# FCDS – Lab

Summer Semester 2014

# Outline

- Introduction
- Goals
- Organization
- Concurrency Concepts
- Tasks

# Your Advisors

- Sergey Arnautov, sergei@se.inf.tu-dresden.de
- Rasha Faqeh, rasha.faqeh@tu-dresden.de
- Dmitrii Kuvaiskii, dmitrii.kuvaiskii@tu-dresden.de
- Sebastian Ertel, sebastian.ertel@tu-dresden.de

If you’re stuck, have questions/issues, or want to have a consultation, write to one of us.

# Introduction

**Single-threaded code:**

- Underutilized hardware.
- Not scalable.

**Concurrent code:**

- low-level concurrency using threads, locks…
  - still, make more troubles than they solve.
- Higher level concurrency using fork/join, actors…
- Leverage the multicore hardware.

# Goals

- Introduction to state-of-the-art concurrency technologies.
- Hands-on experience in designing high-performance algorithms.
- First experience on parallel programming.
- Evaluation of different approaches.

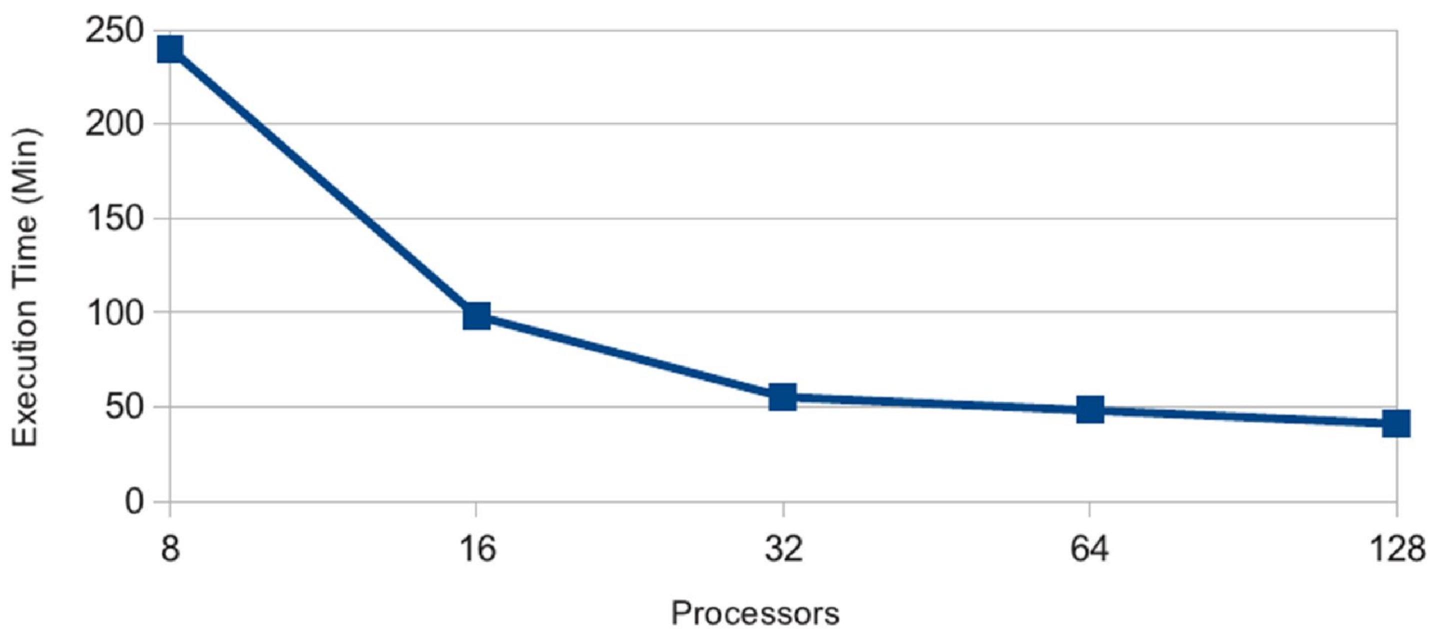
# Submission

- **Intermediate presentation:**
  - Date: 16.6.2014 / 3:00-4:30 pm / room INF3105.
  - Present the ideas/concepts at midterm.

