



# Do PCP 2025 class quiz #1 now

On Amathuba

- Password: **serial**

Amdahl's Law

parallel

concurrent

non-deterministic

*thread of control*

*processors versus processes*  
fork-join parallelism.

# Parallel and concurrent programming

## 1. Introduction: what is parallel programming?

PROTECTION

*data race*

synchronization

divide-and-conquer algorithms

THREAD SAFETY

correctness

MUTUAL EXCLUSION

*locks*

liveness

**Prof. Michelle Kuttel**

DEADLOCK

starvation

HIGH PERFORMANCE COMPUTING

EXECUTORS

*thread pools*

timing

# Who am I?

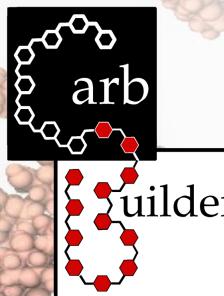
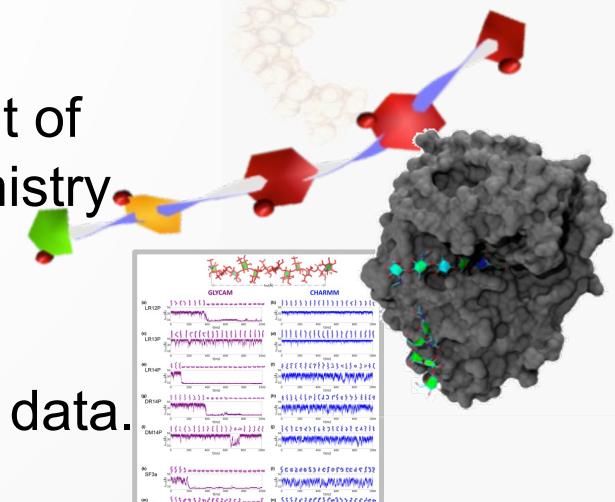
HOD

Main research focus is **computational glycochemistry**:

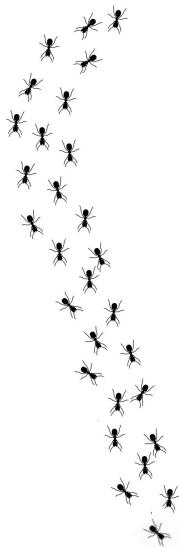
- using molecular modeling to investigate the **conformation** and interactions of bacterial **carbohydrates** that are used in modern **vaccines**

Some other areas:

- **Parallel and High Performance Computing:** using parallel architectures to accelerate compute-intensive algorithms;
- **Research Programming:** the development of software tools to support research in Chemistry
- **Visualization:** developing novel effective visualizations of complex multidimensional data.



# PCP admin



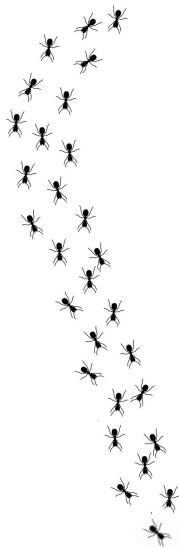
When possible (*loadshedding?!*), course will be filmed for your convenience

- But – you are **required to come to lectures**
- I will be running the **marked quizzes** in most **lectures**.

## Protocol for course queries for PCP.

- All general queries on Amathuba or the **CSC2002S MSTeam**.
- If you need speak to **me**, do so before or after class, or make an appointment on the Google appointment booking page:  
<https://calendar.app.google/PWueVvA7xzTy5jNT9>
- You **cannot** simply **pop in** at other times.

# Time management and your PCP assignments



- Historically, **most DPRs** are because students did not submit a PCP assignment.
  - You have to do the assignments,
  - and you have to hand in on time
    - we are not lenient about this.
- These assignments are hard and long (and fun!)
  - **start as soon as they are released.**
- If you have *bad time management*, now is the time to address this *critical weakness...*



# PCP Content

This module covers the **essential concepts** of **parallelism** and **concurrency**.

The PCP module will cover 10 sections over the 4 weeks that it runs.

**Part I** is primary concerned with **parallel programming**, **Part II** with **concurrency**.

*(Slides will be added before the lectures)*

## Sections

- 1.Introduction:** What is parallel /concurrent programming?
- 2.Introduction:** The shared memory programming model
- 3.Introduction:** Multithreading in Java
- 4.Part I:** Parallel programming in Java
- 5.Part I:** Parallel programming: algorithms
- 6.Part I:** Parallel programming: performance
- 7.Part II:** Concurrent correctness: Mutual Exclusion
- 8.Part II:** Concurrent correctness: Thread Safety
- 9.Part II:** Concurrent correctness: Condition variables
- 10.Part II:** Concurrent liveness: Deadlock



# PCP content

***A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency***

by Dan Grossman (U. Washington, USA)

This course textbook is **free** and available on Vula  
(and a bit *old*, but still valid)

The module will contain other information from other sources; **all content presented online or in class is examinable**.

*Note also that there will be some changes in content from previous years.*

# ✳️ Changing a core assumption

So far, your algorithms/programs have assumed that  
**one thing happens at a time**

this is traditional *sequential programming*.



# ✳️ Changing a core assumption

So far, your algorithms/programs have assumed that  
**one thing happens at a time**

this is traditional *sequential programming*.

