High-performance Computing for Economists

Lukas Mann[1]

[1]Adapted from notes by R. Cioffi, J. Fernández-Villaverde, P. Guerrón, and D. Zarruk

May 2024

C++

C/C++

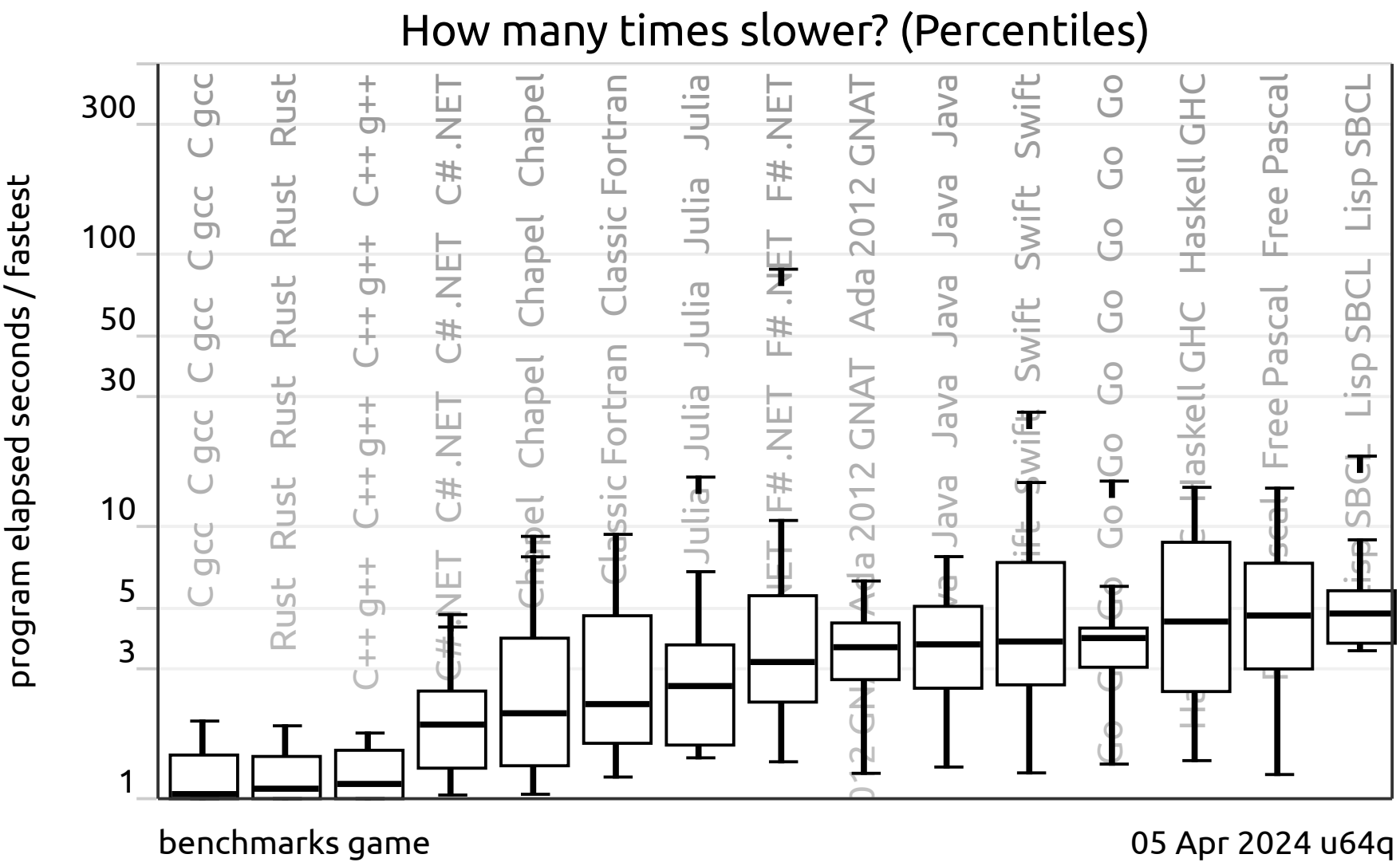
- `C/C++` is the infrastructure of much of the modern computing world.
- If you know `Unix` and `C/C++`, you can probably master everything else easily (think of Latin and Romance languages!).
- In some sense, `C++` is really a “federation” of languages.
- What is the difference between `C` and `C++`?
- `C++` introduced full OOP support.

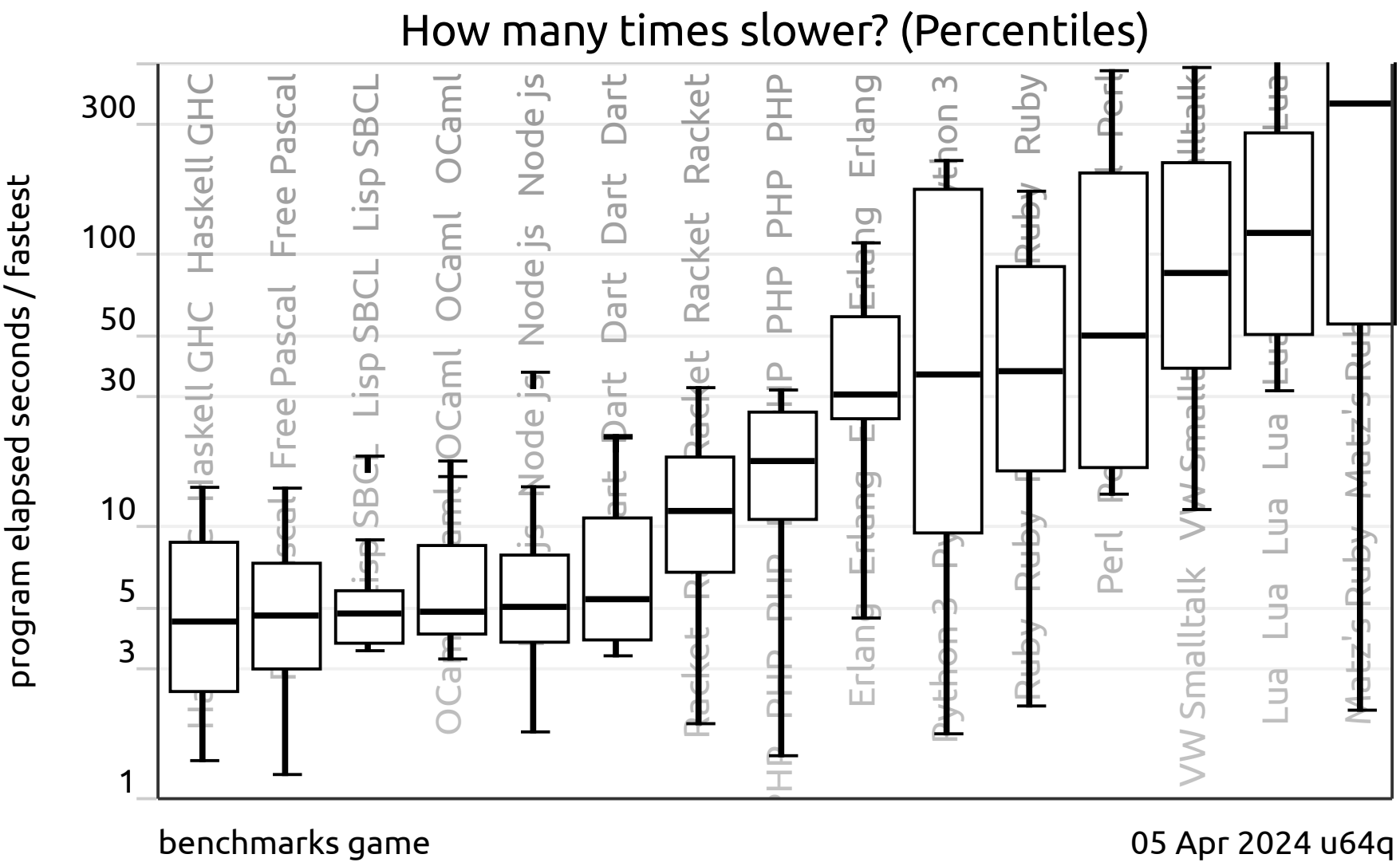
C++

- General-purpose, multi-paradigm, static partially inferred typed, compiled language.
- Current standard is `C++20` (published in December 2020).
- Developed by Bjarne Stroustrup at Bells Labs in the early 1980s.
- `C++` has a more modern design and approach to programming than predecessor languages. Enormously influential in successor languages.
- If you know `C++`, you will be able to read `C` legacy code without much problem.
- In fact, nearly all `C` programs are valid `C++` programs (the converse is not true).
- But you should try to "think" in `C++20`, not in `C` or even in older `C++` standards.

C++: advantages

1. Powerful language: you can code anything you can imagine in `C++`.
2. Continuously evolving but with a standard: new implementations have support for meta-programming, functional programming,...
3. Rock-solid design.
4. Top performance in terms of speed.
5. Wide community of users.
6. Excellent open-source compilers and associated tools.
7. One of most detailed object-orientation programming (OOP) implementation available.
8. Easy integration with multiprocessor programming (`OpenMP`, `MPI`, `CUDA`, `OpenCL`, ...).
9. Allows low-level memory manipulation.


How many times slower? (Percentiles)
program elapsed seconds / fastest
300
100
50
30
10
5
3
1
C gcc
Rust
C++ g++
C# .NET
Chapel
Classic Fortran
Julia
F# .NET
Ada 2012 GNAT
Java
Swift
Go
Haskell GHC
Free Pascal
Lisp SBCL
benchmarks game
05 Apr 2024 u64q


How many times slower? (Percentiles)
program elapsed seconds / fastest
300
100
50
30
10
5
3
1
Haskell GHC
Free Pascal
Lisp SBCL
OCaml
Node.js
Dart
Racket
PHP
Erlang
Python 3
Ruby
Perl
VW Smalltalk
Lua
Mutz's Ruby
benchmarks game
05 Apr 2024 u64q

C++: disadvantages

1. Hard language to learn, even harder to master (although for scientific computation –and if you start with control structures, arrays, and functions before pointers, objects and classes– you can get away with a small subset).

2. Large specification: `C++20`, `Standard Template Library`, ...

3. This causes, at times, portability issues.

4. Legacies from the past (you even have a `goto`!).

5. Minimal optimization of OO features.

6. Notation is far away from standard mathematical notation (you can fix much of this with libraries, but then you need to learn them).

Additional resources I

Books with focus on `C++11` and later implementations and on scientific computing.

- Basic:
  1. Starting Out with C++ from Control Structures to Objects, by Tony Gaddis.
  2. Guide to Scientific Computing in C++, by Joe Pitt Francis and Jonathan Whiteley.
- Intermediate:
  1. Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers by Peter Gottschling.
  2. The C++ Programming Language, by Bjarne Stroustrup.

Additional resources II

- Advanced:
  1. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, by Scott Meyers.
  2. The C++ Standard Library: A Tutorial and Reference, by Nicolai M. Josuttis.
  3. C++ Templates: The Complete Guide, by David Vandevoorde and Nicolai M. Josuttis.
- References:
  1. **`http://www.cplusplus.com`**.
  2. **`http://www.cprogramming.com`**.
  3. **`https://isocpp.org/`**

Python

Python

- General-purpose, multi-paradigm, dynamically typed, interpreted language.
- Designed by Guido von Rossum.
- Name inspired by Monty Python's Flying Circus.
- Open source.
- Simple and with full OOP support.
- Elegant and intuitive language.
- Ideal, for instance, to teach high school students or a class in introductory CS.

Python: advantages

1. Great for prototyping: dynamic typing and REPL.
2. Rich ecosystem:
    - 2.1 Scientific computation modules: **`NumPy`**, **`SciPy`**, and **`SymPy`**.
    - 2.2 Statistics modules: **`Pandas`**.
    - 2.3 Plotting modules: **`matplotlib`** and **`ggplot`**.
    - 2.4 Powerful ML library: **`tensorflow`**.
3. Easy unit testing: **`doctest`** is a default module.
4. Manipulates strings surprisingly well (regular expressions)⇒natural language processing, artificial intelligence, big data.
5. Excellent interaction with other languages.
6. If you have a Mac, it is already preinstalled (although, likely, an older version)

Python: disadvantages

- Considerable time penalty.
- Reference distribution by Guido van Rossum: **CPython** with **IDLE** and **Python Launcher** at http://www.python.org/.
- Better distribution for economists: **Anaconda**, with many additional tools https://www.anaconda.com/.
- **Miniconda** is a more lightweight alternative.
- High-performance routes:
  1. **Numba** is an just-in-time specializing compiler which compiles annotated Python and NumPy code to **LLVM**.
  2. **Pypy** is an implementation with a JIT compiler.
  3. **Cython** is a superset of Python with an interface for invoking C/C++ routines.


VENOM

Additional resources

- Books:

  1. Learning Python (5th Edition), by Mark Lutz.
  2. Think Python (2nd Edition), by Allen Downey.
  3. High Performance Python, by Micha Gorelick and Ian Ozsvald.

- Web page of the language: `http://www.python.org/`

Julia

Julia

- Modern, expressive, high-performance programming language designed for scientific computation and data manipulation.
- Open-source.
- LLVM-based just-in-time (JIT) compiler.
- **`Lisp`**-style macros.
- Designed for parallelism and cloud computing.
- Syntax close to **`Matlab`**, **`R`** and **`Python`**, but not just a faster **`Matlab`**.
- External packages.
- Easy to integrate with **`C++`** and **`Python`**.

Additional resources

- Julia: `http://julialang.org/`.
- Julia-vscode: `https://www.julia-vscode.org/`.
- Books:
  1. Julia 1.0 Programming: Dynamic and high-performance programming to build fast scientific applications, 2nd Edition, by Ivo Balbaert.
  2. Mastering Julia 1.0: Solve complex data processing problems with Julia, by Malcolm Sherrington.
  3. Hands-On Design Patterns and Best Practices with Julia: Proven solutions to common problems in software design for Julia 1.x, by Tom Kwong.

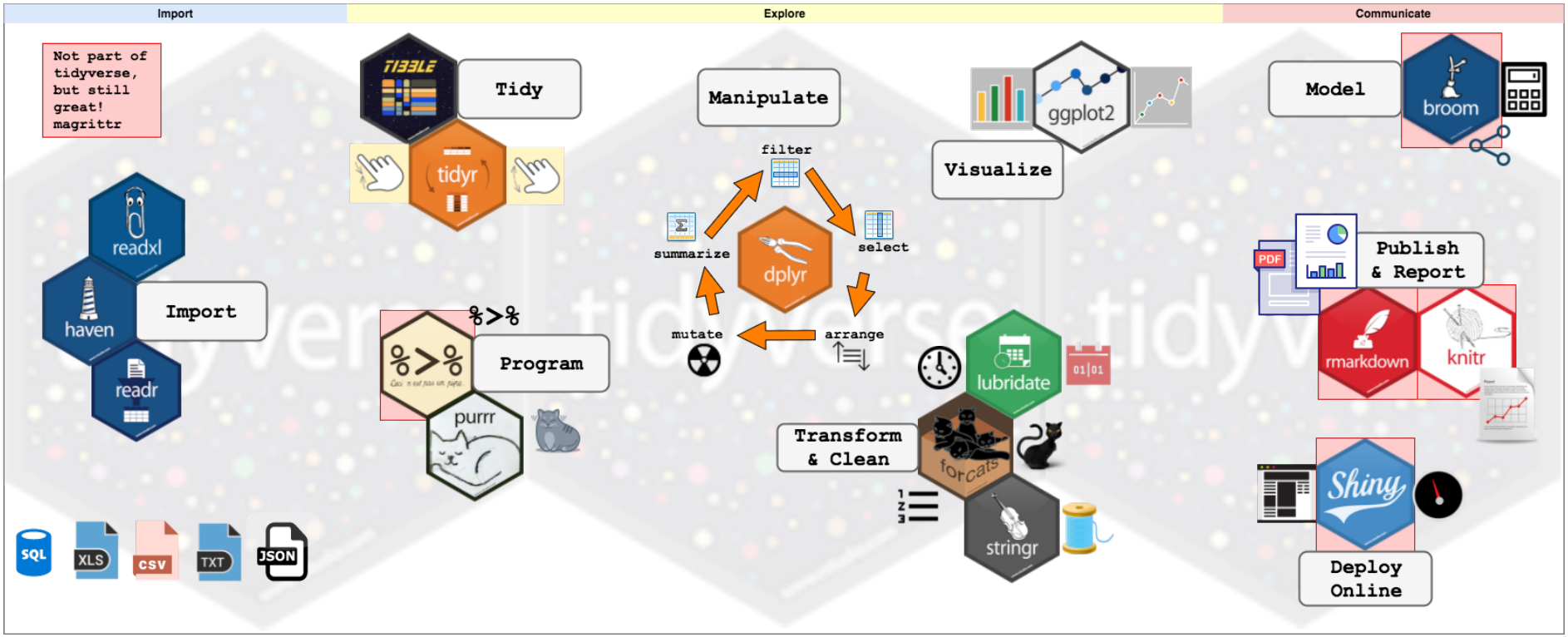
R

R

- High level, open source language for statistical computation.
- Developed by Ross Ihaka and Robert Gentleman as an evolution of **S**, programmed by John Chambers in 1975–1976 (at Bells Labs).
- **S** was itself created to substitute a **Fortran** library for statistics.
- It was soon moved into the public domain: **http://cran.r-project.org/**.
- Key for success.
- **S** has another descendent: **S-Plus**, a commercial implementation that is much less popular.
- Elegant and intuitive syntax.
- Advanced OOP implementation: each estimator is a class.

R: advantages

1. Rich and active community of users:
    - Around 14k packages (**`ggplot2`**, **`data.table`**, **`dplyr`**, **`tidyr`**, **`readr`**, **`knitr`**, **`markdown`**, **`tidyr`**,...).
    - **`https://www.r-bloggers.com`**
    - Dozens of books.
    - Dozens of free tutorials.
2. Widely used for big data (often with **`Hadoop`** and **`Spark`**).
3. Allows for multiprogramming: **`lambda.r`** and **`purrr`** for functional programming.
4. You can easily compile functions into byte code with package **`compiler`**.
5. Interacts well with other languages: **`Rcpp`**, **`Reticulate`**.
6. Easy to parallelize (**`parallel`**).


Import
Explore
Communicate
Not part of tidyverse, but still great! magrittr
Tidy
tidyr
Manipulate
filter
summarize
select
dplyr
mutate
arrange
Visualize
ggplot2
Model
broom
readxl
haven
readr
Import
%>%
Program
purrr
Publish & Report
PDF
rmarkdown
knitr
lubridate
Transform & Clean
forcats
stringr
Shiny
Deploy Online
SQL
XLS
csv
TXT
JSON

RStudio

- **RStudio** is a simple and powerful IDE:

    1. **GIT** integration.
    2. **markdown** support (different versions, including **knitr**).
    3. Package manager.
    4. Install **tidyverse** right away.

Matlab

Matlab

- Matrix laboratory.
- Started in the late 1970s, released commercially in 1984.
- Widely used in engineering and industry.
- Plenty of code around (both for general purposes and in economics).
- Recent versions use **`Java`** extensively in the background⇒allows you to call **`Java`** libraries.
- Many useful toolboxes:
    1. By Mathworks.
    2. By third parties: **`Dynare`**.
- Clones available: **`Octave`**, **`Scilab`**,...

