12. Introduction to Parallel Programming

Mircea Lungu

Based on materials by Oscar Nierstrasz, Fabrizio Perin, Blaise Barney
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency
> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
  — Communication
  — Amdahl’s Law
> Technologies for parallelism
> Parallel examples
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency
> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
  — Communication
  — Amdhal’s Law
> Technologies for parallelism
> Parallel examples
Memory Wall

- CPU Frequency
- DRAM Speeds

CPU -- 2x Every 2 Years
DRAM -- 2x Every 6 Years

Gap
Power Wall: 4GHz

Power = Voltage$^2$ * Frequency
Machine Classification

**SISD**
Old Mac

**SIMD**
Cell (GPU)

**MIMD**
(Super)computers

Legend
- Single
- Multiple
- Instructions
- Data
Why use parallelism?

> Solve problems faster
> Solve larger problems
> Model real-life concurrency
> Address limits to serial computing
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency
> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
  — Communication
  — Amdhal’s Law
> Technologies for parallelism
> Parallel examples
Concurrency

*dealing* with many things at once inherent to the problem

examples: car, bank
Parallelism

*doing* many things at once

goal is to speed up

examples: graphics, simulation
Distributed System-ness

distributing a system over the network, over different hardware
examples: seti@home, facebook
Concurrenty > Parallelism (Conceptually)

- T1 may be executed and finished before T2,
- T2 may be executed and finished before T1,
- T1 and T2 may be executed simultaneously at the same instance of time (parallelism),
- T1 and T2 may be executed alternatively

as a program property, concurrency is more general than parallelism.
Good concurrent design can improve the throughput also in parallelism

*Rob Pike, Concurrency is not Parallelism*
There are many concurrent designs for the Gophers
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency

> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
  — Communication
  — Amdahl’s Law

> Technologies for parallelism
> Parallel examples
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency

> Designing Parallel Programs
  — **Understanding the problem**
  — Partitioning
  — Communication
  — Amdahl’s Law

> Technologies for parallelism

> Parallel examples
Understand the Problem

Some problems are not parallelizable

Some problems are easily parallelizable

Some problems require careful design.

0. Ensure parallelizability
1. Identify hotspots / partition
2. Identify bottlenecks / communication
Design the solution
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency
> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
    — Communication
    — Amdahl’s Law
> Technologies for parallelism
> Parallel examples
Partitioning
Domain Decomposition

Data associated with a problem is decomposed. The parallel tasks work on portions of data.
Domain Decomposition - Examples

1D
- BLOCK
- CYCLIC

2D
- BLOCK, *
- *, BLOCK
- BLOCK, BLOCK
- CYCLIC, *
- *, CYCLIC
- CYCLIC, CYCLIC
Applying a function on a very large matrix

```plaintext
find out if I am MASTER or WORKER
if I am MASTER
     initialize the array
     send each WORKER info on part of array it owns
     send each WORKER its portion of initial array
     receive from each WORKER results
else if I am WORKER
     receive from MASTER info on part of array I own
     receive from MASTER my portion of initial array
     # calculate my portion of array
     do j = my first column, my last column
        do i = 1, n
            a(i, j) = fcn(i, j)
        end do
     end do
     send MASTER results
endif
```
Emergent Sorting on a Spatial Computer

initial positions  10 steps  20 steps

darker... blue... lighter

darker... green... lighter
Functional Decomposition

problem is decomposed based on the work that must be done.

this approach lends itself to problems that can be split in different tasks.
Each task calculates the population for a given group, but each group’s growth depends on the neighbours. Each process calculates its state and then exchanges info with the neighbours.
Audio signal passed through four distinct filters. The filters are working in parallel.
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency
> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
  — Communication
  — Amdahl’s Law
> Technologies for parallelism
> Parallel examples
Communication

When you don’t need it: embarrassingly parallel programs (e.g. image processing)

In all other cases consider: latency, bandwidth, synchronicity, overhead (e.g. 3d heat diffusion)
Parallel Communication overhead can be large

Example of Parallel Communications Overhead and Complexity: actual call graph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.
Computation / Communication Ratio

**Fine-grained Parallelism**
- small C/C ratio
- facilitates load balancing

**Coarse-grained Parallelism**
- high C/C ratio
- more opportunity for performance
Data Dependencies

> **Data dependence** - use of the same storage location by different tasks
  — Loop carried data dependence
  — Loop independent dependence

> Main inhibitors of parallelism

> Communicate data at sync points II synchronize R/W operations
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency
> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
  — Communication
  — Amdahl’s Law
> Technologies for parallelism
> Parallel examples
The speedup of your trip is limited by the portion on the journey for which I let you use fast transportation.
Amdahl’s Law

