Parallel Programming

(Lectures on High-performance Computing for Economists XI)

Jesús Fernández-Villaverde,[1] Pablo Guerrón,[2] and David Zarruk Valencia[3]

March 1, 2022

[1]University of Pennsylvania

[2]Boston College

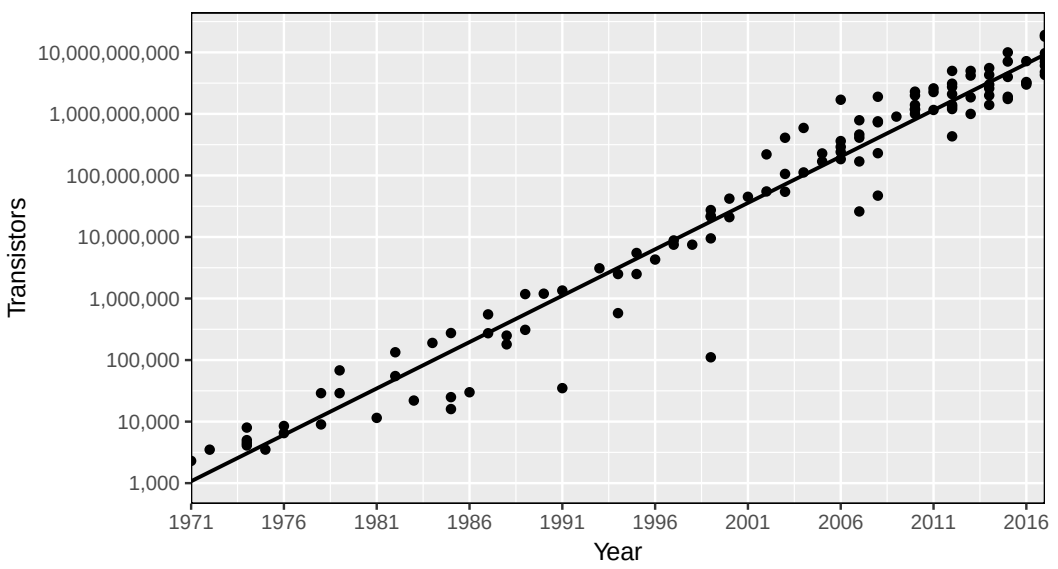
[3]ITAM

Motivation

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

Fujitsu-Riken Fugaku, 2021: 158,976 x 48-core

Parallel programming

- Main idea $\Rightarrow$ divide a complex problem into easier parts:
  1. Numerical computation $\Longrightarrow$ matrix multiplication.
  2. Data handling (`MapReduce` and `Spark`) $\Longrightarrow$ computing moments.
- Two issues:
  1. Algorithms.
  2. Coding.

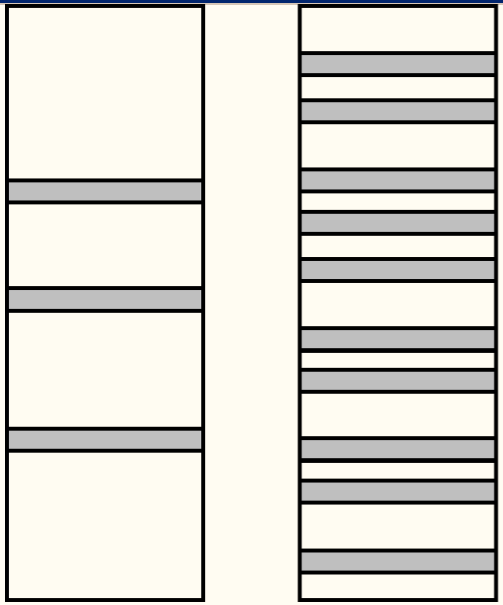
Some references

- *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



Communication

Computation

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

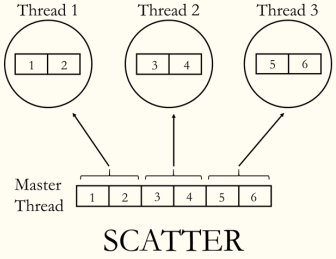
1. Set $t = T$.
2. Given t, the computation of $V(t, e_j, x_i)$ is independent of the computation of $V(t, e_{j'}, x_{i'})$, for $i \neq i'$, $j \neq j'$.
3. One processor can compute $V(t, e_j, x_i)$ while another processor computes $V(t, e_{j'}, x_{i'})$.
4. When the different processors are done at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

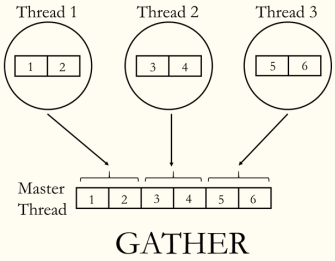
Note that the problem is not parallelizable on t. The computation of $V(t, e, x)$ depends on $V(t + 1, e, x)$!

