DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 37

1. MULTITHREADING, CONCURRENCY, PARALLELISM
2. SIMPLE MULTITHREADING: CRATE `rayon`
3. OTHER THINGS AVAILABLE IN RUST AND IN GENERAL

1. MULTITHREADING, CONCURRENCY, PARALLELISM

2. SIMPLE MULTITHREADING: CRATE `rayon`

3. OTHER THINGS AVAILABLE IN RUST AND IN GENERAL

RUNNING MULTIPLE THINGS AT ONCE

Various reasons:

RUNNING MULTIPLE THINGS AT ONCE

Various reasons:

- Separate GUI from the processing engine in your application
    - more responsive user experience

RUNNING MULTIPLE THINGS AT ONCE

Various reasons:

- Separate GUI from the processing engine in your application
  - more responsive user experience
- Big scale computation: solving big data problems

RUNNING MULTIPLE THINGS AT ONCE

Various reasons:

- Separate GUI from the processing engine in your application
  - more responsive user experience
- Big scale computation: solving big data problems
- Running analytics on your laptop:
  - speeding up a single core more and more challenging
  - more cores even in consumer laptops

RUNNING MULTIPLE THINGS AT ONCE

Various reasons:

- Separate GUI from the processing engine in your application
  - more responsive user experience
- Big scale computation: solving big data problems
- Running analytics on your laptop:
  - speeding up a single core more and more challenging
  - more cores even in consumer laptops
- GPUs offer **a lot** of (restricted) parallelism

TERM EXPLANATION

TERM EXPLANATION

- Parallelism: things running at the very same time, different cores, processors, machines

TERM EXPLANATION

- Parallelism: things running at the very same time, different cores, processors, machines
- Concurrency: the art of sharing resources, even if only one thread is running at a time

TERM EXPLANATION

- Parallelism: things running at the very same time, different cores, processors, machines
- Concurrency: the art of sharing resources, even if only one thread is running at a time
- Threads:
  - minimum organizational unit of your computation on a single machine
  - multiple of them allowed, running at the same or different times

SOLVING A GIVEN PROBLEM MORE EFFICIENTLY VIA PARALLEL COMPUTATION?

- Very problem dependent:

  digging the Suez canal vs. digging a deep well

- What is possible: one of the deepest questions in computer science

PROGRAMMING: DIFFICULT AND VERY ERROR-PRONE

Challenges:

- Information exchange
- Sharing resources
- Taking and returning them properly:
    - Similar to challenges in memory management

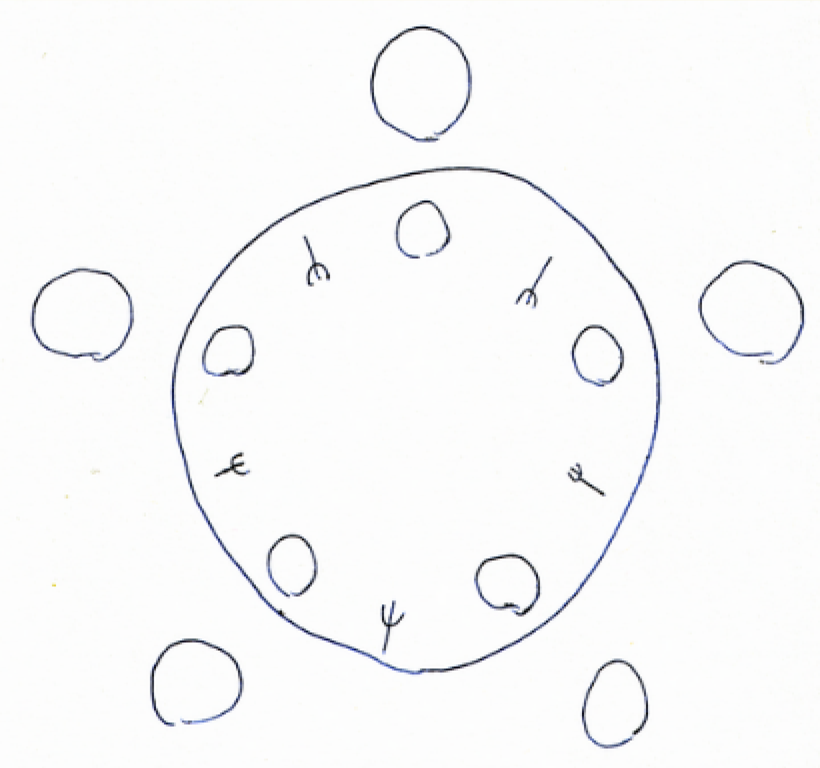
DINING PHILOSOPHERS' PROBLEM

- Multiple philosophers sitting around the table
  - they do two things: think and eat

DINING PHILOSOPHERS' PROBLEM

- Multiple philosophers sitting around the table
    - they do two things: think and eat

