Wrocław University of Science and Technology

Large Scale Data Processing

Lecture 4 – Data processing languages

dr inż. Tomasz Kajdanowicz, Roman Bartusiak, Piotr Bielak

December 27, 2019

Overview

References

Languages

- Python
- Rust
- Erlang
- Scala
- C++
- Go
- MPI

Overview

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

Wrocław University of Science and Technology

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!

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

Strong*: types do not change unexpectedly

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

Strong*, dynamic typing

Languages (Python)

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

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

Basic data types

Languages (Python)

```python
# Text: str
a = 'Hello'

# Numeric: int, float, complex
a = 42
b = 777.0
c = 34.5 + 23.8j

# Sequence: list, tuple, range
a = [1, 2, 'Hello', True]
b = (1, 2, 'Hello', True)
c = range(1, 10, 2)
```

Basic data types

Languages (Python)

```python
# Mapping: dict
a = {'x': 2, 5: 'hello', True: 42}
# Keys must be immutable!

# Sets: set, frozenset
a = set([1, 1, 1, 4])  # a = {1, 4}

# Booleans: bool
a = True
b = False

# Binary Types: bytes, bytearray, memoryview
a = b'Hello'
```

More advanced data types (1)

Languages (Python)

```python
# Namedtuples
from collections import namedtuple

Person = namedtuple('Person', ['name', 'trademark'])
testo = Person(
    name='Lukasz Stanislawowski',
    trademark='Rolex'
)
testo.trademark = 'poramancza'
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: can't set attribute
```

More advanced data types (2)

Languages (Python)

```python
# Custom classes
# not: Person(object) — Python 2 syntax
class Person:
    def __init__(self, name, trademark):
        self._name = name
        self._trademark = trademark

testo = Person(
    name='Lukasz Stanislawowski',
    trademark='Rolex'
)
testo._trademark = 'pomarancza'
# Perfectly fine for interpreter, but avoid that,
    please...
```

More advanced data types (3)

Languages (Python)

```python
# Enums
from enum import Enum

class Trademarks(Enum):
    POMARANCZA = 0
    ROLEX = 1

print(Trademark.POMARANCZA)
# Trademark.POMARANCZA

print(repr(Trademark.POMARANCZA))
# <Trademark.POMARANCZA: 0>
```

Python manifest

Languages (Python)

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

New features in 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 3.8
- 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

```
a: int = 7
a: int = 'Hi'  # Works, but good IDE will complain

a: str = 'Hello'
a: bool = True
a: dict = {'x': 1, 'y': 0}
a: MyClass = MyClass(x=0, y=42)

```


Wrocław University of Science and Technology

Type hints (2)

Languages (Python)

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

```python
from typing import Dict, List, Set

a: List[int] = [1, 2, 3]
a: Set[str] = {'Hello', 'Hi'}
a: Dict[str, int] = {'x': 0, 'y': 1}
```

Type hints (3)

Languages (Python)

Type hints can be also applied to functions and methods

```python
from typing import List

def contains(x: List[int], val: int) -> bool:
    return val in x

class Person:
    def __init__(self, name: str, trademark: Trademark) -> None:
        self._name = name
        self._trademark = trademark

```

Type hints (4)

Languages (Python)

Worth to mention: Data classes

```python
from dataclass import dataclass, field
from typing import List

@dataclass
class Person:
    name: str
    age: int
    trademark: Trademark
    videos: List[str] = field(
        init=False,
        repr=False,
        default_factory=list
    )  # NOT: videos: List[str] = []
```

Type hints (5)

Languages (Python)

Worth to mention:

```python
from typing import Optional, Sequence, Tuple, Union

# Either str or int
def foo(x: Union[str, int]) -> None: ...

# 3-tuple of str, str and int
def foo(x: Tuple[str, str, int]) -> None: ...

# Any kind of int iterable
def foo(x: Sequence[int]) -> None: ...

# Optional value (not the same as default!)
# Here: There could be a str but None is possible
def foo(x: Optional[str] = None) -> None: ...

# vs standard default value
def foo(x: str = 'Hi') -> None: ...
```

