



Wrocław  
University  
of Science  
and Technology

# Large Scale Data Processing

## Lecture 4 – Data processing languages

dr inż. Tomasz Kajdanowicz, Roman Bartusiak, Piotr Bielak

October 26, 2020



HR EXCELLENCE IN RESEARCH

# Overview

## References

### Languages

Python

Rust

Erlang

Scala

C++

Go

MPI

## References

### Languages

Python

Rust

Erlang

Scala

C++

Go

MPI

# References

- ▶ Frank Mueller - How to Parallelize Your Code: Taking Stencils from OpenMP to MPI, CUDA and TensorFlow
- ▶ David W. Walker - Parallel Programming with OpenMP, MPI, and CUDA
- ▶ Alfio Lazzaro - Code Performance Optimizations

# Overview

## References

### Languages

Python

Rust

Erlang

Scala

C++

Go

MPI



# Python

# Introduction

## Languages (Python)

- ▶ interpreted language
- ▶ strong\*, dynamic typing
- ▶ many packages available
- ▶ most popular language in ML
- ▶ computation speedup via FFI to C, C++



# Strong\*, dynamic typing

Languages (Python)

Dynamic: Not variables, but objects are typed!

```
1 a = 1    # type(a) => <class 'int'>
2 a = 'Hello'  # type(a) => <class 'str'>
```

Strong\*: types do not change unexpectedly

```
1 a = 1
2 b = '23'
3 print(a + b)
4 # Traceback (most recent call last):
5 #   File "<stdin>", line 1, in <module>
6 #   TypeError: unsupported operand type(s) for +: 'int'
   ' and 'str'
```



# Strong\*, dynamic typing

Languages (Python)

... but this works (list behaves like a bool; implicit casting)

```
1 a = [1, 2]
2 if a:
3     print('Not empty!')
```

# Basic data types

## Languages (Python)

```
1 # Text: str
2 a = 'Hello'
3
4 # Numeric: int , float , complex
5 a = 42
6 b = 777.0
7 c = 34.5 + 23.8j
8
9 # Sequence: list , tuple , range
10 a = [1, 2, 'Hello', True]
11 b = (1, 2, 'Hello', True)
12 c = range(1, 10, 2)
```

# Basic data types

## Languages (Python)

```
1 # Mapping: dict
2 a = {'x': 2, 5: 'hello', True: 42}
3 # Keys must be immutable!
4
5 # Sets: set, frozenset
6 a = set([1, 1, 1, 4]) # a = {1, 4}
7
8 # Booleans: bool
9 a = True
10 b = False
11
12 # Binary Types: bytes, bytearray, memoryview
13 a = b'Hello'
```

# More advanced data types (1)

## Languages (Python)

```
1 # Namedtuples
2 from collections import namedtuple
3
4 Person = namedtuple('Person', ['name', 'trademark'])
5 testo = Person(
6     name='Lukasz Stanislawowski',
7     trademark='Rolex '
8 )
9 testo.trademark = 'poramancza '
10 # Traceback (most recent call last):
11 #   File "<stdin>", line 1, in <module>
12 # AttributeError: can't set attribute
```

# More advanced data types (2)

## Languages (Python)

```
1 # Custom classes
2 # not: Person(object) – Python 2 syntax
3 class Person:
4     def __init__(self, name, trademark):
5         self._name = name
6         self._trademark = trademark
7
8 testo = Person(
9     name='Lukasz Stanislawowski',
10    trademark='Rolex'
11 )
12 testo._trademark = 'pomarancza'
13 # Perfectly fine for interpreter, but avoid that,
14    please...
```

# More advanced data types (3)

## Languages (Python)

```
1 # Enums
2 from enum import Enum
3
4 class Trademarks(Enum):
5     POMARANCZA = 0
6     ROLEX = 1
7
8 print(Trademark.POMARANCZA)
9 # Trademark.POMARANCZA
10
11 print(repr(Trademark.POMARANCZA))
12 # <Trademark.POMARANCZA: 0>
```

# Python manifest

## Languages (Python)

```
1 >>> import this
2 The Zen of Python, by Tim Peters
3
4 Beautiful is better than ugly.
5 Explicit is better than implicit.
6 Simple is better than complex.
7 Complex is better than complicated.
8 Flat is better than nested.
9 Sparse is better than dense.
10 Readability counts.
11 Special cases aren't special enough to break the rules.
12 Although practicality beats purity.
13 Errors should never pass silently.
14 Unless explicitly silenced.
15 In the face of ambiguity, refuse the temptation to guess.
16 There should be one— and preferably only one —obvious way to do it.
17 Although that way may not be obvious at first unless you're Dutch.
18 Now is better than never.
19 Although never is often better than *right* now.
20 If the implementation is hard to explain, it's a bad idea.
21 If the implementation is easy to explain, it may be a good idea.
22 Namespaces are one honking great idea — let's do more of those!
```