Matlab: advantages

- Great IDE:
  1. Editor, with syntax highlight, smart indent, cells, folding nested statements, easy comparing tools.
  2. Debugger.
  3. checkcode (a linting function).
  4. Automatic TODO/FIXME Report
- Other tools:
  1. Profiler.
  2. Unit testing.
  3. Coder.
- Good OO capabilities.
- Interacts reasonably well with `C/C++`, `Fortran`, and `R` (type !R)

Matlab: disadvantages

- Very expensive if you do not have a university license.
- Age starts to show.
- Many undocumented features: `https://undocumentedmatlab.com/`.
- Features creep causes problems.
- Use of `Java` makes it prone to crashing and some numerical instabilities.
- Parallelization tools are often disappointing.
- Clones (such as `Octave`) are quite slow.

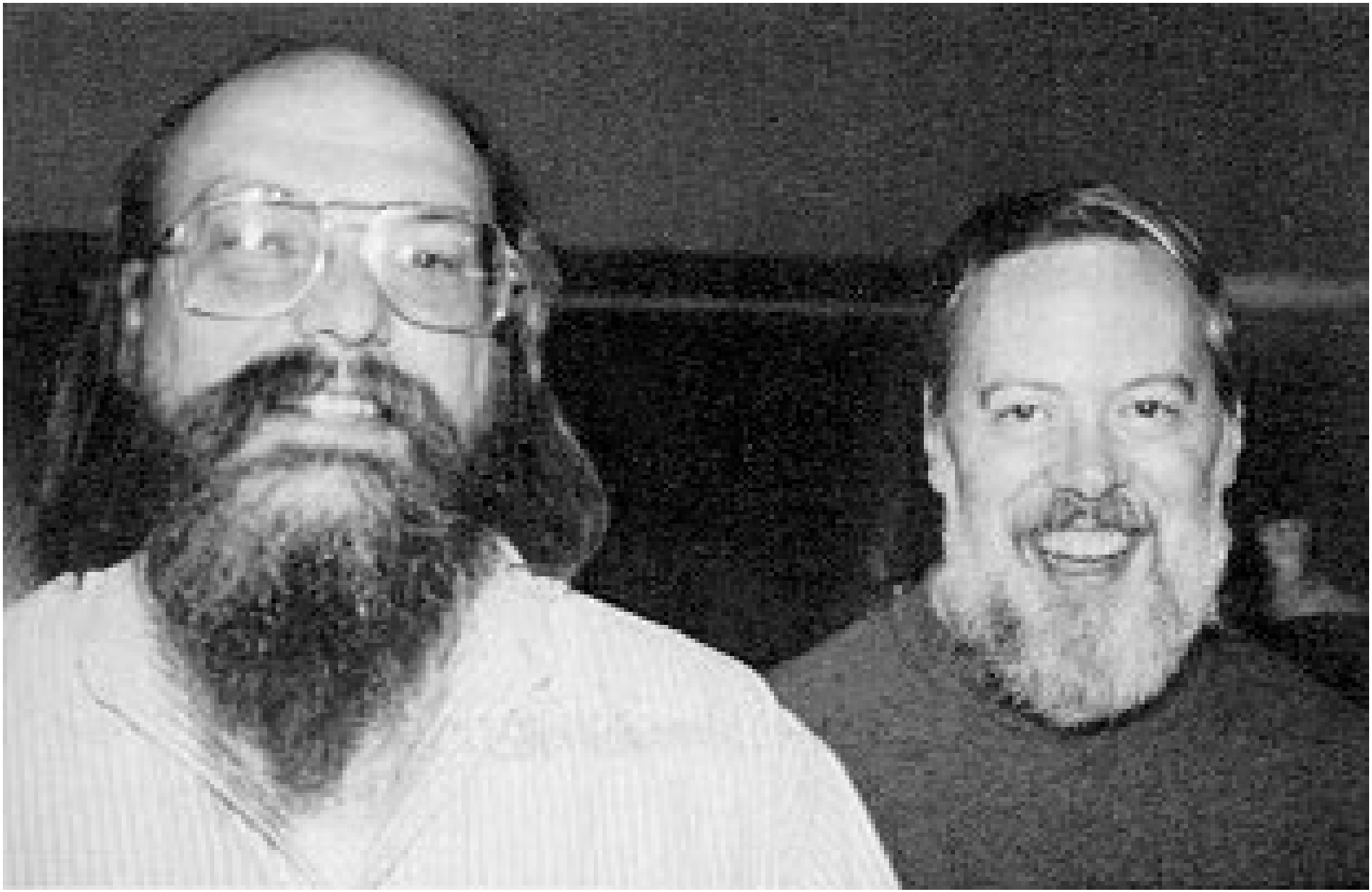
Other choices for scientific computation

- General-purpose languages (GPL):
  1. `C`.
  2. `Fortran`.
  3. `Java`.
  4. `Scala`.
- Domain-specific languages (DSL):
  1. `Mathematica`.
  2. `Stata`.

C

C |

- **C** was created by Dennis Ritchie at Bell Labs to port **UNIX** to different machines.
- The name come because it was created after failing with:
  1. **Fortran**.
  2. With a previous language called **B** developed by Ken Thompson. B itself derived from a previous language called Basic Combined Programming Language (**BCPL**), the first brace programming language.
- **C** is used nowadays for mainly for systems interfaces, embedded controllers, and real-time applications.
- For instance, **Linux** and **R** source code are written in **C**.
- A typical project in a CS class is to code a **Lisp** interpreter in **C**.



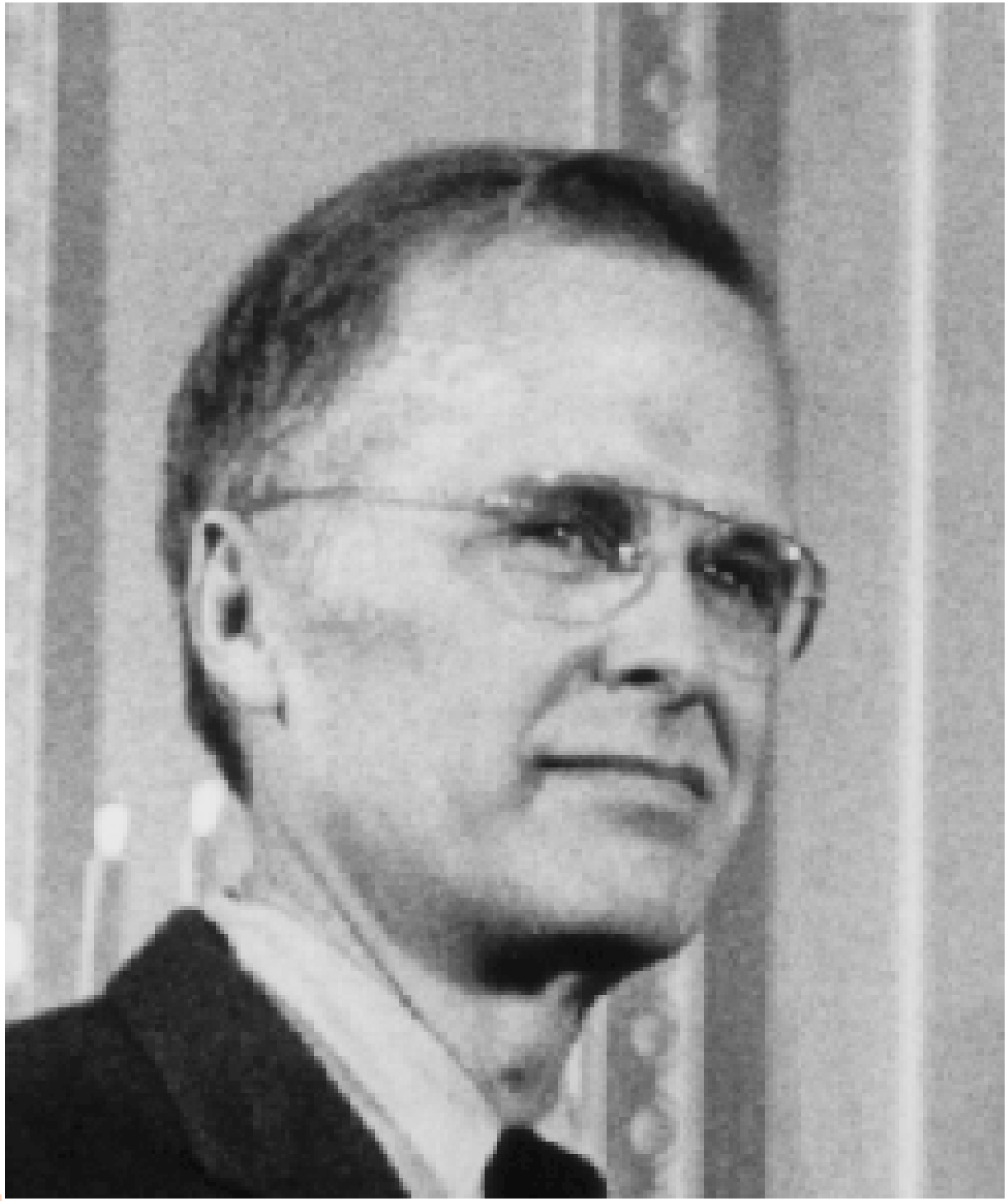
C II

- `C` is a transitional language between a high-level language and a low-level language.
- Preprocessor includes header files, macro expansions, conditional compilation, and line control.
- Lean and fast.
- Easy to do everything, easier to really screw up.
- Original philosophy behind `C` was that programmer needs to check for error (such as vector overflows) not the compiler. This makes `C` fast, but difficult to debug.
- Pointers and memory management.
- Standard reference: The C Programming Language (2nd Edition), by Brian W. Kernighan and Dennis M. Ritchie (of "Hello, world!" fame).

Fortran

Fortran

- Grandfather of all modern languages: “Real Programmers can write Fortran in any language.”
- **`Fortran`** (“Formula Translation”) developed by John Backus and coworkers in 1957 for the IBM 704.
- However, it has kept updating itself:
  1. **`Fortran 2018`** (**`Fortran 2023`** expected soon!).
  2. Introduction of modern features such as (limited) OO.
- Surprising resilience: still used for scientific and engineering problems (weather forecast, nuclear bombs design, chemical plants,….).
- Used by many economists.



Fortran: advantages

- Large set of best-of-pack numerical libraries.
- Small and compact language:
    1. Easy to learn.
    2. Portable.
    3. Easy to implement quality control by software engineering.
    4. Easy to debug.
- Nice array support.
- Easy to parallelize in shared-memory.

Fortran: disadvantages

- Small community of users: only used for technical computations.
- Expiration date? The 1990s foretells of its death were exaggerated. Rush toward **C++** was often costly.
- No speed advantage any longer.
- Limitations of the language (limited data structures, no functional and meta-programming, next-to-none text processing,...).
- Yes, easy to learn but:

1. Far away from modern approaches to programming.
2. You would probably not find many classes/books on **Fortran** at your institution.

Java

Java

- Created by Sun. Now Oracle and open source.
- Evolution from C++:
  1. Pure OO.
  2. Garbage collection.
  3. Cleaner structure.
- Original idea: “Write once, run anywhere (WORA)”.
- Key component: a virtual machine (VM) that performs JIT.
- You suffer a penalty in time with respect to `C++`.

Scala

- General-purpose, functional plus OO, strong static typed language.
- Developed by Martin Odersky.
- Syntax very close to `Java`, but with much syntactic sugar.
- Runs on the `JVM`.
- Great for interactions with `Spark`.
- Nice sequence of online courses on `Coursera`.

Mathematica

Mathematica: advantages

- Developed by Stephen Wolfram.
- Mainly oriented toward symbolic computation.
- Everything is an expression that gets manipulated.
- Functional programming and list-based (although accepts procedural programming at a performance penalty cost).
- Extensive meta-programming abilities. In particular, easy to generate `Fortran` and `C` code.
- You can compile intensive parts of the code.

Mathematica: disadvantages

- Programming approach is different from other languages.
- Difficult to write idiomatic `Mathematica` code if you come from other traditions.
- Nearly no OO support.
- Cryptic error messages and sparse documentation.
- Smaller community.
- Wolfram's company is high both in undeserved self-promotion and in making life difficult for the user.

Stata

Stata

- Statistical package.
- Very popular in microeconometrics.
- Old design and implementation.
- Commercial software.
- Limits the scope of what you can accomplish.
- Similar (and even more so) comments apply to other statistical/econometrics software (`EViews, RATS, GAUSS,...`).

Alternatives?

- Plenty of alternative GPL that we will not discuss:
  1. **C#**, **Javascript**, **PHP**, **Perl**, **Swift**, and **Objective-C**.
  2. **Visual Basic .NET** plus its ancestors.
  3. **Ruby**.
  4. **Rust**.
  5. **Delphi/Object Pascal** plus its ancestors.
  6. **Go**.
  7. **Lua**.
  8. **Ada**.
- These languages are not oriented toward scientific computing.
- Family of functional languages (see lecture slides on functional programming).
- **Sage** for symbolic computation.

Comparison

A baby example

- A basic RBC model:

$$\max \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \log c_t$$

$$\text{s.t. } c_t + k_{t+1} = e^{z_t} k_t^{\alpha} + (1-\delta) k_t, \forall\, t > 0$$
$$z_t = \rho z_{t-1} + \sigma \varepsilon_t,\ \varepsilon_t \sim \mathcal{N}(0,1)$$

- If $\delta = 1$, by "guess and verify":

$$c_t = (1 - \alpha\beta)\, e^{z_t} k_t^{\alpha}$$
$$k_{t+1} = \alpha\beta e^{z_t} k_t^{\alpha}$$

- Calibration:

Parameter	β	α	ρ	σ
Value	0.95	1/3	0.95	0.007

Table 1: Average and Relative Run Time (Seconds)

	Mac		
Language	Version/Compiler	Time	Rel. Time
C++	GCC-7.3.0	1.60	1.00
	Intel C++ 18.0.2	1.67	1.04
	Clang 5.1	1.64	1.03
Fortran	GCC-7.3.0	1.61	1.01
	Intel Fortran 18.0.2	1.74	1.09
Java	9.04	3.20	2.00
Julia	0.7.0	2.35	1.47
	0.7.0, fast	2.14	1.34
Matlab	2018a	4.80	3.00
Python	CPython 2.7.14	145.27	90.79
	CPython 3.6.4	166.75	104.22
R	3.4.3	57.06	35.66
Mathematica	11.3.0, base	1634.94	1021.84

Table 1 (cont.): Average and Relative Run Time (Seconds)

	Mac		
Language	Version/Compiler	Time	Rel. Time
Matlab, Mex	2018a	2.01	1.26
Rcpp	3.4.3	6.60	4.13
Python	Numba 0.37.9	2.31	1.44
	Cython	2.13	1.33
Mathematica	11.3.0, idiomatic	4.42	2.76

Comparison

- Value function iteration, with 35,640 points in the grid of capital and 5 points in the grid of productivity.
- Code has been written in a relatively uniform way across languages.
- Hence, it does not take advantage of language-specific features.
- Code was not optimized for maximum performance, but for clarity.
- `Mathematica` was a partial exception: because of its functional programming structure, a "matlabish" version is too slow.
- Hence, we may be biased in its favor.
- `GCC` compilers were faster than `LLVM` or `Intel` compilers.

Some (opinionated) advice I

- As a DSL for numerical computing, `Julia` is a great choice.
- If you decide to learn one GPL, my advice would be to learn `C++`.
    1. You will have the most powerful tool existing.
    2. You will understand everything else.
    3. Good programming habits.
    4. Keeps your options open.
    5. You can start with a relatively small subset of the language and move later to OO, meta-programming, and functional programming.
- `Python`: Mostly viable if you are interested in deep learning, text analysis, etc. Extremely marketable for industry jobs.

Some (opinionated) advice II

- You should only learn **`Fortran`** if you need to use legacy code or it is the language that your advisor asks you to code in:

  1. No real differences in performance: back-end of GCC is the same, only front-end differs.

  2. Furthermore, once you have learned **`C++`**, you can probably pick up **`Fortran`** in a couple of days if you really need to.

- **`Java`** or **`Scala`**: little reason to pay the speed penalty with respect to C++ unless:

  1. The use of the VM is relevant for the project.

  2. You want to link **`Java`** libraries in **`Matlab`**, **`Mathematica`**, or other DSL or use **`Spark`** at a high-level.

Some (opinionated) advice III

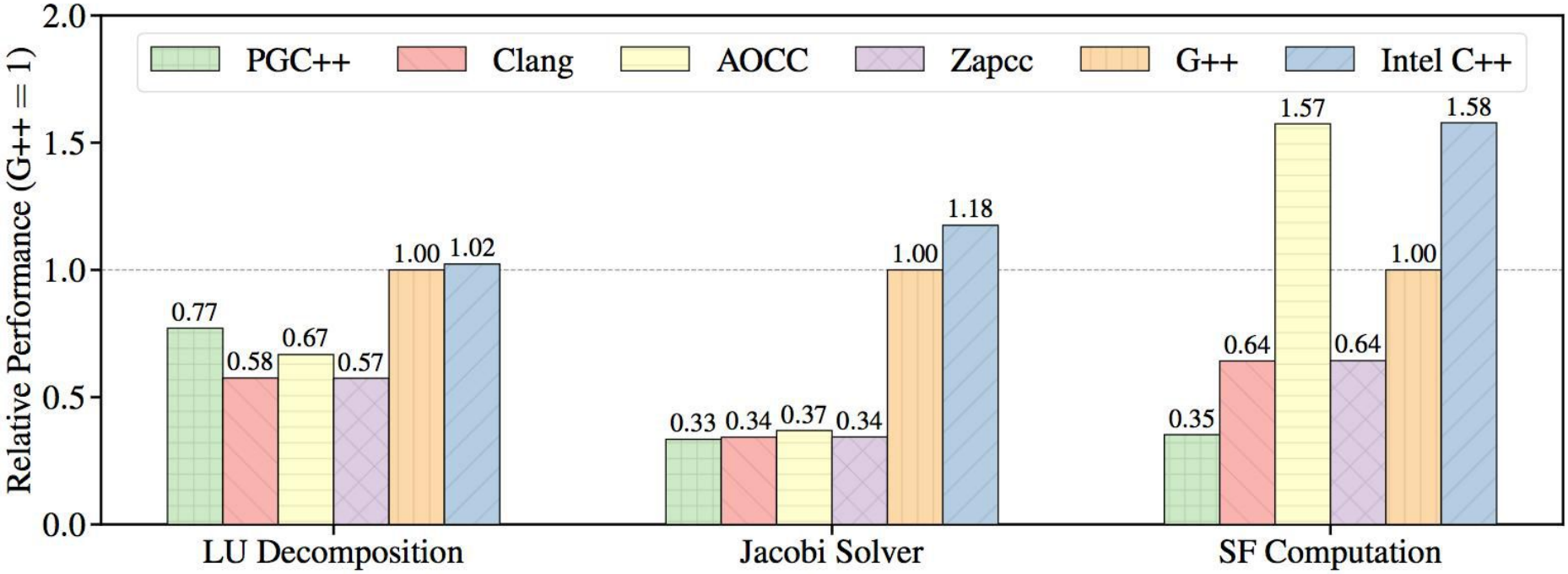
- Even if you use a GPL for "heavy duty" computation, a DSL is convenient for:
  1. Simple tasks (editing some data).
  2. Making graphs (great packages available).
  3. Using specialized packages (for most statistical procedures there is a nice `R` package).
  4. Prototyping (we prototype all the time).
- If the expected computation time of your code is less than a few minutes in `Julia` or `R`, the cost of coding in a GPL language probably is too high.
- Given `Julia`'s current speed, one can think about it as a good "default" language.
- `Julia` and `R` are the most useful DSLs for economists. `Matlab` will remain useful for a long time given the large amount of legacy code on it. `Mathematica` is powerful, but small installed base. `Stata` is dominated by `R`.

