Notes on Parallelization

Jesús Fernández-Villaverde
David Zarruk Valencia

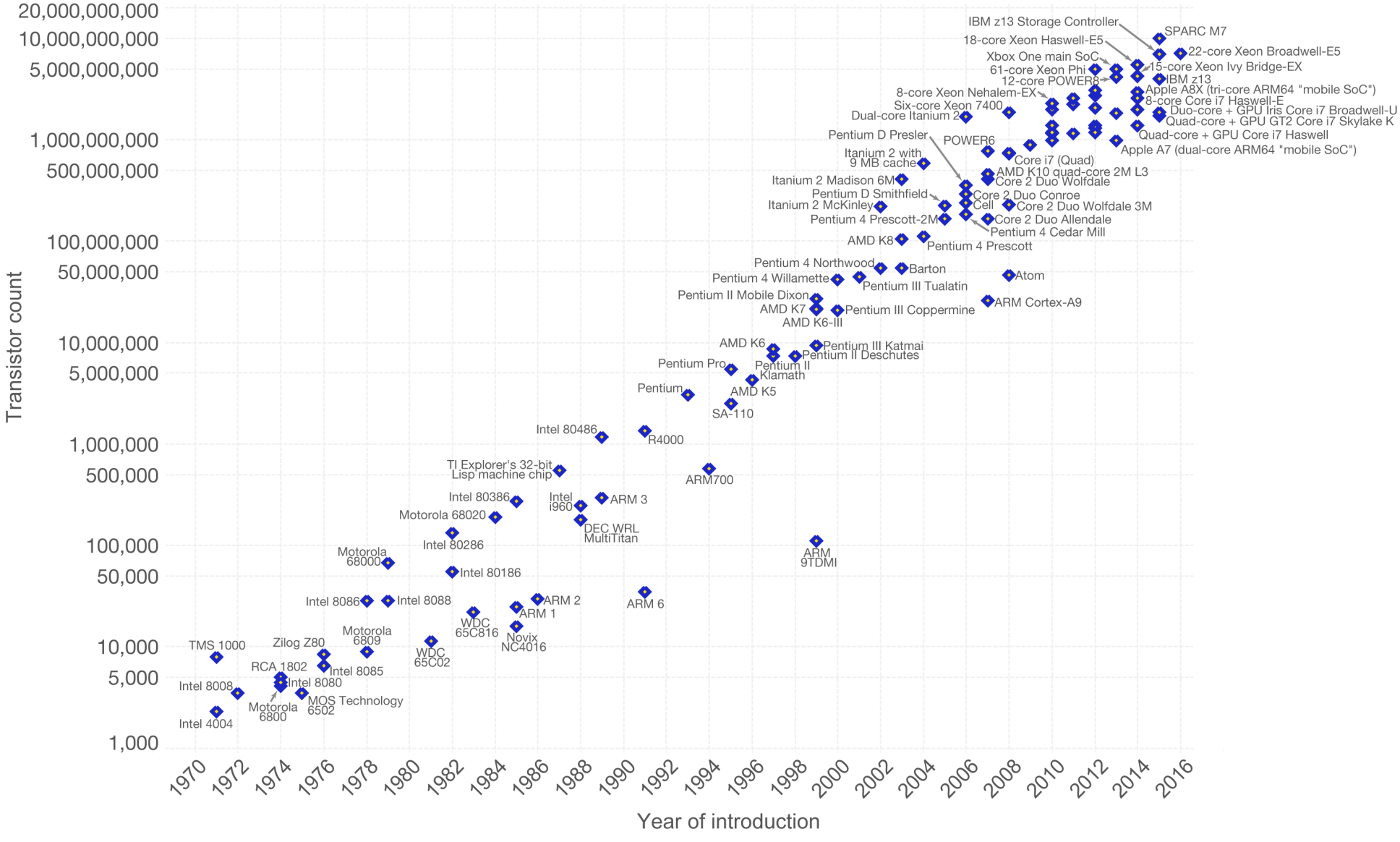
October 11, 2017

Table of Contents

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

Sunway TaihuLight, 2016

Parallel programming

- Main idea⇒divide complex problem into easier parts:
  1. Numerical computation.
  2. Data handling (`MapReduce` and `Hadoop`).
- Two issues:
  1. Algorithms.
  2. Coding.