Everything is part of **one sequence**

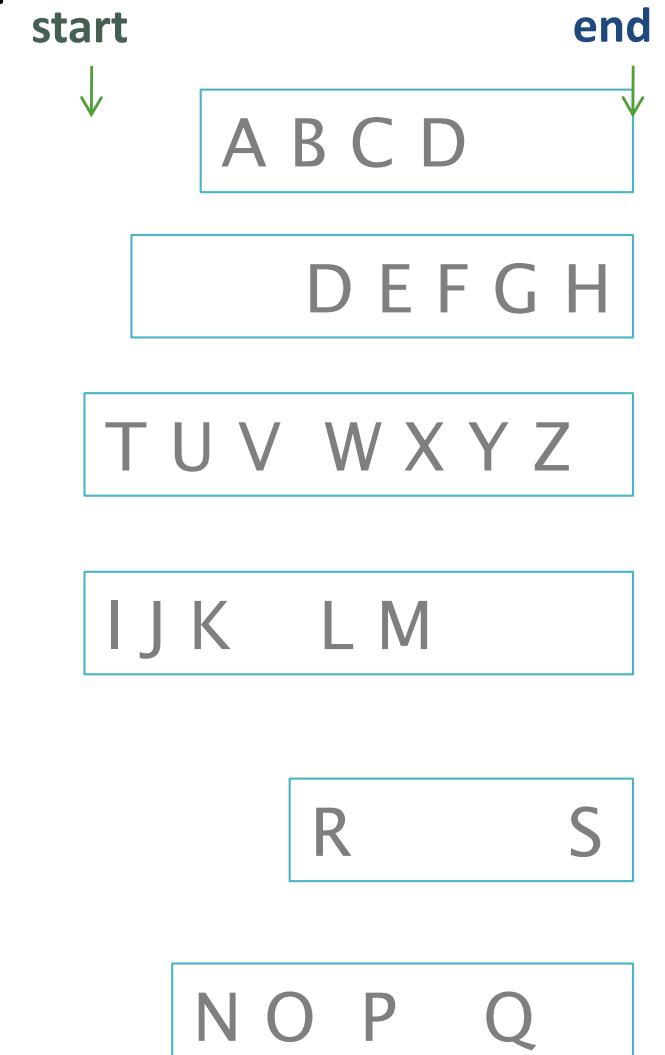
- execution **starts at the beginning** and proceeds one assignment/call/return/arithmetic operation at a time





# Changing a core assumption

We now **remove the sequential**  
assumption and allow  
**many things** to happen **at once**.

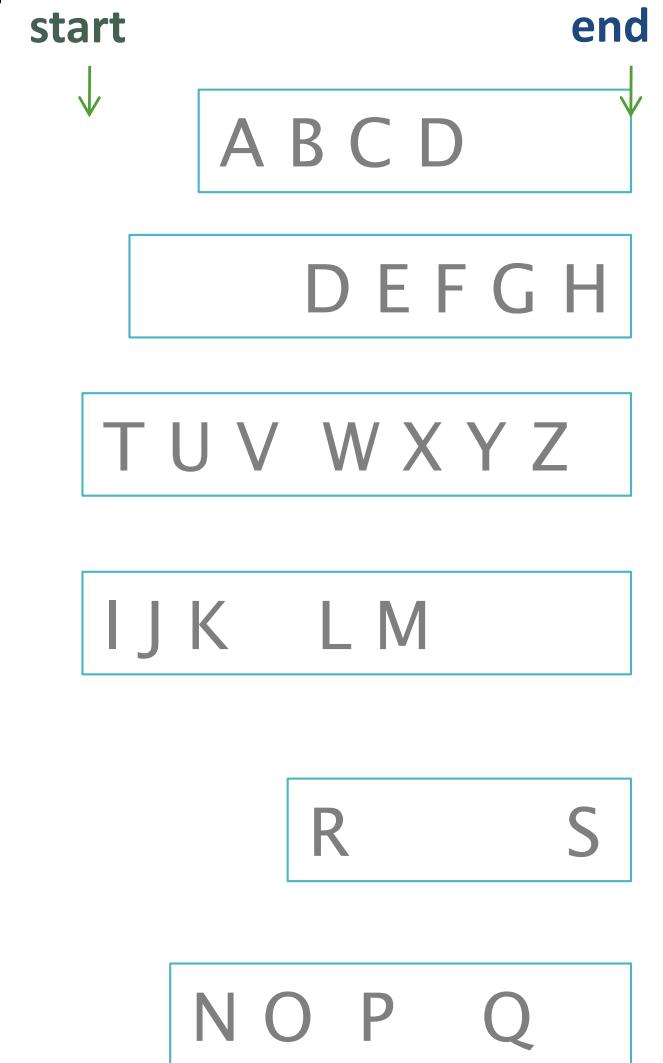




# Changing a core assumption

We now remove the sequential assumption and allow **many things to happen at once.**

This **complicates** things...





# Changing a core assumption

- Parallel and concurrent programming removes the restrictions of sequential programming that require **computational steps** to occur in a **serial order** in a **single place**.