Compilers

Compilers

- If you use a compiled language such as **C/C++** or **Fortran**, you have another choice: which compiler to use?
- Huge differences among compilers in:
  1. Performance.
  2. Compatibility with standards.
  3. Implementation of new features:

     http://en.cppreference.com/w/cpp/compiler_support.
  4. Extra functionality (**MPI**, **OpenMP**, **CUDA**, **OpenACC**. ...).
- High return in learning how to use your compiler proficiently.
- Often you can mix compilers in one project.



Last updated November 2017

The GCC compiler collection

- A good default option: **`GNU GCC 12.1`** compiler.

  1. Open source.
  2. **`C`**, **`C++`**, **`Objective-C`**, **`Java`**, **`Fortran`**, **`Ada`**, and **`Go`**.
  3. Integrates well with other tools, such as **`JetBrains`**' IDEs.
  4. Updated (**`C++20`**).
  5. Efficient.
  6. An Introduction to GCC, by Brian Gough,

The LLVM compiler infrastructure

1. **LLVM** (**http://llvm.org/**), including **Clang**.
    1.1 It comes with OS/X and Xcode.
    1.2 Faster for compiling, uses less memory.
    1.3 Run time is slightly worse than **GCC**.
    1.4 Useful for extensions: **Cling** (**https://github.com/root-project/cling**).
    1.5 Architecture of **Julia**.
2. **DragonEgg**: uses **LLVM** as a **GCC** backend.

Commercial compilers

1. **`Intel Parallel Studio XE`** (in particular with MKL) for **`C`**, **`C++`**, and **`Fortran`** (plus a highly efficient **`Python`** distribution). Community edition available.

2. **`PGI`**. Community edition available. Good for **`OpenACC`**.

3. **`Microsoft Visual Studio`** for **`C`**, **`C++`**, and other languages less relevant in scientific computation. Community edition available.

4. **`C/C++`**: **`C++Builder`**.

5. **`Fortran`**: Absoft, Lahey, and NAG.

Libraries

Libraries I

- Why libraries?
- Well-tested, state-of-the-art algorithms.
- Save on time.
- Classic ones
    1. **`BLAS`** (Basic Linear Algebra Subprograms).
    2. **`Lapack`** (Linear Algebra Package).

Libraries II

- More modern implementations:
  1. **`Accelerate Framework`** (OS/X).
  2. **`ATLAS`** (Automatically Tuned Linear Algebra Software).
  3. **`MKL`** (Math Kernel Library).
- Open source libraries:
  1. **`GNU Scientific Library`**.
  2. **`GNU Multiple Precision Arithmetic Library`**.
  3. **`Armadillo`**.
  4. **`Boost`**.
  5. **`Eigen`**.

Linting

Linting

- Lint was a particular program that flagged suspicious and non-portable constructs in `C` source code.
- Later: Generic word for a tool that discovers errors in a code (syntax, typos, incorrect uses) before the code is compiled (or run)
- It also enforces coding standards.
- Good practice: never submit anything to version control (or exit the text editor) unless your linting tool is satisfied.
- Examples:
    1. Good **`IDEs`** and **`GCC`** (and other compilers) have excellent linting tools. See e.g. VS Code extensions.
    2. **`C/C++`**: **`clang-tidy`** and **`ccpcheck`**.
    3. **`Julia`**: Use **`VS Code`** extension for linting.
    4. **`R`**: **`lintr`**.
    5. **`Matlab`**: **`checkcode`** in the **`editor`**.

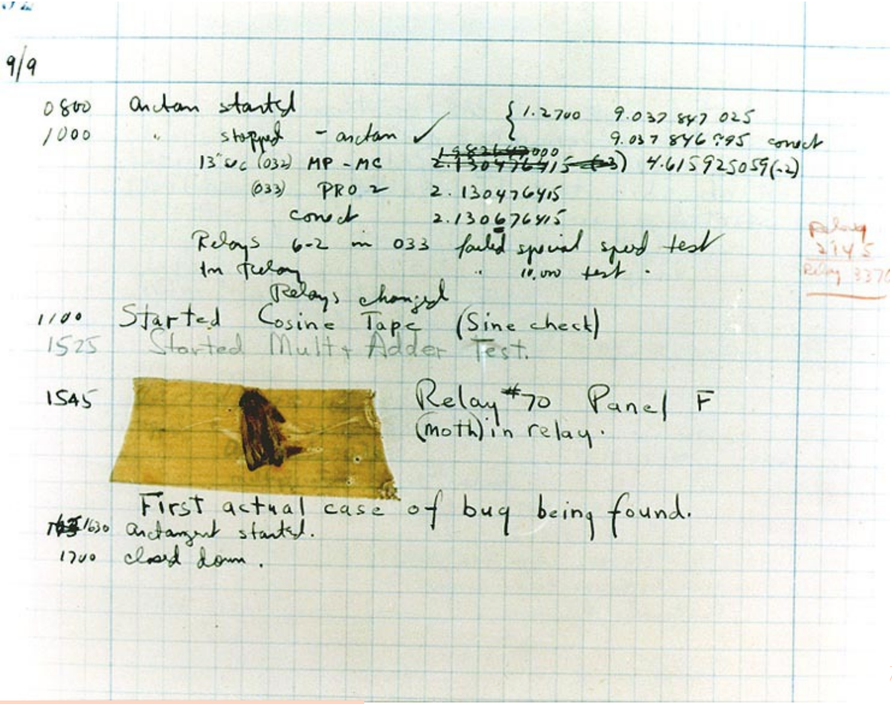
Debugging

Debugging

C. Titus Brown

If you're confident your code works, you're probably wrong. And that should worry you.

- Why bugs? Harvard Mark II, September 9, 1947.
- Find and eliminate mistakes in the code.
- In practice more time is spent debugging than in actual coding.
- Complicated by the interaction with optimization.
- Difference between a bug and a wrong algorithm.



9/9

0800 Antan started
1000 " stopped - antan ✓ {1.2700 9.037 847 025
9.037 846 995 correct
13" sec (032) MP - MC ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

Relay 2145
Relay 3370

Typical bugs

- Memory overruns.
- Type errors.
- Logic errors.
- Loop errors.
- Conditional errors.
- Conversion errors.
- Allocation/deallocation errors.

How to avoid them

- Techniques of good coding.
- Error handling.
- Strategies of debugging:
    1. Tracing: line by line.
    2. Stepping: breakpoints and stepping over/stepping out commands.
    3. Variable watching.

Debuggers

- Manual inspection of the code. Particularly easy in interpreted languages and short scripts.
- Use **`assert`**.
- More powerful: debuggers:
    1. Built in your application: **`RStudio`**, **`Matlab`** or **`IDEs`**.
    2. Explicit debugger:
        - 2.1 **`GNU Debugger (GDB)`**, installed in your Unix machine.
        - 2.2 **`Python`**: **`pdb`**.
        - 2.3 **`Julia`**: Excellent debugging tools in **`VS Code`**.


BEFORE GOING DOWN A STEEP HILL LIKE THIS, ONE SHOULD ALWAYS GIVE HIS SLED A SAFETY CHECK.
RIGHT.
SEAT BELTS?
NONE.
SIGNALS?
NONE.
BRAKES?
NONE.
STEERING?
NONE.
WHEEEEEEE

Unit testing

- Tools:
    1. xUnit framework (CppUnit, testthat in **`R`**, ...).
    2. In **`Julia`**: Test module.
    3. In **`Matlab`**: matlab.unittest framework.
- Regression testing.

Profiler

Profiler

- You want to identify the hot spots of performance.
- Often, they are in places you do not suspect and small re-writtings of the code bring large performance improvements.
- Technique:
    1. Sampling.
    2. Instrumentation mode.
- We will come back to code optimization.
- In `Julia`: `Profile.jl; ProfileView.jl, TimerOutputs.jl`.

Programming Style

Motivation

- In the same way than when writing in a human language, a good style is paramount when writing code.
- Creates code that is:
  1. Clear.
  2. Robust.
  3. Easier to maintain.
  4. Easier to share.
  5. With less bugs.
- Particularly important when working with coauthors.

Style guides and books

- Google coding standard:
  `https://google.github.io/styleguide/cppguide.html`
- The Elements of Programming Style (2nd Edition), by Brian W. Kernighan and P. J. Plauger.
- The Elements of C++ Style, by Trevor Misfeldt, Gregory Bumgardner, and Andrew Gray.
- The Elements of Matlab Style, by Richard Johnson.
- Writing Scientific Software: A Guide to Good Style, by Suely Oliveira and David E. Stewart.

Main ideas

- Two guiding principles:
  1. Consistency. You can have your own rules, but apply them consistenly.
  2. Doing it from start (no window dressing).
- Goals from The Elements of C++ Style:
  1. Simplicity.
  2. Clarity.
  3. Completeness.
  4. Consistency.
  5. Robustness.

Formatting

Formatting

- Keep lines short (80 columns).
- Indentation for nested statements.
- White spaces.
- Block code.
- Use parenthesis even if not extremely needed.

Naming

Naming: a motivating example

- What does this code do?

```
a = b^c*d^e
```

- And this one?

```
y = (k^a)*(l^(1-a))
```

- And this third one?

```
output = (capital^aalpha)*(labor^(1-aalpha))}
```

- Which one do you want to use?

Naming variables

- Variables should have names that are easy to understand:
  1. `output` is a good name.
  2. `a` is not.
- What is a good name is somewhat dependent of the context.
- For instance:
  1. If you are coding an RBC model, names of variables such as `y`, `c`, `i`, or `k` are probably adequate.
  2. Names for counters can be easier than names for variables.

Variable names and programming languages

- Modern programming languages allow for long names.
- For instance, in **`Matlab`**:

```
namelengthmax
ans = 63
```

- Is your programming language case sensitive?
- Does your variable already exist? In **`Matlab`**:

```
exist  myvariable
ans = 1
```

Naming conventions

- Five main conventions:
    1. lowerCamelCase: consumptionDurablesHousehold.
    2. UpperCamelCase: ConsumptionDurablesHousehold.
    3. period.separated: consumption.durables.household.
    4. underscore_separated: consumption_durables_household.
    5. Hungarian notation: doubleconsumption_durables_household,
- lowerCamelCase is perhaps the most used.
- period.separated: confusion with objects, structures, and dataframes (in `R`).
- underscore_separated: for files names.
- Hungarian notation is not very useful with modern IDEs that show workspaces.

Naming functions and objects

- Same ideas for functions, subroutines, objects, and classes, etc.
- Best practices:
    1. Use lowercase only for functions.
    2. Use verbs in the name of the functions.
    3. Use nouns to name classes: **`household`**.
    4. Reserve **`get`** and **`set`** for interactions with objects.
    5. is for Boolean functions: **`isUtilityPositive`**.

Comments

Comments I

- In the ideal case, code is understandable without adding comments (although you always want a header).
- However, complicated pieces of code may need clarifying remarks.
- In that case, describe what the code is aimed to do, not how it does it.
- Share with others and with your future self.
- You will probably spend more time reading code than writing code.
- Document early.
- However, realize that just like code, comments have to be maintained.
- So writing code that is readable without comments can save you a lot of time when fixing bugs or updating your compendium.
- It is better to have no comments than comments that are wrong.

Comments II

- Use TODO/FIXME/NOTE comments: many IDEs will gather them together automatically.
- Automatic systems to Generate documentation from source code.
- **Doxigen**: **http://www.stack.nl/\protect\char126\relaxdimitri/doxygen/**
- Literate programming (Knuth): **weave.jl**, **knitr**.
- Reproducible Research with R and RStudio (2nd Edition) by Christopher Gandrud.
- Dynamic Documents with R and knitr (2nd Edition) by Yihui Xie.

Functions

Functions

1. Write small functions.
2. Name them properly.
3. Modular use.
4. Hide implementation details.
5. Document inmediatedly.
6. Tests.
7. Use function handles.
8. Avoid not passing default parameters.

Optimization

Optimization

Donald Knuth

Premature optimization is the root of all evil.

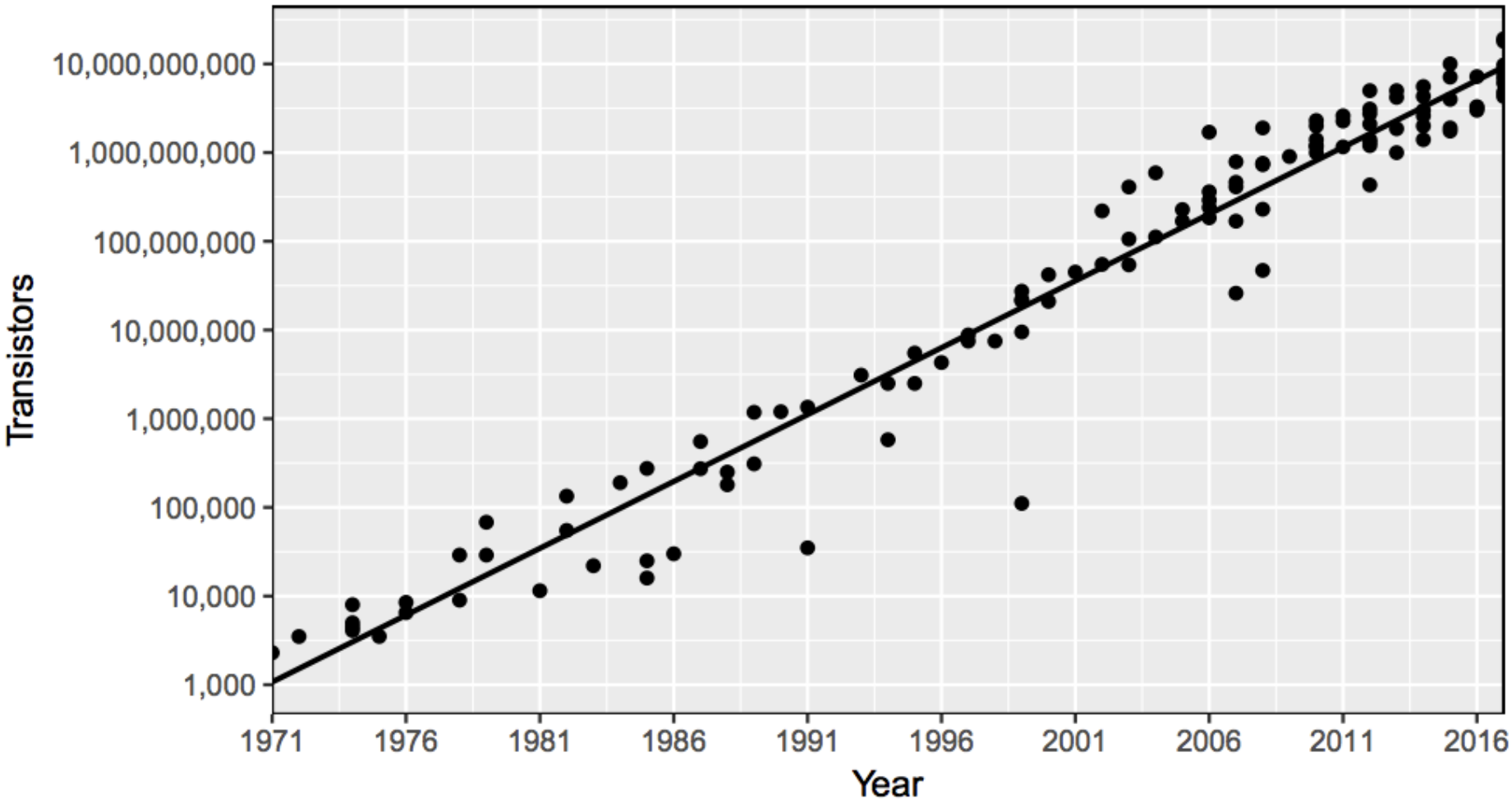
- You first want to be sure your code runs properly.
- Then, you optimize.
- Some automatic tools: `-O` flags in your compiler.
- You want to identify algorithmic bootlenecks and computer hotspots.
- Strategies:
  1. Benchmarking.
  2. Profiling.
  3. Vectorization.
  4. Pre-allocating memory and memorization.
  5. Unrolling loops.
  6. Inlining.

Parallelization

Why parallel?

- Moore's Law (1965): transistor density of semiconductor chips would double roughly every 18 months.
- Problems when transistor size falls by a factor x:
  1. Electricity consumption goes up by x^4.
  2. Heat goes up.
  3. Manufacturing costs go up.
- Inherent limits on serial machines imposed by the speed of light (30 cm/ns) and transmission limit of copper wire (9 cm/ns): virtually impossible to build a serial Teraflop machine with current approach.
- Furthermore, real bottleneck is often memory access (RAM latency has only improved around 10% a year).
- Alternative: having more processors!

Number of transistors



Cray-1, 1975

IBM Summit, 2018

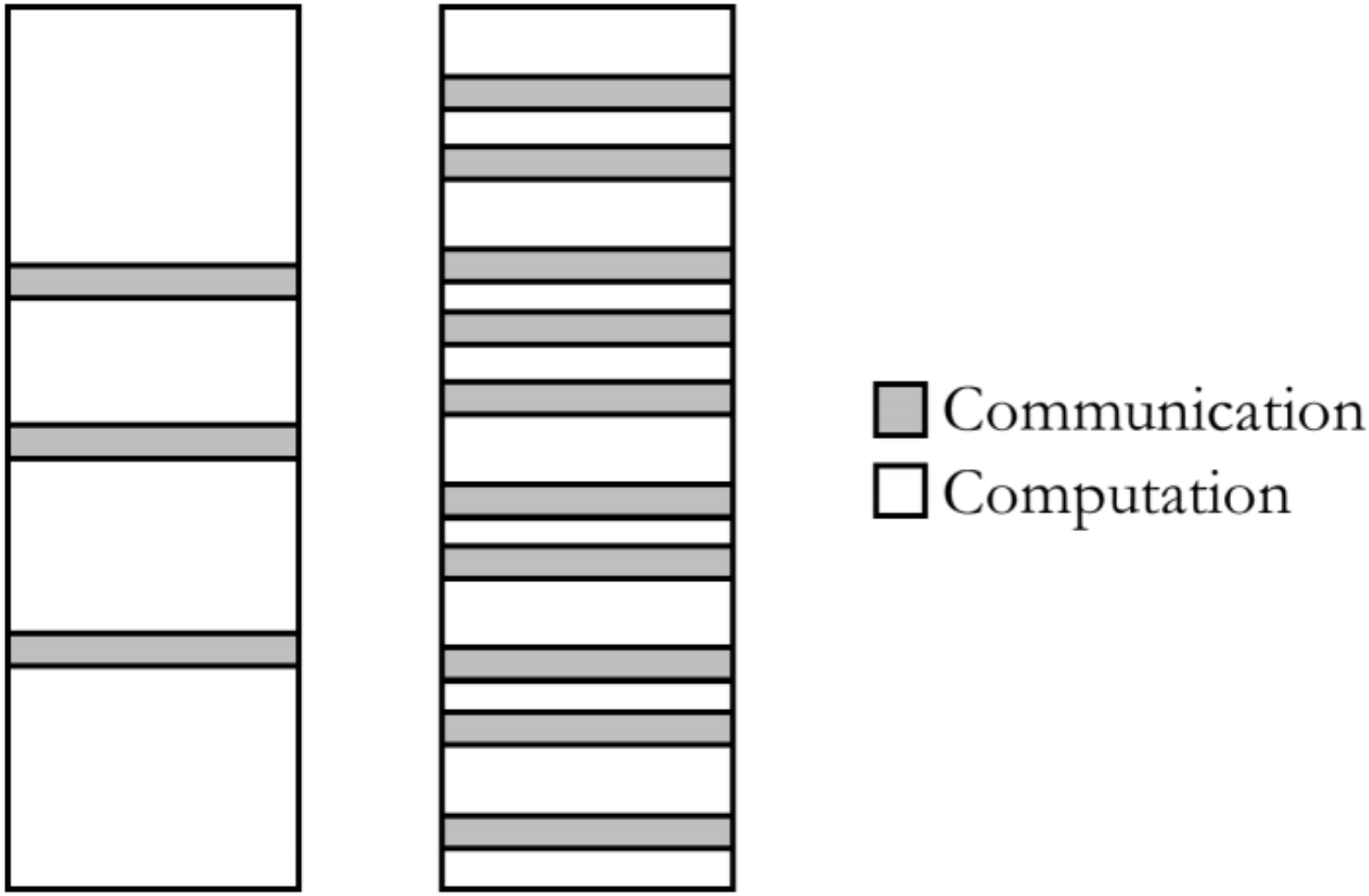
Parallel programming - some references

- Introduction to High Performance Computing for Scientists and Engineers by Georg Hager and Gerhard Wellein.
- Parallel Computing for Data Science: With Examples in R, C++ and CUDA, by Norman Matloff.
- Parallel Programming: Concepts and Practice by Bertil Schmidt, Jorge González-Domínguez, and Christian Hundt.
- An Introduction to Parallel Programming by Peter Pacheco.
- Principles of Parallel Programming by Calvin Lin and Larry Snyder.
- Structured Parallel Programming: Patterns for Efficient Computation by Michael McCool, James Reinders, and Arch Robison.