- **Final presentation:**
  - Date: 21.7.2014 / 3:00-4:30 pm / room INF3105.
  - 5 tasks must be solved to pass the lab!
  - Your program will be evaluated at the end of the lab ( deadline: 14.7.2014 / 11:59pm ).
  - Your presentation includes:
    Program architecture,
    Experience gathered,
    Algorithm tricks (= creativity).

# Required Measurements

- Total execution time for 1, 2, 4, 8 cores.
- Show that your solution scales.



# Testing Machine

- ssh fcdsrl08.zih.tu-dresden.de
- 8 CPU machine
- accounts: (XX ∈ {01, ... ,05})<br>login: studentXX<br>password: FCDstunXX
- This is not a debugging machine!

# Concurrency Concepts:

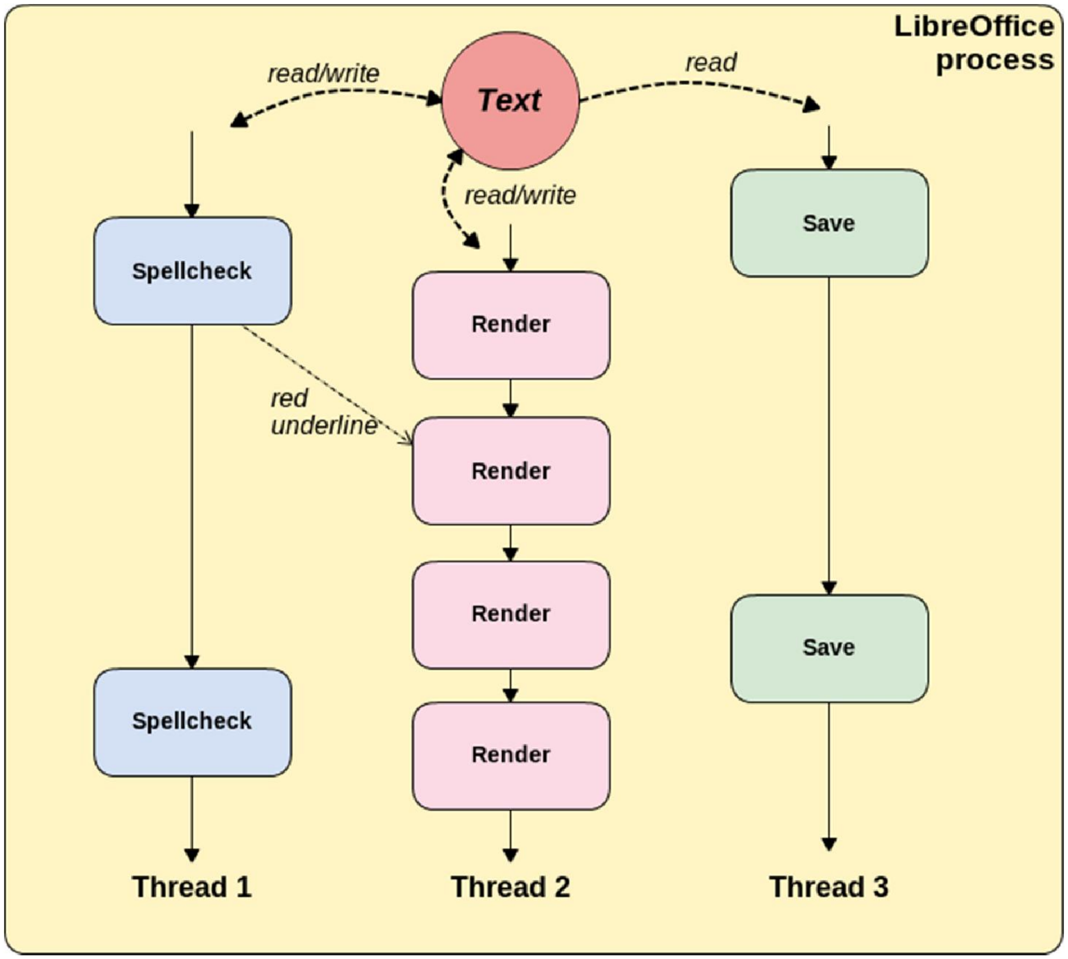
- Thread Model
- Fork-Join Model
- Message Passing Model
- Actors Model