Type hints (6)

Languages (Python)

Worth to mention:

```
from typing import Callable, List, TypeVar

T = TypeVar('T')

def my_map(
    vals: List[T],
    fn: Callable[[T,], T]
) -> List[T]:
    return [fn(x) for x in vals]

def double(x: int) -> int:
    return x * 2

def custom_len(x: str) -> int:
    return len(x)


my_map(vals=[1, 2], fn=double)  # OK
my_map(vals=[1, 'Hi'], fn=double)  # WRONG, why?
my_map(vals=['A', 'B'], fn=double)  # WRONG, why?
my_map(vals=['A', 'B'], fn=custom_len)  # OK
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:

```python
# ...
import multiprocessing as mp

def make_work(x: int, y: List[int]) -> int: ...

def worker_fn(args: tuple) -> int:
    return make_work(*args)

def run():
    args: List[Tuple[int, List[int]]] = [
        (1, [2, 3, 4]),
        ...
    ]

    with mp.Pool(processes=mp.cpu_count()) as pool:
        results = pool.map(worker_fn, args)
```

Native parallelization (2)

Languages (Python)

Threads:

```
# ...
import multiprocessing as mp
from multiprocessing.pool import ThreadPool

# same code as previously

def run():
    # same code as previously

    with Threadpool(processes=mp.cpu_count()) as pool:
        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

```python
from celery import Celery

app = Celery('myapp', broker='amqp://')

@app.task
def add(x, y):
    return x + y


if __name__ == '__main__':
    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

```python
from functional import seq

(seq(1, 2, 3, 4)
    .map(lambda x: x * 2)
    .filter(lambda x: x > 4)
    .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

```python
import ray
ray.init()

@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(4)]
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,

```python
# Arrays implement the Numpy API
import dask.array as da
x = da.random.random(size=(...), chunks=(...))
x + x.T - x.mean(axis=0)

# Dataframes implement the Pandas API
import dask.dataframe as dd
df = dd.read_csv('s3://file.csv')
df.groupby(df.account_id).balance.sum()

# Dask-ML implements the Scikit-Learn API
from dask_ml.linear_model import LogisticRegression
lr = LogisticRegression()
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)

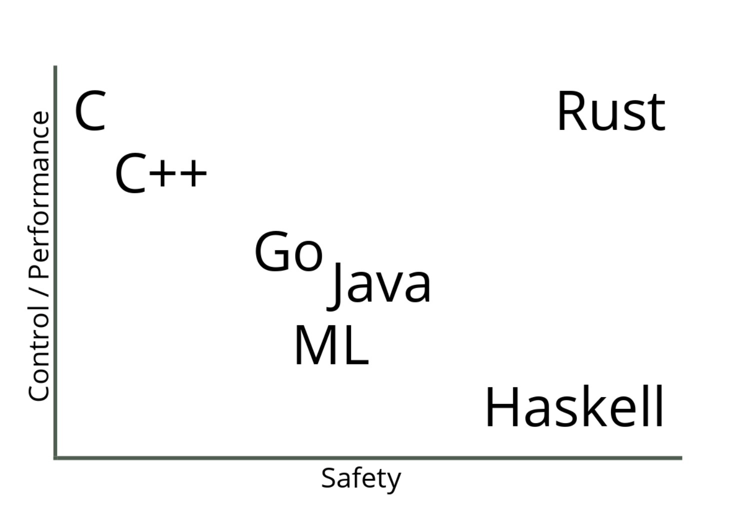
Introduction (2)

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)

```
[package]
name = "docker-cmd"
version = "0.2.0"
authors = ["pbielak"]
edition = "2018"

[dependencies]
console = "0.7.5"
dialoguer = "0.3.0"
derive_more = "0.13.0"
nix = "0.13.0"
structopt = "0.2"
tabwriter = "1.1.0"
```