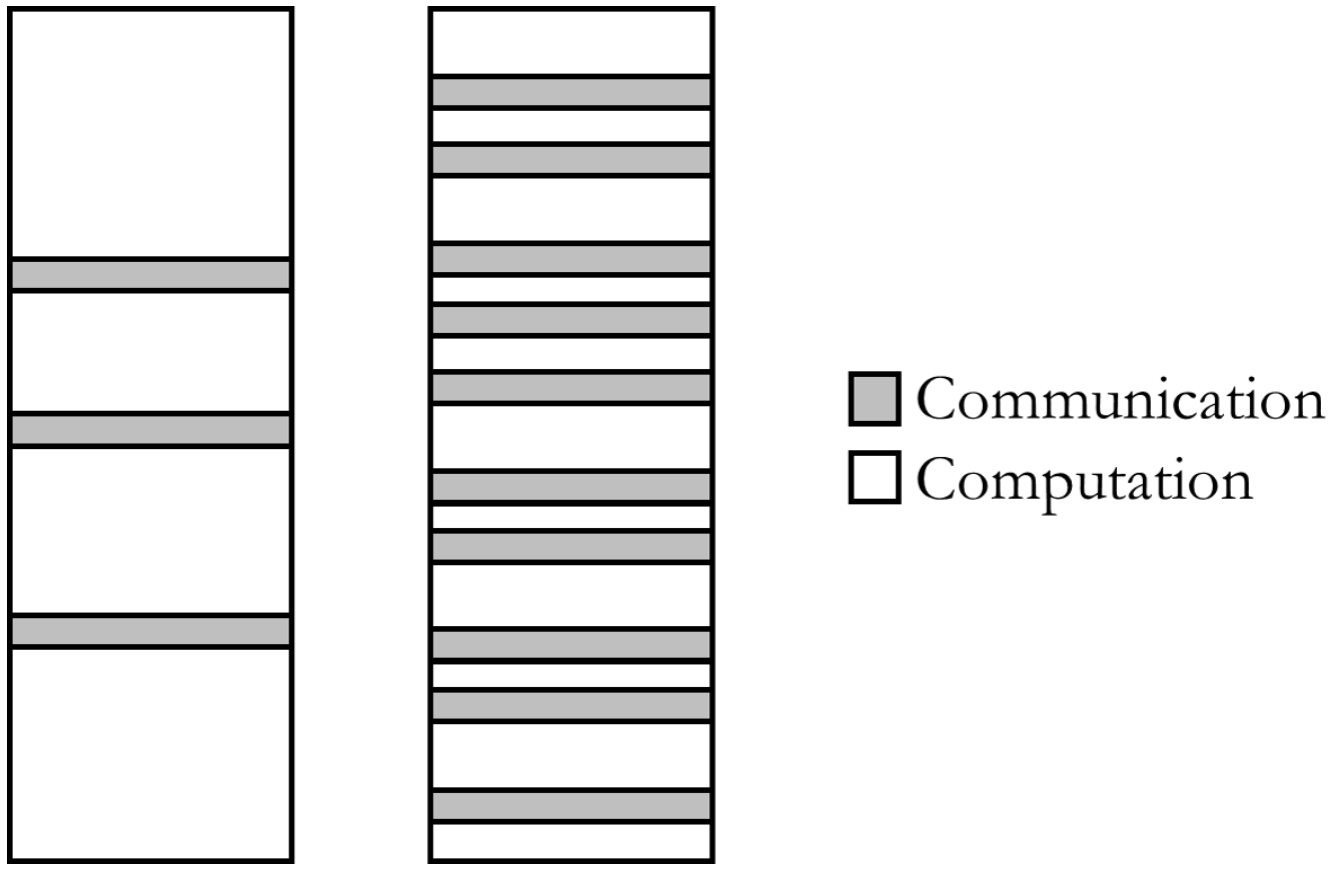
Some references

- Parallel Programming for Multicore and Cluster Systems by Thomas Rauber and Gudula Rünger.
- 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.
- Introduction to High Performance Computing for Scientists and Engineers by Georg Hager and Gerhard Wellein.

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

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

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:

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

   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).

Table of Contents

Life-Cycle Model

- Households solve:

$$V(t,e,x) = \max_{\{c,x'\}} \frac{c^{1-\sigma}}{1-\sigma} + \beta \mathbb{E} V(t+1,e',x') \quad s.t.$$