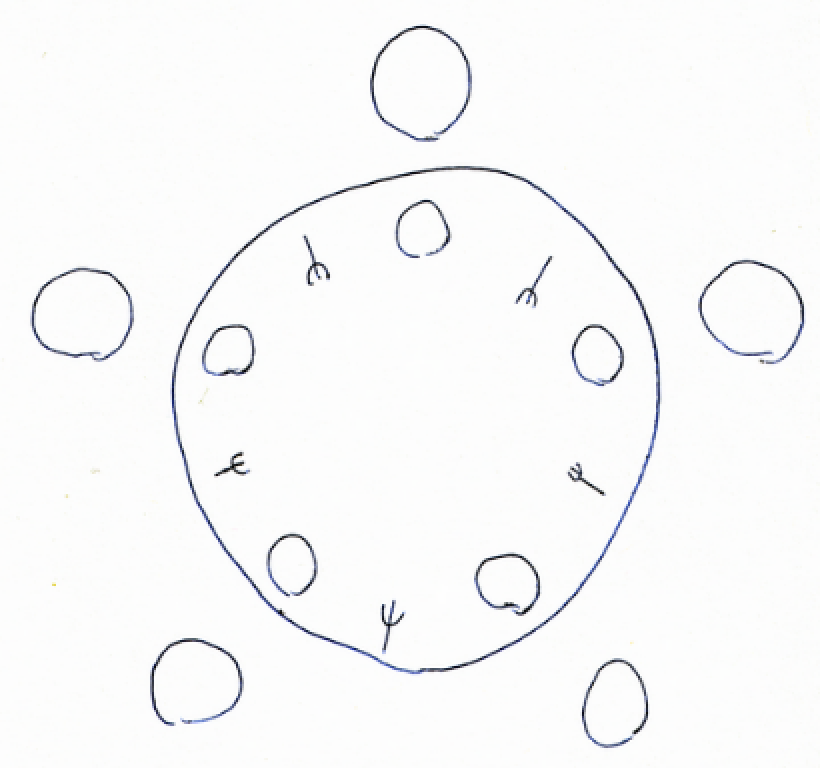
- A single fork between each two of them
- A philosopher needs two forks to eat



DINING PHILOSOPHERS' PROBLEM

- Multiple philosophers sitting around the table
  - they do two things: think and eat

- A single fork between each two of them
- A philosopher needs two forks to eat



What algorithm could the philosophers use to achieve their life goals: eating and thinking?

POTENTIAL PROBLEMS

POTENTIAL PROBLEMS

How about this algorithm?

```
repeat:
   think
   take left fork when available
   take right fork when available
   eat
   return left fork
   return right fork
```

POTENTIAL PROBLEMS

How about this algorithm?

```
repeat:
   think
   take left fork when available
   take right fork when available
   eat
   return left fork
   return right fork
```

- All philosophers could reach for the left fork at the same time!
- They are all stuck
- This is called **deadlock**

POTENTIAL PROBLEMS

How about this algorithm?

```
repeat:
   think
   take any of the forks
   if the other available:
       take it
       eat
   return all forks you have
```

POTENTIAL PROBLEMS

How about this algorithm?

```
repeat:
   think
   take any of the forks
   if the other available:
       take it
       eat
   return all forks you have
```

- A philosopher may never eat!
- This is called **starvation**

1. MULTITHREADING, CONCURRENCY, PARALLELISM

2. SIMPLE MULTITHREADING: CRATE `rayon`

3. OTHER THINGS AVAILABLE IN RUST AND IN GENERAL

CRATE `rayon`

General case difficult:

- manual management of threads
- communication and sharing work by them

Often you may want to speed up simple tasks:

- sorting
- a loop with independent iterations

(Similar in many ways to OpenMP for C/C++/Fortran)

AUXILIARY DEFINITIONS

In [2]:

```
:dep rayon
:dep rand
use rayon::prelude::*;
use std::thread;
use std::time::{Duration,SystemTime};
use rand::Rng;
use std::time::

// see how long something is executing
fn time_it(f: impl FnOnce() -> ()) {
    let before = SystemTime::now();
    f();
    let after = SystemTime::now();
    println!("Time: {:.3?}", after.duration_since(before).unwrap())
}

// do nothing for a specific number of milliseconds
fn wait(millis:u64) {
    std::thread::sleep(Duration::from_millis(millis));
}
```

EXAMPLE OF SORTING

In [3]:

```
// random
const N: usize = 30_000_000;
let mut v = Vec::new();
for i in 0..N {
    v.push(rand::thread_rng().gen_range(0..(N as i32)));
};
```

EXAMPLE OF SORTING

In [3]:

```
// random
const N: usize = 30_000_000;
let mut v = Vec::new();
for i in 0..N {
    v.push(rand::thread_rng().gen_range(0..(N as i32)));
};
```

In [4]:

```
let mut v_copy = v.clone();
time_it(|| v_copy.sort_unstable());

let mut v_copy = v.clone();
time_it(|| v_copy.sort());
```

```
Time: 779.893ms
Time: 1.772s
```