When do we parallelize? I

- Scalability:
    1. Strongly scalable: problems that are inherently easy to parallelize.
    2. Weakly scalable: problems that are not.
- Granularity:
    1. Coarse: more computation than communication.
    2. Fine: more communication.
- Overheads and load balancing.

Granularity



When do we parallelize? II

- Whether or not the problem is easy to parallelize may depend on the way you set it up.
- Taking advantage of your architecture.
- Trade off between speed up and coding time.
- Debugging and profiling may be challenging.
- You will need a good IDE, debugger, and profiler.

Example I: value function iteration

$$V(k) = \max_{k'} \{u(c) + \beta V(k')\}$$
$$c = k^{\alpha} + (1-\delta)k - k'$$

1. We have a grid of capital with 100 points, $k \in [k_1, k_2, ..., k_{100}]$.
2. We have a current guess $V^n(k)$.
3. We can send the problem:

$$\max_{k'} \{u(c) + \beta V^n(k')\}$$
$$c = k_1^{\alpha} + (1-\delta)k_1 - k'$$

   to processor 1 to get $V^{n+1}(k_1)$.
4. We can send similar problem for each k to each processor.
5. When all processors are done, we gather the $V^{n+1}(k_1)$ back.

Example II: random walk Metropolis-Hastings

- Draw $\theta \sim P(\cdot)$
- How?
  1. Given a state of the chain θ_{n-1}, we generate a proposal:

$$\theta^* = \theta_{n-1} + \lambda\varepsilon,\ \varepsilon \sim \mathcal{N}(0,1)$$

  2. We compute:

$$\alpha = \min\left\{1, \frac{P(\theta^*)}{P(\theta_{n-1})}\right\}$$

  3. We set:

$$\begin{aligned} \theta_n &= \theta^*\ w.p.\ \alpha \\ \theta_n &= \theta_{n-1}\ w.p.\ 1-\alpha \end{aligned}$$

- Problem: to generate θ^* we need to θ_{n-1}.
- No obvious fix (parallel chains violate the asymptotic properties of the chain).

The Model

Life-cycle model

- Households solve:

$$
\begin{aligned}
V(t,e,x) \quad = \quad & \max_{\{c,x'\}} \quad \frac{c^{1-\sigma}}{1-\sigma} + \beta \mathbb{E} V(t+1,e',x') \\
& s.t. \\
& c + x' \leq (1+r)x + ew \\
& \mathbb{P}(e'|e) = \Gamma(e) \\
& x' \geq 0 \\
& t \in \{1,\ldots,T\}
\end{aligned}
$$

Computing the model

1. Choose grids for assets $X = \{x_1, \ldots, x_{n_x}\}$ and shocks $E = \{e_1, \ldots, e_{n_e}\}$.

Computing the model

1. Choose grids for assets $X = \{x_1, \ldots, x_{n_x}\}$ and shocks $E = \{e_1, \ldots, e_{n_e}\}$.

2. Backwards induction:

Computing the model

1. Choose grids for assets $X = \{x_1, \ldots, x_{n_x}\}$ and shocks $E = \{e_1, \ldots, e_{n_e}\}$.

2. Backwards induction:

   2.1 For $t = T$ and every $x_i \in X$ and $e_j \in E$, solve the static problem:

$$V(t, e_j, x_i) = \max_{\{c\}} u(c) \quad s.t. \quad c \leq (1 + r)x_i + e_j w$$

Computing the model

1. Choose grids for assets $X = \{x_1, \ldots, x_{n_x}\}$ and shocks $E = \{e_1, \ldots, e_{n_e}\}$.

2. Backwards induction:

   2.1 For $t = T$ and every $x_i \in X$ and $e_j \in E$, solve the static problem:

$$V(t, e_j, x_i) = \max_{\{c\}} u(c) \quad s.t. \quad c \leq (1 + r)x_i + e_j w$$

   2.2 For $t = T - 1, \ldots, 1$, use $V(t + 1, e_j, x_i)$ to solve:

$$\begin{aligned} V(t, e_j, x_i) \quad = \quad & \max_{\{c, x' \in X\}} \quad u(c) + \beta \mathbb{E} V(t + 1, e', x') \quad s.t. \\ & c + x' \leq (1 + r)x_i + e_j w \\ & \mathbb{P}(e' \in E | e_j) = \Gamma(e_j) \end{aligned}$$

Code Structure

```
for(age = T:-1:1)
  for(ix = 1:nx)
    for(ie = 1:ne)
      VV = -10^3;
      for(ixp = 1:nx)

        expected = 0.0;
        if(age < T)
          for(iep = 1:ne)
            expected = expected + P[ie, iep]*V[age+1, ixp, iep];
          end
        end

        cons = (1+r)*xgrid[ix] + egrid[ie]*w - xgrid[ixp];
        utility = (cons^(1-ssigma))/(1-ssigma) + bbeta*expected;

        if(cons <= 0)
          utility = -10^5;
        end
        if(utility >= VV)
          VV = utility;
        end
      end
      V[age, ix, ie] = VV;
    end
  end
end
```

In parallel

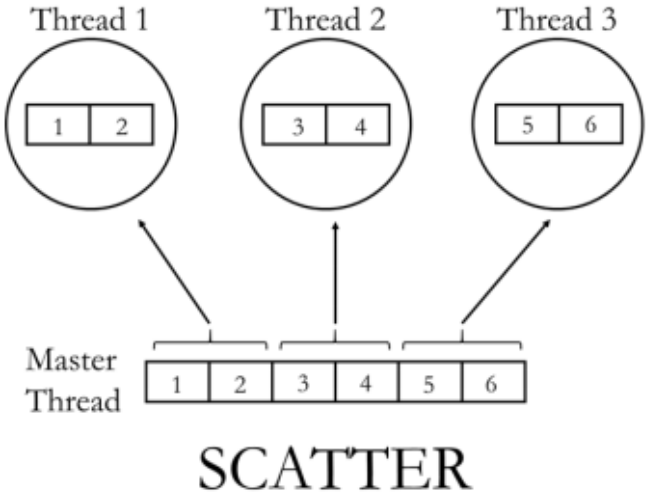
1. Set $t = T$.
2. Given t, the computation of $V(t, e_j, x_i)$ is independent of the computation of $V(t, e_{j'}, x_{i'})$, for $i \neq i'$, $j \neq j'$.
3. One processor can compute $V(t, e_j, x_i)$ while another processor computes $V(t, e_{j'}, x_{i'})$.
4. When the different processors are done at computing $V(t, e_j, x_i)$, $\forall x_i \in X$ and $\forall e_j \in E$, set $t = t - 1$.
5. Go to 1.

Note that the problem is not parallelizable on t. The computation of $V(t, e, x)$ depends on $V(t + 1, e, x)$!

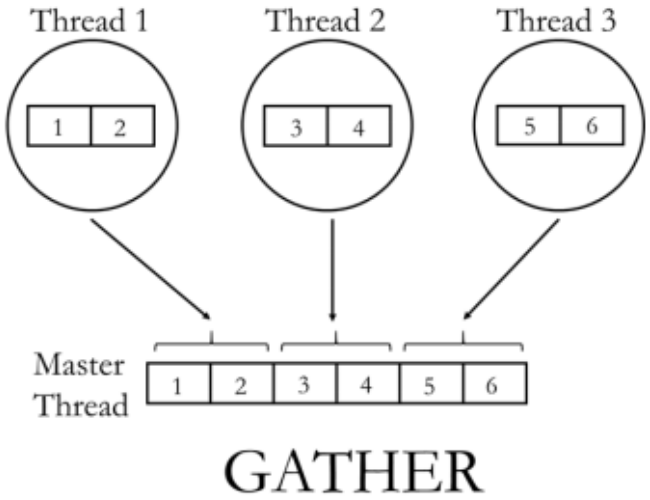
Computational features of the model

1. The simplest life-cycle model.
2. Three state variables:
   - 2.1 Age.
   - 2.2 Assets.
   - 2.3 Productivity shock.
3. Parallelizable only on assets and shock, not on age.
4. May become infeasible to estimate:
   - 4.1 With more state variables:
     - Health.
     - Housing.
     - Money.
     - Different assets.
   - 4.2 If embedded in a general equilibrium.

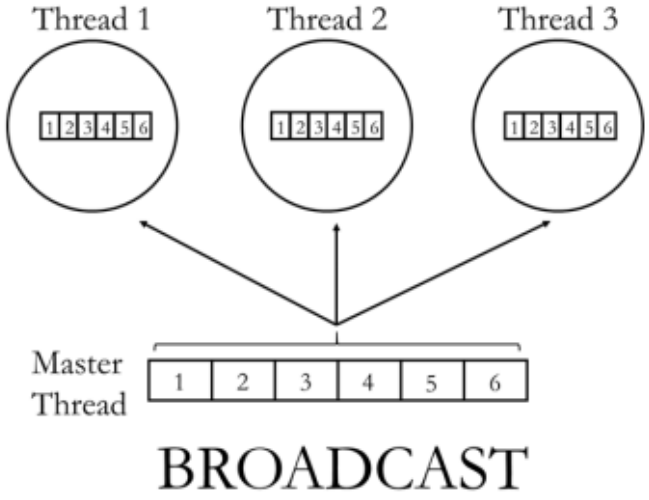
In parallel



SCATTER

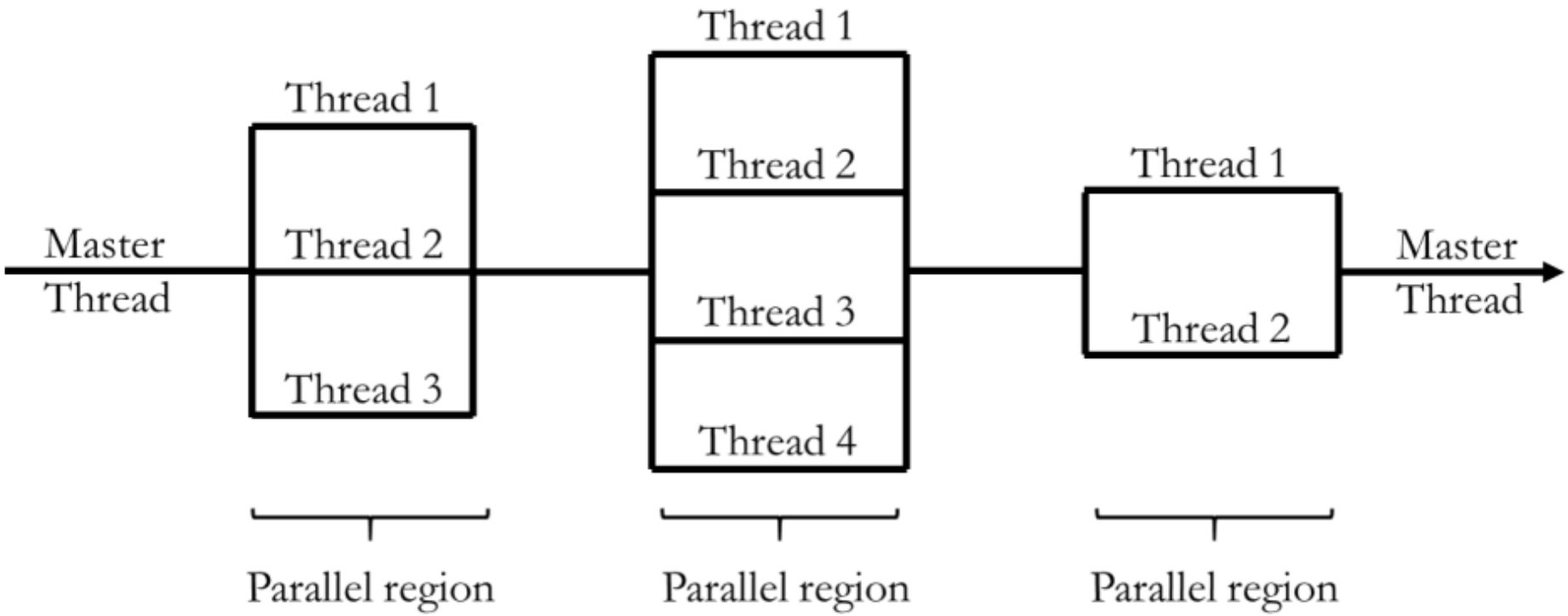


GATHER



BROADCAST

Parallel execution of the code



Many workers instead of one

Figure 1: 1 Core Used for Computation

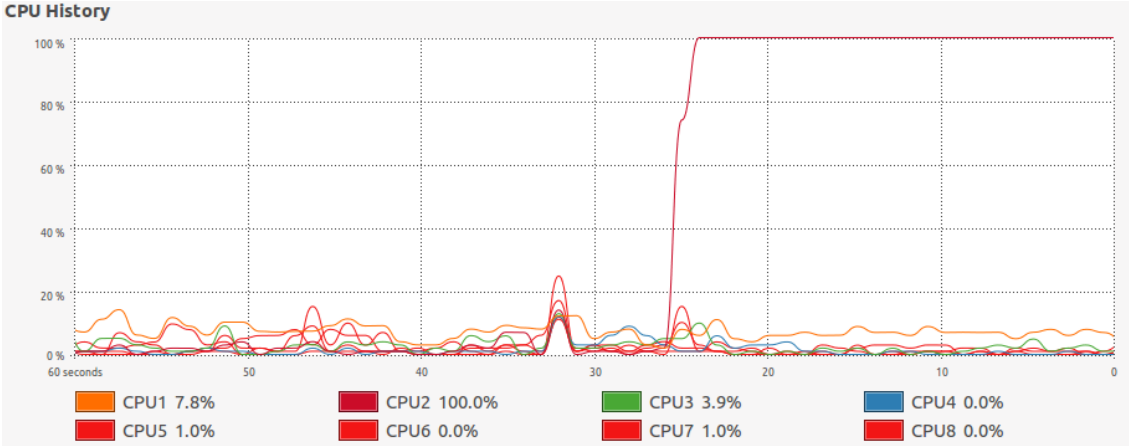
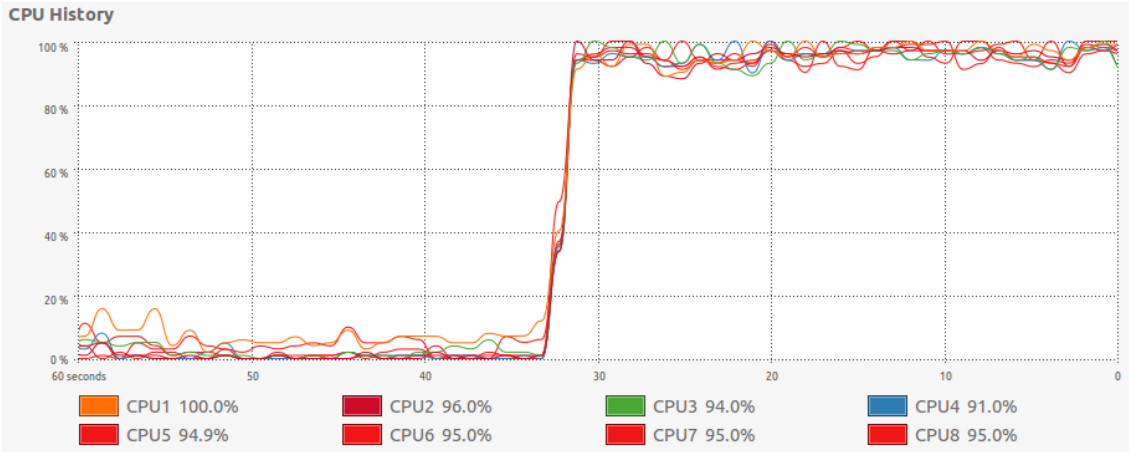


Figure 2: 8 Cores Used for Computation



Parallelization limits

Costs of parallelization

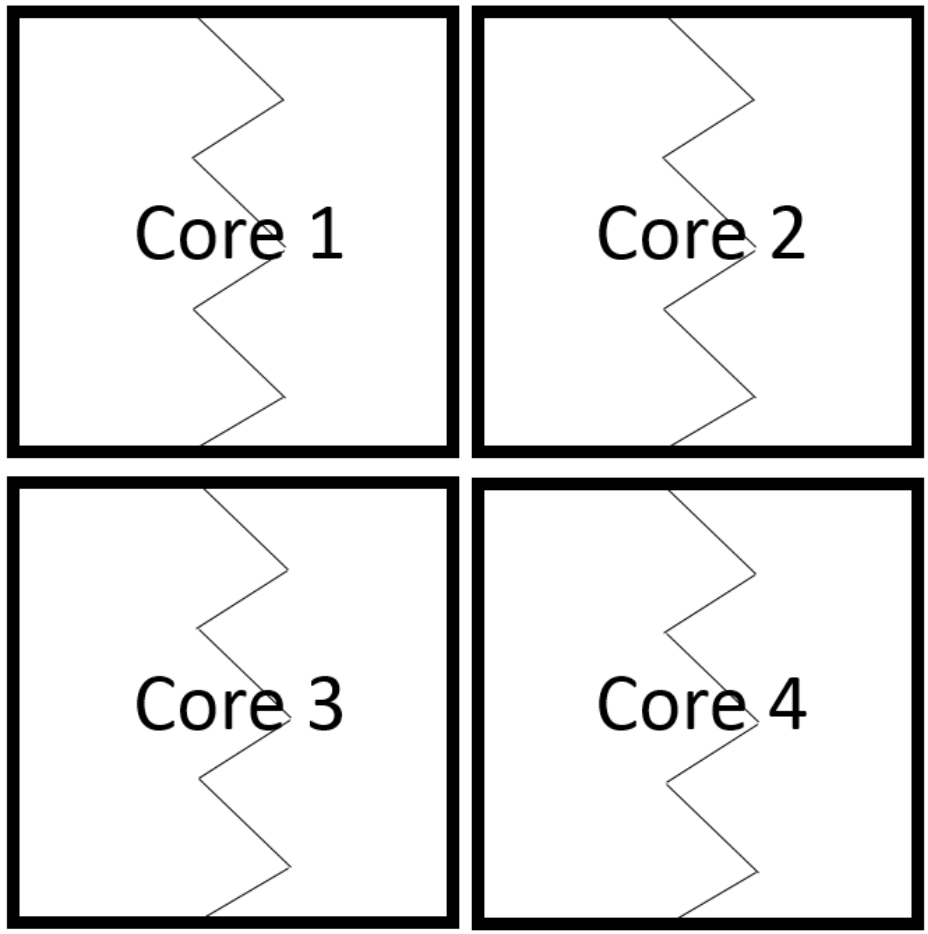
- Amdahl's Law: the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

- Costs:
  - Starting a thread or a process/worker.
  - Transferring shared data to workers.
  - Synchronizing.