# From the ACM/IEEE/AAAI Computer Science curriculum recommendations

In most conventional usage,

- “**parallel**” programming focuses on arranging that **multiple activities co-occur**,
- “**distributed**” programming focuses on arranging that **activities occur in different places**, and
- “**concurrent**” programming focuses on **interactions of ongoing activities** with each other and the environment.

However, **all three** terms may apply in most contexts.

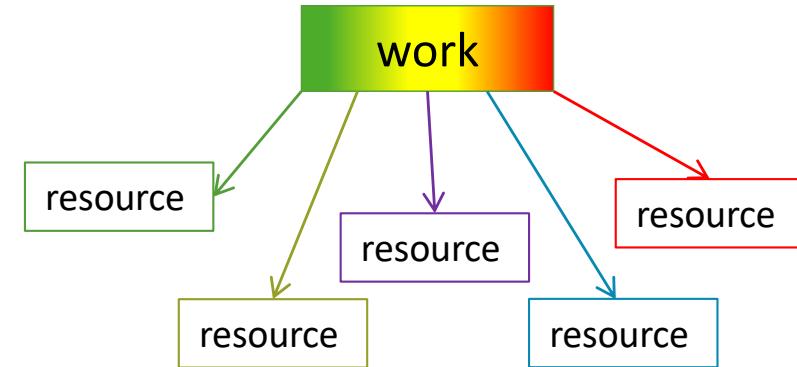
e.g. Parallelism generally implies some form of distribution because multiple activities occurring without sequential ordering constraints happen in multiple physical places.



# Parallelism versus *Concurrency*

## Parallel program:

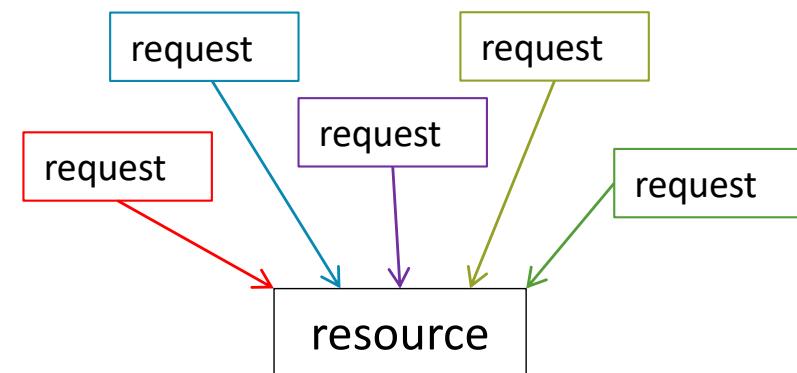
- Use extra computational resources to solve a problem faster



## Concurrent program:

- Correctly and efficiently manage access to shared resources

These terms are NOT standard, but perspective is important.  
Many programmers confuse these terms.



# Parallel algorithm example

Use **extra computational resources** to solve a problem **faster** (increasing throughput via simultaneous execution)

Pseudocode for array sum

- Bad style for reasons we'll see, but may get roughly 4x speedup

```
int sum(int[] arr) {
    res = new int[4];
    len = arr.length;
    FORALL(i=0; i < 4; i++) { //parallel iterations
        res[i] = sumRange(arr,i*len/4,(i+1)*len/4);
    }
    return res[0]+res[1]+res[2]+res[3];
}
int sumRange(int[] arr, int lo, int hi) {
    result = 0;
    for(j=lo; j < hi; j++)
        result += arr[j];
    return result;
}
```

A pseudocode **FORALL** loop is like a *for* loop, except that it does all the iterations in parallel (**at once**).



# Concurrency Example

## Multiple operations on hash table

- What happens if two **inserts** at same time?  
    Duplicate key?
- We need to **prevent this.**



# Concurrency Example

**Concurrency:** Correctly and efficiently **manage access** to **shared resources** (from multiple possibly-simultaneous clients)

Pseudocode for a **shared hashtable**

- Prevent *bad interleaving* (correctness)
- But allow some concurrent access (performance)

```
class Hashtable<K, V> {  
    ...  
    void insert(K key, V value) {  
        int bucket = ...;  
        prevent-other-inserts/lookups in table[bucket]  
        do the insertion  
        re-enable access to arr[bucket]  
    }  
    V lookup(K key) {  
        (like insert, but can allow concurrent  
        lookups to same bucket)  
    }  
}
```