# New features in Python

## Languages (Python)

- ▶ writing code in Python is all about readability
- ▶ it's worth to follow the newest Python releases
- ▶ new features to improve code readability and maintainability
- ▶ currently: Python 3.9
- ▶ let's have a look into some of them...



# Type hints (1)

## Languages (Python)

Not the same as static typing! Only hints for linter tools

```
1 a: int = 7
2 a: int = 'Hi' # Works, but good IDE will complain
3
4 a: str = 'Hello'
5 a: bool = True
6 a: dict = {'x': 1, 'y': 0}
7 a: MyClass = MyClass(x=0, y=42)
8
```

## Type hints (2)

### Languages (Python)

Type hints like: "dict", "set", "list" do not carry information about the inner types. However there exists "typing" module.

```
1 from typing import Dict, List, Set
2
3 a: List[int] = [1, 2, 3]
4 a: Set[str] = {'Hello', 'Hi'}
5 a: Dict[str, int] = {'x': 0, 'y': 1}
6
```

# Type hints (3)

## Languages (Python)

Type hints can be also applied to functions and methods

```
1 from typing import List
2
3 def contains(x: List[int], val: int) -> bool:
4     return val in x
5
6 class Person:
7     def __init__(self, name: str, trademark:
8         Trademark) -> None:
9         self._name = name
10        self._trademark = trademark
```

# Type hints (4)

## Languages (Python)

### Worth to mention: Data classes

```
1 from dataclass import dataclass, field
2 from typing import List
3
4 @dataclass
5 class Person:
6     name: str
7     age: int
8     trademark: Trademark
9     videos: List[str] = field(
10         init=False,
11         repr=False,
12         default_factory=list
13     ) # NOT: videos: List[str] = []
```

# Type hints (5)

## Languages (Python)

Worth to mention:

```
1 from typing import Optional, Sequence, Tuple, Union
2
3 # Either str or int
4 def foo(x: Union[str, int]) -> None: ...
5
6 # 3-tuple of str, str and int
7 def foo(x: Tuple[str, str, int]) -> None: ...
8
9 # Any kind of int iterable
10 def foo(x: Sequence[int]) -> None: ...
11
12 # Optional value (not the same as default!)
13 # Here: There could be a str but None is possible
14 def foo(x: Optional[str] = None) -> None: ...
15
16 # vs standard default value
17 def foo(x: str = 'Hi') -> None: ...
```

# Type hints (6)

## Languages (Python)

Worth to mention:

```
1 from typing import Callable, List, TypeVar
2
3 T = TypeVar('T')
4
5 def my_map(
6     vals: List[T],
7     fn: Callable[[T], T]
8 ) -> List[T]:
9     return [fn(x) for x in vals]
10
11 def double(x: int) -> int:
12     return x * 2
13
14 def custom_len(x: str) -> int:
15     return len(x)
16
17
18 my_map(vals=[1, 2], fn=double) # OK
19 my_map(vals=[1, 'Hi'], fn=double) # WRONG, why?
20 my_map(vals=['A', 'B'], fn=double) # WRONG, why?
21 my_map(vals=['A', 'B'], fn=custom_len) # OK
22 my_map(vals=[1, 2], fn=custom_len) # WRONG, why?
```

# Other features

## Languages (Python)

- ▶ f-strings,
- ▶ breakpoint(),
- ▶ positional only arguments,
- ▶ literal types,
- ▶ typed dicts,
- ▶ final objects

# How to parallelize?

## Languages (Python)

- ▶ native:
  - ▶ threads,
  - ▶ processes,
- ▶ 3rd party:
  - ▶ celery,
  - ▶ pyfunctional,
  - ▶ ray,
  - ▶ dask,
  - ▶ ...