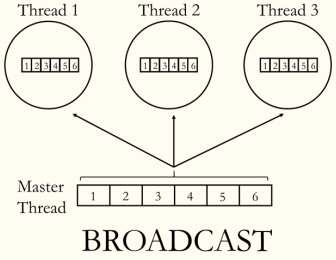
Computational features of the model

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.

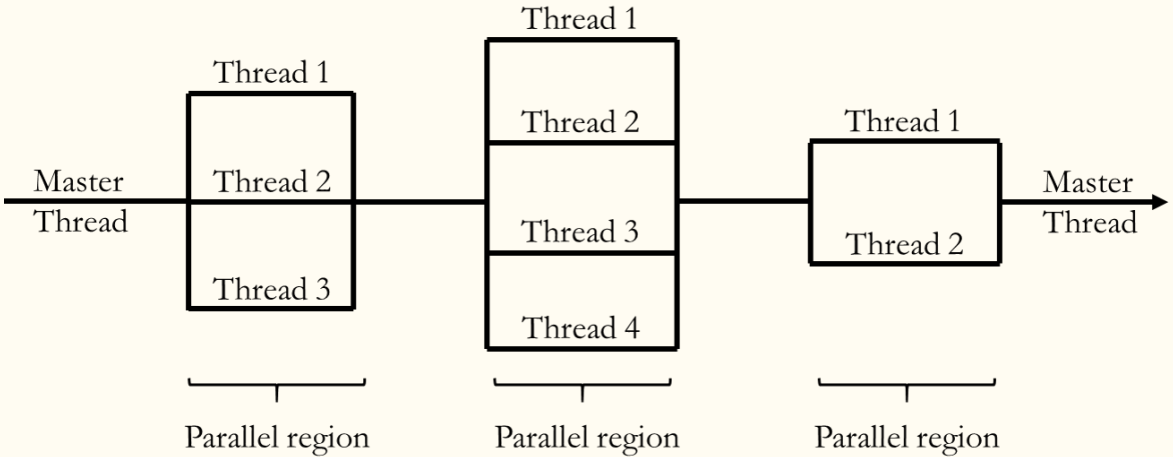
In parallel


Thread 1
Thread 2
Thread 3
1 2
3 4
5 6
Master Thread
1 2 3 4 5 6
SCATTER


Thread 1
Thread 2
Thread 3
1 2
3 4
5 6
Master Thread
1 2 3 4 5 6
GATHER


Thread 1
Thread 2
Thread 3
1 2 3 4 5 6
Master Thread
1 2 3 4 5 6
BROADCAST

Parallel execution of the code



Many workers instead of one

Figure 1: 1 Core Used for Computation

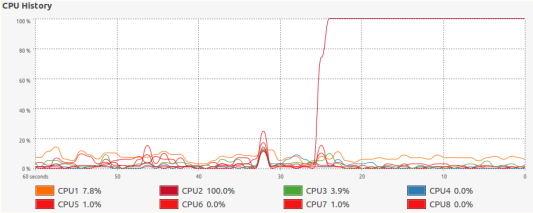
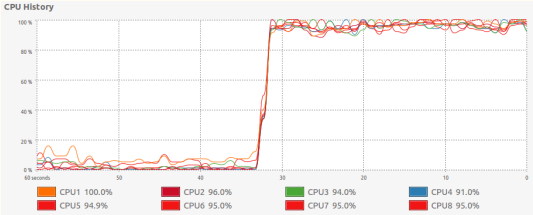


Figure 2: 8 Cores Used for Computation



Parallelization limits

Costs of parallelization

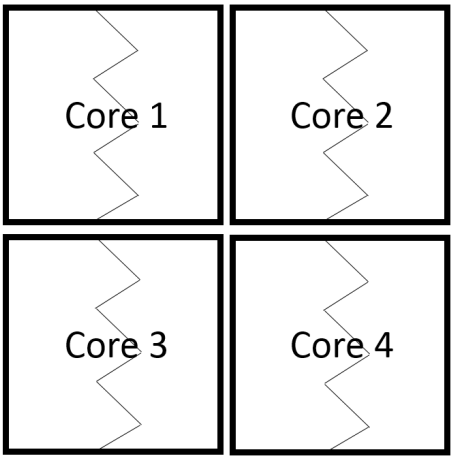
- Amdahl's Law: the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

- 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, 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.

Your laptop is not the limit!

- Cluster servers.
- 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 16Gib, general purpose: $0.204 per hour.
    - 36 processors with 60Gib, compute optimized: $1.591 per hour.
    - 448 processors with 6144 GiB, compute optimized: $54.60 per hour.

Running an instance on AWS

- Go to: https://console.aws.amazon.com/
- Click on EC2.
- Click on Launch Instance and follow the window links (for example, Ubuntu Server 18.04).
- Public key:
  - Create a new key pair.
  - Download key.
  - Store it in a secure place (usually ∼./ssh/).
- Run instance.