Borrow checker

- 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)

```rust
// Main function definition
fn main() {
    println!("Hello, world!");
}

// Variables
let a = true;
let b: bool = true;

a = false; // Error!

// Mutable variables
let mut a = true;
a = false; // OK!

// Variable bindings
let (x, y) = (1, 2);
```

Language basics (2)

Languages (Rust)

```rust
// Function definition
// No return statement needed!
// (If it is the last one*)
fn double(x: i32) -> i32 {
    2 * x
}

// Functions can be assigned to variables
let f = double;
println!("2 * {} = {}", 1, f(1));

// If statements assignment
let is_below_eighteen = if age < 18 { true } else {
    false };
```

Language basics (3)

Languages (Rust)

```rust
// Pattern matching
let f = File::open("hello.txt");

let f = match f {
    Ok(file) => file,
    Err(error) => panic!("Oh noes: {:?}", error),
};
```

Language basics (4)

Languages (Rust)

```rust
// No classes — structs + traits
#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
    age: u8,
}

let p = Person {
    name: "Lukasz Stanislawowski",
    age: 32
};
```

Language basics (5)

Languages (Rust)

```rust
// No classes — structs + traits
#[derive(Debug)]
struct Person<'a> {
    name: &'a str,
    age: u8,
}

impl<'a> Person<'a> {
    fn new(name: &'a str, age: u8) -> Person<'a> {
        Person { name, age }
    }

    fn is_online(&self) -> bool {
        false
    }
}

let p = Person::new("Lukasz Stanislawowski", 32);
```

Language basics (6)

Languages (Rust)

```rust
// ... same code ...

trait OnlineChecker {
    fn is_online(&self) -> bool;
}

impl<'a> Person<'a> {
    fn new(name: &'a str, age: u8) -> Person<'a> {
        Person { name, age }
    }
}

impl<'a> OnlineChecker for Person<'a> {
    fn is_online(&self) -> bool { false }
}

impl OnlineChecker for i32 {
    fn is_online(&self) -> bool { true }
}
```

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)

```rust
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
    v.push(34);
}
```

Borrow checker example (1.2)

Languages (Rust)

```
// error: cannot borrow  v  as mutable because it is
   also borrowed as immutable
//     v.push(34);
//     ^
// note: previous borrow of  v  occurs here; the
   immutable borrow prevents
// subsequent moves or mutable borrows of  v  until
   the borrow ends
// for i in &v {
//           ^
// note: previous borrow ends here
// for i in &v {
//     println!("{}", i);
//     v.push(34);
// }
// ^
```

Borrow checker example (2.1)

Languages (Rust)

```
let mut x = 5;
let y = &mut x;

*y += 1;

println!("{}", x);
```

Borrow checker example (2.2)


```
// error: cannot borrow  x  as immutable because it is also borrowed as mutable
//     println!("{}", x);
//                    ^


let mut x = 5;

let y = &mut x;     // --+ &mut borrow of x starts here
                    //   |
*y += 1;            //   |
                    //   |
println!("{}", x);  // --+ - try to borrow x here
                    // --+ &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)

```rust
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("Hello from a thread!");
    });
}
```

Threads (2)

Languages (Rust)

```rust
use std::thread;

fn main() {
    let x = 1;
    thread::spawn(move || {
        println!("x is {}", x);
    });
}
```

Threads (3.1)

Languages (Rust)

```rust
// NOT WORKING!
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = vec![1, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[0] += i;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

Threads (3.2)

Languages (Rust)

```rust
// OK!
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[0] += i;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

Threads (4)

Languages (Rust)