EXAMPLE OF SORTING

In [3]:
```
// random
const N: usize = 30_000_000;
let mut v = Vec::new();
for i in 0..N {
    v.push(rand::thread_rng().gen_range(0..(N as i32)));
};
```

In [4]:
```
let mut v_copy = v.clone();
time_it(|| v_copy.sort_unstable());

let mut v_copy = v.clone();
time_it(|| v_copy.sort());
```

```
Time: 779.893ms
Time: 1.772s
```

In [5]:
```
let mut v_copy = v.clone();
time_it(|| v_copy.par_sort_unstable());

let mut v_copy = v.clone();
time_it(|| v_copy.par_sort());
```

```
Time: 288.935ms
Time: 567.020ms
```

REPLACING ITERATORS WITH PARALLEL ITERATORS

Replace `iter()` with `par_iter()`, `into_iter()` with `into_par_iter()`, etc.

In [6]:
```
// standard version
(1..=20).for_each(|x| {println!("{}",x);});
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

REPLACING ITERATORS WITH PARALLEL ITERATORS

Replace `iter()` with `par_iter()`, `into_iter()` with `into_par_iter()`, etc.

In [7]:
```
// add explicit iterator construction
(1..=20).into_iter().for_each(|x| {println!("{}",x);});
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

REPLACING ITERATORS WITH PARALLEL ITERATORS

Replace `iter()` with `par_iter()`, `into_iter()` with `into_par_iter()`, etc.

In [8]:

```
// replace into_iter() with into_par_iter() and wait for 500 ms to slow things down
(1..=20).into_par_iter().for_each(|x| {wait(500); println!("{}",x);});
```

```
1
11
18
16
2
17
12
19
3
13
14
20
4
15
6
5
8
9
7
10
```

REPLACING ITERATORS WITH PARALLEL ITERATORS

Replace `iter()` with `par_iter()`, `into_iter()` with `into_par_iter()`, etc.

In [9]:
```
// make the wait time variable to see other patterns of execution
(1..=20).into_par_iter().for_each(|x| {wait(x*x*10); println!("{}",x);});
```

```
1
2
3
4
6
5
8
7
11
9
10
12
16
13
18
19
17
14
15
20
```

BENCHMARKING PARALLEL PROCESSING OF A LONG VECTOR

In [10]:

```
let mut v1 : Vec<i32> = (1..=50_000_000).collect();
let mut v2 = v1.clone();
```

BENCHMARKING PARALLEL PROCESSING OF A LONG VECTOR

In [10]:
```
let mut v1 : Vec<i32> = (1..=50_000_000).collect();
let mut v2 = v1.clone();
```

In [11]:
```
// non-parallel version
time_it(|| v1.iter_mut().for_each(|x| *x += 100 / *x + *x / 100));
```

```
Time: 81.415ms
```

BENCHMARKING PARALLEL PROCESSING OF A LONG VECTOR

In [10]:
```
let mut v1 : Vec<i32> = (1..=50_000_000).collect();
let mut v2 = v1.clone();
```

In [11]:
```
// non-parallel version
time_it(|| v1.iter_mut().for_each(|x| *x += 100 / *x + *x / 100));
```

```
Time: 81.415ms
```

In [12]:
```
// using parallel iterators
time_it(|| v2.par_iter_mut().for_each(|x| *x += 100 / *x + *x / 100));
```

```
Time: 31.021ms
```

1. MULTITHREADING, CONCURRENCY, PARALLELISM

2. SIMPLE MULTITHREADING: CRATE `rayon`

3. OTHER THINGS AVAILABLE IN RUST AND IN GENERAL

SAMPLE OTHER THINGS AVAILABLE IN RUST AND BEYOND

SAMPLE OTHER THINGS AVAILABLE IN RUST AND BEYOND

- Starting separate threads and channels to share data
    - communicating to share data:
        - data sent between threads via channels
        - data that was transmitted cannot be accessed anymore:
            - verification via Rust's ownership rules
            - checked at compile time!

SAMPLE OTHER THINGS AVAILABLE IN RUST AND BEYOND

- Starting separate threads and channels to share data
    - communicating to share data:
        - data sent between threads via channels
        - data that was transmitted cannot be accessed anymore:
            - verification via Rust's ownership rules
            - checked at compile time!

- Mutex
    - a lock for accessing a specific resource
    - various versions:
        - only one thread has access at a time
        - or multiple threads with read access / only one thread with write access

SAMPLE OTHER THINGS AVAILABLE IN RUST AND BEYOND

- Starting separate threads and channels to share data
  - communicating to share data:
    - data sent between threads via channels
    - data that was transmitted cannot be accessed anymore:
      - verification via Rust's ownership rules
      - checked at compile time!

- Mutex
  - a lock for accessing a specific resource
  - various versions:
    - only one thread has access at a time
    - or multiple threads with read access / only one thread with write access

- Chapter 16 of "The Rust Programming Language": overview of some mechanisms available in Rust