# Native parallelization (1)

## Languages (Python)

### Processes:

```
1 # ...
2 import multiprocessing as mp
3
4 def make_work(x: int, y: List[int]) -> int: ...
5
6 def worker_fn(args: tuple) -> int:
7     return make_work(*args)
8
9 def run():
10     args: List[Tuple[int, List[int]]] = [
11         (1, [2, 3, 4]),
12         ...
13     ]
14
15     with mp.Pool(processes=mp.cpu_count()) as pool:
16         results = pool.map(worker_fn, args)
17
```

# Native parallelization (2)

## Languages (Python)

### Threads:

```
1 # ...
2 import multiprocessing as mp
3 from multiprocessing.pool import ThreadPool
4
5 # same code as previously
6
7 def run():
8     # same code as previously
9
10     with ThreadPool(processes=mp.cpu_count()) as
11         pool:
12             results = pool.map(worker_fn, args)
```

# 3rd party parallelization (1)

Languages (Python)

Celery:

- ▶ distributed task queue,
- ▶ uses RabbitMQ underneath,
- ▶ perfectly known to all students – used during labs

```
1 from celery import Celery
2
3 app = Celery('myapp', broker='amqp://')
4
5 @app.task
6 def add(x, y):
7     return x + y
8
9
10 if __name__ == '__main__':
11     app.start()
```

## 3rd party parallelization (2)

Languages (Python)

PyFunctional:

- ▶ provides functional API over streams (collections),
- ▶ can be used in sequential or parallel manner,
- ▶ parallelization only on single host,
- ▶ can directly read / write to from DBs, CSVs etc.,
- ▶ lazy execution,
- ▶ for parallel processing import "pseq"
- ▶ <https://github.com/EntilZha/PyFunctional>

```
1 from functional import seq
2
3 (seq(1, 2, 3, 4)
4  .map(lambda x: x * 2)
5  .filter(lambda x: x > 4)
6  .reduce(lambda x, y: x + y))
```

# 3rd party parallelization (3)

## Languages (Python)

### Ray:

- ▶ framework for distributed processing,
- ▶ uses Redis underneath,
- ▶ deploy on AWS, GCE, K8s

```
1 import ray
2 ray.init()
3
4 @ray.remote
5 def f(x):
6     return x * x
7
8 futures = [f.remote(i) for i in range(4)]
9 print(ray.get(futures))
```

## 3rd party parallelization (4)

Languages (Python)

Dask:

- ▶ framework for distributed processing,
- ▶ integrated with Numpy, Pandas, Scikit-learn, XGBoost,
- ▶ deploy on K8s, Hadoop/YARN, SSH,

```
1 # Arrays implement the Numpy API
2 import dask.array as da
3 x = da.random.random(size=(...), chunks=(...))
4 x + x.T - x.mean(axis=0)
5
6 # Dataframes implement the Pandas API
7 import dask.dataframe as dd
8 df = dd.read_csv('s3://file.csv')
9 df.groupby(df.account_id).balance.sum()
10
11 # Dask-ML implements the Scikit-Learn API
12 from dask_ml.linear_model import LogisticRegression
13 lr = LogisticRegression()
14 lr.fit(train, test)
```

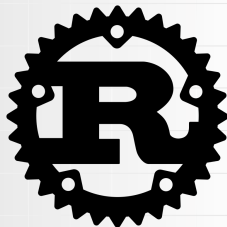


# Rust

# Introduction (1)

## Languages (Rust)

- ▶ fairly new, but already good developed
- ▶ created by Mozilla
- ▶ used by Microsoft (2019)
- ▶ compiled language
- ▶ fast (LLVM backend – same as C++)
- ▶ excellent toolkit (cargo)
- ▶ compiler **guarantees memory safety**
- ▶ borrow checker (no GC)

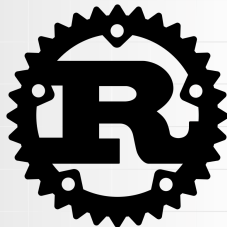




# Introduction (2)

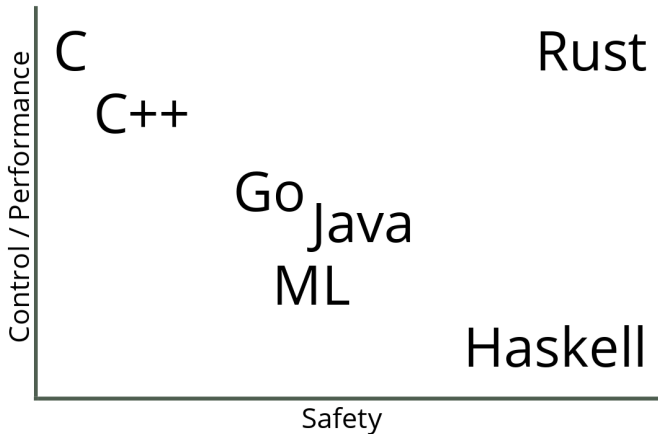
## Languages (Rust)