```rust
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = i * i;

            tx.send(answer).unwrap();
        });
    }

    for _ in 0..10 {
        println!("{}", rx.recv().unwrap());
    }
}
```

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)

Wroclaw University of Science and Technology

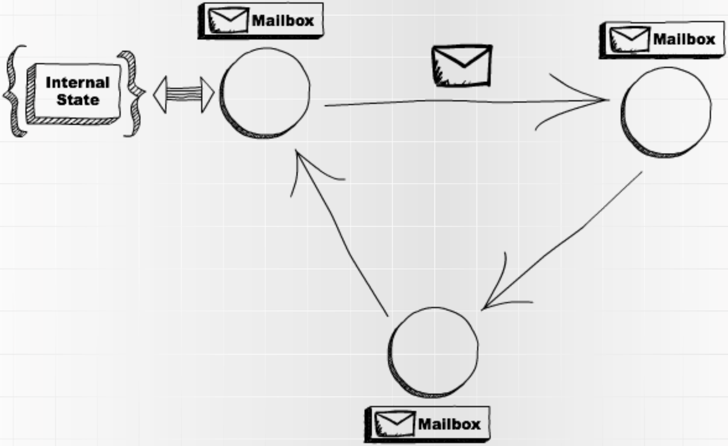
Actor programming (1)

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

```scala
var a = 0
a = a + 1
val b = 0
b = b + 1 // NOPE!
```

Scala

Languages

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

Scala

Languages

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

Scala

Languages

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

Wroclaw University of Science and Technology

Wroclaw University of Science and Technology

Scala

Languages

```scala
def someDefinition(a:String): String = a

val someLambda = (a: String) => a

def higherOrderFunction(f: String=>String): String=>
    String = f
```

Scala

Languages

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

Scala

Languages

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

Scala

Languages

```scala
val a = 10
val b = 11

val c: String = s"$a + $b = ${a+b}"
```

Scala

Languages

- generics
- implicits
- type classes

Scala

Languages

```scala
case class Container[T](data: T)
Container("string")
Container(1)
Container(Container("wow"))
//Container[Container[String]]
```

Scala

Languages

```scala
case class Context(data: String)
val context = Context("data")

def functionThatRequiresContext(data: String, ctx:
    Context): String = data

functionThatRequiresContext("notNice", context)
```

Scala

Languages

```scala
case class Context(data: String)
implicit val context = Context("data")

def functionThatRequiresContext(data: String)(
    implicit ctx: Context): String = data

functionThatRequiresContext("nice!!!")
```

Scala

Languages

```scala
val s: String = "fun"

s.makeFun //NOPE!
```

Scala

Languages

```scala
implicit class Funner(s: String){
    def makeFun: String = "funHasBeenMade"
}
val s: String = "fun"

s.makeFun //WOOOOOW!
```

Scala

Languages

```scala
object Show {
    trait Show[A] {
      def show(a: A): String
    }

    def show[A](a: A)(implicit sh: Show[A]) = sh.
    show(a)

    implicit val intCanShow: Show[Int] =
        new Show[Int] {
          def show(int: Int): String = s"int $int"
        }
}
```

Scala

Languages

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

Scala

Languages

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

Scala

Languages

```scala
import Show._
case class Person(name: String)

implicit val personCanBeShown: Show[Person] =
    new Show[Int] {
        def show(p: Person): String = p.name
    }

print(show(Person("Lukasz S.")))
```

Scala

Languages

Parallel collections

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

Scala

Languages

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

Scala

Languages

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

Scala

Languages

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

Wroclaw University of Science and Technology

Scala

Languages

Multithreading

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

Scala

Languages

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

Scala

Languages

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

Scala

Languages

```scala
import scala.concurrent.{ Future, ExecutionContext}
object FutureeUsingGlobalExecutionContext extends
    App {
    implicit val ctx = ExecutionContext.global
    Future {
        print("Hey, I am on a separate thread!.")
    }
    Thread.sleep(1000)
}
```

Scala

Languages

- failure
- map
- flatMap
- sequential

Scala

Languages

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

Wroclaw University of Science and Technology

C++

C++

Languages

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

Optimizations

Languages (C++)

Optimization Options