- Implementations: language/library

# Thread Model

- Shared **memory model.**
- **Singl**e "heavy weight" process has **multiple** "light weight", concurrent execution paths (threads).
- Threads communicate via shared variables (need to be careful: locks/semaphores) and/or sending signals.
- Threads split the **tasks**.
- Most control, least safety/comfort.
- Implementations:
  - C (Pthreads)
  - C++ (Boost Threads)
  - Java (Thread/Runnable classes)
  - Python (threading module)

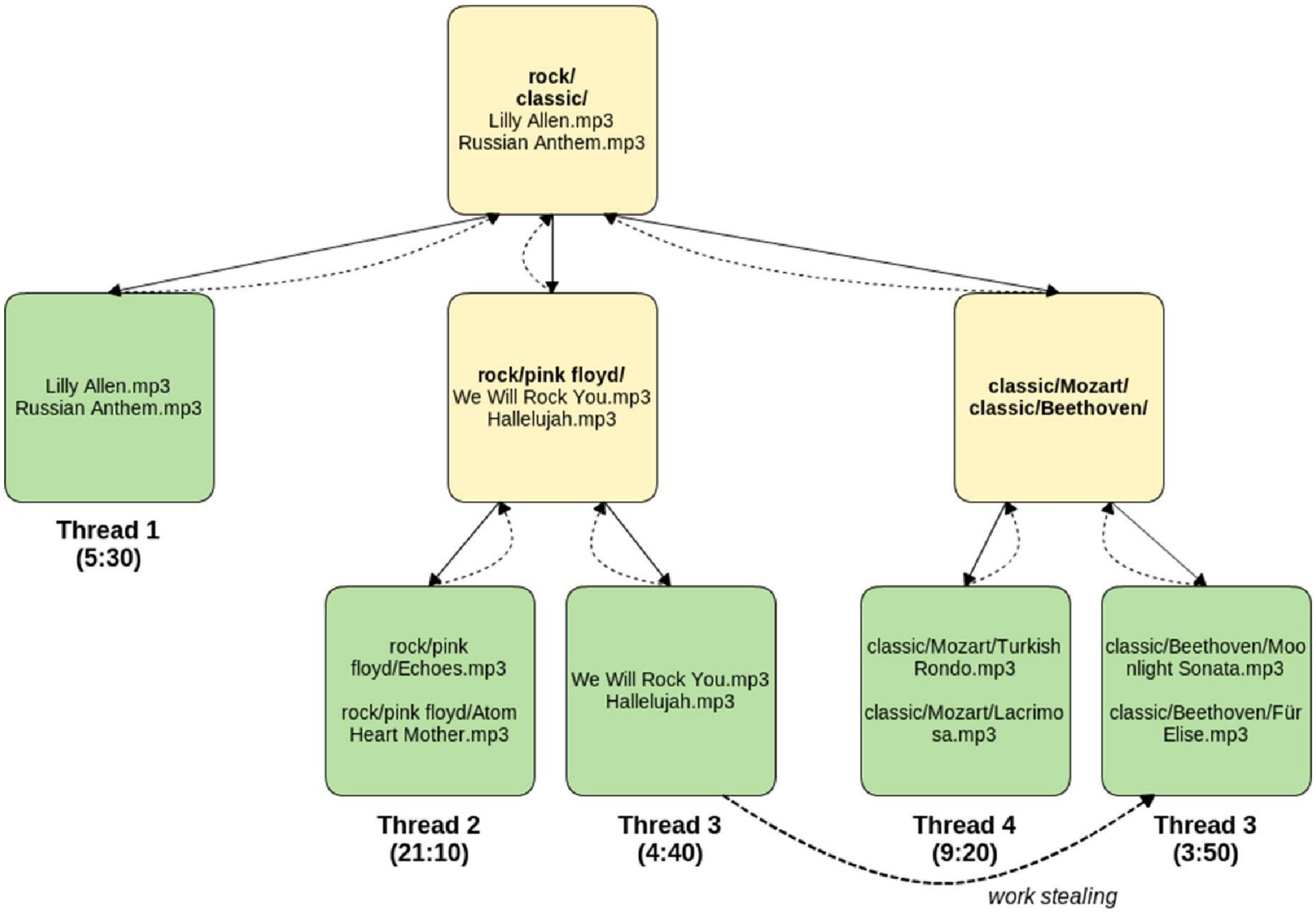
# Threading Example



# Fork-Join Model

- Divide and Conquer model to solve hierarchical problems.
- Split a problem into smaller sub-problems and recursively apply the same algorithm to each sub-problem (**Fork**)…Split the **data**
- Solutions of all sub-problems are combined to solve the initial problem (**Join**).
- Sub-problems do not share data: no locks, no races!
- Implementations:
  - Java (ForkJoinPool)
  - C:
    - OpenMP -- uses pragmas, gcc 4.3
    - Cilk Plus -- extension of C/C++, gcc 4.9
    - Intel Thread Building Blocks -- C++ template library, gcc 3.4