- ▶ **static typing,**
- ▶ systems programming language,
- ▶ access to low level OS resources - like in C (pipes, sockets, message queues),
- ▶ easy FFI,
- ▶ suitable for embedded devices,
- ▶ WASM – compile with JavaScript,
- ▶ compile to native GPU kernels\*,
- ▶ **steep learning curve,**



# Comparison

## Languages (Rust)



# Cargo (1)

## Languages (Rust)

- ▶ dependency management,
- ▶ code linter (cargo clippy),
- ▶ documentation (cargo doc),
- ▶ code formatting (cargo fmt),
- ▶ tests execution (cargo test),
- ▶ toolset upgrading (rustup)

# Cargo (2)

## Languages (Rust)

```
1 [package]
2 name = "docker-cmd"
3 version = "0.2.0"
4 authors = ["pbielak"]
5 edition = "2018"
6
7 [dependencies]
8 console = "0.7.5"
9 dialoguer = "0.3.0"
10 derive_more = "0.13.0"
11 nix = "0.13.0"
12 structopt = "0.2"
13 tabwriter = "1.1.0"
```

# Borrow checker

## Languages (Rust)

- ▶ no dangling pointers,
- ▶ no double free (security risk!),
- ▶ no data races (easy concurrency),
- ▶ if it compiles then it's valid code (in 99% of all cases),

# Language basics (1)

## Languages (Rust)

```
1 // Main function definition
2 fn main() {
3     println!("Hello , world!");
4 }
5
6 // Variables
7 let a = true;
8 let b: bool = true;
9
10 a = false; // Error!
11
12 // Mutable variables
13 let mut a = true;
14 a = false; // OK!
15
16 // Variable bindings
17 let (x, y) = (1, 2);
```

# Language basics (2)

## Languages (Rust)

```
1 // Function definition
2 // No return statement needed!
3 // (If it is the last one*)
4 fn double(x: i32) -> i32 {
5     2 * x
6 }
7
8 // Functions can be assigned to variables
9 let f = double;
10 println!("2 * {} = {}", 1, f(1));
11
12 // If statements assignment
13 let is_below_eighteen = if age < 18 { true } else {
    false };
```

# Language basics (3)

## Languages (Rust)

```
1 // Pattern matching
2 let f = File::open("hello.txt");
3
4 let f = match f {
5     Ok(file) => file ,
6     Err(error) => panic!("Oh noes: {:?}", error),
7 };
```



# Language basics (4)

## Languages (Rust)

```
1 // No classes – structs + traits
2 #[derive(Debug)]
3 struct Person <'a> {
4     name: &'a str,
5     age: u8,
6 }
7
8 let p = Person {
9     name: "Lukasz Stanislawowski",
10    age: 32
11 };
```

# Language basics (5)

## Languages (Rust)

```
1 // No classes — structs + traits
2 #[derive(Debug)]
3 struct Person<'a> {
4     name: &'a str,
5     age: u8,
6 }
7
8 impl<'a> Person<'a> {
9     fn new(name: &'a str, age: u8) -> Person<'a> {
10         Person { name, age }
11     }
12
13     fn is_online(&self) -> bool {
14         false
15     }
16 }
17
18 let p = Person::new("Lukasz Stanislawowski", 32);
```

# Language basics (6)

## Languages (Rust)

```
1 // ... same code ...
2
3 trait OnlineChecker {
4     fn is_online(&self) -> bool;
5 }
6
7 impl<'a> Person<'a> {
8     fn new(name: &'a str, age: u8) -> Person<'a> {
9         Person { name, age }
10    }
11 }
12
13 impl<'a> OnlineChecker for Person<'a> {
14     fn is_online(&self) -> bool { false }
15 }
16
17 impl OnlineChecker for i32 {
18     fn is_online(&self) -> bool { true }
19 }
```

# Borrow checker rules

## Languages (Rust)

First, any borrow must last for a scope no greater than that of the owner. Second, you may have one or the other of these two kinds of borrows, but not both at the same time:

- ▶ one or more references (&T) to a resource,
- ▶ exactly one mutable reference (&mut T).