- Load imbalance: for large machines, it is often difficult to use more than 10% of its computing power.

Parallelization limits on a laptop

- Newest processors have plenty of processor.
- For example, for the examples in these slides, we used 4 physical cores + 4 virtual cores = 8 logical cores.



Multi-core processors

Intel® Core™ X-Series Processors

- High-performance desktops
- First 18-core processor
- Extreme gaming, mega-tasking, and high-end content creation

Know your limits!

- Spend some time getting to know you laptop's limits and the problem to parallelize.
- In our life-cycle problem with many grid points, parallelization improves performance almost linearly, up to the number of physical cores.
- Parallelizing over different threads of the same physical core does not improve speed if each thread uses 100% of core capacity.
- For computationally heavy problems, adding more threads than cores available may even reduce performance.
- But: Your laptop is not the limit

Clusters: Some terminology

- Cluster: Interconnected set of nodes.
- Node: "Computer", single component of a cluster.
  - Nodes can fulfill different functions, e.g. compute nodes, login nodes, etc.
  - Compute nodes offer processors, RAM, disk space, accelerators (e.g. GPU) etc.
- Core: Part of a processor that does the computations. Can consist of one or two hardware threads.

Clusters: Some terminology

- Processor: Consists of multiple cores + other components.
  - Sometimes referred to as a socket (= slot on the motherboard that hosts the processor)
- CPU: General context: Same as a processor. Slurm context: Consumable resource offered by a node, can refer to a socket, a core, or a hardware thread, based on the Slurm configuration.

Clusters at Princeton

- Princeton has freely accessible clusters: Nobel, Adroit, Della, Tigressdata, Tiger, Stellar, Traverse.
- Most relevant for Economists: Nobel, Adroit, Della.
- Nobel requires no approval, Adroit requires an application, Della a proposal.
- Job scheduling works through SLURM (=Simple Linux Utility for Resource Management)
- SLURM requires a short bash script with the description of the job and resources required.
- Let's take a look!

AWS

- Alternative: Amazon Web Services - EC2 at **`https://aws.amazon.com/ec2/`**:
  - Almost as big as you want!
  - Replace a large initial capital cost for a variable cost (use-as-needed).
  - Check: **`https://aws.amazon.com/ec2/pricing/`**
    - 8 processors with 32Gb, general purpose: $0.332 per hour.
    - 64 processors with 256Gb, compute optimized: $3.20 per hour.

Programming modes I

- More common in economics.
  1. Packages/libraries/toolboxes within languages:
     - 1.1 **`Julia`**.
     - 1.2 **`Python`**.
     - 1.3 **`R`**.
     - 1.4 **`Matlab`**.
  2. Explicit parallelization:
     - 2.1 OpenMP.
     - 2.2 MPI.
     - 2.3 GPU programming: CUDA, OpenCL, and OpenACC.

Two ways of parallelizing

1. `for` loop:
    - Adding a statement before a `for` loop that wants to be parallelized.
2. `Map` and `reduce`:
    - Create a function that depends on the state variables over which the problem can be parallelized:
        - In our example, we have to create a function that computes the value function for a given set of state variables.
    - `Map` computes in parallel the function at a vector of states.
    - `Reduce` combines the values returned by `map` in the desired way.

Julia

Parallelization in Julia - for loops

- Parallelization of for loops is worth for “small tasks.”
- “Small task” == “few computations on each parallel iteration”:
  - Few control variables.
  - Few grid points on control variables.
  - Our model is a “small task.”

Parallelization in Julia - for loops

1. Load distributed module

```
using Distributed
```

2. Set number of workers:

```
addprocs(5)
```

3. Remove workers:

```
rmprocs(2,3,5)
```

4. Checking workers:

```
workers()
```

Parallelization in Julia - for loops

1. Load distributed and SharedArrays modules

```
using Distributed
using SharedArrays
```

2. Declare variables used inside the parallel for loop that are not modified inside parallel iterations to be `@everywhere`:

```
@everywhere nx = 1500;
```

3. Declare variables used inside the parallel for loop that are modified inside parallel iterations as `SharedArray`:

```
tempV = SharedArray{Float64}(ne*nx);
```

Parallelization in Julia - for loops

4. Data structure of state and exogenous variables

```
@everywhere struct modelState
  ind::Int64
  ne::Int64
  nx::Int64
  T::Int64
  age::Int64
  P::Array{Float64,2}
  xgrid::Vector{Float64}
  egrid::Vector{Float64}
  ssigma::Float64
  bbeta::Float64
  V::Array{Float64,2}
  w::Float64
  r::Float64
end
```

Parallelization in Julia - for loops

5. Define a function that computes value function for a given state:

```
@everywhere function value(currentState::modelState)

    ind    = currentState.ind
    age    = currentState.age
        # ...
        VV     = -10.0^3;

        ixpopt = 0;

        for ixp = 1:nx
            # ...
        end

        return(VV);

end
```

Parallelization in Julia - for loops

6. For paralellizing a for loop, add **`@distributed`** before the **`for`** statement:

```
@distributed for ind = 1:(ne*nx)
    # ...
end
```

7. To synchronize before the code continues its execution, add **`@sync`** before the **`@distributed for`** statement:

```
@sync @distributed for ind = 1:(ne*nx)
    # ...
end
```

Parallelization in Julia - for loops

- Choose appropriately the dimension(s) to parallelize:

```
nx = 350;
ne = 9;
for(ie = 1:ne)
  @sync @distributed for(ix = 1:nx)
    # ...
  end
end
```

```
nx = 350;
ne = 9;
for(ix = 1:nx)
  @sync @distributed for(ie = 1:ne)
    # ...
  end
end
```

- The first one is much faster, as there is less communication.

Parallelization in Julia - for loops

- OR convert the problem so all state variables are computed by iterating over a one-dimensional loop:

```
@sync @distributed for ind = 1:(ne*nx)

        ix      = convert(Int,ceil(ind/ne));
        ie      = convert(Int,floor(mod(ind-0.05, ne))+1);
    # ...

end
```

- Communication time is minimized!

Parallelization in Julia - Performance

Figure 3: Julia - 1 core used for computation

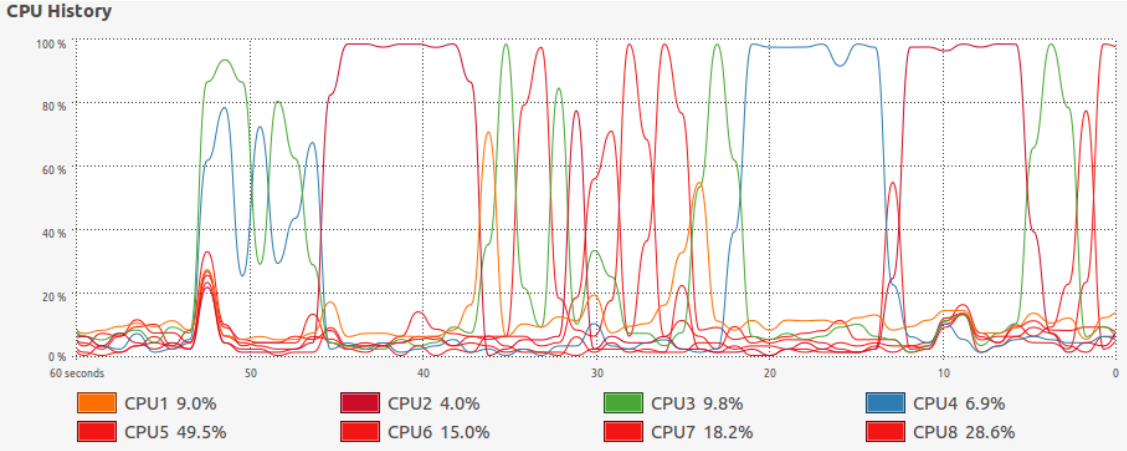
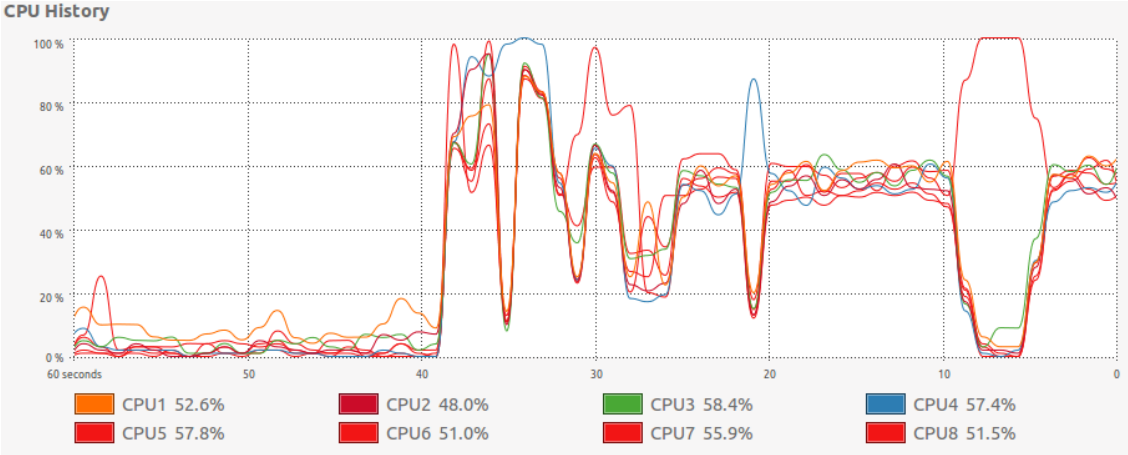


Figure 4: Julia - 8 cores used for computation



Parallelization in Julia - Performance

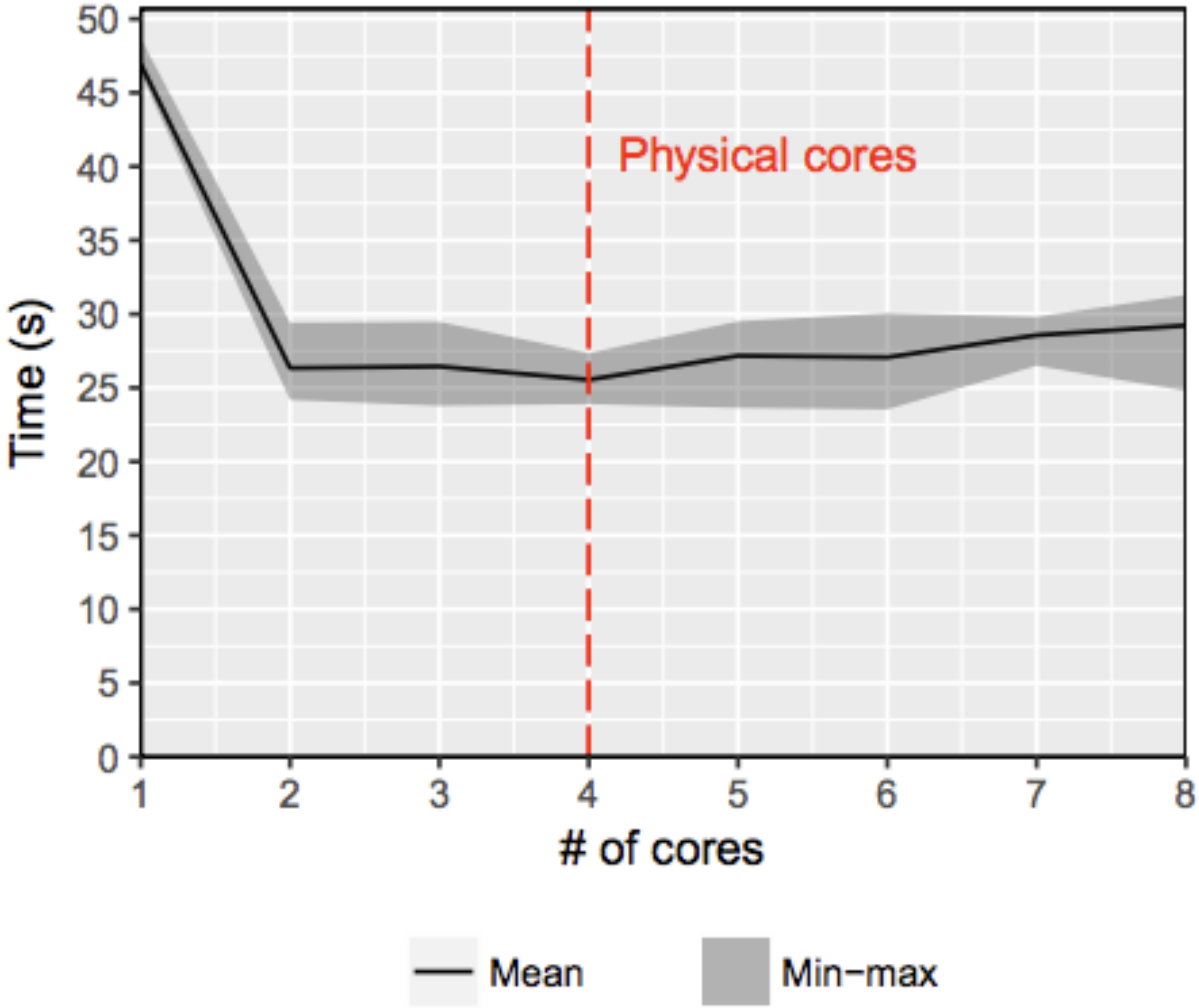


Figure 5: Computing time (s)

Parallelization in Julia - for loops

- Speed decreases with the number of global variables used.
- Very sensitive to the use of large `SharedArray` objects.
- Can be faster without paralellization than with large shared objects.
- See code on github

Parallelization in Julia - Map

- Problems with more computations per iteration.
- Value function/life-cycle models with more computations per state:
  - Many control variables.
  - Discrete choice (marry-not marry, accept-reject work offer, default-repay, etc.).
- If problem is “small”, using map for parallelization is slower.

Parallelization in Julia - Map

1. Most of the code as as in the for case.

2. The function `pmap(f,s)` computes the function `f` at every element of `s` in parallel:

```
for(age = T:-1:1)
    pars = [modelState(ix, age, ..., w, r) for ix in 1:nx];
    s = pmap(value,pars);
    for(ind = 1:nx)
        V[age, ix, ie] = s[ix];
    end
end
```

Parallelization in Julia - Performance

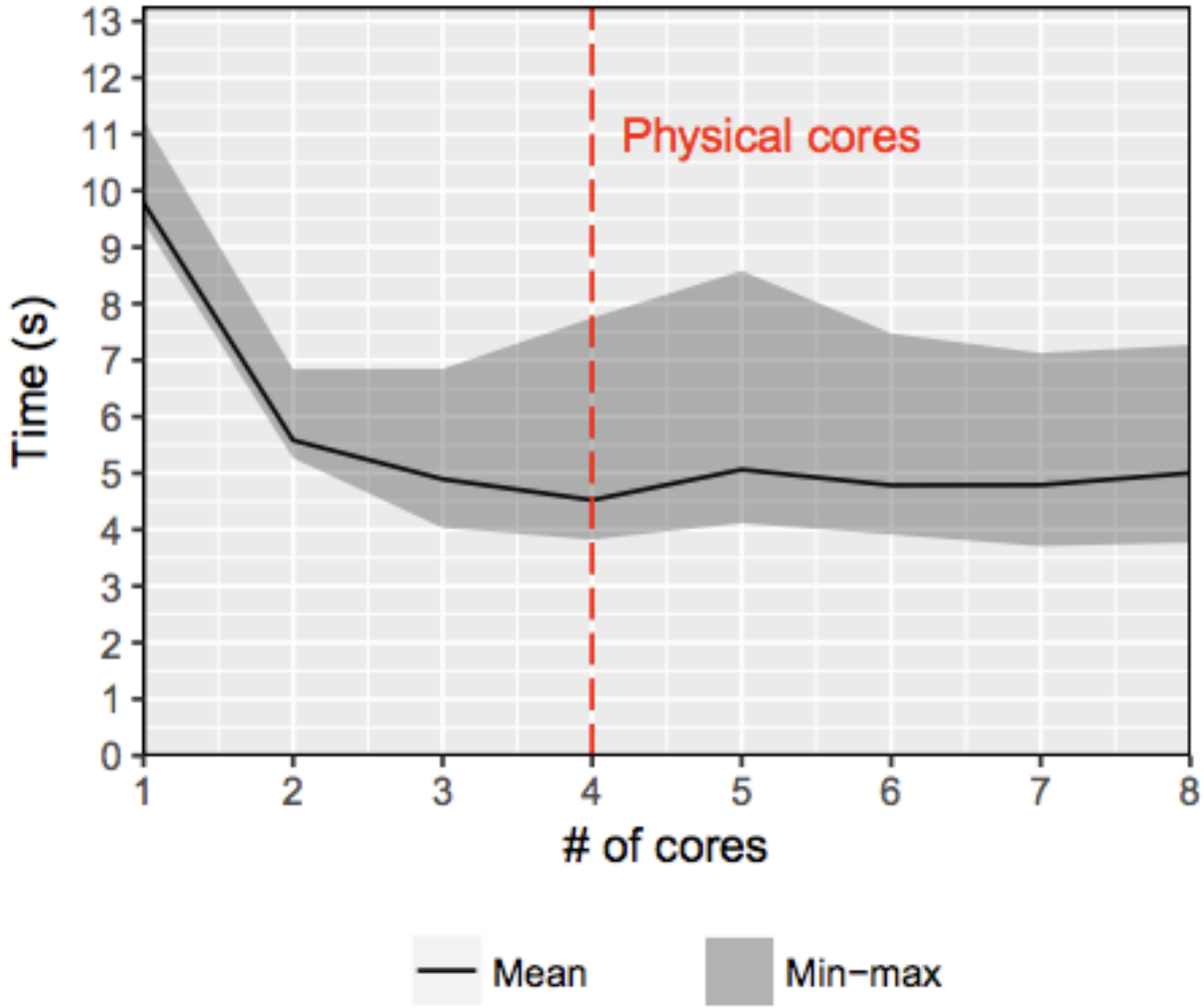


Figure 6: Computing time (s)

Parallelization in Julia - Final advice

- Assess size of problem, but usually problem grows as paper evolves!
- Wrapping value function computation for every state might significantly increase speed (even more than parallelizing).

Python

Parallelization in Python - Map

1. Use `joblib` package

```
from joblib import Parallel, delayed
import multiprocessing
```

2. Define a parameter structure for value function computation:

```
class modelState(object):
    def __init__(self, age, ix, ...):
        self.age      = age
        self.ix    = ix
        # ...
```

Parallelization in Python

3. Define a function that computes value for a given input **`states`** of type **`modelState`**:

```
def value_func(states):
    nx = states.nx
    age = states.age
    # ...
    VV = math.pow(-10, 3)
    for ixp in range(0,nx):
        # ...
    return[VV];
```

Parallelization in Python

4. The function `Parallel`:

```
results = Parallel(n_jobs=num_cores)(delayed(value_func)
    (modelState(ix, age, ..., w, r)) for ind in range(0,nx*ne))
```

maps the function `value_func` at every element of `modelState(ix, age, ..., w, r)` in parallel using `num_cores` cores.