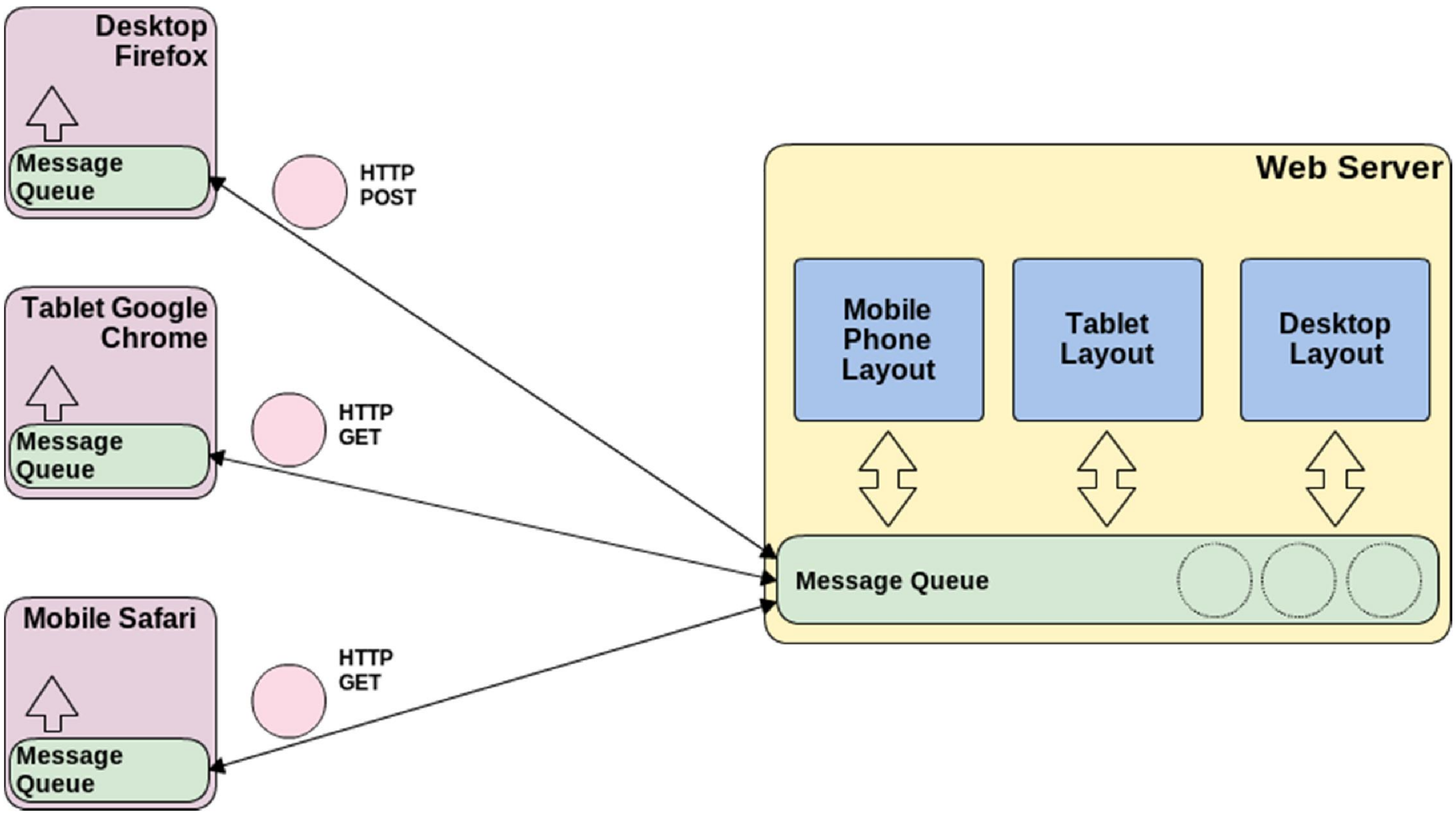
# Fork-Join Example



# Message Passing Model

- Different **objects** (*actors*, *agents*) communicate only via sending and receiving messages.
- No shared data -- messages contain **full copies**.
- Need an infrastructure to communicate -- **channels** (*message queues, pipes, sockets*).
- Synchronous or Asynchronous MP.
- Great for distributed programming, useful for concurrent programming.
- Implementations:
  - ZeroMQ (bindings to C, C++, Java, Python, PHP, Ruby)

# Message Passing Example



# Actor Model

- Specific implementation of message passing.