Source:

<https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html>

# Borrow checker example (1.1)

## Languages (Rust)

```
1 let mut v = vec![1, 2, 3];  
2  
3 for i in &v {  
4     println!("{}", i);  
5     v.push(34);  
6 }
```

# Borrow checker example (1.2)

## Languages (Rust)

```
1 // error: cannot borrow v as mutable because it is
    // also borrowed as immutable
2 //     v.push(34);
3 //     ^
4 // note: previous borrow of v occurs here; the
    // immutable borrow prevents
5 // subsequent moves or mutable borrows of v until
    // the borrow ends
6 // for i in &v {
7 //     ^
8 // note: previous borrow ends here
9 // for i in &v {
10 //     println!("{}", i);
11 //     v.push(34);
12 // }
13 // ^
```

# Borrow checker example (2.1)

## Languages (Rust)

```
1 let mut x = 5;  
2 let y = &mut x;  
3  
4 *y += 1;  
5  
6 println!("{}", x);
```

# Borrow checker example (2.2)

## Languages (Rust)

```

1 // error: cannot borrow x as immutable because it
  // is also borrowed as mutable
2 //     println!("{}", x);
3 //           ^
4
5
6 let mut x = 5;
7
8 let y = &mut x;    // → &mut borrow of x starts
  here
9
10      // |
  *y += 1;          // |
11      // |
  println!("{}", x); // → - try to borrow x here
12
13                  // → &mut borrow of x ends here
  
```



# How to parallelize?

## Languages (Rust)

- ▶ mostly threads with channels are being used,
- ▶ 3rd party tools: actix, rayon, tokio
- ▶ `async*`,

# Threads (1)

## Languages (Rust)

```
1 use std::thread;  
2  
3 fn main() {  
4     thread::spawn(|| {  
5         println!("Hello from a thread!");  
6     });  
7 }
```

# Threads (2)

## Languages (Rust)

```
1 use std::thread;  
2  
3 fn main() {  
4     let x = 1;  
5     thread::spawn(move || {  
6         println!("x is {}", x);  
7     });  
8 }
```

# Threads (3.1)

## Languages (Rust)

```
1 // NOT WORKING!
2 use std::thread;
3 use std::time::Duration;
4
5 fn main() {
6     let mut data = vec![1, 2, 3];
7
8     for i in 0..3 {
9         thread::spawn(move || {
10             data[0] += i;
11         });
12     }
13
14     thread::sleep(Duration::from_millis(50));
15 }
```

# Threads (3.2)

## Languages (Rust)

```
1 // OK!
2 use std::sync::{ Arc , Mutex };
3 use std::thread;
4 use std::time:: Duration ;
5
6 fn main() {
7     let data = Arc::new(Mutex::new(vec![1, 2, 3]));
8
9     for i in 0..3 {
10         let data = data.clone();
11         thread::spawn(move || {
12             let mut data = data.lock().unwrap();
13             data[0] += i;
14         });
15     }
16
17     thread::sleep(Duration::from_millis(50));
18 }
```

# Threads (4)

## Languages (Rust)

```
1 use std::thread;
2 use std::sync::mpsc;
3
4 fn main() {
5     let (tx, rx) = mpsc::channel();
6
7     for i in 0..10 {
8         let tx = tx.clone();
9
10        thread::spawn(move || {
11            let answer = i * i;
12
13            tx.send(answer).unwrap();
14        });
15    }
16
17    for _ in 0..10 {
18        println!("{}", rx.recv().unwrap());
19    }
20 }
```

# Erlang

# Introduction

## Languages (Erlang)

- ▶ concurrent,
- ▶ functional programming language,
- ▶ garbage collection,
- ▶ **actor programming**,
- ▶ most popular software: RabbitMQ, WhatsApp





# Erlang runtime

## Languages (Erlang)

- ▶ distributed,
- ▶ fault tolerant,
- ▶ soft real-time
- ▶ HA, non-stop applications
- ▶ Hot swapping (change code without system stop)

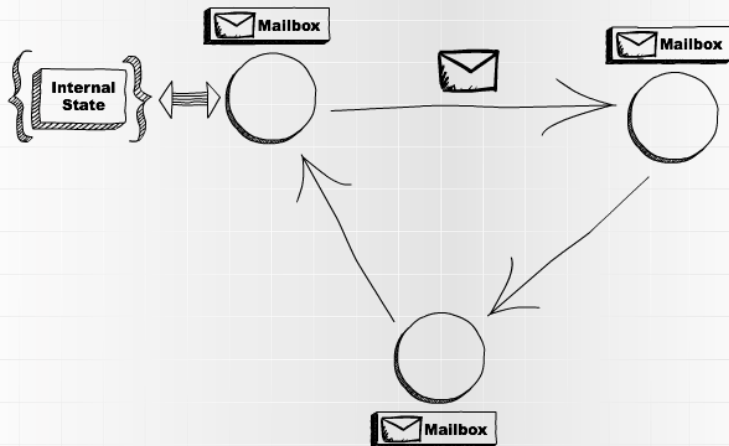
# Actor programming (1)

