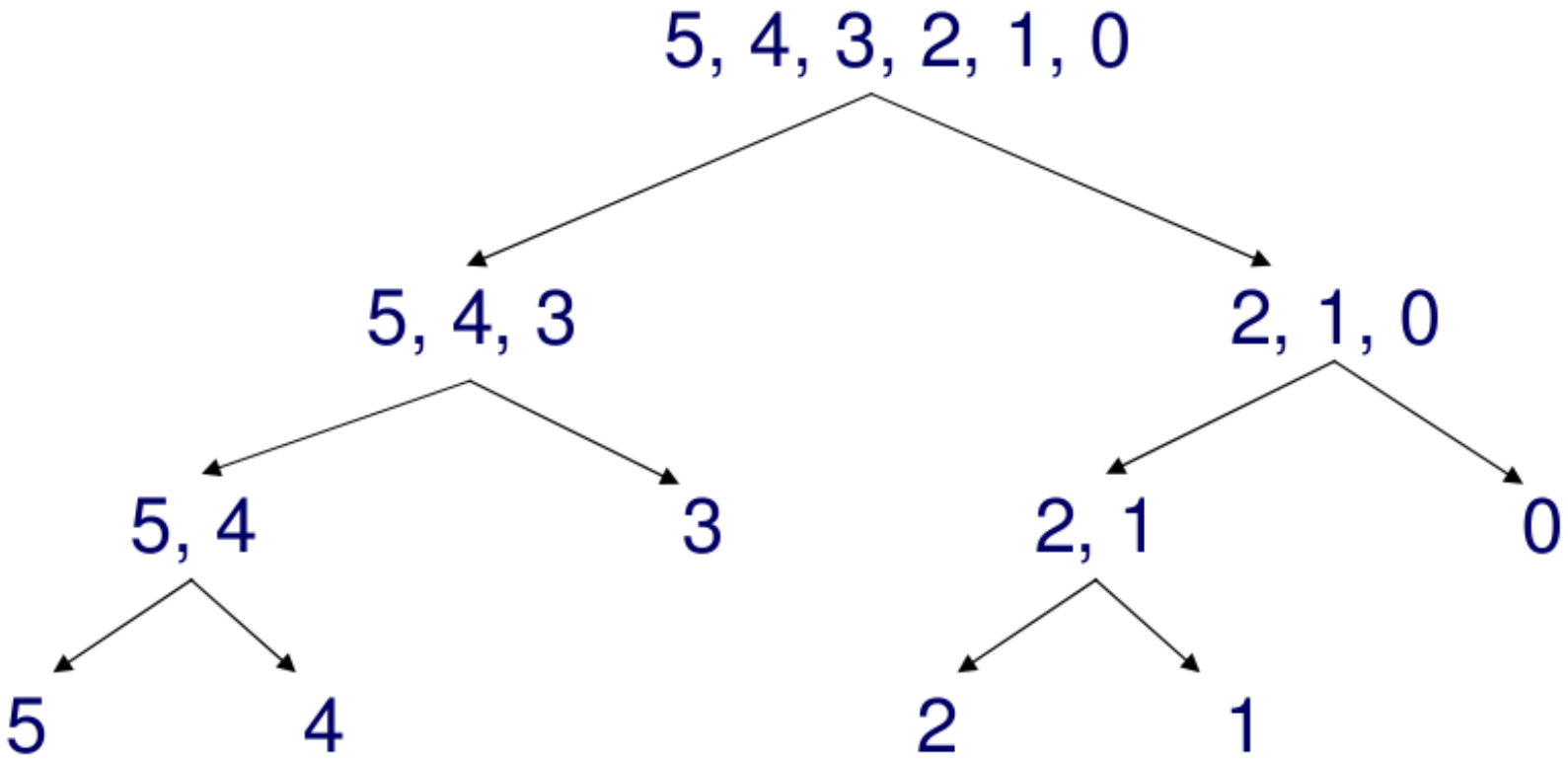
MergeSort

- Problem: Sort a given list l of n numbers
- Example:
  - Input: 9 8 7 6 5 4 3 2 1 0
  - Output: 0 1 2 3 4 5 6 7 8 9

Algorithm

- Divide l into two sub-lists of size $n/2$
- Sort each sub-list recursively by re-applying MergeSort
  - End of recursion: Size of the sub-list becomes 1
  - If size of a sub-list $> 1 \Rightarrow$ other sorting needed
- Merge the two sub-lists back into one sorted list