- Actors are **independent isolated** objects -- no shared data.
- Actors communicate only via **asynchronous** messages:
  - Actor can send message to itself --> recursion;
  - Actors can create new actors and send their addresses to other actors.
- Actors reuse the same threads from thread pool--> don't require excessive system resources.
- Implementations:
  – Erlang
  – Rust / D / Google Go
  – Scala (w/wo Akka)

# Tasks

- Tasks provided by 7th Marathon of Parallel Programming 2012

  http://regulus.pcs.usp.br/marathon/12/problems.html

- **Main requirements:**

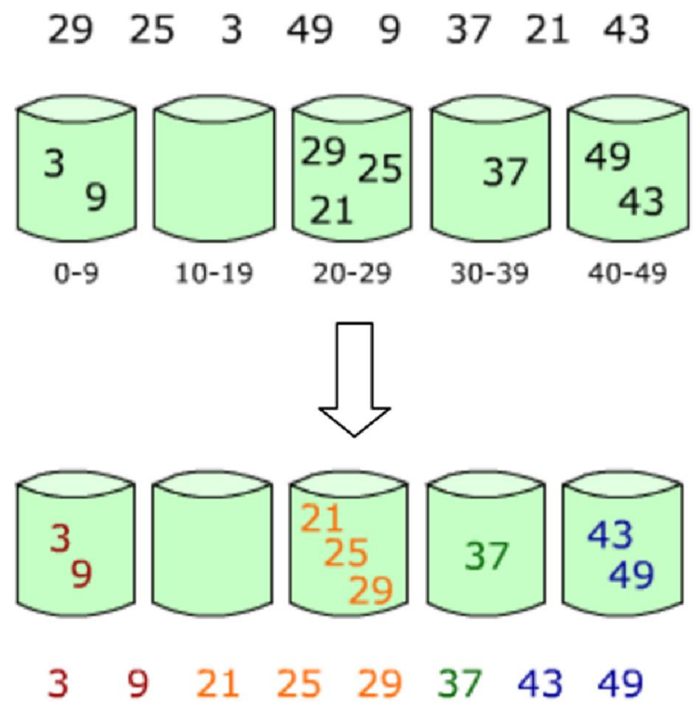
  program correctness and concurrency.