## Languages (Erlang)

- ▶ actor programming = real OOP,
- ▶ actor = primitive unit of computation,
- ▶ actor receives a message and do some kind of computation based on it,
- ▶ actors are completely separated (no shared memory etc.),

# Actor programming (2)

Languages (Erlang)



# Actor programming (3)

## Languages (Erlang)

When an actor receives a message, it can do one of these 3 things:

- ▶ create more actors,
- ▶ send messages to other actors,
- ▶ designate what to do with the next message.

# Actor programming (4)

## Languages (Erlang)

- ▶ you shouldn't care about fault tolerance,
- ▶ create a supervisor that can retrieve other actors when they fail,
- ▶ distribution is easy (just serialize messages)

# Scala

# Scala

## Languages

- ▶ JVM
- ▶ breaking changes
- ▶ can use all from JVM world
- ▶ multi-paradigm



# Scala

## Languages

- ▶ immutable first
- ▶ val vs var
- ▶ immutable case classes
- ▶ lazy
- ▶ higher order functions



# Scala

## Languages

```
1 var a = 0  
2 a = a + 1  
3 val b = 0  
4 b = b + 1 // NOPE!
```

# Scala

## Languages

```
1 case class Person(name: String)
2 val person = Person("Lukasz Stanislawowski")
3 person.name="Testo" //NOPE!
```

# Scala

## Languages

```
1 case class Person(name: String)
2 var person = Person("Lukasz Stanislawowski")
3 person.name="Testo" //NOPE!
```

# Scala

## Languages

```
1 case class Person(name: String)
2 var person = Person("Lukasz Stanislawowski")
3 var newPerson = person.copy(name = "Testo") //WEEEE!
```

# Scala

## Languages

```
1 def someDefinition(a: String): String = a
2
3 val someLambda = (a: String) => a
4
5 def higherOrderFunction(f: String=>String): String=>
  String = f
```

# Scala

## Languages

```
1 val a = print("show must go on")
```



# Scala

## Languages

```
1 lazy val a = print("show must go on, not")
```

# Scala

## Languages

```
1 val a = 10  
2 val b = 11  
3  
4 val c: String = s"$a + $b = ${a+b}"
```



# Scala

## Languages

- ▶ generics
- ▶ implicits
- ▶ type classes

# Scala

## Languages

```
1 case class Container[T](data: T)
2   Container("string")
3   Container(1)
4   Container(Container("wow"))
5   //Container[Container[String]]
```

# Scala

## Languages

```
1 case class Context(data: String)
2 val context = Context("data")
3
4 def functionThatRequiresContext(data: String , ctx:
   Context): String = data
5
6 functionThatRequiresContext("notNice", context)
```

# Scala

## Languages

```
1 case class Context(data: String)
2 implicit val context = Context("data")
3
4 def functionThatRequiresContext(data: String)(
5     implicit ctx: Context): String = data
6 functionThatRequiresContext("nice !!!")
```

# Scala

## Languages

```
1 val s: String = "fun"  
2  
3 s.makeFun //NOPE!
```

# Scala

## Languages

```
1 implicit class Funner(s: String){  
2     def makeFun: String = "funHasBeenMade"  
3 }  
4 val s: String = "fun"  
5  
6 s.makeFun //WOOOOOW!
```

# Scala

## Languages

```
1 object Show {  
2     trait Show[A] {  
3         def show(a: A): String  
4     }  
5  
6     def show[A](a: A)(implicit sh: Show[A]) = sh.  
    show(a)  
7  
8     implicit val intCanShow: Show[Int] =  
9         new Show[Int] {  
10             def show(int: Int): String = s"int $int"  
11         }  
12 }
```

# Scala

## Languages

```
1 import Show._  
2 print (show (1))
```



# Scala

## Languages

```
1 import Show._  
2 print (show (1) (intCanShow))
```

# Scala

## Languages

```
1 import Show._  
2 case class Person(name: String)  
3  
4 implicit val personCanBeShown: Show[Person] =  
5     new Show[Int] {  
6         def show(p: Person): String = p.name  
7     }  
8  
9 print(show(Person("Lukasz S.")))
```