$$c + x' \leq (1+r)x + ew$$

$$\mathbb{P}(e'|e) = \Gamma(e)$$

$$x' \geq 0$$

$$t \in \{1,\ldots,T\}$$

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

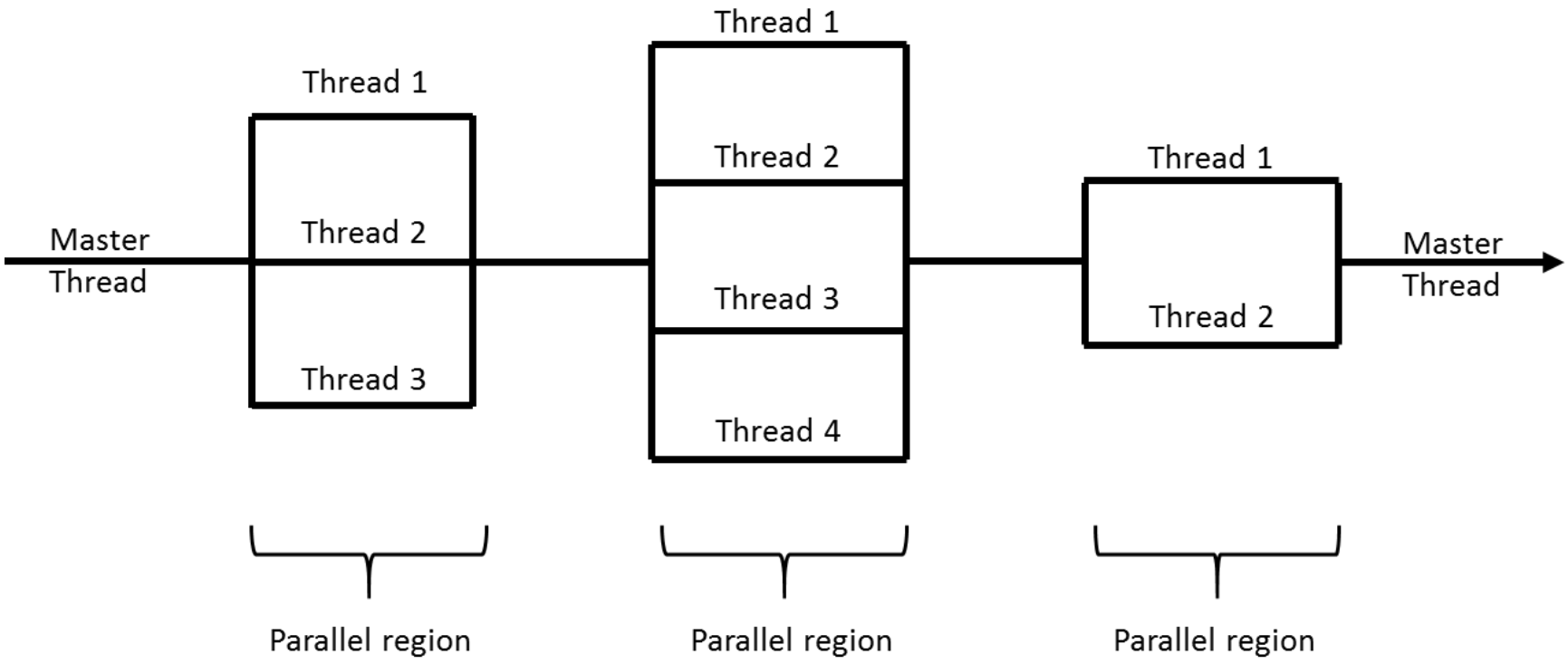
```
for(age = T:-1:1)
  for(ix = 1:nx)
    for(ie = 1:ne)
      VV = -10^3;
      for(ixp = 1:nx)

        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 $V(t + 1, e, x)$!

Parallel Execution of the Code



Many Workers Instead of One

Figure : 1 Core Used for Computation

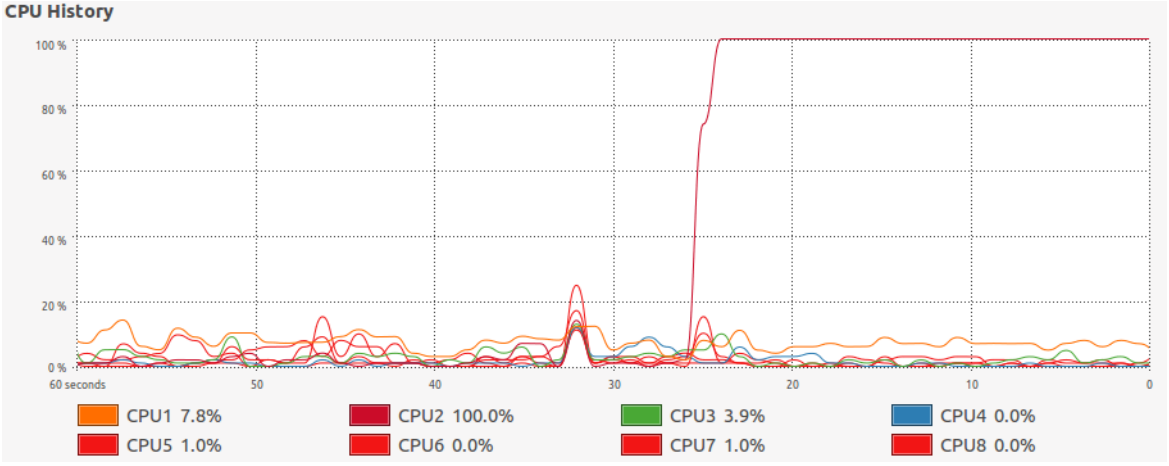
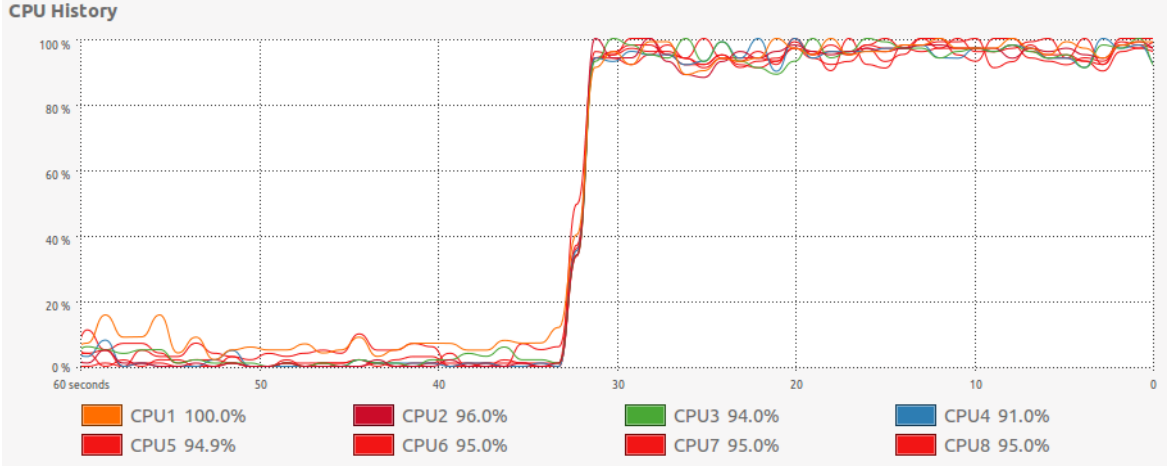


Figure : 8 Cores Used for Computation



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.

Table of Contents

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.

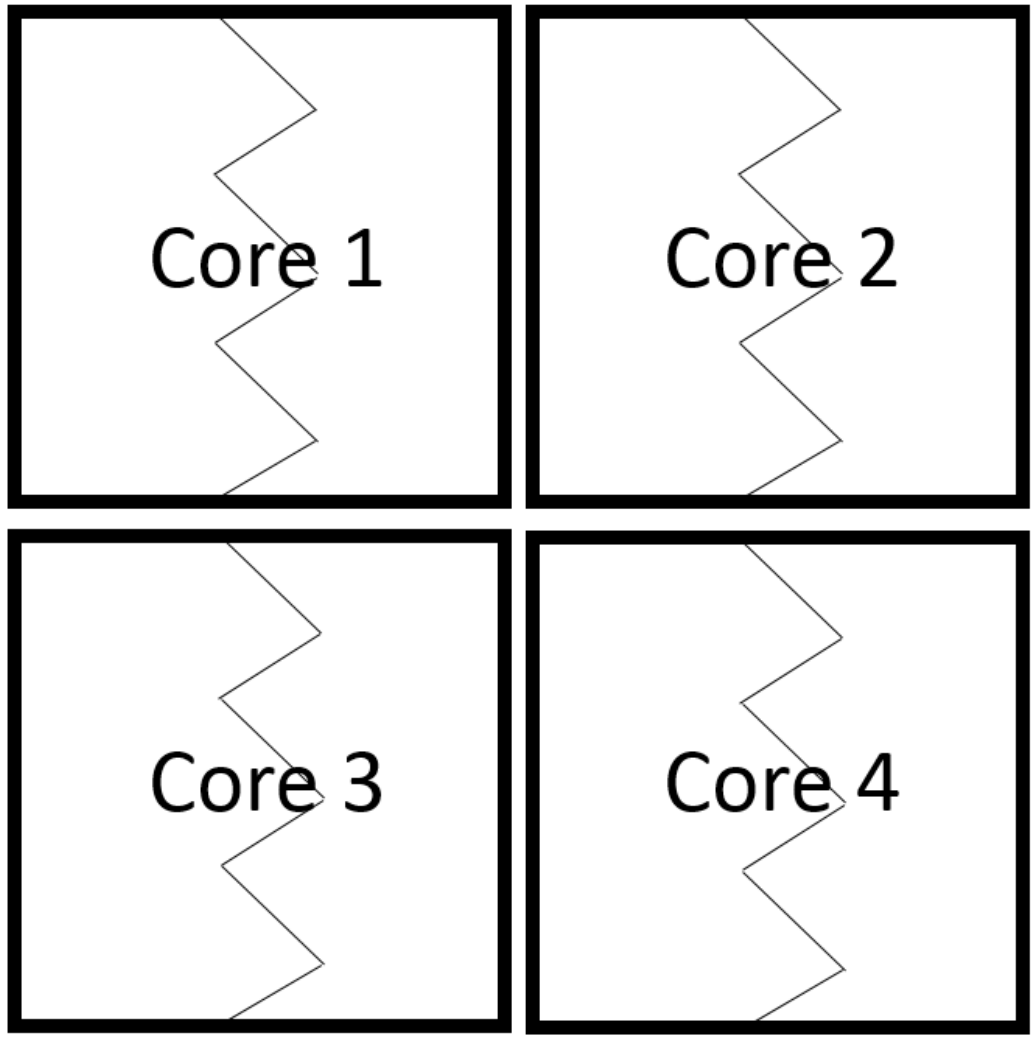
- Costs:
  - Starting a thread or a process/worker.
  - Transferring shared data to workers.
  - Synchronizing.

- 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:

  4 physical cores + 4 virtual cores = 8 logical cores



Multi-core Processors

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!

- Many other resources:
  - Tesla server:
    - 61 Cores
  - Hawk server:
    - 72 Cores
  - Amazon Web Services - EC2:
    - Almost as big as you want!

Amazon Web Services

- 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: $0.479 per hour.
  - 64 processors with 256Gb, compute optimized: $3.83 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 14.04).
- Public key:
  - Create a new key pair.
  - Download key.
  - Store it in a secure place (usually $\sim$`./ssh/`).
- Run instance.

Working on AWS instance

On Ubuntu terminal:

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