- **5 Tasks:**

  Bucketsort, Mutually Friendly Numbers,
  Haar Wavelets, Unbounded Knapsack Problem,
  3SAT

# Task 1: Bucketsort

1. Divide and Conquer algorithm.
2. Partition input array into **buckets**.
3. Sort each bucket **individually**.



# Task 2: Mutually Friendly Numbers

1. Two numbers are **mutually friendly**
   - if the ratio of the sum of all divisors of the number
   - and the number itself
   - is equal to the corresponding ratio of the other number.
2. Find all pairs of numbers that are mutually friendly in specified range.

$$\frac{1+2+3+5+6+10+15+30}{30} = \frac{72}{30} = \frac{12}{5}$$

$$\frac{1+2+4+5+7+10+14+20+28+35+70+140}{140} = \frac{336}{140} = \frac{12}{5}$$

# Task 3: Haar Wavelets

1. Transformation to prepare images for compression.
2. Input: matrix of ZxZ greyscale pixels.
3. Each pass: calculate approximation and details coefficients.
4. Next pass on smaller matrix.

$t_0 = [\ 420\ \ 680\ \ 448\ \ 709\ \ 1420\ \ 1260\ \ 1600\ \ 1600\ ]$

$a_1 = (420+680) \div 2,\ d_1 = (420-680) \div 2$

$a_2 = (448+709) \div 2,\ d_2 = (448-709) \div 2$

$a_3 = (1420+1260) \div 2,\ d_3 = (1420-1260) \div 2$

$a_4 = (1600+1600) \div 2,\ d_4 = (1600-1600) \div 2,\ \therefore$

$t_1 = [\ 550\ \ 578\ \ 1340\ \ 1600\ \ -130\ \ -130\ \ 80\ \ 0\ ]$

$a_1 = (550+578) \div 2,\ d_1 = (550-578) \div 2$

$a_2 = (1340+1600) \div 2,\ d_2 = (1340-1600) \div 2$

$t_2 = [564\ \ 1470\ \ -14\ \ -130\ \ -130\ \ -130\ \ 80\ \ 0\ ]$

$a_1 = (564+1470) \div 2,\ d_1 = (564-1470) \div 2$

$t_3 = [1017\ \ -453\ \ -14\ \ -130\ \ -130\ \ -130\ \ 80\ \ 0\ ]$

# Task 4: Unbounded Knapsack Problem

1. Resource allocation problem.
2. You have a knapsack with **weight capacity** *M*.
3. You also have *n* **types of items** with their **weights** and **values**.
4. Cram so many items in the knapsack that:
   - the **total value** is the **maximum possible** and
   - the **total weight** does not **exceed** *M.*
5. *Unbounded* means as many copies of each type of item as you like!

# Task 5:We're Back: 3SAT

1. 3-satisfiability, where each clause contains **exactly** 3 literals.
2. Literal is **variable** or a **negation of variable**.
3. Input: amount of clauses, amount of variables.
4. Prove satisfiability:
   - If **at least one assignment** of variables exists when formula becomes TRUE, then function is **satisfiable**;
   - If **no such assignment** exists (formula is always FALSE), then function is **unsatisfiable**.

# Our Suggestions

- You always wanted to try that new language/library?
  - Try it for this lab, we're happy with new approaches.
- You don't have any preferences?
  - Choose from one of our suggestions.

| **Language/ library** | C with Pthreads | Java with Fork/Join | Python with ZeroMQ | Google Go with go-routines | D (or Rust) with actors |
|---|---|---|---|---|---|
| **Parallel model** | Threads | Fork/Join | Message Passing | Actors | Actors |