### Parallel collections

- ▶ Separate library since 2.13
- ▶ Better to use Vector
- ▶ IT IS NOT ALWAYS WORTH TO PARALLELIZE

# Scala

## Languages

```
1 import scala.collection.parallel.  
   CollectionConverters._  
2 val lastNames = List("Smith", "Jones", "Frankenstein",  
   "Bach", "Jackson", "Rodin").par  
3 lastNames.map(_.toUpperCase)
```

# Scala

## Languages

```
1 import scala.collection.parallel.  
   CollectionConverters._  
2 val parArray = (1 to 10000).toArray.par  
3 parArray.fold(0)(_ + _)
```

# Scala

## Languages

```
1 import scala.collection.parallel.  
   CollectionConverters._  
2 val lastNames = List("Smith", "Jones", "Frankenstein",  
   "Bach", "Jackson", "Rodin").par  
3 lastNames.filter(_.head >= 'J')
```

### Multithreading

- ▶ Runnable
- ▶ Futures
- ▶ Promises
- ▶ ExecutionContext
- ▶ context switching has a cost

- ▶ Fork join pool - asynchronous operations
- ▶ Thread pool - blocking operations



# Scala

## Languages

```
1 object RunnableUsingGlobalExecutionContext extends
  App {
2   val ctx = scala.concurrent.ExecutionContext.
    global
3   ctx.execute(new Runnable {
4     def run() = print("Hey, I am on a separate
    thread!.")
5   })
6   Thread.sleep(1000)
7 }
```

# Scala

## Languages

```
1 import scala.concurrent.{ Future , ExecutionContext}  
2 object FutureUsingGlobalExecutionContext extends  
   App {  
3     implicit val ctx = ExecutionContext.global  
4     Future {  
5         print("Hey, I am on a separate thread!.")  
6     }  
7     Thread.sleep(1000)  
8 }
```

# Scala

## Languages

- ▶ failure
- ▶ map
- ▶ flatMap
- ▶ sequential

# Scala

## Languages

- ▶ Akka
- ▶ Play
- ▶ Akka-Stream
- ▶ Akka-Typed



# C++

# C++

## Languages

- ▶ high performance
- ▶ optimizations
- ▶ low-level
- ▶ full control

95/121

# Optimizations

## Languages (C++)

### Common Sub-expression removal:

```
1 a = b * c + d;  
2 e = b * c + 3;
```

```
1 tmp = b * c  
2 a = tmp + d;  
3 e = tmp + 3;
```



# Optimizations

## Languages (C++)

Redundant code removal:

```
1 int main() {  
2     int v[2];  
3     for (int i = 0; i < 2; i++)  
4         v[i] = i * i;  
5     return 0;  
6 }
```

# Optimizations

## Languages (C++)

### (Auto)-vectorization

- ▶ SIMD instructions for the execution of a loop
- ▶ depends on the available SIMD instructions
- ▶ 2x for Single Precision ops
- ▶ can introduce different rounding

# Optimizations

## Languages (C++)

Loop-invariant code motion:

```
1 for (int i=0; i<n; i++) {  
2     x = y * z;  
3     a[i] = 2 * i + x * x;  
4 }
```

```
1 x = y * z;  
2 tmp = x * x;  
3 for (int i=0; i<n; i++) {  
4     a[i] = 2 * i + tmp;  
5 }
```

# Optimizations

## Languages (C++)

Induction variable:

```
1 x = y * z;  
2 tmp = x * x;  
3 for (int i=0; i<n; i++) {  
4     a[i] = 2 * i + tmp;  
5 }
```

```
1 x = y * z;  
2 tmp = x * x;  
3 for (int i=0; i<n; i++, tmp+=2) {  
4     a[i] = tmp;  
5 }
```

# Optimizations

## Languages (C++)

### Loop unrolling:

```
1 for (int i=0; i<n; i++) {  
2     a[i] += 2.2 * b[i];  
3 }
```

```
1 for (int i=0; i<n; i+=4) {  
2     a[i] += 2.2 * b[i];  
3     a[i+1] += 2.2 * b[i+1];  
4     a[i+2] += 2.2 * b[i+2];  
5     a[i+3] += 2.2 * b[i+3];  
6 }
```