Example: Divide into sub-lists

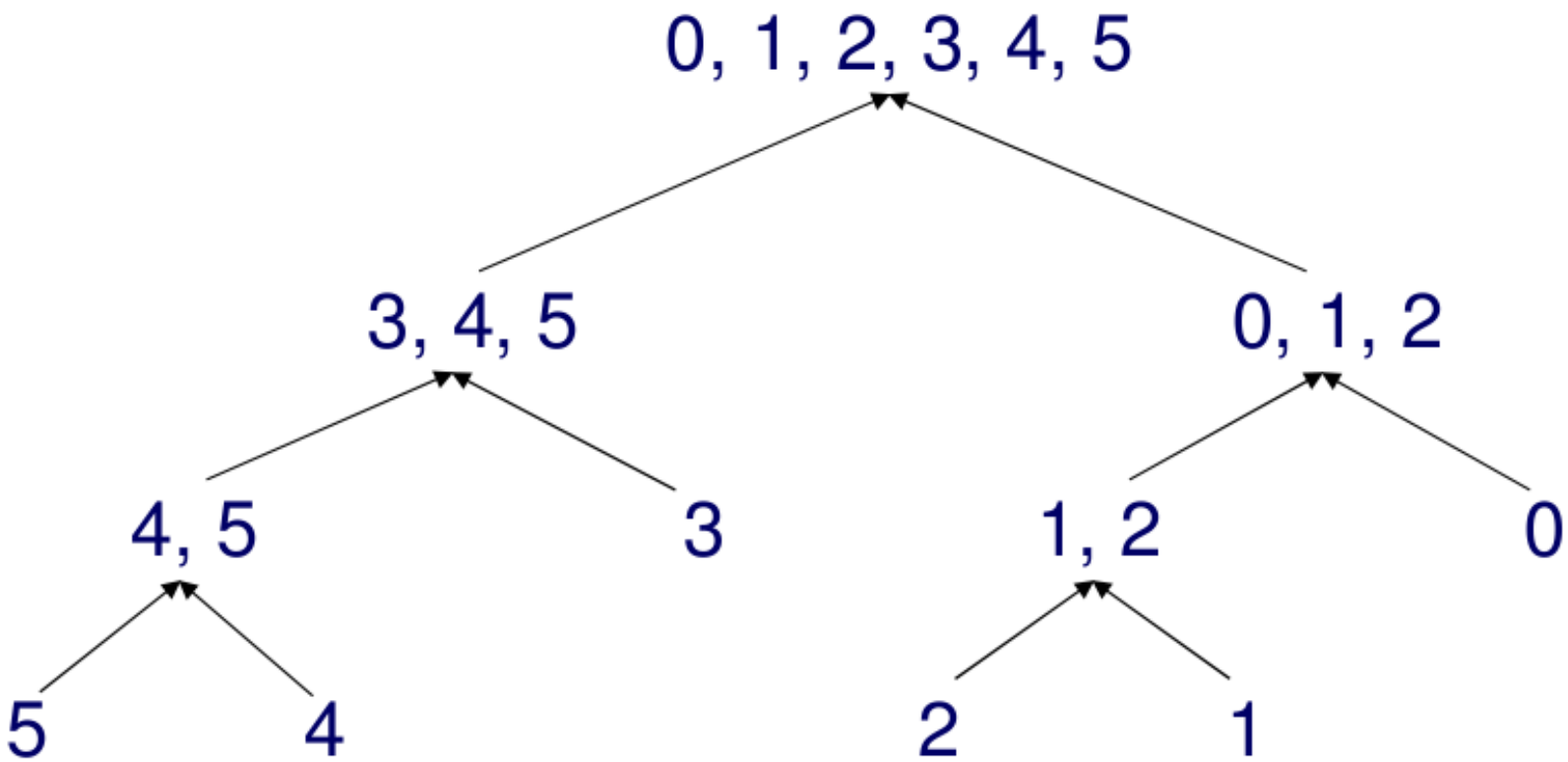


Merging

- Combine two sorted lists into sorted list
- Example:
  - List 1: 0, 5
  - List 2: 3, 4, 45
  - Output: 0, 3, 4, 5, 45

List 1	List 2	Merged list
0, 5	3, 4, 45	$0 < 3 \rightarrow$ insert 0 in merged list: 0
0, 5	3, 4, 45	$3 < 5 \rightarrow$ insert 3 in merged list: 0, 3
0, 5	3, 4, 45	$4 < 5 \rightarrow$ insert 4 in merged list: 0, 3, 4
0, 5	3, 4, 45	$5 < 45 \rightarrow$ insert 5 in merged list: 0, 3, 4, 5
0, 5	3, 4, 45	Finally, insert 45 in merged list: 0, 3, 4, 5, 45

Example: Merging sorted sub-lists



Code skeletons

Eclipse

Outline

Which operations can be done in parallel?