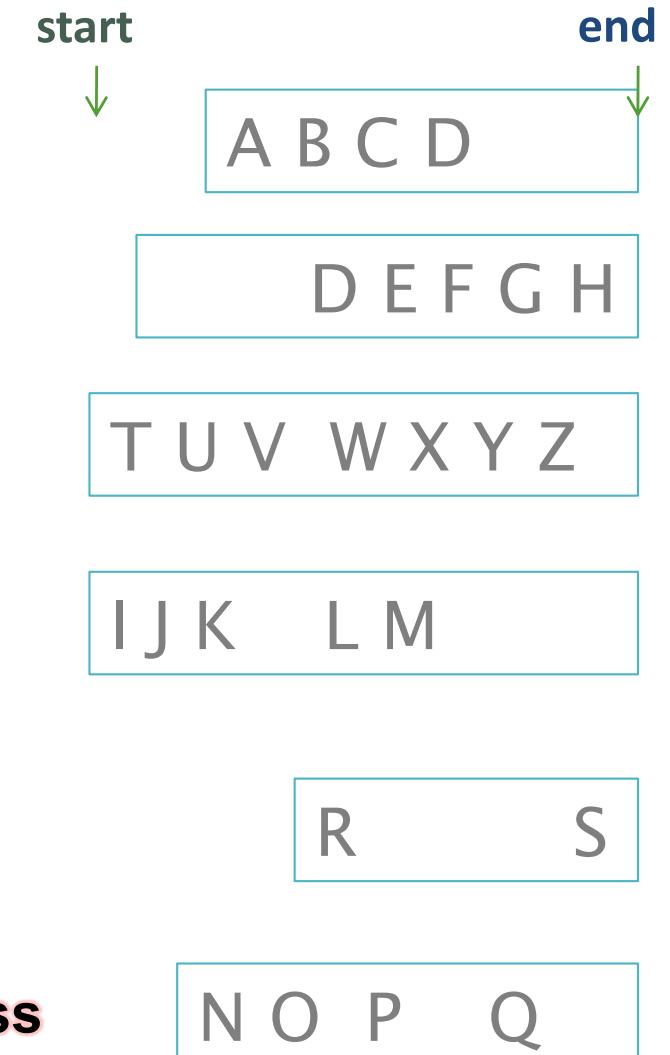
concurrent programs use **synchronization** to prevent multiple operations from interleaving in a way that leads to incorrect results



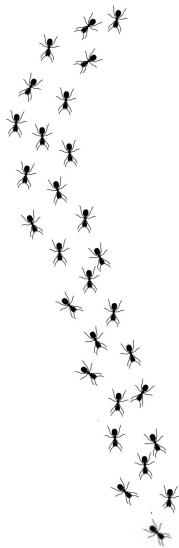
# Summary: Changing a core assumption

Removing the sequential assumption creates both **opportunities** and **challenges**

- Programming:
  - **Divide** and **coordinate work**
- Algorithms:
  - more **throughput**: work done per unit wall-clock time = speedup
  - more **organization required**
- Data structures:
  - May need to support **concurrent access**



# Parallelism versus Concurrency

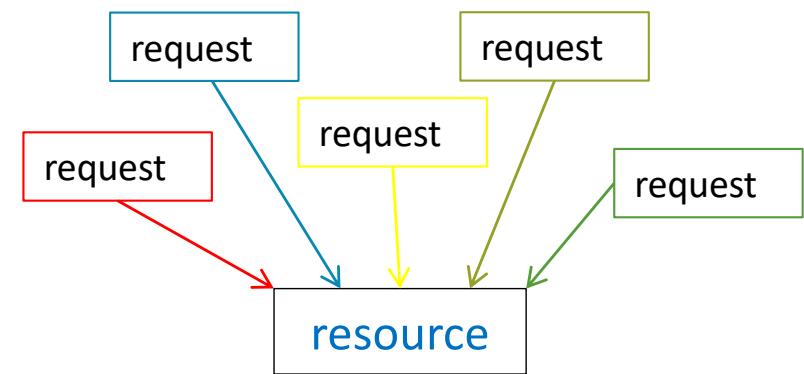
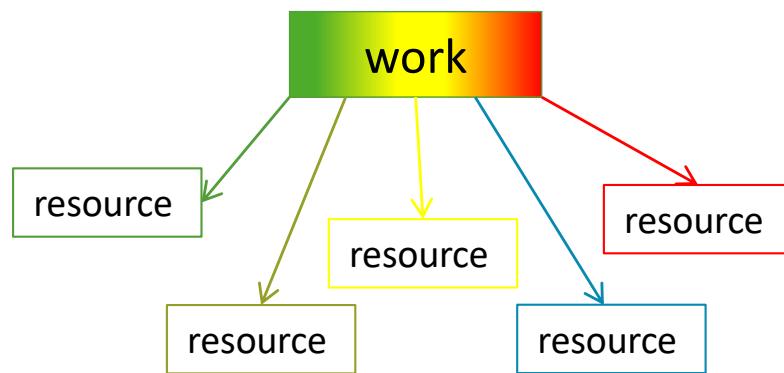


There is some connection:

- If parallel computations need access to shared resources, then the concurrency needs to be managed

We first cover **parallelism**, then **concurrency**:

**one assignment in each**



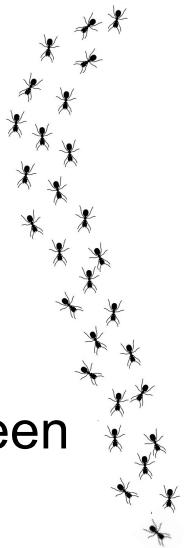


# Why is Parallel/Concurrent Programming Hard?

- Try preparing a seven-course banquet for 300 people
  - by yourself...
  - with one friend....
  - with 32 friends ...
  - with 1024 friends.



# Restaurant analogy contd.



A serial program is a recipe for **one cook**

- the cook does one thing at a time (sequential), maybe switching between tasks (time-slicing)

Options for **lots of cooks**:

- use helpers to assist with a big task, like slicing lots of potatoes for hot chips (**parallel**)
- dedicate helpers to different tasks, such as beating eggs, slicing onions, washing up  
(**concurrent**)
- have a list of tasks and assign them to workers as they are free  
(parallel/concurrent **work pool** )

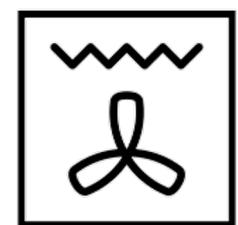
But.... **too many chefs = too much time coordinating**, bumping into each other etc!