The **maximum speedup** of a program using multiple processors in parallel computing is **limited by** the time needed for the **sequential fraction** of the program.

\[
\text{Speedup} = \frac{1}{1 - P} = \frac{1}{P/N + S}
\]

- **P** = Parallel fraction
- **N** = Number of processors
- **S** = Sequential section

**Example:**

If **P = 0.8** and **N = 4**, then the speedup is

\[
\text{Speedup} = \frac{1}{1 - 0.8} = \frac{1}{0.2} = 5
\]

This means using 4 processors would result in a 5x speedup, indicating that the sequential section takes up 20% of the total execution time.
Amdahl’s Law (example)
Amdahl’s Law (example)

\[
\text{Speedup} = \frac{\text{old running time}}{\text{new running time}}
\]

\[
\text{Speedup} = \frac{140}{65} = 2.15
\]

(parallel version is 2.15 times faster)
Amdahl’s Law (example)

\[
\text{Speedup} = \frac{\text{old running time}}{\text{new running time}} = 2.15
\]
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency
> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
  — Communication
  — Amdahl’s Law
> Technologies for parallelism
> Parallel examples
Parallelism in Java

> In java.util.concurrent
  — Executors / Thread pools
  — Executor Services / The Fork/Join framework
Executor

> Interface used to define custom thread-like systems.

> Strategies to execute tasks and manage them.

> Tasks may execute
  > in a newly created thread, or
  > in an existing task-execution thread, or
  > in the thread calling execute().

> sequentially or concurrently.

```java
public interface Executor {
    Void execute(Runnable command);
}
```

(new Thread(r).start();
is replaced with
```
    e.execute(r)
```
Executor Implementation: **Thread Pools**

- Manage a set of worker threads

- **Threads have a simple life cycle**
  - Request the next task from the queue of tasks
  - Execute
  - Wait for another task

- **Advantages**
  - Reduce the costs of thread creation and teardown
  - Increases responsiveness
  - By tuning the pool you always have the correct number of threads
Thread Pools

- **newFixedThreadPool**: Creates threads as tasks are submitted, up to the maximum pool size, and then attempts to keep the pool size constant.

- **newCachedThreadPool**: Can add new threads when demand increases, no bounds on the size of the pool.
Thread Pools

> **newSingleThreadExecutor**: Single worker thread to process tasks, Guarantees order of execution based on the queue policy (FIFO, LIFO, priority order).

> **newScheduledThreadPool**: Fixed-size, supports delayed and periodic task execution.
class ThreadPerTaskWebServer{
    public static void main(String[] args) throws IOException{
        ServerSocket socket = new ServerSocket(80);
        while(true){
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run(){
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}
Web server with *Executor*

class TaskExecutionWebServer{
    private static final int NTHREADS = 100;
    private static final Executor exec
        = Executor.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException{
        ServerSocket socket = new ServerSocket(80);
        while(true){
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run(){
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
public interface ExecutorService extends Executor {
    <T> Future <T> void submit (Callable <T> task)
}

> Adds submit to the Executor
> Accepts Callable objects
> Produces a Future result
> Manages termination
ExecutorService implementation: **Fork/Join**

> Allows taking advantage of multiple processors

> RecursiveAction & RecursiveTask implement Future (much more lightweight than a thread)

> **Work-stealing** algorithm
  — automatic and efficient load-balancing
  — recommended for divide and conquer, tree traversals

*Cilk: An Efficient Multithreaded Runtime System*
Blumofe, R. D et al.
Roadmap

> Parallelism
  — Overview
  — ~ vs. Concurrency
> Designing Parallel Programs
  — Understanding the problem
  — Partitioning
  — Communication
  — Amdahl’s Law
> Technologies for parallelism
> Parallel examples
public class SelectMaxProblem {

    public int solveSequentially(){
        int max = Integer.MIN_VALUE;
        for (int i = start; i<end; i++){
            int n = numbers[i];
            if(n > max)
                max = n;
        }
        return max;
    }

    ...
}
import java.util.concurrent.*;

public class MaxWithFJ extends RecursiveAction {
    ...
    @Override
    protected void compute() {
        if (problem.getSize() < threshold) {
            result = problem.solveSequentially();
            println(Thread.currentThread() + " solving seq on:" + problem.getSize());
        } else {
            int midpoint = problem.getSize() / 2;