Sorting

Each sub-list can be sorted by a separate thread

Merging

Two ordered sub-lists can be merged by a thread

Parallelization issues

Synchronization issues

Limitations in parallelization?

↪ Merge can only happen if two sub-lists are sorted

Performance issues

- Number of threads?
- Size of array to sort?

Load balancing

What do we do if the threads cannot be evenly distributed?

```
size of array % numThreads != 0
```

Simple (proposed) solution

Assign remaining elements to one thread

Balanced (more complicated) solution

Distribute remaining elements to more threads

Outline

Performance Measurement

Array size	Number of threads								
	1	2	4	8	16	32	64	...	1024
$100'000$									
$500'000$									
...									
$10'000'000$									

How to measure time?

```
public class StopWatch {
  private long startTime = 0;
  private long stopTime = 0;
  private boolean running = false;

  public void start() {
    this.startTime = System.currentTimeMillis();
    this.running = true;
  }

  public void stop() {
    this.stopTime = System.currentTimeMillis();
    this.running = false;
  }

  public long getElapsedTime() {
    if (running) {
      return System.currentTimeMillis() - startTime;
    } else {
      return stopTime - startTime;
    }
  }
}
```

Measure time

- `System.currentTimeMillis()` might not be exact
  - Granularity might be higher than a millisecond
  - Might be slightly inaccurate
- `System.nanoTime()`: Nanosecond precision, but not nanosecond accuracy
- For our measurements `System.currentMillis()` is good enough

K-Best Measurement Scheme

- Measure the K best execution times of the program
- K measurements should be close to the best performance value
- Three parameters required:
  - Number of measurement ($K = 3$)
  - How close the measurements should be ($\epsilon = 5\%$)
  - The maximum number of measurements/time before we give up
- In the table: arithmetic average of measured values

Questions to be answered

- Is the parallel version faster?
- How many threads give the optimum performance?
- What is the influence of the CPU model/CPU frequency?

Outline

The Harsh Realities of Parallelization

Ideally

Upgrading from uni-processor to n-way multiprocessor should provide an n-fold increase in computational power

Real world

Most computations cannot be efficiently parallelized
$\hookrightarrow$ Sequential code, synchronization, communication

Speedup

$$\text{Speedup} = \frac{\text{time}(\text{single processor})}{\text{time}(n \text{ concurrent processors})}$$

Amdahl's Law

Speedup of any complex job is limited by how much of the job must be executed sequentially

Amdahl's Law

$$S = \frac{1}{\underbrace{1 - p}_{\text{serial}} + \underbrace{\frac{p}{n}}_{\text{parallel}}}$$

Parameters

- 1: normalized sequential execution time
- p: fraction that can be executed in parallel
- n: number of processors

Example

Question

5 painters, and 5 rooms, 4 small rooms have the same size, one big room has twice the size of a small room.
What is the speedup?

Solution

$n = 5$, $p = \frac{5}{6}$, $1 - p = \frac{1}{6}$

$$S = \frac{1}{1-p+\frac{p}{n}} = \frac{1}{\frac{1}{6}+\frac{1}{6}} = 3$$

Even worse?

Question

10 painters, and 10 rooms, 9 small rooms have the same size, 1 big room has twice the size of a small room.
What is the speedup?

Solution

$n = 10$, $p = \frac{10}{11}$, $1 - p = \frac{1}{11}$

$$S = \frac{1}{1-p+\frac{p}{n}} = \frac{1}{\frac{1}{11}+\frac{1}{11}} = 5.5$$

$\hookrightarrow$ Even if we manage to parallelize 90% of the application, but not the remaining 10% we end up with a five-fold speedup

Summary

- Please send me your questions and remarks
- Synchronization
- Pre and Post Increment
- MergeSort and parallel MergeSort
- How to measure time in Java
- Amdahl's law