# Why is Parallel/Concurrent Programming so Hard?

Often there are **multiple steps** that can be done at the same time - with access to more cooks.

- More cooks require **extra coordination**.
  - A cook may have to wait for another cook to finish something.
- And there are **limited resources**:
  - If you have only one oven, two cooks won't be able to bake cakes at different temperatures at the same time.
  - And some recipes will require exclusive use of the oven (e.g. a souffle)
- Multiple cooks present efficiency opportunities, but also **significantly complicate** the process of producing a meal.





# Restaurant analogy contd.

## Timing (*concurrency*) issues

- Lots of cooks making different things, but only 2 ovens
  - One approach is to allow access to ovens but only for one cake at a time (for safety, cakes can flop)
  - Cooks **must wait** until oven free



# Do PCP 2025 class quiz #2 now

On Amathuba

- Password: **parallel**

# Restaurant analogy contd.

## Timing (*concurrency*) issues

- Lots of cooks making different things, but only 2 ovens
  - One approach is to allow access to ovens but only for one cake at a time (for safety)
  - Cooks **must wait** until oven free

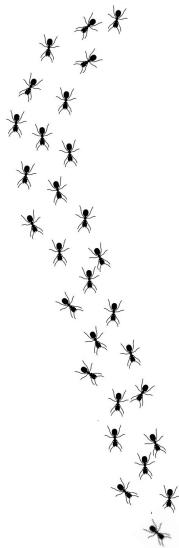
```
if (!ovenBusyLightOn) {  
    ovenBusyLightOn=true;  
    bakeCake();  
    ovenBusyLightOn=false;  
}
```

This pseudo code is fine for a single threaded program.

Will it work in a multithreaded program?



# Restaurant analogy contd.



## Timing (*concurrency*) issues

- Lots of cooks making different things, but only 2 ovens
  - One approach is to allow access to ovens but only for one cake at a time (for safety)
  - Cooks **must wait** until oven free



```
while (ovenBusyLightOn);  
ovenBusyLightOn=true;           //Cooks now wait for a signal.  
bakeCake();  
ovenBusyLightOn=false;
```

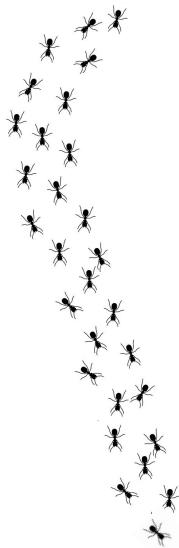


# In Java...

```
public class BusyThread extends java.lang.Thread {  
    private int i;  
    public static boolean ovenBusyLightOn=false;  
    BusyThread(int i) { this.i = i; }  
  
    public void run() {  
        while (ovenBusyLightOn);  
        ovenBusyLightOn=true;  
        bakeCake();  
        ovenBusyLightOn=false;  
    }  
  
    public void bakeCake() {  
        System.out.println("Baker " + i+ " is baking!");  
        System.out.println("Baker " + i+ " finished baking!");  
    }  
  
    public static void main(String[] args) {  
        for(int i=1; i <= 4; ++i) {  
            BusyThread c = new BusyThread(i);  
            c.start();  
        }  
    }  
}
```

Baker 1 is baking!  
Baker 1 finished baking!  
Baker 2 is baking!  
Baker 2 finished baking!  
Baker 3 is baking!  
Baker 3 finished baking!  
Baker 4 is baking!  
Baker 4 finished baking!

# Restaurant analogy contd.



## Timing (*concurrency*) issues

- Lots of cooks making different things, but only 2 ovens
  - One approach is to allow access to ovens but only for one cake at a time (for safety)
  - Cooks **must wait** until oven free



```
while (ovenBusyLightOn);  
ovenBusyLightOn=true;           //Cooks now wait for a signal.  
bakeCake();  
ovenBusyLightOn=false;
```

*BUT the validity of this code depends on the implementation of the **ovenBusyLightOn** Boolean ...*

*validity depends on the implementation of the **ovenBusyLightOn** Boolean ...*

## In Java...

```
public class BusyThread extends java.lang.Thread {  
    private int i;  
    public static boolean ovenBusyLightOn=false;  
    BusyThread(int i) { this.i = i; }  
  
    public void run() {  
        while (ovenBusyLightOn);  
        ovenBusyLightOn=true;  
        bakeCake();  
        ovenBusyLightOn=false;  
    }  
  
    public void bakeCake() {  
        System.out.println("Baker " + i+ " is baking!");  
        System.out.println("Baker " + i+ " finished baking!");  
    }  
  
    public static void main(String[] args) {  
        for(int i=1; i <= 10; ++i) {  
            BusyThread c = new BusyThread(i);  
            c.start();  
        }  
    }  
}
```

Baker 1 is baking!  
Baker 1 finished baking!  
Baker 5 is baking!  
Baker 5 finished baking!  
Baker 2 is baking!  
Baker 2 finished baking!  
Baker 4 is baking!  
Baker 3 is baking!  
Baker 3 finished baking!  
Baker 4 finished baking!  
Baker 9 is baking!  
Baker 9 finished baking!  
Baker 8 is baking!  
Baker 8 finished baking!  
Baker 6 is baking!  
Baker 10 is baking!  
Baker 6 finished baking!  
Baker 10 finished baking!  
Baker 7 is baking!  
Baker 7 finished baking!