Working on AWS instance

On Unix/Linux terminal:

- Transfer folder from local to instance with `scp`:

```
$ scp -i "/path/"Penn_Fall_2021.pem"" -r "/pathfrom/FOLDER/"
    ec2-user@ec2-34-226-147-93.compute-1.amazonaws.com:~
```

- Make sure key is not publicly available:

```
$ chmod 400 "Penn_Fall_2021.pem"
```

- Connect to instance with `ssh`:

```
$ ssh -i "Penn_Fall_2021.pem"
    ec2-35-173-135-140.compute-1.amazonaws.com
```

Also, on your browser.

Parallelization

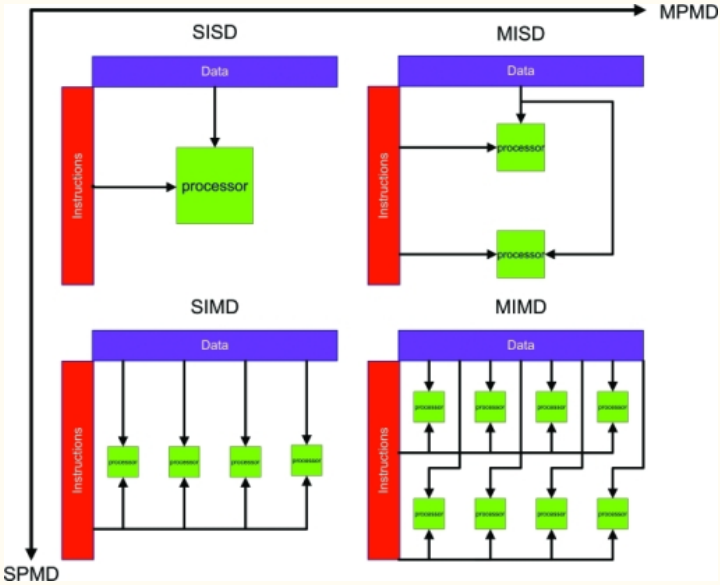
Programming modes I

- 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.

Programming modes II

- Less common in economics.
  1. Automatic parallelization: `AutoParInGCC`, `Intel compilers`.
  2. Partitioned Global Address Space Languages (PGAS):
     - 2.1 `Coarray Fortran`.
     - 2.2 `UPC`.
     - 2.3 `X10`.
     - 2.4 `Chapel`.
  3. Pthreads (POSIX threads).
  4. TPUs (Tensor processing units).
  5. FPGAs (field programmable gate arrays).
  6. Hybrids.

Flynn's taxonomy



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!

FIGURE 3: Julia - 1 core used for computation

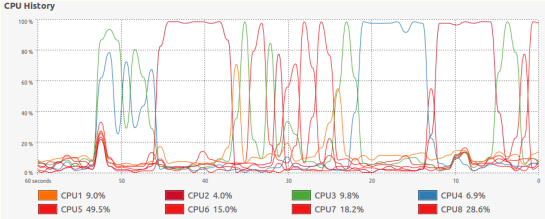
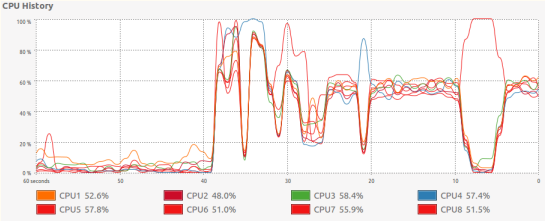
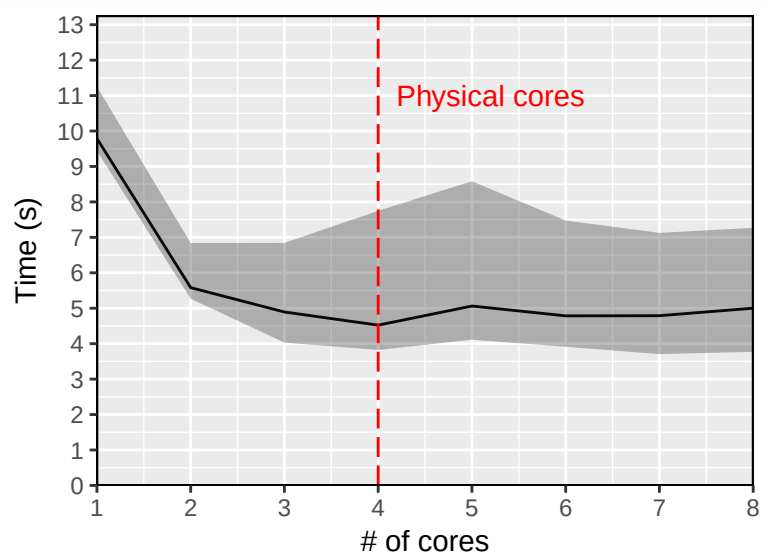


FIGURE 4: Julia - 8 cores used for computation



Parallelization in Julia - Performance



Parallelization in Julia - *for* loops

- Speed decreases with the number of global variables used.
- Very sensible 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.
- See examples 3 and 4 on github.