```
    -faggressive-loop-optimizations -falign-functions[=n[:m:[n2[:m2]]]] -falign-jumps[=n[:m:[n2[:m2]]]] -falign-labels[=n[:m:[n2[:m2]]]] -falign-loops[=n[:m:[n2[:m2]]]] -fassociative-math
    -fauto-profile  -fauto-profile[=path] -fauto-inc-dec  -fbranch-probabilities -fbranch-target-load-optimize  -fbranch-target-load-optimize2 -fbtr-bb-exclusive  -fcaller-saves
    -fcombine-stack-adjustments  -fconserve-stack -fcompare-elim  -fcprop-registers  -fcrossjumping -fcse-follow-jumps  -fcse-skip-blocks  -fcx-fortran-rules -fcx-limited-range -fdata-sections  -fdce
    -fdelayed-branch -fdelete-null-pointer-checks  -fdevirtualize  -fdevirtualize-speculatively -fdevirtualize-at-ltrans  -fdse -fearly-inlining  -fipa-sra  -fexpensive-optimizations
    -ffat-lto-objects -ffast-math  -ffinite-math-only  -ffloat-store  -fexcess-precision=style -fforward-propagate  -ffp-contract=style  -ffunction-sections -fgcse  -fgcse-after-reload  -fgcse-las
    -fgcse-lm  -fgraphite-identity -fgcse-sm  -fhoist-adjacent-loads  -fif-conversion -fif-conversion2  -findirect-inlining -finline-functions  -finline-functions-called-once  -finline-limit=n
    -finline-small-functions  -fipa-cp  -fipa-cp-clone -fipa-bit-cp  -fipa-vrp  -fipa-pta  -fipa-profile  -fipa-pure-const -fipa-reference  -fipa-reference-addressable -fipa-stack-alignment
    -fipa-icf  -fira-algorithm=algorithm -flive-patching=level -fira-region=region  -fira-hoist-pressure -fira-loop-pressure  -fno-ira-share-save-slots -fno-ira-share-spill-slots
    -fisolate-erroneous-paths-dereference  -fisolate-erroneous-paths-attribute -fivopts  -fkeep-inline-functions  -fkeep-static-functions -fkeep-static-consts  -flimit-function-alignment
    -flive-range-shrinkage -floop-block  -floop-interchange  -floop-strip-mine -floop-unroll-and-jam  -floop-nest-optimize -floop-parallelize-all  -flra-remat  -flto  -flto-compression-level
    -flto-partition=alg  -fmerge-all-constants -fmerge-constants  -fmodulo-sched  -fmodulo-sched-allow-regmoves -fmove-loop-invariants  -fno-branch-count-reg -fno-defer-pop
    -fno-fp-int-builtin-inexact  -fno-function-cse -fno-guess-branch-probability  -fno-inline  -fno-math-errno  -fno-peephole -fno-peephole2  -fno-printf-return-value  -fno-sched-interblock
    -fno-sched-spec  -fno-signed-zeros -fno-toplevel-reorder  -fno-trapping-math  -fno-zero-initialized-in-bss -fomit-frame-pointer  -foptimize-sibling-calls -fpartial-inlining  -fpeel-loops
    -fpredictive-commoning -fprefetch-loop-arrays -fprofile-correction -fprofile-use  -fprofile-use=path  -fprofile-values -fprofile-reorder-functions -freciprocal-math  -free  -frename-registers
    -freorder-blocks -freorder-blocks-algorithm=algorithm -freorder-blocks-and-partition  -freorder-functions -frerun-cse-after-loop  -freschedule-modulo-scheduled-loops -frounding-math
    -fsave-optimization-record -fsched2-use-superblocks  -fsched-pressure -fsched-spec-load  -fsched-spec-load-dangerous -fsched-stalled-insns-dep[=n]  -fsched-stalled-insns[=n]
    -fsched-group-heuristic  -fsched-critical-path-heuristic -fsched-spec-insn-heuristic  -fsched-rank-heuristic -fsched-last-insn-heuristic  -fsched-dep-count-heuristic -fschedule-fusion
    -fschedule-insns  -fschedule-insns2  -fsection-anchors -fselective-scheduling  -fselective-scheduling2 -fsel-sched-pipelining  -fsel-sched-pipelining-outer-loops -fsemantic-interposition
    -fshrink-wrap  -fshrink-wrap-separate -fsignaling-nans -fsingle-precision-constant  -fsplit-ivs-in-unroller  -fsplit-loops -fsplit-paths -fsplit-wide-types  -fssa-backprop  -fssa-phiopt
    -fstdarg-opt  -fstore-merging  -fstrict-aliasing -fthread-jumps  -ftracer  -ftree-bit-ccp -ftree-builtin-call-dce  -ftree-ccp  -ftree-ch -ftree-coalesce-vars  -ftree-copy-prop  -ftree-dce
    -ftree-dominator-opts -ftree-dse  -ftree-forwprop  -ftree-fre  -fcode-hoisting -ftree-loop-if-convert  -ftree-loop-im -ftree-phiprop  -ftree-loop-distribution  -ftree-loop-distribute-patterns
    -ftree-loop-ivcanon  -ftree-loop-linear  -ftree-loop-optimize -ftree-loop-vectorize -ftree-parallelize-loops=n  -ftree-pre  -ftree-partial-pre  -ftree-pta -ftree-reassoc  -ftree-scev-cprop
    -ftree-sink  -ftree-slsr  -ftree-sra -ftree-switch-conversion  -ftree-tail-merge -ftree-ter  -ftree-vectorize  -ftree-vrp  -funconstrained-commons -funit-at-a-time  -funroll-all-loops
    -funroll-loops -funsafe-math-optimizations  -funswitch-loops -fipa-ra  -fvariable-expansion-in-unroller  -fvect-cost-model  -fvpt -fweb  -fwhole-program  -fwpa  -fuse-linker-plugin --param
    name=value -O  -O0  -O1  -O2  -O3  -Os  -Ofast  -Og
```

Optimizations

Languages (C++)

Common Sub-expression removal:

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

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

Optimizations

Languages (C++)

Redundant code removal:

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

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:

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

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

Optimizations

Languages (C++)

Induction variable:

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

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

Optimizations

Languages (C++)

Loop unrolling:

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

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

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:

```cpp
for (int j=0; i<columns; j++) {
    for (int i=0; i<rows; i++) {
        mymatrix[i][j] += increment;
    }
}
```

Correct:

```cpp
for (int i=0; i<rows; i++) {
    for (int j=0; i<columns; j++) {
        mymatrix[i][j] += increment;
    }
}
```

Manual optimizations

Languages (C++)

Loop Fusion:

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

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

Wroclaw University of Science and Technology

OpenMP

Languages (C++)

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

OpenMP

Languages (C++)

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

Wroclaw University of Science and Technology

OpenMP

Languages (C++)

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

OpenMP

Languages (C++)

```
#pragma omp parallel
int id = omp_get_thread_num();
int total = omp_get_num_threads();
#pragma omp for
for (int i=0; i<10; i++) {
    // do something with i
}
```

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

Wroclaw University of Science and Technology

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

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]){
    MPI_Init(&argc, &argv);
    printf("Hello, world!\n");
    MPI_Finalize();
    return 0;
}
```

MPI

Languages

- how many processes?
- which process am I?

MPI

Languages

```c
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] ){
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

MPI

Languages

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

Wrocław University of Science and Technology

Large Scale Data Processing

Lecture 4 – Data processing languages

dr inż. Tomasz Kajdanowicz, Roman Bartusiak, Piotr Bielak

December 27, 2019