```
$ scp -i "/path/PUBLICKEY.pem" -r "/pathfrom/FOLDER/"
    ubuntu@52.3.251.249:~
```

- Make sure key is not publicly available:

```
$ chmod 400 "/path/PUBLICKEY.pem"
```

- Connect to instance with `ssh`:

```
$ ssh -i "/path/PUBLICKEY.pem" ubuntu@52.3.251.249
```

Table of Contents

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.

Table of Contents

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. Set number of workers:

```
addprocs(6)
```

2. Variables are not observable by workers $\Longrightarrow$ Declare the variables inside the parallel *for* loop that are not modified inside parallel iterations to be global:

```
@everywhere T = 10;
#...
@everywhere gridx = zeros(nx);
```

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

```
V = SharedArray(Float64, (T, nx, ne),
      init = V -> V[Base.localindexes(V)] = myid());
```

Parallelization in Julia - *for* loops

4. For paralellizing a *for* loop, add `@parallel` before the `for` statement:

```
@parallel for(ix = 1:1:nx)
    # ...
end
```

5. To synchronize before the code continues its execution, add `@sync` before the `@parallel for` statement:

```
@sync @parallel for(ix = 1:1:nx)
    # ...
end
```

Parallelization in Julia - *for* loops

- Choose appropriately the dimension(s) to parallelize:

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

```
nx = 350;
ne = 9;
for(ix = 1:nx)
  @sync @parallel 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 @parallel 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 - *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 examples 1 and 2 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. Initialize number of workers:

```
addprocs(6)
```

2. To avoid declaring all variables as global (makes computation slower), define a structure of inputs:

```
@everywhere type modelState
    ix::Int64
    age::Int64
    # ...
end
```

Parallelization in Julia - *Map*

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

```
@everywhere function value(currentState::modelState)
    ix      = currentState.ix;
    age     = currentState.age;
    # ...
    VV      = -10^3;
    for(ixp = 1:nx)
       # ...
    end
    return(VV);
end
```

Parallelization in Julia - *Map*

4. 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 - 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).

Table of Contents

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];
```

Table of Contents

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]
  }
}
```

Table of Contents

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

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

Table of Contents

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 (`Eclipse`).
- 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.

OpenMP IV

- Compiler directives to tell what to parallelize:

```
#pragma omp parallel default(shared) private(beta,pi)
```

- Compiler generates explicitly threaded code when OpenMP flag is invoked (`-fopenmp`).

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

- Most implementations (although not the standard!) allow for nested parallelization and dynamic thread changes.

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++){
   // ...
}
```

Table of Contents

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")
```