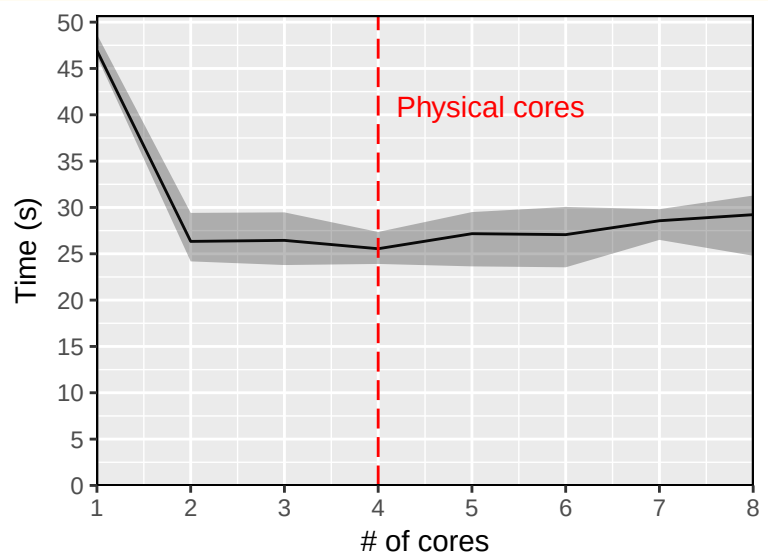
Parallelization in Julia - *Map*

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



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
        # ...
```

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];
```

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))
```

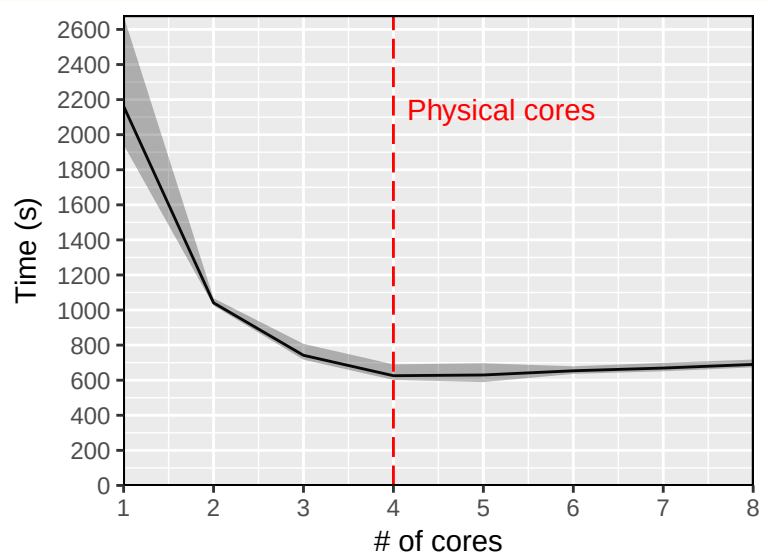
maps the function `value_func` at every element of `modelState(ix, age, ..., w, r)` in parallel using `num_cores` cores.

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



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))
```

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

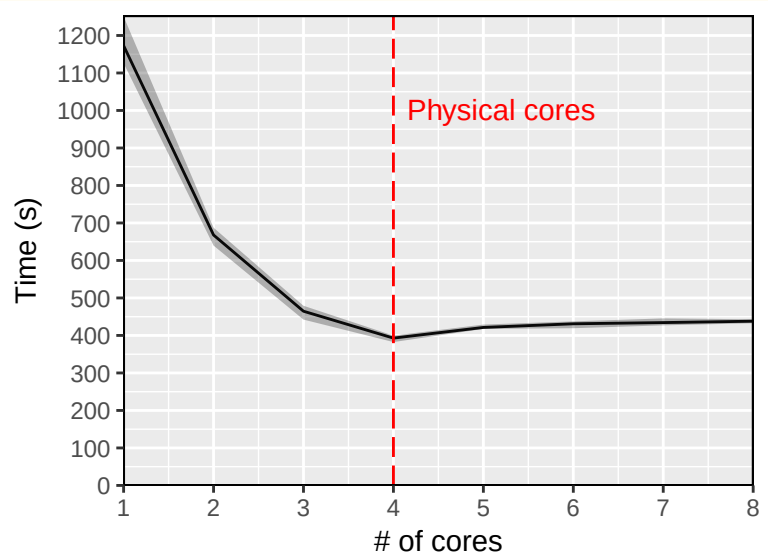
4. Define the cluster with desired number of cores:

```
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



Matlab

Parallelization in Matlab - *for* loop

Using the `parallel toolbox`:

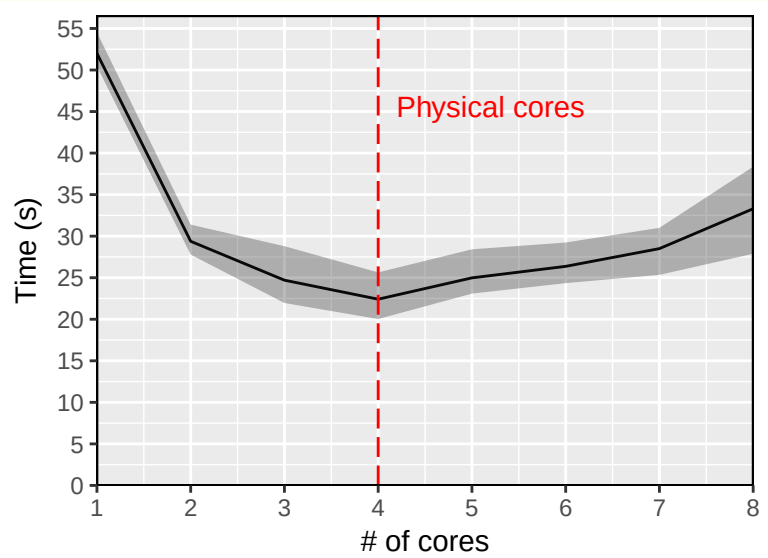
1. Initialize number of workers with `parpool()`:

```
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



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.

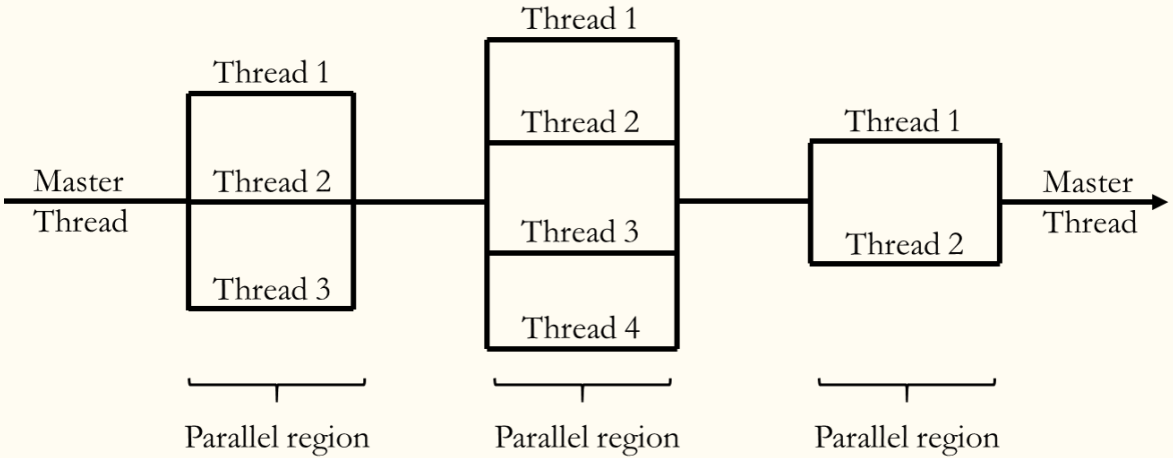
OpenMP II

- 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


Thread 1
Thread 1
Thread 2
Thread 1
Master
Thread
Thread 2
Thread 3
Thread 3
Master
Thread
Thread 4
Thread 2
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
```

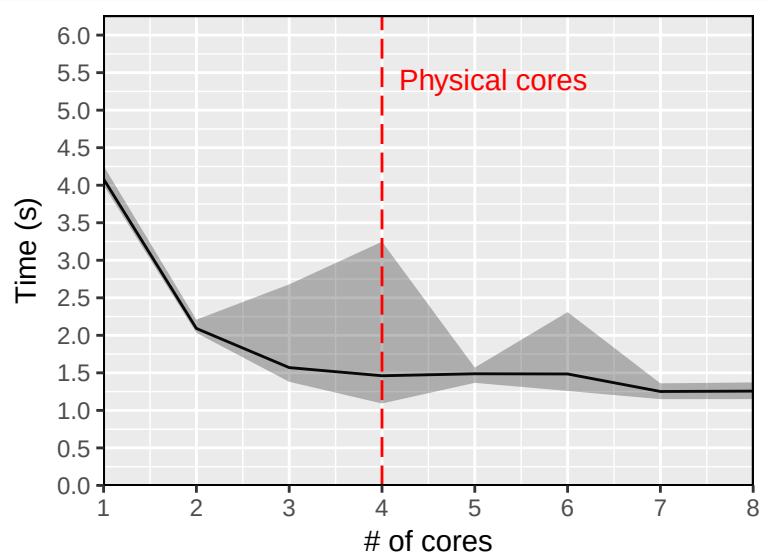
3. Add line before loop:

```
#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 nested parallelization and dynamic thread changes.