Parallelization in Python

5. Life-cycle model:

```
for age in reversed(range(0,T)):
    results = Parallel(n_jobs=num_cores)(delayed(value_func)
         (modelState(ix, age, ..., w, r)) for ix in range(0,nx))
    for ix in range(0,nx):
        V[age, ix] = results[ix][0];
```

Parallelization in Python - Performance

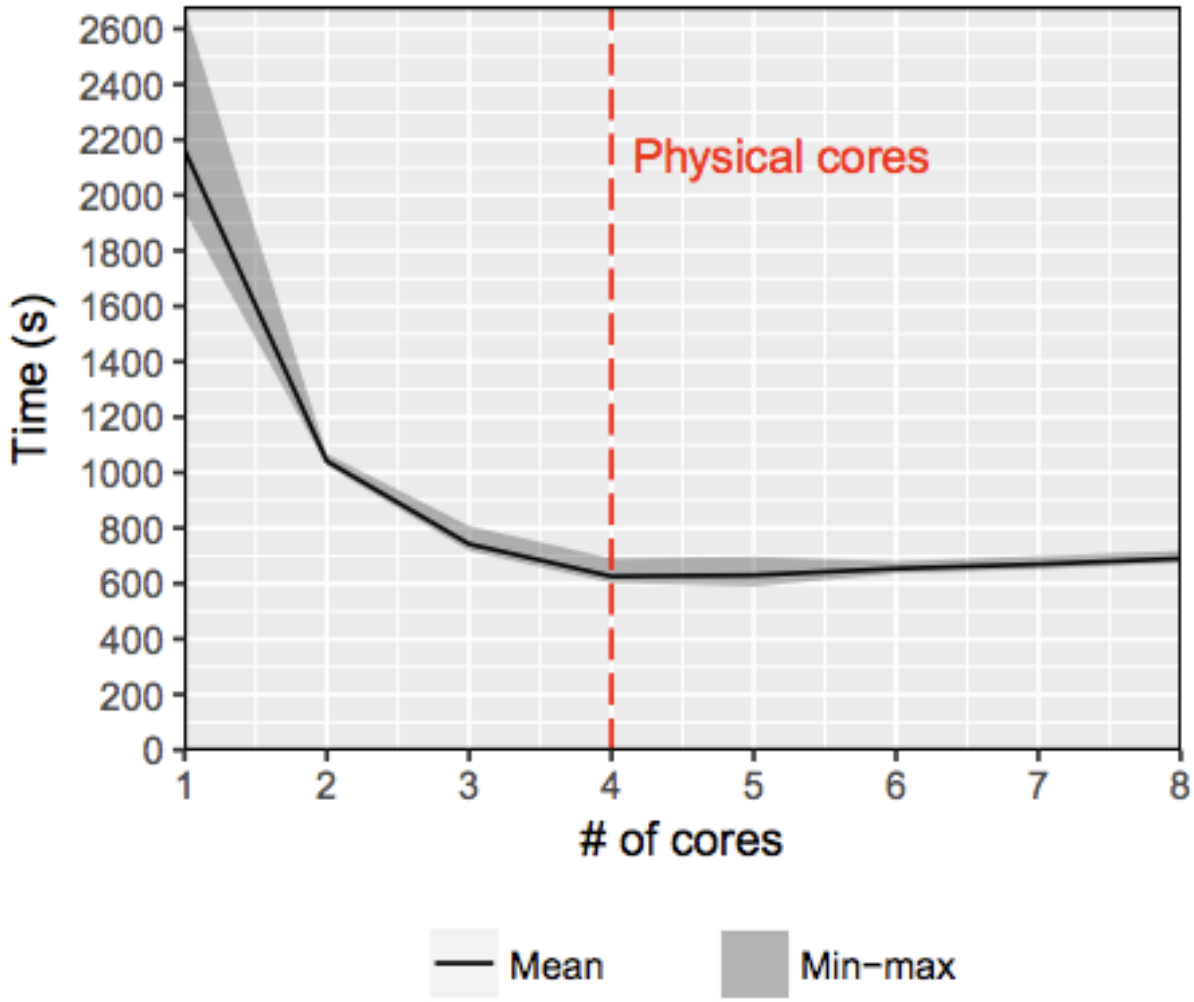


Figure 7: Computing time (s)

R

Parallelization in R - Map

1. Use package **parallel**:

```
library("parallel")
```

2. Create the structure of parameters for the function that computes the value for a given state as a list:

```
states = lapply(1:nx, function(x) list(age=age,ix=x, ...,r=r))
```

Parallelization in R

3. Create the function that computes the value for a given state:

```
value = function(x){
    age   = x$age
    ix    = x$ix
    ...
    VV = -10^3;
    for(ixp in 1:nx){
        # ...
    }
    return(VV);
}
```

Parallelization in R

4. Define the cluster with desired number of cores:

```
cl <- makeCluster(no_cores)
```

5. Use function `parLapply(cl, states, value)` to compute `value` at every state in `states` with `cl` cores:

```
for(age in T:1){
  states = lapply(1:nx, ...)
  for(ix in 1:nx){
    V[age, ix] = s[[ix]][1]
  }
}
```

Parallelization in R - Performance

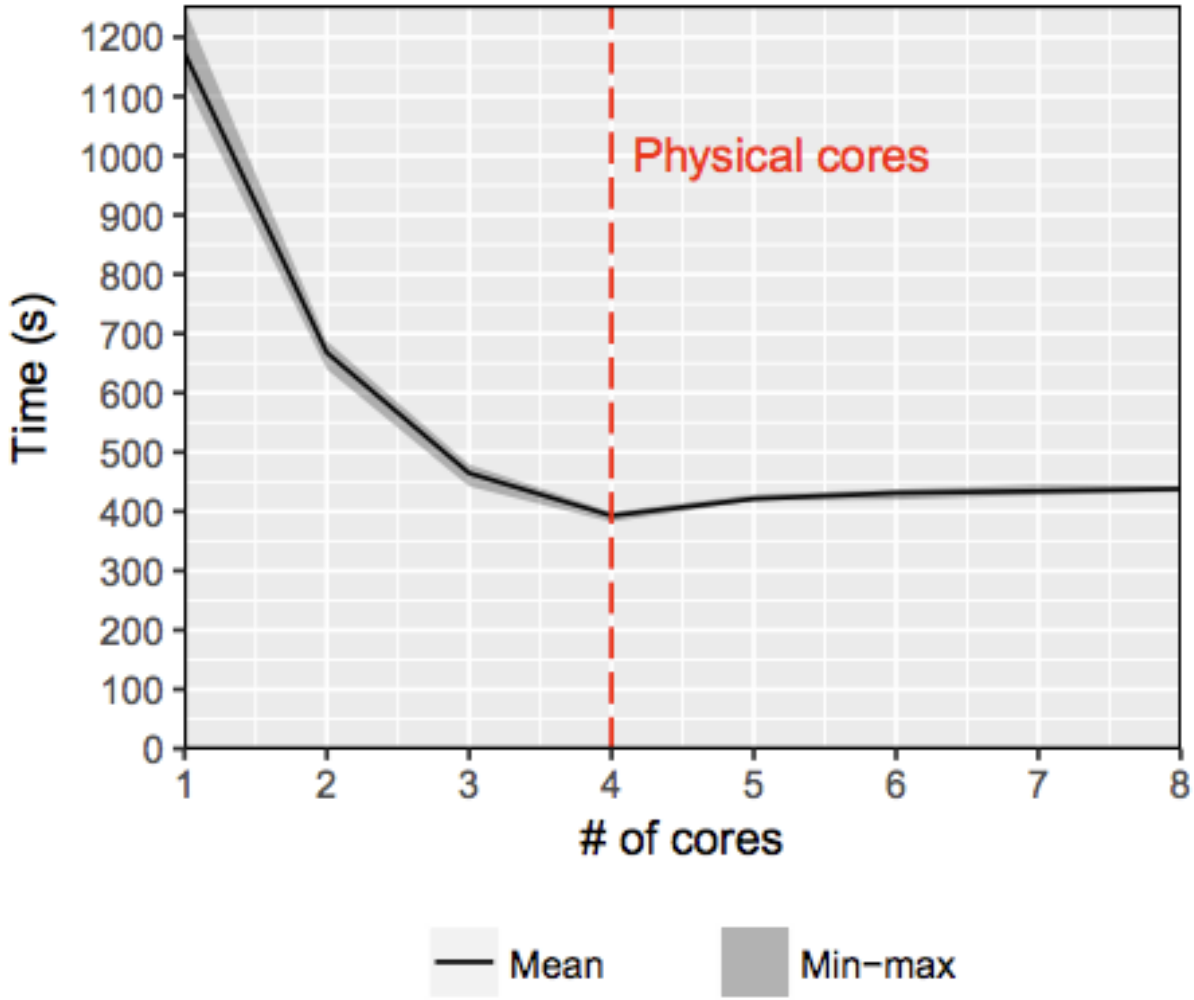


Figure 8: Computing time (s)

Matlab

Parallelization in Matlab - for loop

Using the **parallel toolbox**:

1. Initialize number of workers with **parpool()**:

```
parpool(6)
```

2. Replace the **for** loop with **parfor**:

```
for age = T:-1:1
    parfor ie = 1:1:ne
        % ...
    end
end
```

Parallelization in Matlab - Performance

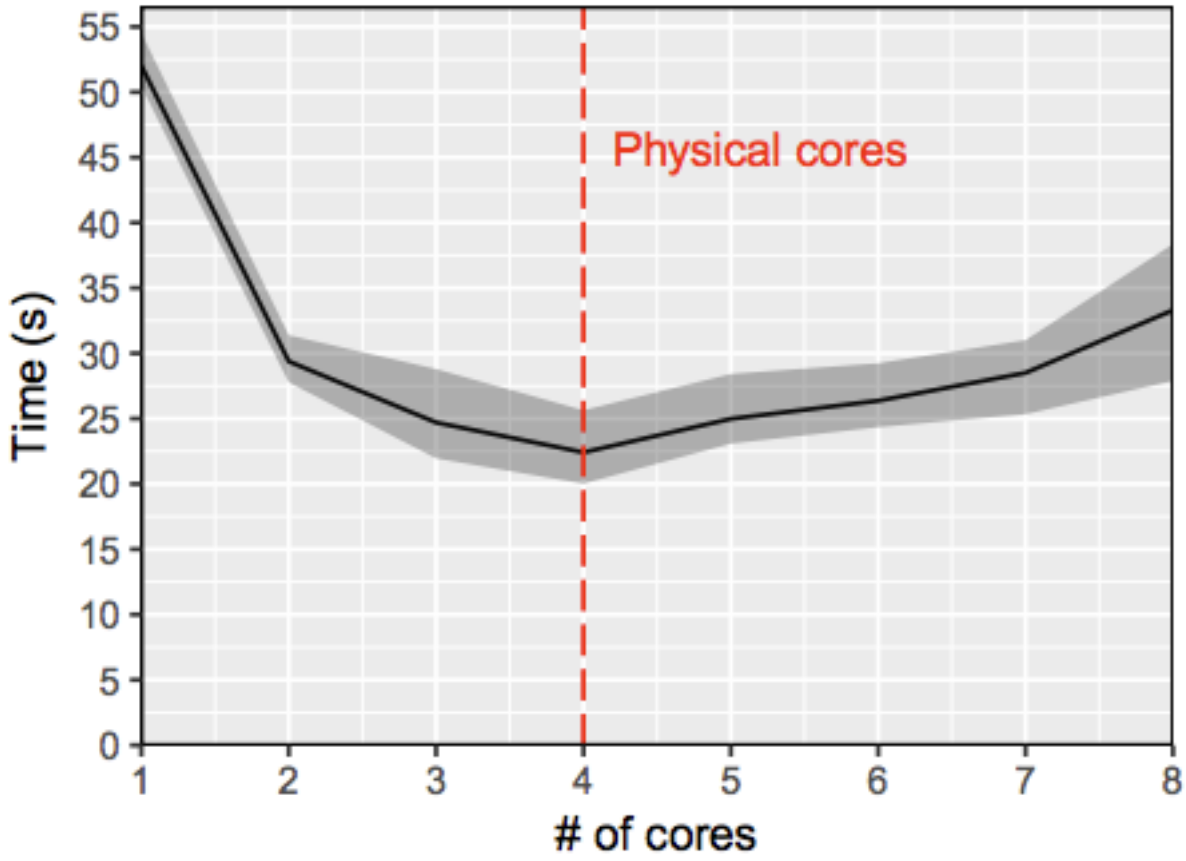


Figure 9: Computing time (s)

Parallelization in Matlab

- Extremely easy.
- Also simple to extend to GPU.
- There is no free lunch $\Longrightarrow$ very poor performance.

OpenMP

OpenMP I

- Open specifications for multi-processing.
- It has been around for two decades. Current version 4.5.
- Official web page: `http://openmp.org/wp/`
- Tutorial: `https://computing.llnl.gov/tutorials/openMP/`
- Using OpenMP: Portable Shared Memory Parallel Programming by Barbara Chapman, Gabriele Jost, and Ruud van der Pas.
- Fast to learn, reduced set of instructions, easy to code, but you need to worry about contention and cache coherence.

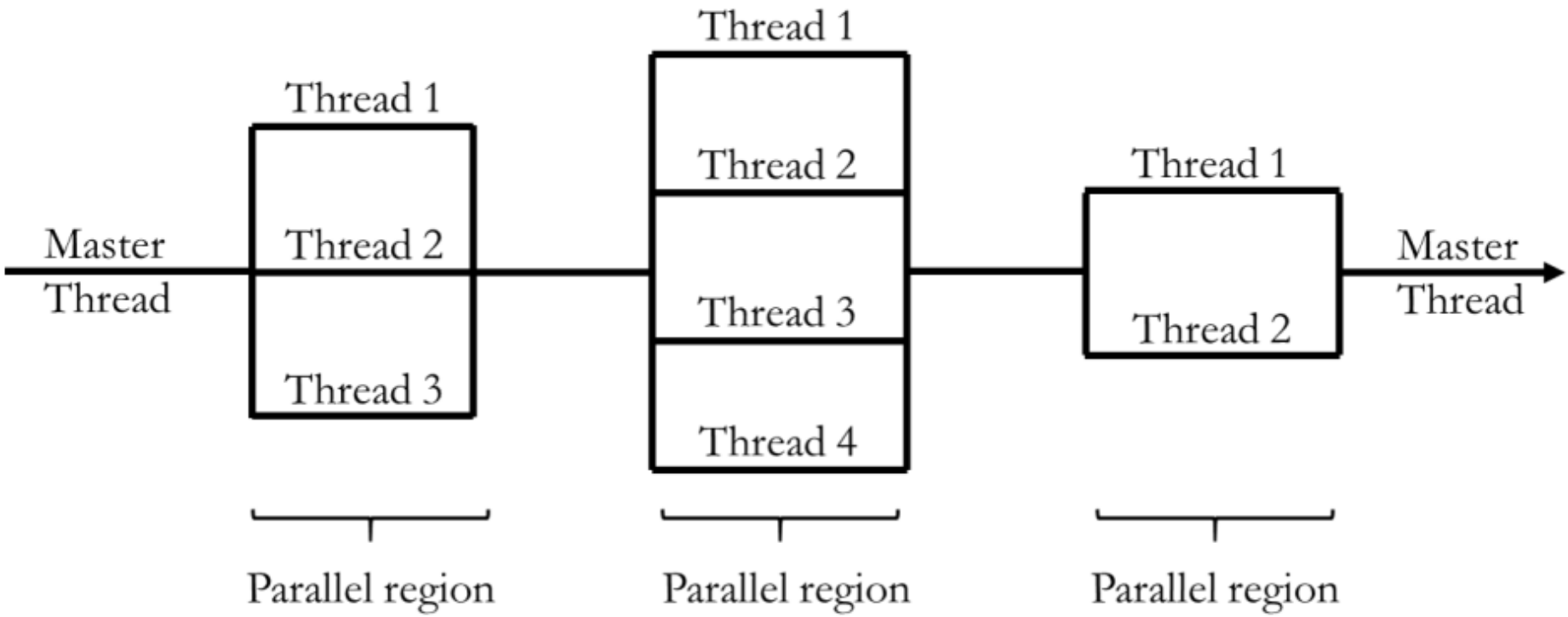
OpenMP II

- API for multi-processor/core, shared memory machines defined by a group of major computer hardware and software vendors.
- **`C++ and Fortran`**. Extensions to other languages.
- For example, you can have **`OpenMP`** in **`Mex`** files in **`Matlab`**.
- Supported by major compilers (**`GCC`**) and IDEs (**`Clion`**).
- Thus, it is usually straightforward to start working with it.

OpenMP III

- Multithreading with fork-join.
- Rule of thumb: One thread per processor.
- Job of the user to remove dependencies and syncronize data.
- Heap and stack (`LIFO`).
- Race conditions: you can impose fence conditions and/or make some data private to the thread.
- Remember: synchronization is expensive and loops suffer from overheads.

Fork-join


Master
Thread
Thread 1
Thread 2
Thread 3
Thread 1
Thread 2
Thread 3
Thread 4
Thread 1
Thread 2
Master
Thread
Parallel region
Parallel region
Parallel region

Parallelization in C++ using OpenMP

1. At compilation, add flag:

```
-fopenmp
```

2. Set environmental variable `OMP_NUM_THREADS`:

```
export OMP_NUM_THREADS=32
```

3. Add line before loop:

```
#pragma omp parallel for shared(V, ...) private(VV, ...)
   for(int ix=0; ix<nx; ix++){
      // ...
      }
```

4. We can always recompile without the flag and compiler directives are ignored.

5. Most implementations (although not the standard!) allow for

Parallelization in C++ using OpenMP - Performance

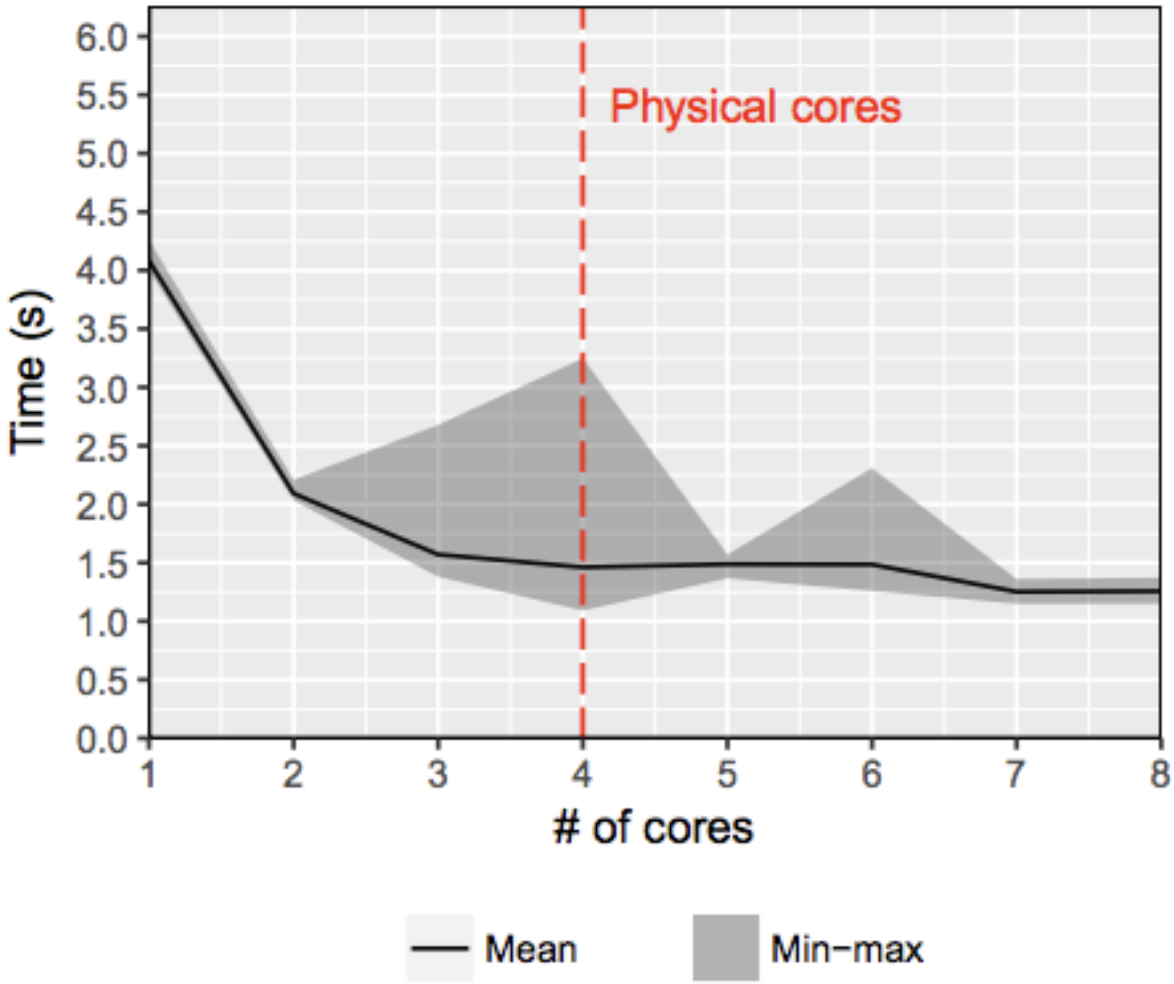


Figure 10: Computing time (s)