# Optimizations

## Languages (C++)

### Loop unrolling:

- ▶ better pipelining
- ▶ better vectorization
- ▶ increase binary size

# Optimizations

## Languages (C++)

### Function inlining:

- ▶ function call puts previous results on the stack
- ▶ function calls stops further optimizations (vectorization etc.)
- ▶ increase binary size

# Optimizations

## Languages (C++)

- ▶ check compiler reports
- ▶ compiler will not always work



# Manual optimizations

Languages (C++)

Loop Interchange:

Wrong:

```
1 for (int j=0; j<columns; j++) {  
2     for (int i=0; i<rows; i++) {  
3         mymatrix[i][j] += increment;  
4     }  
5 }
```

Correct:

```
1 for (int i=0; i<rows; i++) {  
2     for (int j=0; j<columns; j++) {  
3         mymatrix[i][j] += increment;  
4     }  
5 }
```

# Manual optimizations

## Languages (C++)

### Loop Fusion:

```
1  for (int j=0; j<columns; j++) {  
2      for (int i=0; i<rows; i++) {  
3          mymatrix[i][j] = othermatrix[i][j]*2;  
4      }  
5  }  
6  for (int j=0; j<columns; j++) {  
7      for (int i=0; i<rows; i++) {  
8          mymatrix[i][j] += 1;  
9      }  
10 }
```

```
1  for (int j=0; j<columns; j++) {  
2      for (int i=0; i<rows; i++) {  
3          mymatrix[i][j] = othermatrix[i][j]*2;  
4          mymatrix[i][j] += 1;  
5      }  
6  }
```

# OpenMP

## Languages (C++)

- ▶ multithreading
- ▶ explicit
- ▶ API or pragmas
- ▶ task or data parallel
- ▶ SIMD

# OpenMP

## Languages (C++)

```
1 #pragma omp parallel
2 #pragma omp for
3 for (int i=0; i<10; i++) {
4     // do something with i
5 }
```

# OpenMP

## Languages (C++)

```
1 #pragma omp parallel num_threads(3)
2 #pragma omp for
3 for (int i=0; i<10; i++) {
4     // do something with i
5 }
```

# OpenMP

## Languages (C++)

```
1 #pragma omp parallel
2 int id = omp_get_thread_num();
3 int total = omp_get_num_threads();
4 #pragma omp for
5 for (int i=0; i<10; i++) {
6     // do something with i
7 }
```

Go

# Go

## Languages

- ▶ Google
- ▶ high performance
- ▶ easy development
- ▶ simple syntax



# Go

## Languages

### Protoactor

- ▶ blazing fast
- ▶ protobuffers
- ▶ virtual actors
- ▶ up to 10x speed of erlang
- ▶ up to 100x speed of Akka.NET
- ▶ Kotlin, C#, GO



# MPI

# MPI

## Languages

- ▶ widely used standard for message passing distributed-memory concurrent computers
- ▶ sender and receiver must specify data type
- ▶ point-to-point communication
- ▶ collective communication
- ▶ defines common data types

# MPI

## Languages

- ▶ generally we are not using all-to-all communication
- ▶ models application topology
- ▶ support for Cartesian topology
- ▶ data shifting among dimension
- ▶ collective communication among dimension

# MPI

## Languages

```
1 #include "mpi.h"
2 #include <stdio.h>
3 int main(int argc, char *argv[]){
4     MPI_Init(&argc, &argv);
5     printf("Hello, world!\n");
6     MPI_Finalize();
7     return 0;
8 }
```

# MPI

## Languages

- ▶ how many processes?
- ▶ which process am I?

# MPI

## Languages

```
1 #include "mpi.h"
2 #include <stdio.h>
3 int main( int argc, char *argv[] ){
4     int rank, size;
5     MPI_Init( &argc, &argv );
6     MPI_Comm_rank( MPI_COMM_WORLD, &rank );
7     MPI_Comm_size( MPI_COMM_WORLD, &size );
8     printf( "I am %d of %d\n", rank, size );
9     MPI_Finalize();
10    return 0;
11 }
```

# MPI

## Languages

- ▶ broadcast
- ▶ multicast
- ▶ all-to-all
- ▶ barrier
- ▶ scatter
- ▶ gather
- ▶ all gather
- ▶ reduce



# Large Scale Data Processing

## Lecture 4 – Data processing languages

dr inż. Tomasz Kajdanowicz, Roman Bartusiak, Piotr Bielak

October 26, 2020