Parallelization in C++ using OpenMP - Performance



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")
```

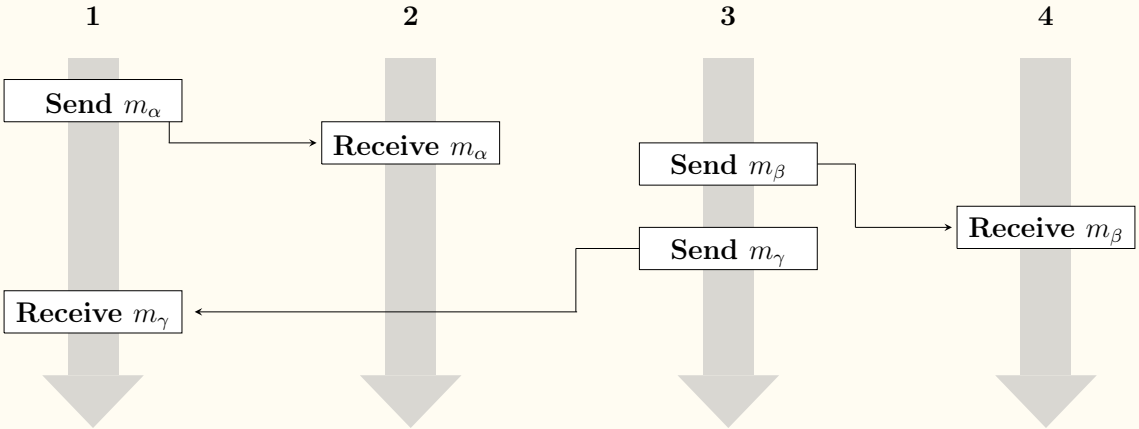
MPI

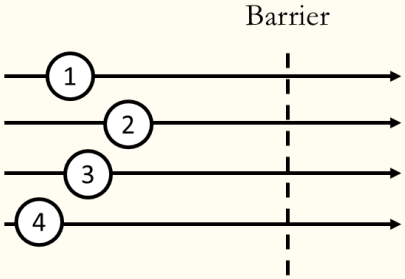
MPI I

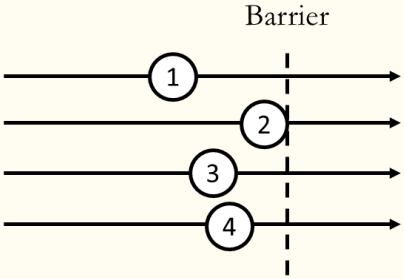
- Message Passing Interface (`MPI`) is a standardized and portable message-passing system based on the consensus of the MPI Forum.
- Official web page (and for downloads): `http://www.open-mpi.org/`
- Tutorial: `https://computing.llnl.gov/tutorials/mpi/`
- A couple of references:
  1. *Using MPI : Portable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

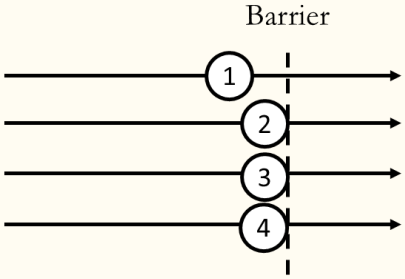
MPI II

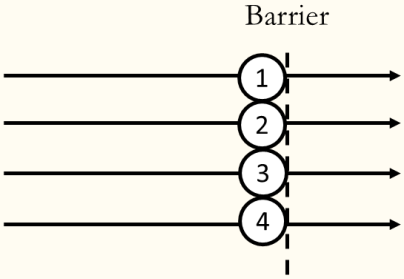
- `MPI` is organized as a library performed with routine calls.
- Bindings for `C++ and Fortran`. Also for `Python`, `Julia`, `R`, and other languages.
- For example, you can have `MPI` in `Mex` files in `Matlab`.
- Harder to learn (`MPI 3.0` standard has more than 440 routines) and code, but extremely 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

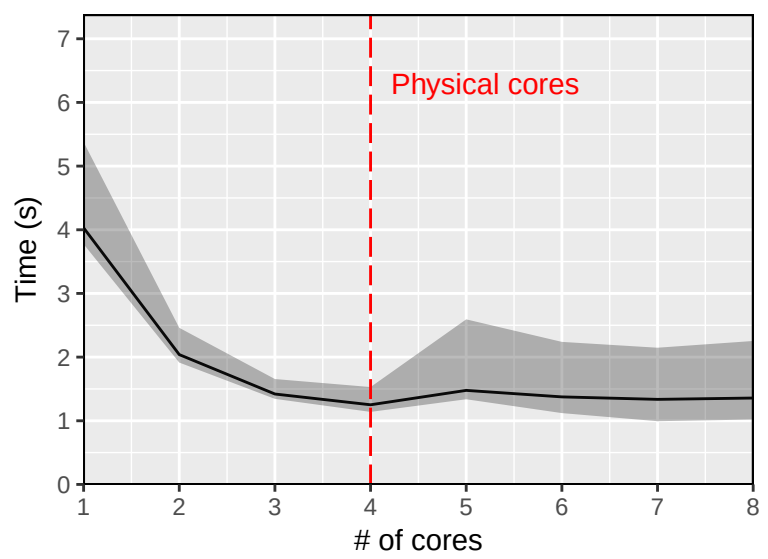
```
#include "mpi.h"
#include <iostream>
int main( int argc, char *argv[] ){
   int rank, size;
   MPI::Init(argc, argv);
   rank = MPI::COMM\_WORLD.Get\_rank();
   size = MPI::COMM\_WORLD.Get\_size();
   std::cout<< "I am " << rank << " of " << size << \qquad "n";
   MPI::Finalize();
   return 0;
 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