Parallelization in Rcpp using OpenMP

1. Write your code in C++, adding the parallelization statement

```
#pragma omp parallel for shared(...) private(...)
```

2. In the C++ code, add the following line to any function that you want to import from R:

```
// [[Rcpp::export]]
```

3. In R, load the `Rcpp` package:

```
library("Rcpp")
```

Parallelization in Rcpp using OpenMP

4. Set the environmental variable `OMP_NUM_THREADS` using the `Sys.setenv()` function:

```
Sys.setenv("OMP_NUM_THREADS"="8")
```

5. Add the `-fopenmp` flag using `Sys.setenv()` function:

```
Sys.setenv("PKG_CXXFLAGS"=" -fopenmp")
```

6. Compile and import using `sourceCpp`:

```
sourceCpp("my_file.cpp")
```

MPI

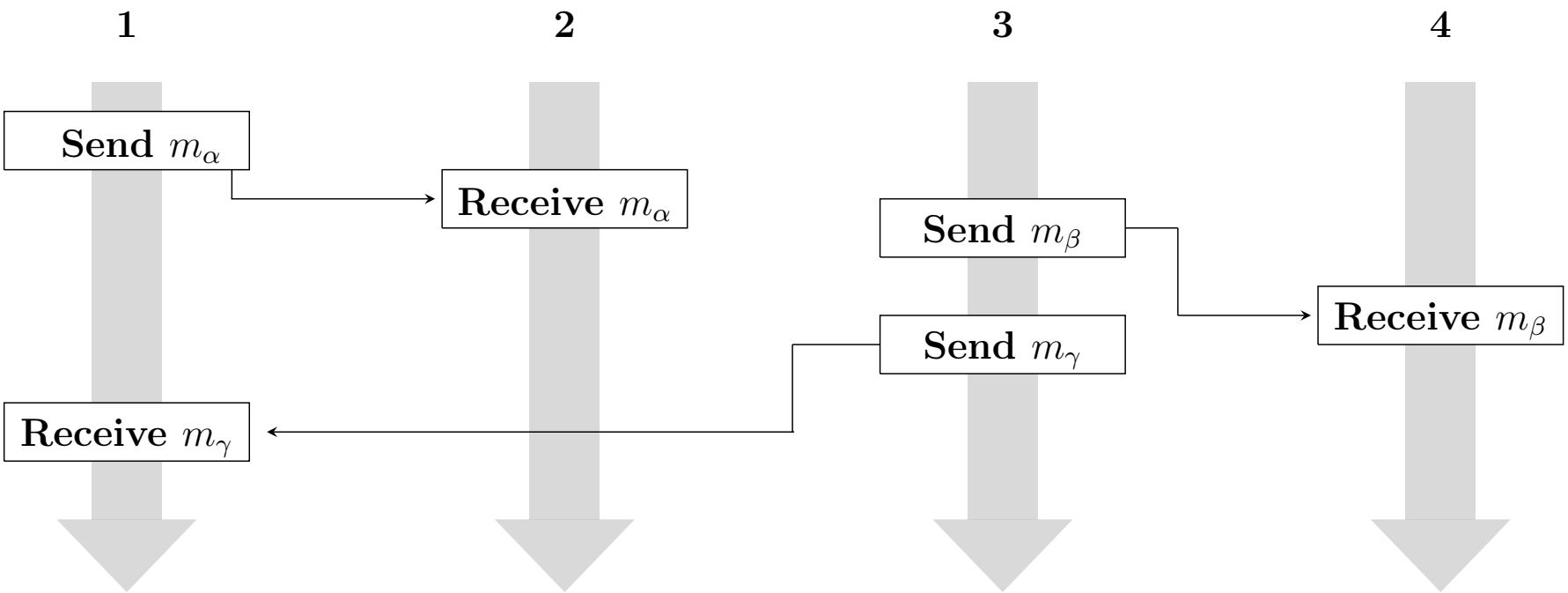
MPI I

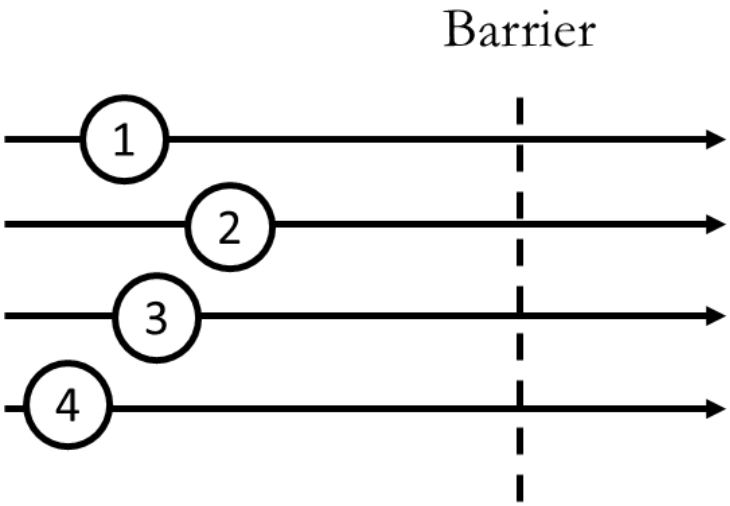
- Message Passing Interface (`MPI`) is a standardized and portable message-passing system based on the consensus of the MPI Forum.
- Official web page (and for downloads): `http://www.open-mpi.org/`
- Tutorial: `https://computing.llnl.gov/tutorials/mpi/`
- A couple of references:

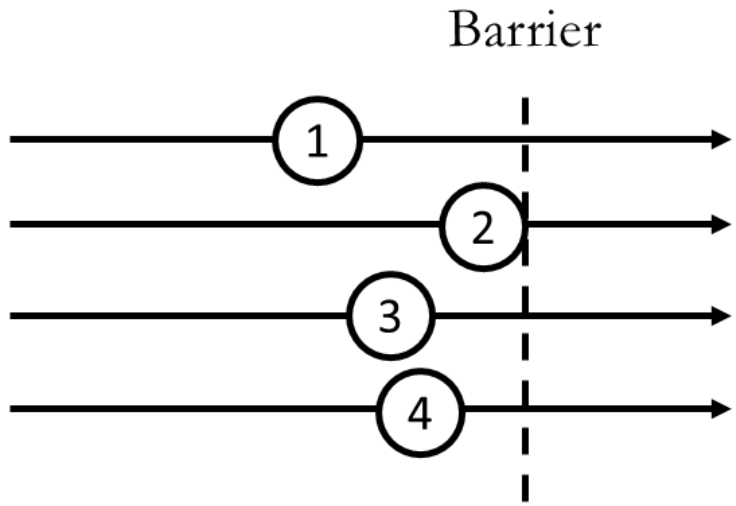
1. Using MPI : Portable Parallel Programming with the Message Passing Interface (2nd edition) by William Gropp, Ewing L. Lusk, and Anthony Skjellum.
2. MPI: The Complete Reference - Volumes 1 and 2, by several authors.

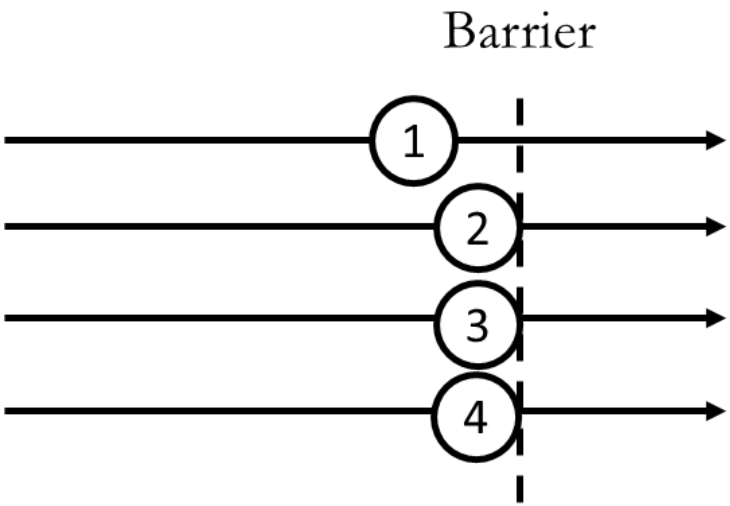
MPI II

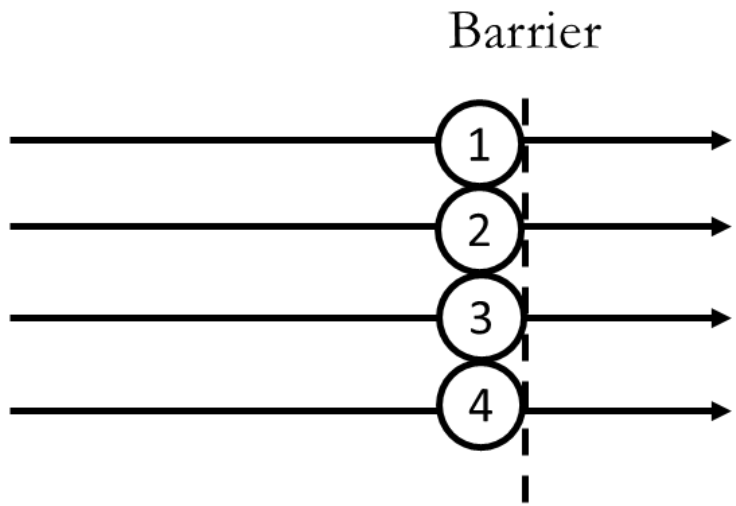
- `MPI` is organized as a library performed with routine calls.
- Bindings for `C++ and Fortran`. Also for `Python`, `Julia`, `R`, and other languages.
- For example, you can have `MPI` in `Mex` files in `Matlab`.
- Harder to learn (`MPI 3.0` standard has more than 440 routines) and code, but extremely powerful $\Rightarrow$ used for state-of-the-art computations.
- Multiple processes (thread with its own controller).
- Thus, better for coarse parallelization.


1
2
3
4
Send m_α
Receive m_α
Send m_β
Receive m_β
Send m_γ
Receive m_γ


Barrier
1
2
3
4


Barrier
1
2
3
4


Barrier
1
2
3
4


Barrier
1
2
3
4

MPI III

- Invoked with a compiler wrapper

```
mpic++ -o ClassMPI ClassMPI.cpp
```

- Plenty of libraries (`PLAPACK`, `Boost.MPI`).
- Parallel I/O features.

Example code

```
#include "mpi.h"
#include <iostream>
int main( int argc, char *argv[] ){
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM\_WORLD.Get\_rank();
    size = MPI::COMM\_WORLD.Get\_size();
    std::cout<< "I am " << rank << " of " << size << \qquad "n";
    MPI::Finalize();
    return 0;
    }
```

Routines

- Communication:
  1. Send and receive: between two processors.
  2. Broadcast, scatter, and gather data on all processors.
  3. Compute and move (sum, product, max of, ...) data on many processors.
- Synchronization.
- Enquiries:
  1. How many processes?
  2. Which process is this one?
  3. Are all messages here?

MPI derived types

- `MPI` predefines its primitive data types:
  1. `MPI_CHAR`
  2. `MPI_DOUBLE_PRECISION`
  3. `MPI_C_DOUBLE_COMPLEX`
- Also for structs and vectors.
- Particularly important for top performance.

Parallelization in C++ using MPI - Performance

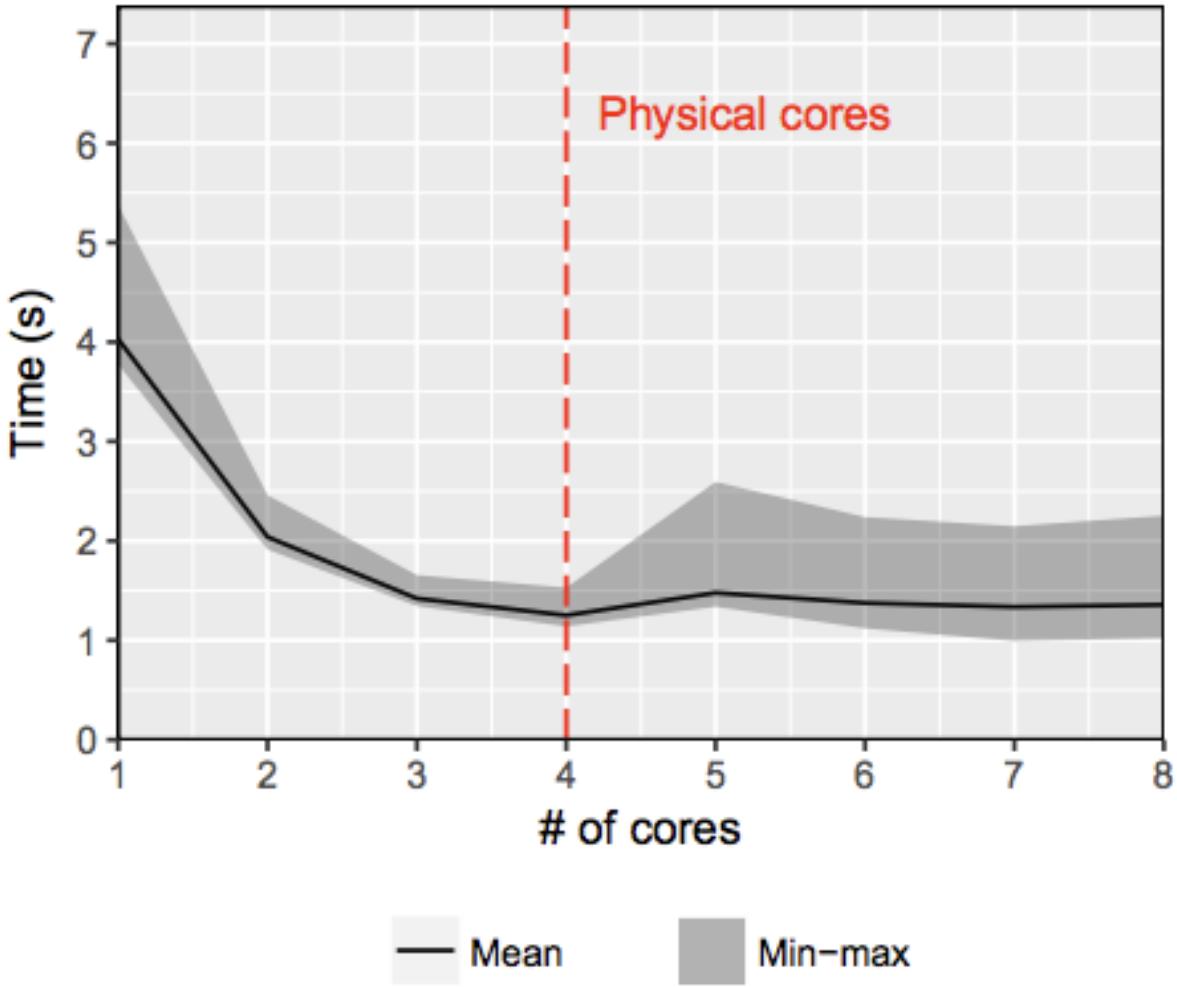


Figure 11: Computing time (s)

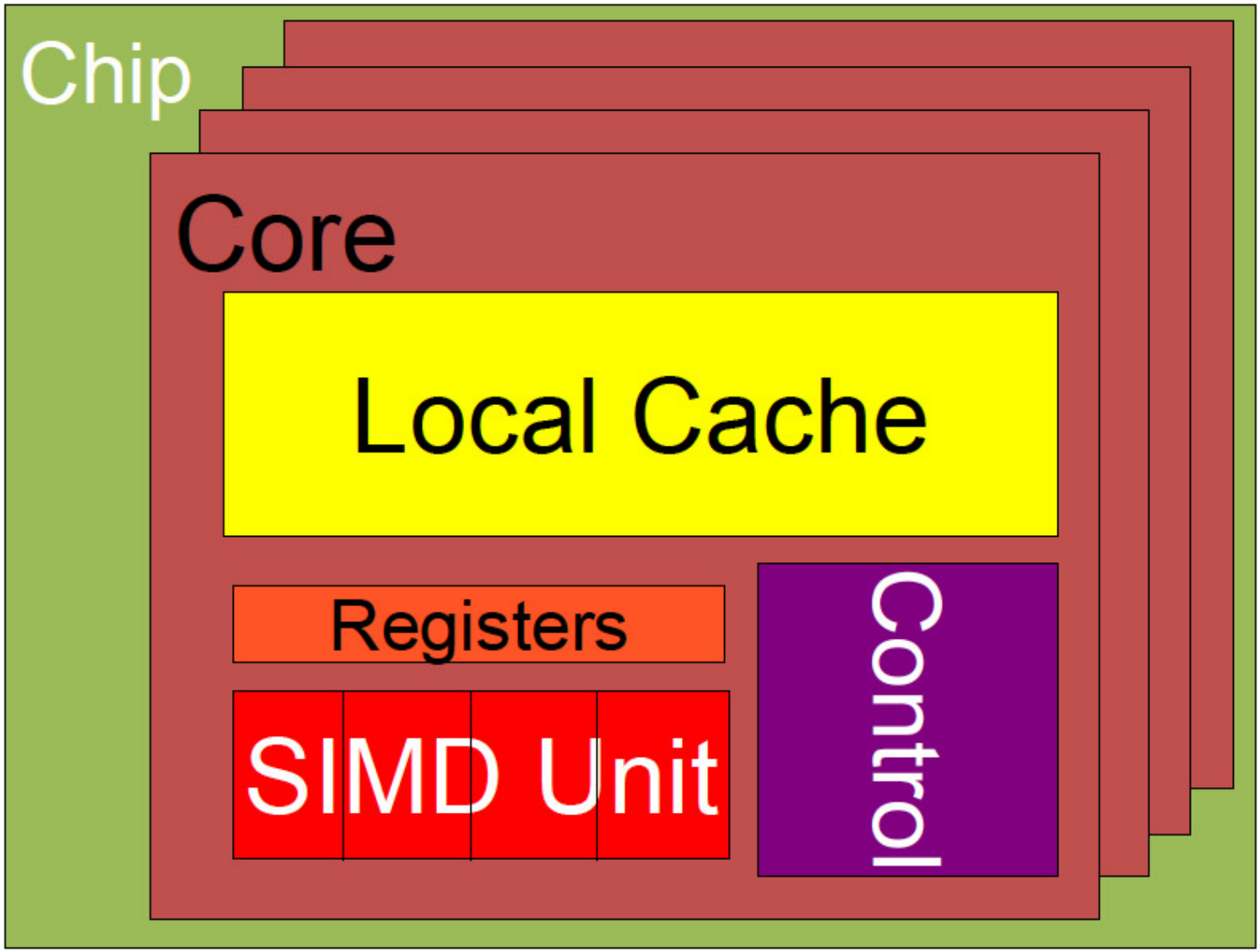
GPUs

Big difference

- Latency: amount of time required to complete a unit of work.
- Throughput: amount of work completed per unit of time.
- Latency devices: CPU cores.
- Throughput devices: GPU cores.
- Intermediate: Coprocessors.
- Nature of your application?