*More about this in Part II...*



# Concurrency problems: Race Conditions

important for tests & exams to know

A **race condition** is an error in a program where the output and/or result of the process is **unexpectedly** and **critically** dependent on the relative sequence or timing of other events.

The idea is that the events **race** each other to influence the output first.

***Unintended race conditions*** are *incredibly common in concurrent programs.*



# Nondeterminism

In sequential programs, instructions are executed **in a fixed order** determined by the program and its input. The execution of one procedure does not overlap in time with another. **Deterministic**

In concurrent programs, computational activities may overlap in time and the subprogram executions describing these activities proceed concurrently.

**Nondeterministic**

it is not possible to tell, by looking at the program, what will happen when it executes.

# Simple example of non-determinism: An Audition

**Bob's** lines are:

**“Hi Alice!”**

**“Bye Alice!”**

**Alice’s** lines are:

**“Hi Bob!”**

**“Bye Bob!”**

They cannot **communicate** - they are in separate sound-proof rooms for recording purposes.

Assume that you are taping the output from both rooms.

**What will you tape?**

# Simple example of non-determinism: An Audition

Bob's lines are:

**“Hi Alice!”**

**“Bye Alice!”**

Alice's lines are:

**“Hi Bob!”**

**“Bye Bob!”**

Program to demonstrate

Note: in Java, concurrent programming is done by **threads**.

*Why Alice and Bob? – see [https://en.wikipedia.org/wiki/Alice\\_and\\_Bob](https://en.wikipedia.org/wiki/Alice_and_Bob)*

# Seriously expensive race conditions (#1)

1985-1987

- **Therac-25**, a radiation therapy machine for cancer treatment used software safety systems that had a **race condition** between operator text input and software configuring the electron beam
  - caused by users hitting the up arrow too quickly, preventing the system from properly registering their edits.
- Resulted in an occasional 100X radiation dose, that lead to **several deaths**.



Nancy Leveson and Clark Turner, “The Investigation of the Therac-25 Accidents”, Computer, 26, 7 (July 1993) pp 18-41.

# Seriously expensive race conditions (#2)

2003

a **race condition** in General Electric Energy's Unix-based **energy management system** aggravated the **USA Northeast Blackout** - biggest blackout in North American history

- trip alarm failed due to a **race condition** in alarm subsystem
  - three sagging power lines were tripped simultaneously, but the race condition prevented alerts from being raised to the monitoring technicians, delaying their awareness of the problem
- created a **cascade** of power failures
- 50 million people lost power for up to two days
- cost an estimated \$6 billion



source: Scientific American

# Seriously expensive race conditions (#3)

## Mars Rover "Spirit"

- a six-wheeled driven, four-wheeled steered vehicle designed by NASA to navigate the surface of Mars in order to gather videos, images, samples and other possible data about the planet.

### ***sol 136-139, June 02, 2004: Spirit Recovers from a Low Probability Software Error***

- The Spirit designed to reboot its operating system whenever it ran into a problem that it could not resolve.
- The rover's computer was programmed to "write-protect" its software unless a specific command is given to lift that protection to make a change.
- Race condition: a "lift" command was given as part of one software routine — but before the change could be written to the rover's memory, a different routine put the write protection back in place.
- "It caused the system to say, 'I can't write to that area, so this is a software problem I can't do anything about, so therefore I have to reboot,'"
- But during the boot process found the problem with the file system. And so it rebooted itself again.
- a **reboot loop**



Eventually, the batteries drained, a scenario that activated a setting similar to "Safe Mode" on Windows PCs, where only essential files are loaded at startup. The technicians on earth could then fix this remotely.



<https://mars.nasa.gov/mer/mission/rover-status/spirit/2004/all/>



## Race conditions

Samuel Gibbs and  
Matthew Weaver

Wed 29 Nov 2017 10.28 GMT

 Share

### Apple "Root Bug" (2017)

Race condition in macOS High Sierra allowed users to enable the root user with a blank password just by trying to log in as root multiple times.

Anyone could gain **full root access** without authentication due to a flaw in how the system handled authentication and user creation concurrently.

### MacOS High Sierra bug: blank password let anyone take control of a Mac

Apple provides emergency fix for flaw that allows access to secure preferences with username 'root' and subsequent bypass of lock screen



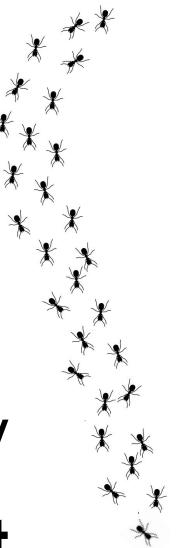
 Taking Mac control ... the security flaw was initially discovered a couple of weeks ago and disclosed in an Apple developer support forum. Photograph: Apple

A serious security flaw was found in the latest version of Apple's macOS High Sierra that could allow anyone to access locked settings on a Mac using the user name "root" and no password, and subsequently unlock the computer.