            MaxWithFJ left =
                new MaxWithFJ(problem.subproblem(0, midpoint), threshold);
            MaxWithFJ right =
                new MaxWithFJ(problem.subproblem(midpoint + 1, problem.getSize()), threshold);

            invokeAll(left, right);

            println("solving in parallel on: " + ForkJoinTask.getPool());

            result = Math.max(left.getResult(), right.getResult());
        }
    }
    ...
}
Finding the maximum in parallel

```java
public class MaxWithFJTests {

    private final SelectMaxProblem problem;
    private final int threshold;
    private final int nThreads;
    private final int[] number = new int[500000];

    public MaxWithFJTests(){
    ...
    }

    @Test
    public void parallelTest(){
        MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
        ForkJoinPool fjPool = new ForkJoinPool(nThreads);

        fjPool.invoke(mfj);
        int result = mfj.getResult();

        ...
        assertEquals(result, max);
    }
}
```
> **Divide et Impera algorithm:**
> 
> > **Divide**: split your problem into sub-problems that are smaller parts of the original problem
> > **Impera**: solve the sub-problems recursively. (If the sub-problem is small enough than it is solve in a straightforward manner).
> > **Combine**: the solutions to the sub-problems into the solution for the original problem.
> *Divide et Impera* algorithm:
> > **Divide** the $n$-element sequence to be sorted into two sub-sequences of $n/2$ elements each
> > **Impera:** Sort the sub-sequence recursively using merge sort
> > **Combine:** Merge the two sorted sub-sequences to produce the sorted answer
public class MergeSort extends RecursiveAction {

private void merge(MergeSort left, MergeSort right) {
    int i=0,
    leftPos=0,
    rightPos=0,
    leftSize = left.size(),
    rightSize = right.size();

    while (leftPos < leftSize && rightPos < rightSize)
        result[i++] = (left.result[leftPos] <= right.result[rightPos])
            ? left.result[leftPos++]
            : right.result[rightPos++];

    while (leftPos < leftSize)
        result[i++] = left.result[leftPos++];

    while (rightPos < rightSize)
        result[i++] = right.result[rightPos++];
}

...
public class MergeSort extends RecursiveAction {

    public int size() {return endPos-startPos;}

    protected void compute() {
        if (size() < SEQUENTIAL_THRESHOLD) {
            System.arraycopy(numbers, startPos, result, 0, size());
            Arrays.sort(result, 0, size());
        } else {
            int midpoint = size() / 2;
            MergeSort left =
                new MergeSort(numbers, startPos, startPos+midpoint);
            MergeSort right =
                new MergeSort(numbers, startPos+midpoint, endPos);
            invokeAll(left, right);
            merge(left, right);
        }
    }

    public int[] getResult() {return result;}
}
package mergeSortParallel;
import java.util.Arrays;
import java.util.concurrent.*;

public class MergeSort extends RecursiveAction {
    private static final int SEQUENTIAL_THRESHOLD = 50000;
    ...

    private void merge(MergeSort left, MergeSort right) {
        int i=0, leftPos=0, rightPos=0, leftSize = left.size(), rightSize = right.size();
        while (leftPos < leftSize && rightPos < rightSize)
            result[i++] = (left.result[leftPos] <= right.result[rightPos])
                ? left.result[leftPos++]
                : right.result[rightPos++];
        while (leftPos < leftSize)
            result[i++] = left.result[leftPos++];
        while (rightPos < rightSize)
            result[i++] = right.result[rightPos++];
    }

    public int size() {return endPos-startPos;}

    protected void compute() {
        if (size() < SEQUENTIAL_THRESHOLD) {
            System.arraycopy(numbers, startPos, result, 0, size());
            Arrays.sort(result, 0, size());
        }
        else {
            int midpoint = size() / 2;
            MergeSort left = new MergeSort(numbers, startPos, startPos+midpoint);
            ...
        }
    }
}
What you should know!

> What is the difference between Concurrent Computing and Parallel Computing?

> What are the main two types of partitioning that you use for concurrent programming? Give examples of problems adequate for the two partitioning types.

> When do you use the Fork/Join framework and when do you use Thread Pools?

> What limits the speedup obtained through parallelism?

> Which are the main functionalities of the java.util.concurrent package of Java?
Sources

> Section 4.4 of *Concurrent Programming in Java* (Doug Lea, Prentice Hall PTR, November 1999)
  — Covers parallel decomposition in greater detail.

> Section 6-7-8 of *Java concurrency in practice* (Brian Goetz, et al., Addison Wesley Professional May 09, 2006)

> Doug Lea's concurrency-interest website:
  — read the paper on the design of the fork-join framework.
  — http://gee.cs.oswego.edu/dl/concurrency-interest/index.html

> Introduction to Parallel Computing, Blaise Barney
  — https://computing.llnl.gov/tutorials/parallel_comp/
Attribution-ShareAlike 3.0

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/3.0/