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

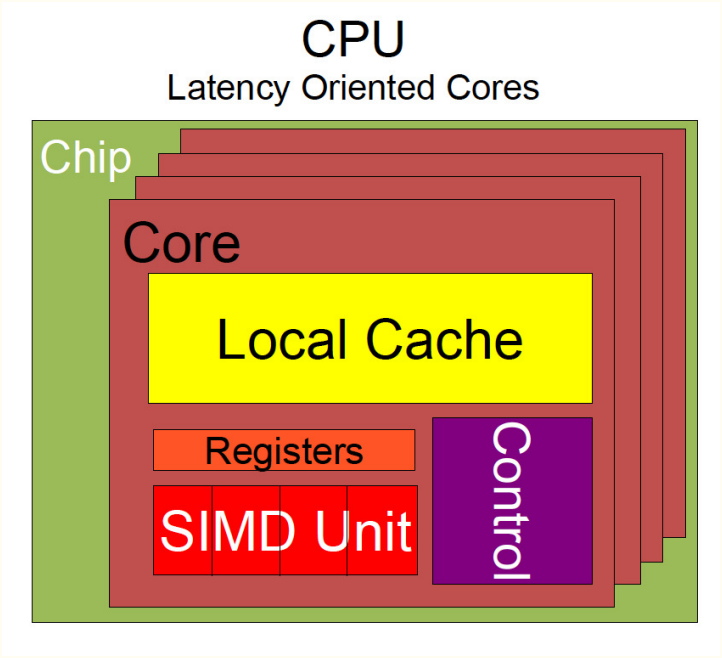
- MPI predefines its primitive data types:
  1. MPI_CHAR
  2. MPI_DOUBLE_PRECISION
  3. MPI_C_DOUBLE_COMPLEX
- Also for structs and vectors.
- Particularly important for top performance.

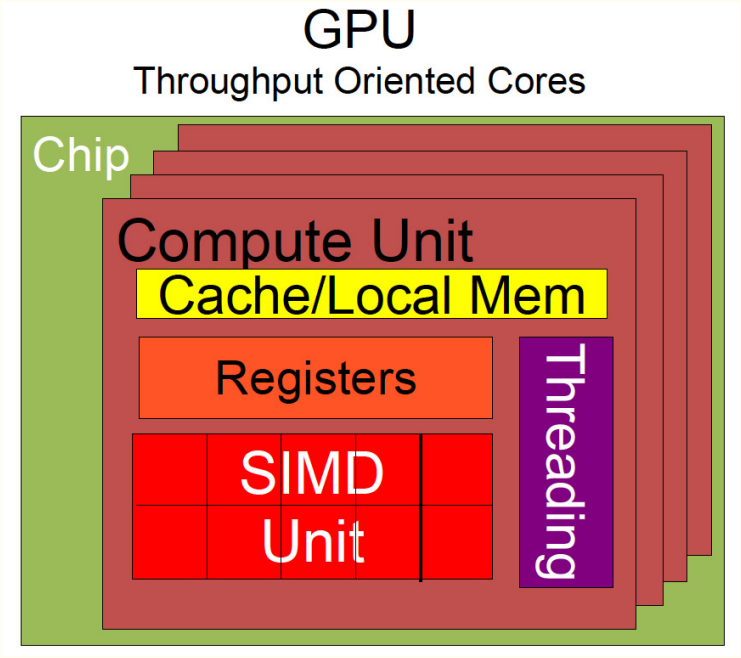

Physical cores
Time (s)
# of cores
0
1
2
3
4
5
6
7
1
2
3
4
5
6
7
8

GPUs

Big difference

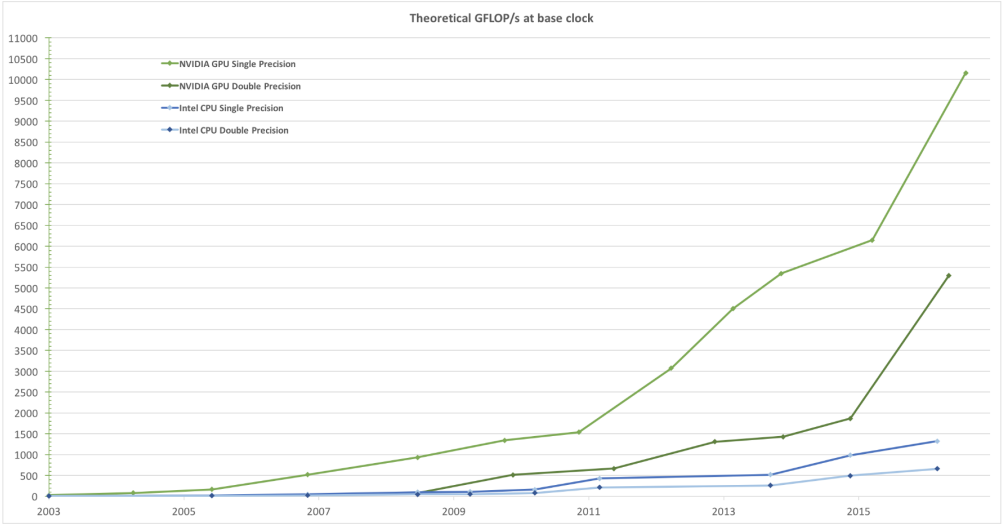
- Latency: amount of time required to complete a unit of work.
- Throughput: amount of work completed per unit of time.
- Latency devices with larger memory: CPU cores.
- Throughput devices with memory with more bandwidth: GPU cores.
- Intermediate: Coprocessors.
- Nature of your application?