## Checkpoint: Race conditions

- Give an example of a race condition in everyday life.



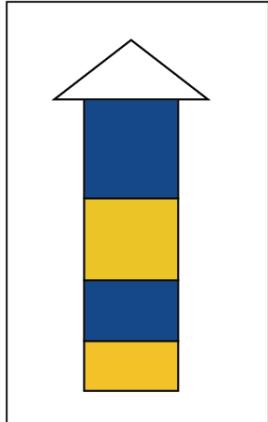
# Parallel/Concurrent complexity

**Managing parallel complexity**

and the **principles and techniques** necessary  
for the construction of **well-behaved** and **efficient**  
**parallel** and *concurrent* programs

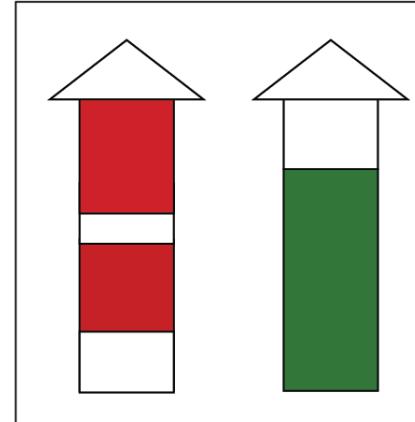
is the main subject matter of this module

Concurrency



Concurrency is about *dealing with*  
lots of things at once

Parallelism



Parallelism is about *doing*  
lots of things at once



# Difficult task

Parallel/concurrent programming is a **lot of work...**

Writing **correct** and **efficient** multithreaded programs is often **much more difficult**

than single-threaded (sequential) code

- Increased **complexity (pain)** of writing programs, especially in common languages like Java and C

So a general rule is to **stay sequential** if possible.

When is parallel/concurrent  
programming necessary?



# When is concurrent programming necessary?

Complex programs like

- **operating systems,**
- **web browsers,**
- **Real-time systems,**
- **event-based** implementations of graphical user interfaces,
- multiuser games, chats and ecommerce.

handle **multiple things** happening **at once** and therefore are **multithreaded**.

- Many concurrent threads run simultaneously.





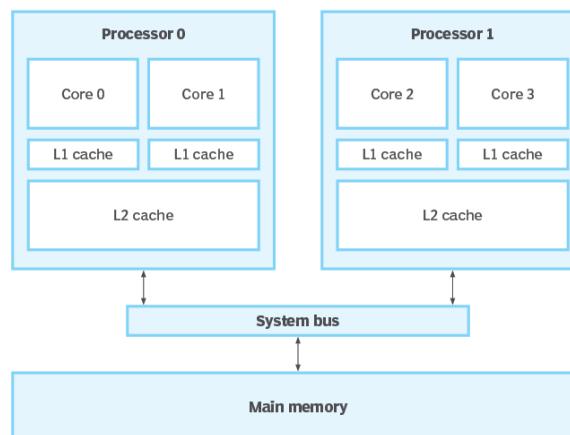
# Using multiple cores

Parallel/concurrent programming **is essential** to use the computing power of **multicore architectures**

And modern desktop and laptop computers all have **multiple cores**.

Couldn't keep making "wires exponentially smaller" so put **multiple processors** on the same chip (**multicore**)

Architecture of multicore processors



# ✳️ The death of Moore's Law

Between 1980-2005 Moore's Law\* held:

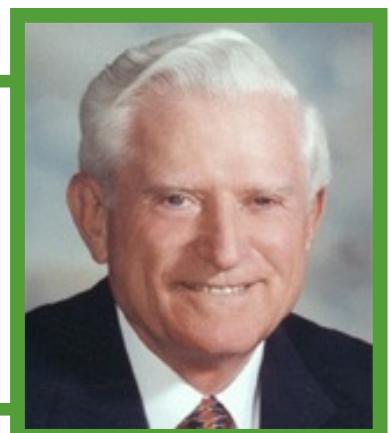
- clock speeds of desktop computers doubled every 18–24 months (“wires exponentially smaller”)
- exponentially faster at running sequential programs

But eventually reached the **power wall**

- Increasing clock rate generates **too much heat**: power and cooling constraints limit increases in microprocessor clock speeds
- Relative cost of memory access is too high

\*Not a physical Law, merely an observation.

Statement made by  
Gordon Moore  
(1929– ),  
Cofounder of Intel,  
in 1965





Note: You don't need multiple cores/CPUs to have race conditions!

## Concurrent execution versus time-slicing

If the computer has multiple processors then instructions from a number of processes/threads, equal to the number of physical processors, can be executed **at the same time**.

- sometimes referred to as **parallel** or ***real concurrent*** execution.

However it is usual to have **more active processes than processors**.

- the available processes are switched between processors – **timeslicing**.

**NB Processor scheduling** is managed by the **operating system** and the programmer has (almost) no control over it.

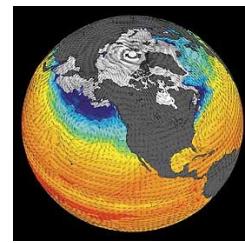
# High Performance Computing

HPC uses **cutting-edge parallel computers** to solve “grand-challenge” problems.