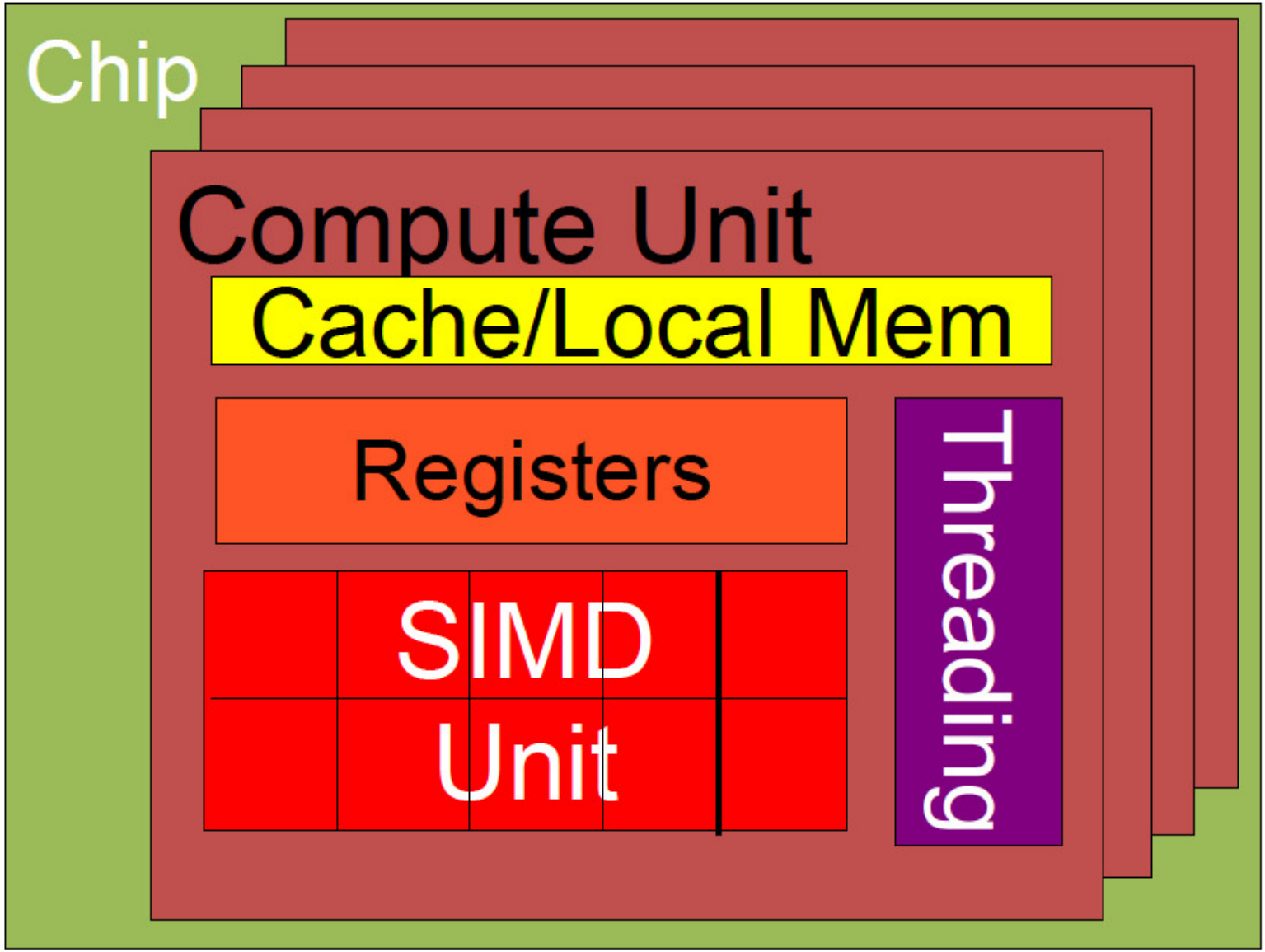
CPU

Latency Oriented Cores



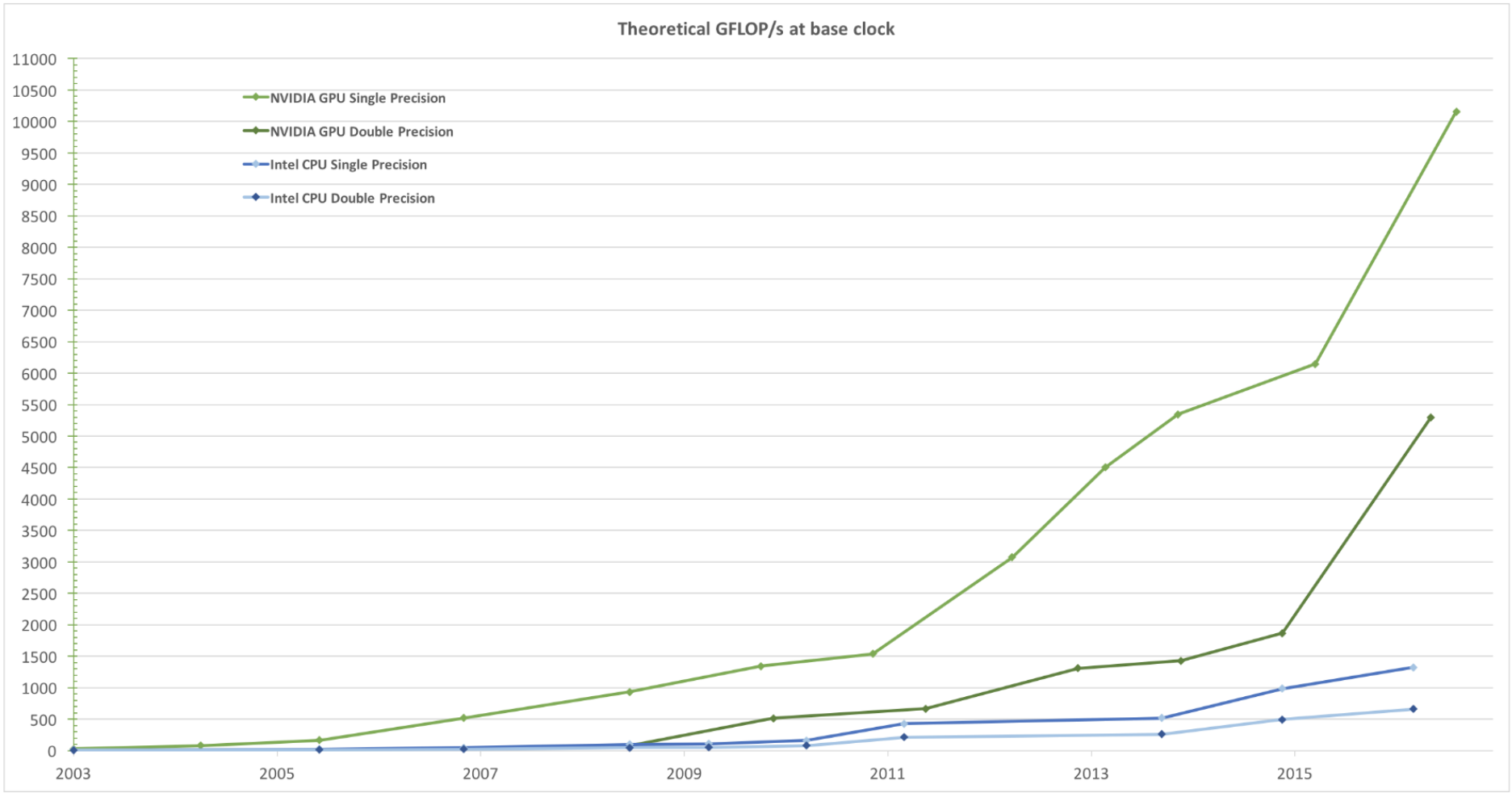
GPU

Throughput Oriented Cores




intel

Floating-point operations per second for the CPU and GPU



When to go to the GPU?

1. Problem is easily scalable because computation is massively parallel.

2. Much more time spent on computation than on communication.

Remember: a GPU is attached to the CPU via a PCI (Peripheral Component Interconnect) Express bus.

CUDA, OpenCL, and OpenACC

- Three approaches to code in the GPU:

  1. **`CUDA`** (Compute Unified Device Architecture):

  2. **`OpenCL`** (Open Computing Language).

  3. **`OpenACC`**.

- Also, using some of the packages/libraries of languages such as **`R`** or **`Matlab`**.

References

- Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors.
- Programming Massively Parallel Processors: A Hands-on (3rd ed.) by David B. Kirk and Wen-mei W. Hwu.
- CUDA Handbook: A Comprehensive Guide to GPU Programming by Nicholas Wilt.
- Heterogeneous Computing with OpenCL 2.0 by David R. Kaeli and Perhaad Mistry.
- OpenACC for Programmers: Concepts and Strategies by Sunita Chandrasekaran and Guido Juckeland.

CUDA

CUDA

- **CUDA** (ComputeUnifiedDeviceArchitecture) was created by Nvidia to facilitate GPU programming.
- It is based on C/C++ with a set of extensions to enable heterogenous programing.
- Introduced in 2007, it is being actively developed (current version 10).
- Many toolboxes: **cuBlas**, **curand**, **cuSparse**, **thrust**.
- It can be access from other languages such as **Fortran**, **Matlab**, **Python**.
- Widely used in data mining, computer vision, medical imaging, bio-informatics.

CUDA: advantages

- Massive acceleration for parallelizable problems.
- Brings `C++11`, since version 7, and `C++14`, since version 9, language features, albeit only a subset available in Device.
- Fast shared memory that can be accessed by threads.
- Rapidly expanding third-party libraries: `OpenCV` machine learning, `CULA` linear algebra, `HIPLAR` linear algebra for `R`.
- Enter `Thrust`:

    1. Library of parallel algorithms and data structures.
    2. Flexible, high-level interface for GPU programming.
    3. A few lines of code to perform GPU-accelerated sort, scan, transform, and reduction operations

Example code

```
// Functions to be executed only from GPU
__device__ float utility(float consumption, float
    ssigma){
  float utility = pow(cons, 1-ssigma) / (1-ssigma);
  // ...
  return(utility);
}
// Functions to be executed from CPU and GPU
__global__ float value(parameters params, float* V,
    ...){
  // ...
}
```

CUDA: disadvantages

- Runs only in Nvidia devices.
- High startup cost. Tricky to program even for experienced programmers.
- Tracking host and device codes.
- Demands knowledge of architecture: grid, blocks, threats. Memory management.
- Copying between host and device may reduce speed gains.
- Not all applications benefit from parallelization.
- Limited community, most information comes from Nvidia and third-party developers.

Additional resources

Some additional books and references for CUDA programming.

- Books:
  1. CUDA by Example, by Jason Sanders and Edward Kandrot.
  2. CUDA C Programming, by John Cheng, Max Grossman, and Ty McKercher.
- References:
  1. **`https://developer.nvidia.com/cuda-zone`**.
  2. **`https://developer.nvidia.com/thrust`**.
  3. **`https://devblogs.nvidia.com/`**

Thrust

Thrust I

- `Thrust` brings the power of GPUs to the masses (at least those familiar with `C++`).
- `Thrust` is a parallel algorithms library in the spirit of `C++`'s Standard Template Library.
- `Thrust`'s main goal is to solve problems that
  1. "can be implemented efficiently without a detailed mapping to the target architecture," and
  2. "don't merit or won't receive (for whatever reason) significant optimization attention from the programmer."
- The idea is that the programmer spends more time on the problem, rather than on the implementation of the algorithms solving the problem.

Thrust II

- Low-level costumization and easy interaction with `CUDA`, `OpenMP`, or `TBB`.[1]
- `Thrust` has two main features.
  1. An STL-style vector container for host and device, and
  2. A set of high-level algorithms for copying, merging, sorting, transforming.
- `Thrust` can be used for parallel computing for multicore CPUs.
- `Thrust` incorporates tuned implementation for each backend: `CUDA`, `OpenMP`, and `TBB`
- This results in portability across parallel frameworks and hardware architecture without losing performance.

[1] Intel's `TBB` – Threading Building Blocks – is a `C++` template library for task parallelism.

Thrust III

- Of course, **Thrust** has limitations.
- No multidimensional data structures libraries.
- **Thrust** is "entirely defined in header files." Hence, each modification in code requires recompilation.
- **Thrust** is not for situations in which performance, customization are crucial.
- Documentation is limited and mostly based on examples. But it has improved over the years.
- Although the last release, version 1.8.1, dates back to 2015, Nvidia seems to be working on an update.[2]

[2] https://www.reddit.com/r/cpp/comments/7erub1/anybody_still_using_thrust/

Example code

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <algorithm>
#include <cstdlib>

int main(void){
    // generate random data serially
    thrust::host_vector<int> h_vec(100);
    std::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer to device and compute sum
    thrust::device_vector<int> d_vec= h_vec;
    int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0,
         thrust::plus<int>());
    return 0;
    }
```

Thrust IV

- Let's take a more detailed peek at some of **Thrust**'s capabilities.
- **Thrust** provides two vector containers:
  1. host_vector stored in the CPU's memory

     thrust::host_vector<int> hexample(10,1) // host vector with 10 elements set to 1
  2. device_vector resides in the GPU's device memory.

     thrust::device_vector<int> dexample(hexample.begin(),hexample.begin()+5) // device vector with first 5 elements of hexample
- Some algorithms that operate on vectors are: thrust::fill(), thrust::copy(), thrust::sequence().
- Last algorithm creates a sequence of equally spaced values.

Thrust V – Algorithms

- Transformations are “algorithms that apply an operation to each element in a set of input ranges and stores result in destination range.”

- Compute $Y = -X$:

  thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>());

- Compute $Y = X mod 2$

  thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), thrust::modulus<int>());

Thrust V – Algorithms

- Reduction uses “a binary operation to reduce an input sequence to a single value.”
- Sum elements in device vector Y:

```
int sum = thrust::reduce(Y.begin(), Y.end(), (int) 0, thrust::plus<int>());
```

- **`Thrust`** includes other reduction operations:
  1. thrust::count number of instances of specific value,
  2. thrust::min_element // find minimum in vector,
  3. thrust::max_element,,
  4. thrust::inner_product,,

Thrust VI – Algorithms

- **`Thrust`** offers many more algorithms to, for example, reordering, sorting, and prefix-sums.

- Another important feature in **`thrust`** is the fancy iterators.

    1. thrust::constant_iterator< > iterator returns same value when dereference it,

    2. transform_iterator,

    3. permutation_iterator fuse, gather, and scatter operations with **`thrust`** algorithms,

    4. zip_iterator takes multiple input sequences and yields a sequence of tuples.

       zip_iterator can be used to create “a virtual array of 3d vectors” that can be fed to other algorithms.

Additional resources

Some books and references for **thrust** programming.

- Books:
  1. Sorry! No books that we are aware of.
- References:
  1. **https://devblogs.nvidia.com/expressive-algorithmic-programming-thrust/**.
  2. **https://github.com/thrust/thrust/wiki/Quick-Start-Guide**.
  3. **http://thrust.github.io/**.
  4. **https://docs.nvidia.com/cuda/thrust/index.html**

OpenACC

OpenACC I

- Like **Thrust**, **OpenACC** tries to bring heterogenous HPC to the masses.
- Its motto is “More Science, Less Programming.”
- **OpenACC** is a “user-driven directive-based performance-portable parallel programming model.”
- Main idea is to take existing serial code, say **C++**, and give hints to compiler to what should be parallelized.
- **OpenACC** is a model designed to allow parallel programming across different computer architectures with minimum effort by the developer. Portability means that the code should be independent of hardware/compiler.
- **OpenACC** specification supports **C/C++** and **Fortran** and runs in CPUs and GPUs.

OpenACC II

- **OpenACC** is built around a very simple set of directives, very similar in design to **OpenMP**: **OpenACC** uses the **fork-join** paradigm.

- The same program can be compiled to be executed in parallel using the CPU or the GPU (or mixing them), depending on the hardware available.

- Communication between the master and worker threads in the parallel pool is automatically handled, although the user can state directives to grant explicit access to variables, and to transfer objects from the CPU to the GPU when required.

- The OpenACC website describes multiple compilers, profilers, and debuggers for **OpenACC**.

- We use the **PGI Community Edition** compiler. The **PGI** compiler can be used with the CPUs and with NVIDIA Tesla GPUs. In this way, it suffices to select different flags at compilation time to execute the code in the CPU or the GPU.

Using OpenACC I: Analyze

- Use a profiler to check where your code spends lots of time. Example of bottlenecks are loops.

- Check if there is an optimized library that implements some of your code: `cuBlas`, `Armadillo`.

Using OpenACC II: Parallelize

- Expose your code to parallelism starting with functions/operations that are time consuming on CPU.

- To initiate parallel execution:

```
# pragma acc parallel
```

- To execute a kernel:

```
# pragma acc parallel kernel
```

- To parellelize a loop:

```
# pragma acc parallel for
```

Example

- Example code of parallelizing a loop in C++.
- Basic parallel loop:

```
#pragma acc parallel loop
for(int ix=0; ix<nx; ix++){
    // ...
}
```

- Ensuring copy of relevant objects:

```
#pragma acc data copy(...)
#pragma acc parallel loop
for(int ix = 0; ix<nx; ix++){
    //...
}
```

Example code II

- By choosing the appropriate compilation flag, we can compile the code to be executed in parallel only in the CPU or in the GPU.
- To compile the code to be executed by the CPU, we must include the `-ta=multicore` flag at compilation. In addition, the `-acc` flag must be added to tell the compiler this is `OpenACC` code:

```
pgc++ Cpp_main_OpenACC.cpp -o Cpp_main -acc
    -ta=multicore
```

- If, instead, we want to execute the program with an NVIDIA GPU, we rely on the `-ta=nvidia` flag:

```
pgc++ Cpp_main_OpenACC.cpp -o Cpp_main -acc
    -ta=nvidia
```

Using OpenACC III: Optimize

- Give info to compiler of parts that can be optimize: data management (minimize copying between host and device).
- Instruct compiler how to parallelize loops.
- Step 3 is not trivial and maybe involved, limiting `OpenACC`'s applicability.

Additional resources

Some books and references for **OpenACC** programming.

- Books:
  1. OpenACC Programming and Best Practices Guide, 2015.
- References:
  1. **https://devblogs.nvidia.com/tag/openacc/**.
  2. **https://www.openacc.org/**.

Comparisons

Comparisons I

- All results are specific to our life-cycle model example.
- There are other ways to improve speed on each language:
  - Function wrapping in **`Julia`**.
  - Vectorizing in **`Matlab`**.
  - Etc.

Comparisons II

- The comparisons regarding parallelization are specific to the packages used on these slides:

	Community	Speed	Parallelization		Time to program	Debug
			Difficulty	Improvement		
Matlab	Large	Medium	Easy	Low	Fast	Easy
Julia	Very small	Fast	Medium	High	Fast	Easy
R	Large	Slow	Medium	High	Fast	Easy
Python	Large	Slow	Medium	High	Fast	Easy
C++	Large	Fast	Easy	High	Slow	Difficult

Advice

- Short-run:
  MATLAB, Python, Julia, or R
- Medium-run:
  Rcpp
- Long-run:
  C++ with OpenMP, MPI, or GPUs

Final comparison