CPU
Latency Oriented Cores
Chip
Core
Local Cache
Registers
SIMD Unit
Control


GPU
Throughput Oriented Cores
Chip
Compute Unit
Cache/Local Mem
Registers
SIMD
Unit
Threading


intel
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 `Julia`, `R`, or `Matlab` (Parallel Computing Toolbox).

References

- Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors.
- *Programming Massively Parallel Processors: A Hands-on (4th 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 can use the `GCC` compiler. The `GCC` compiler can be generate code to be run with CPUs and with NVIDIA 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 parallelized library that implements some of your code: `cuBlas`, `Armadillo`.
- Think about incremental changes to your code: add annotations to expose parallelism.
- Rerun the code to verify correctness and performance.

Using OpenACC II: Pragmas

- A pragma in C/C++ gives instructions to the compiler. If the compiler does not understand the pragma, it will ignore it.
- Generic structure:

```
# pragma acc directive clauses
<code>
```

- A directive is a commands in OpenACC to change the compilation and execution of the code.
- Clauses are modifies of the directives.

Using OpenACC III: Parallelize

- Expose your code to parallelism starting with functions/operations that are time consuming on CPU.
- To initiate parallel execution:

```
# pragma acc parallel
{<code>}
```

- To execute a kernel:

```
# pragma acc parallel kernel
```

- To parallelize a loop:

```
# pragma acc parallel loop
```

- Differences between kernels and loops.

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 copyin(a,b) copyout(c)
#pragma acc parallel
{
   #pragma acc parallel loop
   for(int ix = 0; ix<nx; ix++){
      c[ix] = a[ix]+b[ix];}
}
```

Compilation

- 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 `-fopenacc` flag at compilation.

```
g++ Cpp_main_OpenACC.cpp -o Cpp_main -fopenacc
```

- We can also specify concrete hardware we want to use.

Using OpenACC IV: Optimize

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

```
#pragma acc parallel loop collapse(2)
for(int ix=0; ix<nx; ix++){
   for(int iy=0; iy<ny; iy++){
   a[ix,iy] = 0.0;
}
```

- Also instructions for tiles, gangs, workers, and vectorization.

Additional resources

Some books and references for OpenACC programming.

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

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.

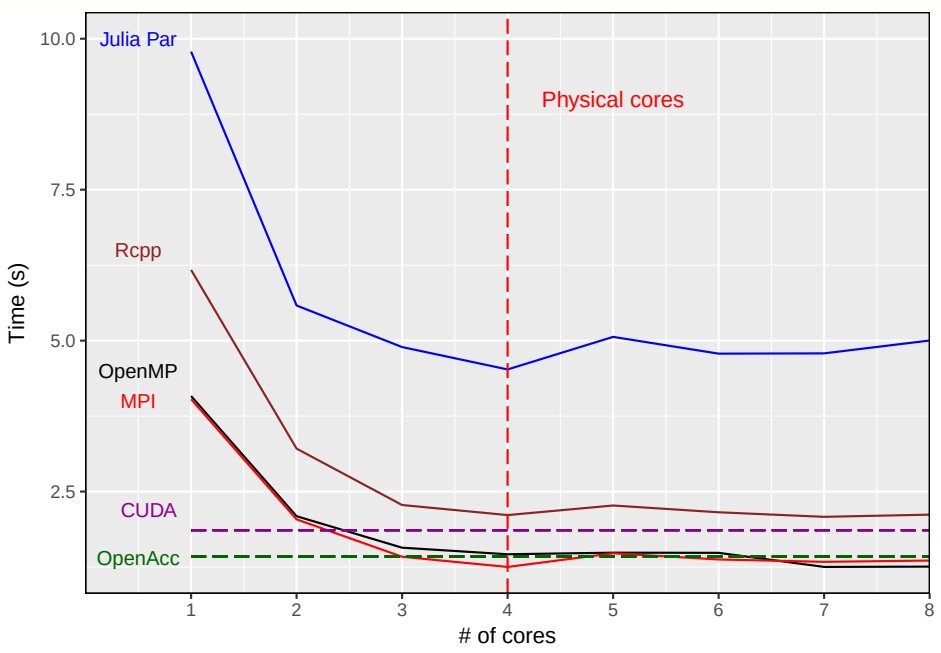
Comparisons II

- 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


Julia Par
Physical cores
Rcpp
OpenMP
MPI
CUDA
OpenAcc
Time (s)
10.0
7.5
5.0
2.5
1
2
3
4
5
6
7
8
# of cores