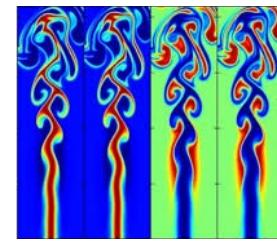
Programmers write programs to solve such **computationally large** problems that it would take years, or centuries, for one computer to finish it.



Proteins and diseases



Climate change



Energy-efficient jet engines



Understanding the universe



HOME    LISTS    STATISTICS    RESOURCES    ABOUT    MEDIA KIT

## El Capitan Retains Top Spot in 65th TOP500 List as Exascale Era Expands

June 10, 2025

The 65th edition of the TOP500 showed that the El Capitan system retains the No. 1 position. With El Capitan, Frontier, and Aurora, there are now 3 Exascale systems leading the TOP500. All three are installed at Department of Energy (DOE) laboratories in the United States.

[read more »](#)



1 **El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE

2 **Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE

3 **Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel

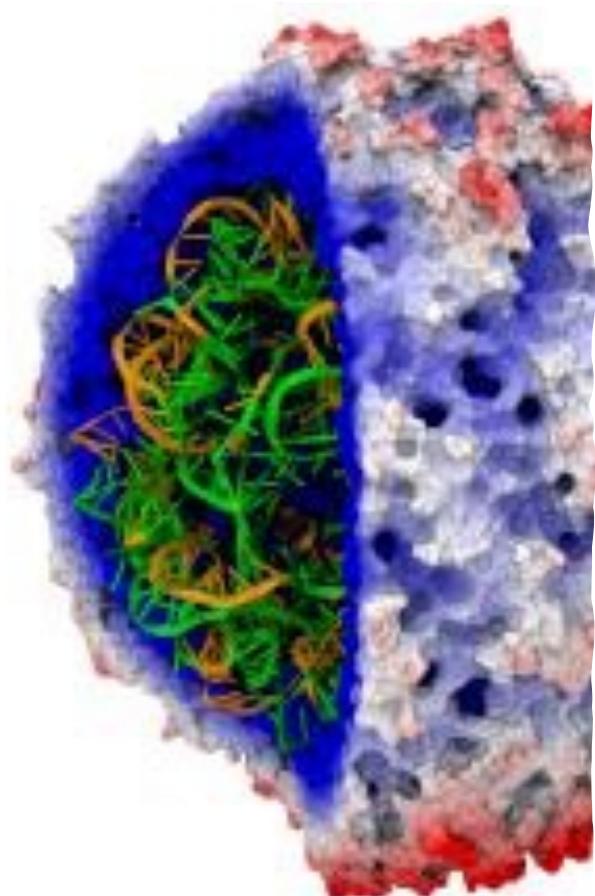
4 **JUPITER Booster** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN

5 **Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA A100, NVIDIA



# HPC in molecular modelling

---



Molecular modelling simulates the behaviour and interactions of molecules to try to understand their properties.

Widely used in **drug development**.

Models can be highly accurate and computationally expensive (quantum mechanics) or less accurate and cheaper (molecular dynamics, coarse-grained models).

Increases in HPC power has allowed simulations to move from isolated molecules in solvent to complex aggregates –

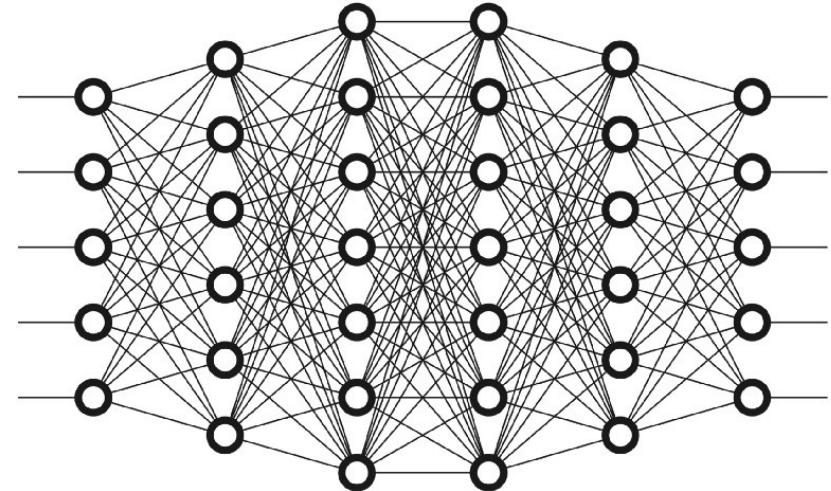
some recent simulations have exceeded **10,000,000 atoms**.

# HPC and deep learning

The recent development of deep learning has revolutionized artificial intelligence, especially for tasks such as speech and image recognition.

Neural networks date back to the 1960's, but HPC has made them widely useful

- The wide availability and low cost of GPUs and accelerated computing parallel programming tools catalysed the modern revolution in AI/deep learning.
- Simple neural networks that were once impossible to train can now be trained in hours or minutes.
- HPC is needed to train large models in a reasonable time.



# HPC for animation and visual effects



- Animation studios need HPC systems.
- Computer graphics must be **rendered** into high-resolution images.
- Rendering turns a virtual 3D scene into a 2D image, requires a LOT of computing power.

